Algorithms and Data Structures

Part 3: Searching and Search Trees

Department of Mathematics
Stockholm University

Searching

So-far we considered sorting. What about searching an element?

Searching in arrays means to determine if a given element (key) is in an array and, in the affirmative case, provide its position.

We focus on the following algorithms:

- Linear Search
- Binary Search
- Jump Search
- Exponential Search

We then focus on a special data structure binary search trees (BST). In particular, we are dealing with two special types of BSTs:

- AVL trees
- Red-Black trees

Searching

So-far we considered sorting. What about searching an element?

Searching in arrays means to determine if a given element (key) is in an array and, in the affirmative case, provide its position.

We focus on the following algorithms:

- Linear Search
- Binary Search
- Jump Search
- Exponential Search

We then focus on a special data structure binary search trees (BST). In particular, we are dealing with two special types of BSTs:

- AVL trees
- Red-Black trees

Searching

So-far we considered sorting. What about searching an element?

Searching in arrays means to determine if a given element (key) is in an array and, in the affirmative case, provide its position.

We focus on the following algorithms:

- Linear Search
- Binary Search
- Jump Search
- Exponential Search

We then focus on a special data structure binary search trees (BST). In particular, we are dealing with two special types of BSTs:

- AVL trees
- Red-Black trees

Searching

So-far we considered sorting. What about searching an element?

Searching in arrays means to determine if a given element (key) is in an array and, in the affirmative case, provide its position.

We focus on the following algorithms:

- Linear Search
- Binary Search
- Jump Search
- Exponential Search

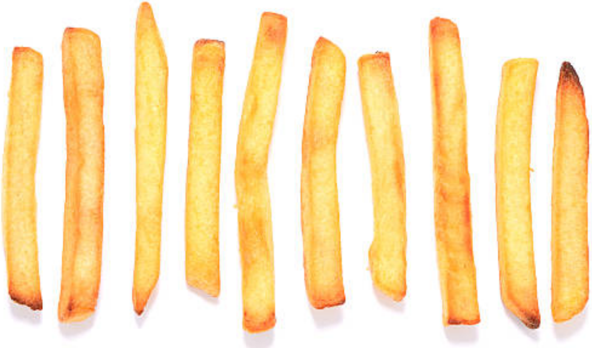
We then focus on a special data structure binary search trees (BST). In particular, we are dealing with two special types of BSTs:

- AVL trees
- Red-Black trees

Part 3: Searching in Arrays

Part 3: Searching in Arrays - Linear Search

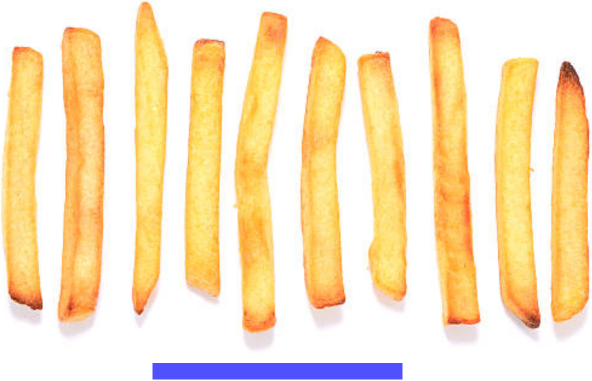
Linear search (often also called sequential search) is a method for finding an element (key) within an array. It sequentially checks each element of the array until a match is found or the whole array has been searched.



Look for the french fry with length

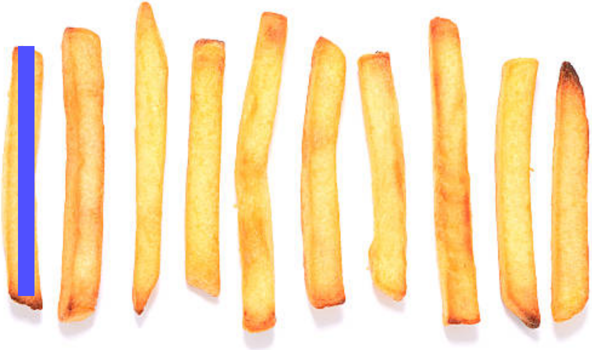
Part 3: Searching in Arrays - Linear Search

Linear search (often also called sequential search) is a method for finding an element (key) within an array. It sequentially checks each element of the array until a match is found or the whole array has been searched.



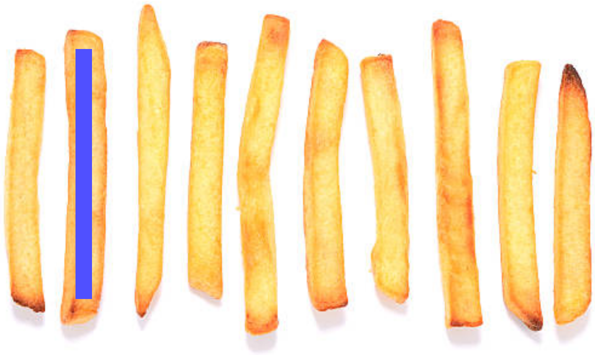
Part 3: Searching in Arrays - Linear Search

Linear search (often also called sequential search) is a method for finding an element (key) within an array. It sequentially checks each element of the array until a match is found or the whole array has been searched.



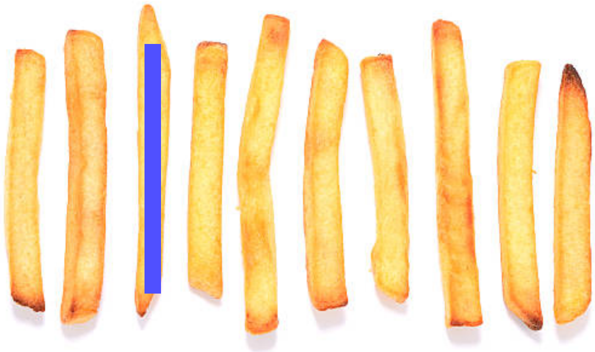
Part 3: Searching in Arrays - Linear Search

Linear search (often also called sequential search) is a method for finding an element (key) within an array. It sequentially checks each element of the array until a match is found or the whole array has been searched.



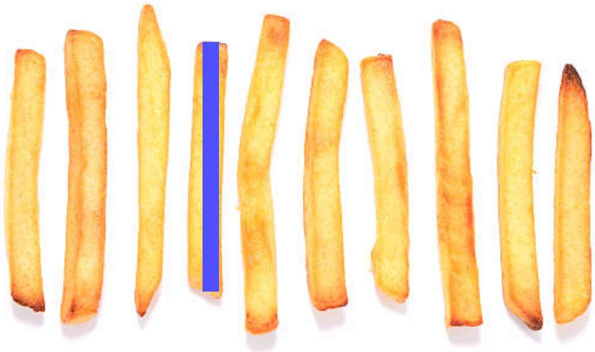
Part 3: Searching in Arrays - Linear Search

Linear search (often also called sequential search) is a method for finding an element (key) within an array. It sequentially checks each element of the array until a match is found or the whole array has been searched.



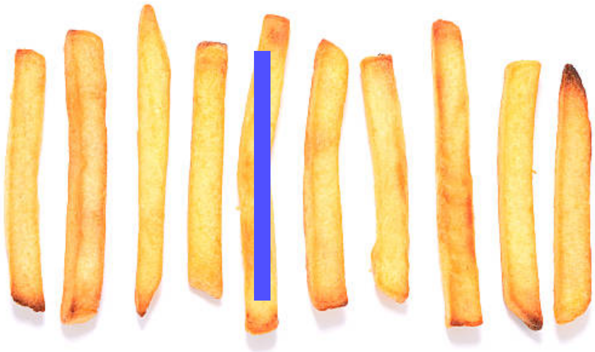
Part 3: Searching in Arrays - Linear Search

Linear search (often also called sequential search) is a method for finding an element (key) within an array. It sequentially checks each element of the array until a match is found or the whole array has been searched.



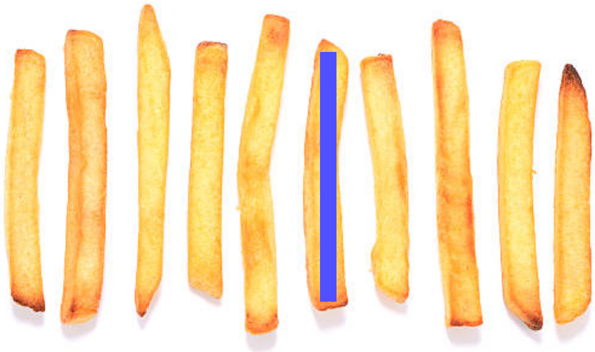
Part 3: Searching in Arrays - Linear Search

Linear search (often also called sequential search) is a method for finding an element (key) within an array. It sequentially checks each element of the array until a match is found or the whole array has been searched.



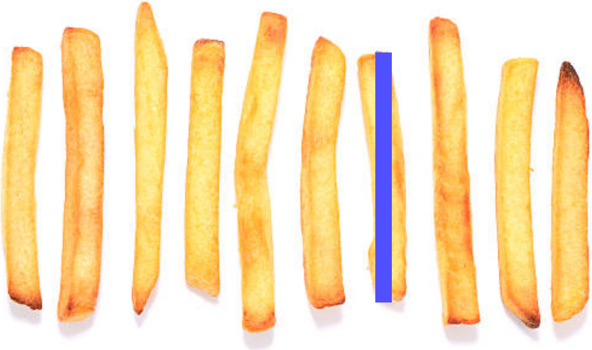
Part 3: Searching in Arrays - Linear Search

Linear search (often also called sequential search) is a method for finding an element (key) within an array. It sequentially checks each element of the array until a match is found or the whole array has been searched.



Part 3: Searching in Arrays - Linear Search

Linear search (often also called sequential search) is a method for finding an element (key) within an array. It sequentially checks each element of the array until a match is found or the whole array has been searched.



French fry found!

Part 3: Searching in Arrays - Linear Search

Linear search (often also called sequential search) is a method for finding an element (key) within an array. It sequentially checks each element of the array until a match is found or the whole array has been searched.

given an array of length n:

- An unsuccessful search requires n key comparisons (all elements must be checked).
- A successful search requires at most n key comparisons in the worst case (the desired element is at the end of the list).
- Time-complexity of Sequential Search: $O(n)$

Part 3: Searching in Arrays - Linear Search

Linear search (often also called sequential search) is a method for finding an element (key) within an array. It sequentially checks each element of the array until a match is found or the whole array has been searched.

given an array of length n:

- An unsuccessful search requires n key comparisons (all elements must be checked).
- A successful search requires at most n key comparisons in the worst case (the desired element is at the end of the list).
- Time-complexity of Sequential Search: $O(n)$

Let A be an array with n pairwise distinct elements that are randomly distributed along A. Then, the average number C_{avg} of key comparisons in a successful sequential search in A is:

$$C_{avg} = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\frac{n(n+1)}{2} = \frac{n+1}{2}$$

Part 3: Searching in Arrays - Linear Search

Linear search (often also called sequential search) is a method for finding an element (key) within an array. It sequentially checks each element of the array until a match is found or the whole array has been searched.

given an array of length n:

- An unsuccessful search requires n key comparisons (all elements must be checked).
- A successful search requires at most n key comparisons in the worst case (the desired element is at the end of the list).
- Time-complexity of Sequential Search: $O(n)$

Let A be an array with n pairwise distinct elements that are randomly distributed along A. Then, the average number C_{avg} of key comparisons in a successful sequential search in A is:

$$C_{avg} = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\frac{n(n+1)}{2} = \frac{n+1}{2}$$

WHY?

Part 3: Searching in Arrays - Linear Search

Linear search (often also called sequential search) is a method for finding an element (key) within an array. It sequentially checks each element of the array until a match is found or the whole array has been searched.

given an array of length n:

- An unsuccessful search requires n key comparisons (all elements must be checked).
- A successful search requires at most n key comparisons in the worst case (the desired element is at the end of the list).
- Time-complexity of Sequential Search: $O(n)$

Let A be an array with n pairwise distinct elements that are randomly distributed along A. Then, the average number C_{avg} of key comparisons in a successful sequential search in A is:

$$C_{avg} = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\frac{n(n+1)}{2} = \frac{n+1}{2}$$

WHY? since, with probability $\frac{1}{n}$ a key x is at position i and to find this key needs then i comparisons.
The necessary key comparisons for all cases (with x at position i) amount to: $1 + 2 + \cdots + n$.

Part 3: Searching in Arrays - Linear Search

Linear search (often also called sequential search) is a method for finding an element (key) within an array. It sequentially checks each element of the array until a match is found or the whole array has been searched.

given an array of length n:

- An unsuccessful search requires n key comparisons (all elements must be checked).
- A successful search requires at most n key comparisons in the worst case (the desired element is at the end of the list).
- Time-complexity of Sequential Search: $O(n)$

Let A be an array with n pairwise distinct elements that are randomly distributed along A. Then, the average number C_{avg} of key comparisons in a successful sequential search in A is:

$$C_{avg} = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\frac{n(n+1)}{2} = \frac{n+1}{2}$$

WHY? since, with probability $\frac{1}{n}$ a key x is at position i and to find this key needs then i comparisons. The necessary key comparisons for all cases (with x at position i) amount to: $1 + 2 + \cdots + n$.

Can we do better, e.g., if A is already sorted?

Part 3: Searching in Arrays - Binary Search

Linear search: checking if x is in $A[1..n]$ works in $O(n)$ time. However, we can do better!

Suppose that A is sorted, then we can apply binary search
[sorting costs $O(n \log n)$ time but needs to be done only once!]:

`BinarySearch`(A, x, $first$, $last$) //find x in sorted $A[first..last]$ initialized with (A, x, 1, n)

`IF` ($first > last$) `THEN RETURN` *"A does not contain x"*
$mid = \lfloor (first + last)/2 \rfloor$
`IF` ($A[mid] = x$) `THEN RETURN` *"A contains x on position mid"*
`IF` ($A[mid] > x$) `THEN BinarySearch`(A, x, $first$, $mid - 1$)
`ELSE BinarySearch`(A, x, $mid + 1$, $last$)

Part 3: Searching in Arrays - Binary Search

Linear search: checking if x is in $A[1..n]$ works in $O(n)$ time. However, we can do better!

Suppose that A is sorted, then we can apply binary search
[sorting costs $O(n \log n)$ time but needs to be done only once!]:

BinarySearch(A, x, *first*, *last*) //find x in sorted $A[\mathit{first}..\mathit{last}]$ initialized with $(A, x, 1, n)$

IF $(\mathit{first} > \mathit{last})$ THEN RETURN *"A does not contain x"*
$\mathit{mid} = \lfloor(\mathit{first} + \mathit{last})/2\rfloor$
IF $(A[\mathit{mid}] = x)$ THEN RETURN *"A contains x on position mid"*
IF $(A[\mathit{mid}] > x)$ THEN BinarySearch(A, x, *first*, $\mathit{mid} - 1$)
ELSE BinarySearch(A, x, $\mathit{mid} + 1$, *last*)

Part 3: Searching in Arrays - Binary Search

Linear search: checking if x is in $A[1..n]$ works in $O(n)$ time. However, we can do better!

Suppose that A is sorted, then we can apply binary search
[sorting costs $O(n \log n)$ time but needs to be done only once!]:

```
BinarySearch(A, x, first, last) //find x in sorted A[first..last] initialized with (A, x, 1, n)
    IF (first > last) THEN RETURN "A does not contain x"
    mid = ⌊(first + last)/2⌋
    IF (A[mid] = x) THEN RETURN "A contains x on position mid"
    IF (A[mid] > x) THEN BinarySearch(A, x, first, mid − 1)
    ELSE BinarySearch(A, x, mid + 1, last)
```

Example. Find $x = 3$ in A

pos	1	2	3	4	5	6	7
A=	[1	2	3	5	7	8	9]

call BinarySearch(A, x, 1, 7)

Part 3: Searching in Arrays - Binary Search

Linear search: checking if x is in $A[1..n]$ works in $O(n)$ time. However, we can do better!

Suppose that A is sorted, then we can apply binary search
[sorting costs $O(n \log n)$ time but needs to be done only once!]:

```
BinarySearch(A, x, first, last) //find x in sorted A[first..last] initialized with (A, x, 1, n)
    IF (first > last) THEN RETURN "A does not contain x"
    mid = ⌊(first + last)/2⌋
    IF (A[mid] = x) THEN RETURN "A contains x on position mid"
    IF (A[mid] > x) THEN BinarySearch(A, x, first, mid − 1)
    ELSE BinarySearch(A, x, mid + 1, last)
```

Example. Find $x = 3$ in A

pos	1	2	3	4	5	6	7
A=	[1	2	3	5	7	8	9]

call BinarySearch$(A, x, 1, 7) \implies mid = \lfloor (1+7)/2 \rfloor = 4$

Part 3: Searching in Arrays - Binary Search

Linear search: checking if x is in $A[1..n]$ works in $O(n)$ time. However, we can do better!

Suppose that A is sorted, then we can apply binary search
[sorting costs $O(n \log n)$ time but needs to be done only once!]:

```
BinarySearch(A, x, first, last) //find x in sorted A[first..last] initialized with (A, x, 1, n)
  IF (first > last) THEN RETURN "A does not contain x"
  mid = ⌊(first + last)/2⌋
  IF (A[mid] = x) THEN RETURN "A contains x on position mid"
  IF (A[mid] > x) THEN BinarySearch(A, x, first, mid − 1)
  ELSE BinarySearch(A, x, mid + 1, last)
```

Example. Find $x = 3$ in A

pos	1	2	3	4	5	6	7
A=	[1	2	3	5	7	8	9]

call BinarySearch$(A, x, 1, 7) \implies mid = \lfloor (1+7)/2 \rfloor = 4$

Since $A[4] = 5 > 3$ and A is sorted, x must be contained in $A[1..3]$ (if x exists in A)

Part 3: Searching in Arrays - Binary Search

Linear search: checking if x is in $A[1..n]$ works in $O(n)$ time. However, we can do better!

Suppose that A is sorted, then we can apply binary search
[sorting costs $O(n \log n)$ time but needs to be done only once!]:

```
BinarySearch(A, x, first, last) //find x in sorted A[first..last] initialized with (A, x, 1, n)
  IF (first > last) THEN RETURN "A does not contain x"
  mid = ⌊(first + last)/2⌋
  IF (A[mid] = x) THEN RETURN "A contains x on position mid"
  IF (A[mid] > x) THEN BinarySearch(A, x, first, mid − 1)
  ELSE BinarySearch(A, x, mid + 1, last)
```

Example. Find $x = 3$ in A

pos	1	2	3	4	5	6	7
A=	[1	2	3	5	7	8	9]

call BinarySearch(A, x, 1, 3)

Part 3: Searching in Arrays - Binary Search

Linear search: checking if x is in $A[1..n]$ works in $O(n)$ time. However, we can do better!

Suppose that A is sorted, then we can apply binary search
[sorting costs $O(n \log n)$ time but needs to be done only once!]:

```
BinarySearch(A, x, first, last) //find x in sorted A[first..last] initialized with (A, x, 1, n)
  IF (first > last) THEN RETURN "A does not contain x"
  mid = ⌊(first + last)/2⌋
  IF (A[mid] = x) THEN RETURN "A contains x on position mid"
  IF (A[mid] > x) THEN BinarySearch(A, x, first, mid − 1)
  ELSE BinarySearch(A, x, mid + 1, last)
```

Example. Find $x = 3$ in A

pos	1	2	3	4	5	6	7
A=	[1	2	3	5	7	8	9]

call BinarySearch$(A, x, 1, 3) \implies mid = \lfloor (1+3)/2 \rfloor = 2$

Part 3: Searching in Arrays - Binary Search

Linear search: checking if x is in $A[1..n]$ works in $O(n)$ time. However, we can do better!

Suppose that A is sorted, then we can apply binary search
[sorting costs $O(n \log n)$ time but needs to be done only once!]:

```
BinarySearch(A, x, first, last) //find x in sorted A[first..last] initialized with (A, x, 1, n)
  IF (first > last) THEN RETURN "A does not contain x"
  mid = ⌊(first + last)/2⌋
  IF (A[mid] = x) THEN RETURN "A contains x on position mid"
  IF (A[mid] > x) THEN BinarySearch(A, x, first, mid − 1)
  ELSE BinarySearch(A, x, mid + 1, last)
```

Example. Find $x = 3$ in A

pos	1	2	3	4	5	6	7
A=	[1	2	3	5	7	8	9]

call BinarySearch$(A, x, 1, 3)$ $\implies$ $mid = \lfloor (1+3)/2 \rfloor = 2$
Since $A[2] = 2 < 3$ and A is sorted, x must be contained in $A[3..3]$ (if x exists in A)

Part 3: Searching in Arrays - Binary Search

Linear search: checking if x is in $A[1..n]$ works in $O(n)$ time. However, we can do better!

Suppose that A is sorted, then we can apply binary search
[sorting costs $O(n \log n)$ time but needs to be done only once!]:

```
BinarySearch(A, x, first, last) //find x in sorted A[first..last] initialized with (A, x, 1, n)
    IF (first > last) THEN RETURN "A does not contain x"
    mid = ⌊(first + last)/2⌋
    IF (A[mid] = x) THEN RETURN "A contains x on position mid"
    IF (A[mid] > x) THEN BinarySearch(A, x, first, mid − 1)
    ELSE BinarySearch(A, x, mid + 1, last)
```

Example. Find $x = 3$ in A

pos	1	2	3	4	5	6	7
A=	[1	2	3	5	7	8	9]

call BinarySearch$(A, x, 3, 3)$ $\implies$ $mid = \lfloor (3+3)/2 \rfloor = 3$

Part 3: Searching in Arrays - Binary Search

Linear search: checking if x is in $A[1..n]$ works in $O(n)$ time. However, we can do better!

Suppose that A is sorted, then we can apply binary search
[sorting costs $O(n \log n)$ time but needs to be done only once!]:

```
BinarySearch(A, x, first, last) //find x in sorted A[first..last] initialized with (A, x, 1, n)
  IF (first > last) THEN RETURN "A does not contain x"
  mid = ⌊(first + last)/2⌋
  IF (A[mid] = x) THEN RETURN "A contains x on position mid"
  IF (A[mid] > x) THEN BinarySearch(A, x, first, mid − 1)
  ELSE BinarySearch(A, x, mid + 1, last)
```

Example. Find $x = 3$ in A

pos	1	2	3	4	5	6	7
A=	[1	2	3	5	7	8	9]

call BinarySearch$(A, x, 3, 3) \implies mid = \lfloor (3+3)/2 \rfloor = 3$

Since $A[3] = 3$ and we found x

Part 3: Searching in Arrays - Binary Search

Linear search: checking if x is in $A[1..n]$ works in $O(n)$ time. However, we can do better!

Suppose that A is sorted, then we can apply binary search
[sorting costs $O(n \log n)$ time but needs to be done only once!]:

```
BinarySearch(A, x, first, last) //find x in sorted A[first..last] initialized with (A, x, 1, n)
    IF (first > last) THEN RETURN "A does not contain x"
    mid = ⌊(first + last)/2⌋
    IF (A[mid] = x) THEN RETURN "A contains x on position mid"
    IF (A[mid] > x) THEN BinarySearch(A, x, first, mid − 1)
    ELSE BinarySearch(A, x, mid + 1, last)
```

Theorem. BinarySearch(A,x,1,n) *correctly determines if x exists in sorted A in* $\Theta(\log_2(n))$

Part 3: Searching in Arrays - Binary Search

Linear search: checking if x is in $A[1..n]$ works in $O(n)$ time. However, we can do better!

Suppose that A is sorted, then we can apply binary search
[sorting costs $O(n \log n)$ time but needs to be done only once!]:

`BinarySearch`$(A, x, \mathit{first}, \mathit{last})$ //find x in sorted $A[\mathit{first}..\mathit{last}]$ initialized with $(A, x, 1, n)$
- `IF` $(\mathit{first} > \mathit{last})$ `THEN RETURN` *"A does not contain x"*
- $\mathit{mid} = \lfloor (\mathit{first} + \mathit{last})/2 \rfloor$
- `IF` $(A[\mathit{mid}] = x)$ `THEN RETURN` *"A contains x on position mid"*
- `IF` $(A[\mathit{mid}] > x)$ `THEN BinarySearch`$(A, x, \mathit{first}, \mathit{mid} - 1)$
- `ELSE BinarySearch`$(A, x, \mathit{mid} + 1, \mathit{last})$

Theorem. `BinarySearch`$(A,x,1,n)$ *correctly determines if x exists in sorted A in* $\Theta(\log_2(n))$

Proof. correctness-sketch: If $x = A[\lfloor \frac{n}{2} \rfloor]$ we are done.

Part 3: Searching in Arrays - Binary Search

Linear search: checking if x is in $A[1..n]$ works in $O(n)$ time. However, we can do better!

Suppose that A is sorted, then we can apply binary search
[sorting costs $O(n \log n)$ time but needs to be done only once!]:

```
BinarySearch(A, x, first, last) //find x in sorted A[first..last] initialized with (A, x, 1, n)
    IF (first > last) THEN RETURN "A does not contain x"
    mid = ⌊(first + last)/2⌋
    IF (A[mid] = x) THEN RETURN "A contains x on position mid"
    IF (A[mid] > x) THEN BinarySearch(A, x, first, mid − 1)
    ELSE BinarySearch(A, x, mid + 1, last)
```

Theorem. BinarySearch(A,x,1,n) *correctly determines if x exists in sorted A in* $\Theta(\log_2(n))$

Proof. correctness-sketch: If $x = A[\lfloor \frac{n}{2} \rfloor]$ we are done.

Otherwise, if $x < A[\lfloor \frac{n}{2} \rfloor]$ it must be contained (if at all) in $A[1..\lfloor \frac{n}{2} \rfloor - 1]$ (Since A is sorted)

Part 3: Searching in Arrays - Binary Search

Linear search: checking if x is in $A[1..n]$ works in $O(n)$ time. However, we can do better!

Suppose that A is sorted, then we can apply binary search
[sorting costs $O(n \log n)$ time but needs to be done only once!]:

```
BinarySearch(A, x, first, last) //find x in sorted A[first..last] initialized with (A, x, 1, n)
    IF (first > last) THEN RETURN "A does not contain x"
    mid = ⌊(first + last)/2⌋
    IF (A[mid] = x) THEN RETURN "A contains x on position mid"
    IF (A[mid] > x) THEN BinarySearch(A, x, first, mid − 1)
    ELSE BinarySearch(A, x, mid + 1, last)
```

Theorem. BinarySearch$(A,x,1,n)$ *correctly determines if x exists in sorted A in* $\Theta(\log_2(n))$

Proof. correctness-sketch: If $x = A[\lfloor \frac{n}{2} \rfloor]$ we are done.

Otherwise, if $x < A[\lfloor \frac{n}{2} \rfloor]$ it must be contained (if at all) in $A[1..\lfloor \frac{n}{2} \rfloor - 1]$ (Since A is sorted)

Otherwise, if $x > A[\lfloor \frac{n}{2} \rfloor]$ it must be contained (if at all) in $A[\lfloor \frac{n}{2} \rfloor + 1, n]$ (Since A is sorted)

Part 3: Searching in Arrays - Binary Search

Linear search: checking if x is in $A[1..n]$ works in $O(n)$ time. However, we can do better!

Suppose that A is sorted, then we can apply binary search
[sorting costs $O(n \log n)$ time but needs to be done only once!]:

BinarySearch(A, x, *first*, *last*) //find x in sorted A[*first*..*last*] initialized with (A, x, 1, n)

```
IF (first > last) THEN RETURN "A does not contain x"
mid = ⌊(first + last)/2⌋
IF (A[mid] = x) THEN RETURN "A contains x on position mid"
IF (A[mid] > x) THEN BinarySearch(A, x, first, mid − 1)
ELSE BinarySearch(A, x, mid + 1, last)
```

Theorem. BinarySearch(A,x,1,n) *correctly determines if x exists in sorted A in* $\Theta(\log_2(n))$

Proof. correctness-sketch: If $x = A[\lfloor \frac{n}{2} \rfloor]$ we are done.

Otherwise, if $x < A[\lfloor \frac{n}{2} \rfloor]$ it must be contained (if at all) in $A[1..\lfloor \frac{n}{2} \rfloor - 1]$ (Since A is sorted)

Otherwise, if $x > A[\lfloor \frac{n}{2} \rfloor]$ it must be contained (if at all) in $A[\lfloor \frac{n}{2} \rfloor + 1, n]$ (Since A is sorted)

Now recurse (exercise)

Part 3: Searching in Arrays - Binary Search

Linear search: checking if x is in $A[1..n]$ works in $O(n)$ time. However, we can do better!

Suppose that A is sorted, then we can apply binary search
[sorting costs $O(n \log n)$ time but needs to be done only once!]:

```
BinarySearch(A, x, first, last) //find x in sorted A[first..last] initialized with (A, x, 1, n)
IF (first > last) THEN RETURN "A does not contain x"
mid = ⌊(first + last)/2⌋
IF (A[mid] = x) THEN RETURN "A contains x on position mid"
IF (A[mid] > x) THEN BinarySearch(A, x, first, mid − 1)
ELSE BinarySearch(A, x, mid + 1, last)
```

Theorem. BinarySearch(A,x,1,n) *correctly determines if x exists in sorted A in* $\Theta(\log_2(n))$

Proof. correctness-sketch: If $x = A[\lfloor \frac{n}{2} \rfloor]$ we are done.

Otherwise, if $x < A[\lfloor \frac{n}{2} \rfloor]$ it must be contained (if at all) in $A[1..\lfloor \frac{n}{2} \rfloor - 1]$ (Since A is sorted)

Otherwise, if $x > A[\lfloor \frac{n}{2} \rfloor]$ it must be contained (if at all) in $A[\lfloor \frac{n}{2} \rfloor + 1, n]$ (Since A is sorted)

Now recurse (exercise)

runtime: any idea?

Part 3: Searching in Arrays - Binary Search

Linear search: checking if x is in $A[1..n]$ works in $O(n)$ time. However, we can do better!

Suppose that A is sorted, then we can apply binary search
[sorting costs $O(n \log n)$ time but needs to be done only once!]:

```
BinarySearch(A, x, first, last) //find x in sorted A[first..last] initialized with (A, x, 1, n)
    IF (first > last) THEN RETURN "A does not contain x"
    mid = ⌊(first + last)/2⌋
    IF (A[mid] = x) THEN RETURN "A contains x on position mid"
    IF (A[mid] > x) THEN BinarySearch(A, x, first, mid − 1)
    ELSE BinarySearch(A, x, mid + 1, last)
```

Theorem. BinarySearch$(A,x,1,n)$ *correctly determines if x exists in sorted A in* $\Theta(\log_2(n))$

Proof. correctness-sketch: If $x = A[\lfloor \frac{n}{2} \rfloor]$ we are done.

Otherwise, if $x < A[\lfloor \frac{n}{2} \rfloor]$ it must be contained (if at all) in $A[1..\lfloor \frac{n}{2} \rfloor - 1]$ (Since A is sorted)

Otherwise, if $x > A[\lfloor \frac{n}{2} \rfloor]$ it must be contained (if at all) in $A[\lfloor \frac{n}{2} \rfloor + 1, n]$ (Since A is sorted)

Now recurse (exercise)

runtime: any idea? $T(n) = T(\frac{n}{2}) + \Theta(1)$

Part 3: Searching in Arrays - Binary Search

Linear search: checking if x is in $A[1..n]$ works in $O(n)$ time. However, we can do better!

Suppose that A is sorted, then we can apply binary search
[sorting costs $O(n \log n)$ time but needs to be done only once!]:

```
BinarySearch(A, x, first, last) //find x in sorted A[first..last] initialized with (A, x, 1, n)
   IF (first > last) THEN RETURN "A does not contain x"
   mid = ⌊(first + last)/2⌋
   IF (A[mid] = x) THEN RETURN "A contains x on position mid"
   IF (A[mid] > x) THEN BinarySearch(A, x, first, mid − 1)
   ELSE BinarySearch(A, x, mid + 1, last)
```

Theorem. BinarySearch(A,x,1,n) *correctly determines if x exists in sorted A in* $\Theta(\log_2(n))$

Proof. correctness-sketch: If $x = A[\lfloor \frac{n}{2} \rfloor]$ we are done.

Otherwise, if $x < A[\lfloor \frac{n}{2} \rfloor]$ it must be contained (if at all) in $A[1..\lfloor \frac{n}{2} \rfloor - 1]$ (Since A is sorted)

Otherwise, if $x > A[\lfloor \frac{n}{2} \rfloor]$ it must be contained (if at all) in $A[\lfloor \frac{n}{2} \rfloor + 1, n]$ (Since A is sorted)

Now recurse (exercise)

runtime: any idea? $T(n) = T(\frac{n}{2}) + \Theta(1)$

Via Mastertheorem: $a = 1, b = 2, d = 0 \implies a = b^d \implies T(n) \in \Theta(n^0 \log_2(n)) = \Theta(\log_2(n))$

Part 3: Searching in Arrays - Binary Search

Theorem. `BinarySearch`$(A,x,1,n)$ *correctly determines if* x *exists in sorted* A *in* $\Theta(\log_2(n))$

To give you a sense of just how fast binary search:

- Player 1 selects a word (e.g. from a printed dictionary)
- Player 2 repeatedly asks true/false questions in an attempt to guess it.
- If the word remains unidentified after 20 questions, the player 1 wins; otherwise, the player 2 takes the honors.

Part 3: Searching in Arrays - Binary Search

Theorem. `BinarySearch`$(A, x, 1, n)$ *correctly determines if x exists in sorted A in* $\Theta(\log_2(n))$

To give you a sense of just how fast binary search:

Twenty questions is a popular children's game:

- Player 1 selects a word (e.g. from a printed dictionary)
- Player 2 repeatedly asks true/false questions in an attempt to guess it.
- If the word remains unidentified after 20 questions, the player 1 wins; otherwise, the player 2 takes the honors.

Part 3: Searching in Arrays - Binary Search

Theorem. `BinarySearch`$(A, x, 1, n)$ *correctly determines if x exists in sorted A in* $\Theta(\log_2(n))$

To give you a sense of just how fast binary search:

Twenty questions is a popular children's game:

- Player 1 selects a word (e.g. from a printed dictionary)
- Player 2 repeatedly asks true/false questions in an attempt to guess it.
- If the word remains unidentified after 20 questions, the player 1 wins; otherwise, the player 2 takes the honors.

Part 3: Searching in Arrays - Binary Search

Theorem. `BinarySearch`$(A,x,1,n)$ *correctly determines if* x *exists in sorted* A *in* $\Theta(\log_2(n))$

To give you a sense of just how fast binary search:

Twenty questions is a popular children's game:

- Player 1 selects a word (e.g. from a printed dictionary)
- Player 2 repeatedly asks true/false questions in an attempt to guess it.
- If the word remains unidentified after 20 questions, the player 1 wins; otherwise, the player 2 takes the honors.

Part 3: Searching in Arrays - Binary Search

Theorem. `BinarySearch`$(A,x,1,n)$ *correctly determines if* x *exists in sorted* A *in* $\Theta(\log_2(n))$

To give you a sense of just how fast binary search:

Twenty questions is a popular children's game:

- Player 1 selects a word (e.g. from a printed dictionary)
- Player 2 repeatedly asks true/false questions in an attempt to guess it.
- If the word remains unidentified after 20 questions, the player 1 wins; otherwise, the player 2 takes the honors.

Is there a winning strategy for player 2

Part 3: Searching in Arrays - Binary Search

Theorem. `BinarySearch`$(A,x,1,n)$ *correctly determines if* x *exists in sorted* A *in* $\Theta(\log_2(n))$

To give you a sense of just how fast binary search:

Twenty questions is a popular children's game:

- Player 1 selects a word (e.g. from a printed dictionary)
- Player 2 repeatedly asks true/false questions in an attempt to guess it.
- If the word remains unidentified after 20 questions, the player 1 wins; otherwise, the player 2 takes the honors.

Is there a winning strategy for player 2
Answer: YES!

Part 3: Searching in Arrays - Binary Search

Theorem. `BinarySearch`$(A,x,1,n)$ *correctly determines if* x *exists in sorted* A *in* $\Theta(\log_2(n))$

To give you a sense of just how fast binary search:

Twenty questions is a popular children's game:

- Player 1 selects a word (e.g. from a printed dictionary)
- Player 2 repeatedly asks true/false questions in an attempt to guess it.
- If the word remains unidentified after 20 questions, the player 1 wins; otherwise, the player 2 takes the honors.

Is there a winning strategy for player 2
Answer: YES!

- Player 2 opens dictionary in the middle, selects a word (say "move"), and asks whether the unknown word is before "move" in alphabetical order. Since standard dictionaries contain $50'000$ to $200'000$ words, we can be certain that the process will terminate within twenty questions since $log_2(200'000) = 17.61$.

  Player 2 always wins!

Part 3: Searching in Arrays - Binary Search

Theorem. `BinarySearch`$(A, x, 1, n)$ *correctly determines if x exists in sorted A in* $\Theta(\log_2(n))$

To give you a sense of just how fast binary search:

Twenty questions is a popular children's game:

- Player 1 selects a word (e.g. from a printed dictionary)
- Player 2 repeatedly asks true/false questions in an attempt to guess it.
- If the word remains unidentified after 20 questions, the player 1 wins; otherwise, the player 2 takes the honors.

Is there a winning strategy for player 2

Answer: YES!

- Player 2 opens dictionary in the middle, selects a word (say "move"), and asks whether the unknown word is before "move" in alphabetical order. Since standard dictionaries contain $50'000$ to $200'000$ words, we can be certain that the process will terminate within twenty questions since $log_2(200'000) = 17.61$.

  Player 2 always wins!

Part 3: Searching in Arrays - Jump search

Pre-condition: (1) Array L sorted (increasing) and (2) keys in L are pairwise distinct

(first element is $L[1]$)

Principle: L is divided into sections of fixed length m. Jump over the sections to determine the section of the key.

$$\text{Sections:} \quad 1 \ldots m, \quad m+1 \ldots 2m, \quad 2m+1 \ldots 3m, \quad \text{and so on.}$$

Simple Jump Search:

Jump to positions $i \cdot m + 1$ (for $i = 1, 2, \ldots$) one after another.

As soon as $x < L[i \cdot m + 1]$, x can only be in the i-th section $(i-1) \cdot m + 1 \ldots i \cdot m$;

In this i-th section, we apply linear search for finding x.

Part 3: Searching in Arrays - Jump search

Example:

$$L = [\ 1\ \ 3\ \ 5\ \ 7\ \ 11\ \ 13\ \ 16\ \ 17\ \ 23\ \ 33\ \ 34\ \ 35\ ]$$

Find key $x = 17$ in L und chose here jump_width $m = 3$.

Example:

$$L = [\ \underline{1\ \ 3\ \ 5}\ \ \underline{7\ \ 11\ \ 13}\ \ \underline{16\ \ 17\ \ 23}\ \ \underline{33\ \ 34\ \ 35}\ ]$$

Subdivision of L into blocks of length $m = 3$.

Part 3: Searching in Arrays - Jump search

Example:

$$L = [\ 1\ \ 3\ \ 5\ \ 7\ \ 11\ \ 13\ \ 16\ \ 17\ \ 23\ \ 33\ \ 34\ \ 35\ ]$$

$i = 1$, $m = 3$ **and jump to** $i \cdot m + 1 = 4$

Since $L[4] = 7 < 17$, we take the next $i := 2$.

Part 3: Searching in Arrays - Jump search

Example:

$$L = [\ \underline{1\ \ 3\ \ 5}\ \ \underline{7\ \ 11\ \ 13}\ \ \underline{16\ \ 17\ \ 23}\ \ \underline{33\ \ 34\ \ 35}\ ]$$

$i = 2$, $m = 3$ **and jump to** $i \cdot m + 1 = 7$

Since $L[7] = 16 < 17$, we take next $i := 3$.

Part 3: Searching in Arrays - Jump search

Example:

L = [1 3 5 7 11 13 16 17 23 33 34 35]

$i = 3$, $m = 3$ **and jump to** $i \cdot m + 1 = 10$

Since $L[10] = 33 > 17$, it follows that 17 must be in section $L[2m + 1..3m]$ (if 17 is in L at all).

Part 3: Searching in Arrays - Jump search

Example:

$$L = [\ \underline{1\ \ 3\ \ 5}\ \ \underline{7\ \ 11\ \ 13}\ \ \boxed{\underline{16\ \ 17\ \ 23}}\ \ \underline{33\ \ 34\ \ 35}\ ]$$

Start linear search in section [16, 17, 23].

Since 16 $<$ 17 (already compared), go to next element in this section

Part 3: Searching in Arrays - Jump search

Example:

$$L = [\ 1\ \ 3\ \ 5\ \ \ 7\ \ 11\ \ 13\ \ \boxed{16\ \ 17\ \ 23}\ \ 33\ \ 34\ \ 35\ ]$$

We find 17 at position 8 in *L*, return 8 and stop searching.

Part 3: Searching in Arrays - Jump search

Assuming that all keys in L are randomly distributed and n is length of L and jumps and comparison can be done in constant time:
Average Search Costs*:

$$C_{avg}(n) \in O(\frac{n}{m} + m)$$

In total, at most $\frac{n}{m}$ jumps are possible
and we need to check one of the blocks of size m to check if x exists via linear search

Question: Is there an optimal jump-width?

Idea: One could attempt to optimize the average key comparisons, i.e., finding the m that minimizes $\frac{n}{m} + m$.

Sketch: Take the derivative of $C_{avg}(n)$, set it to 0, and solve the equation for m:

$$\frac{d}{dm} C_{avg}(n) = 1 - \frac{n}{m^2} = 0 \quad \Rightarrow \quad m^2 = n \quad \Rightarrow \quad m = \sqrt{n}$$

$\Rightarrow$ Optimal jump length $m = \lfloor \sqrt{n} \rfloor$; then complexity is in $O(\sqrt{n})$ [better than linear search!]

Jump seach is better than linear search but worse than binary search.

Why not use binary search instead?

If your data is too large for main memory, with jump search, only parts (blocks) need to be loaded into main memory!

* For more information, see Shneiderman, B. (1978) "Jump searching: a fast sequential search technique" Communications of the ACM, 21(10), pp.831-834.

Part 3: Searching in Arrays - Jump search

Assuming that all keys in L are randomly distributed and n is length of L and jumps and comparison can be done in constant time:
Average Search Costs*:

$$C_{avg}(n) \in O(\frac{n}{m} + m)$$

In total, at most $\frac{n}{m}$ jumps are possible

and we need to check one of the blocks of size m to check if x exists via linear search

Question: Is there an optimal jump-width?

Idea: One could attempt to optimize the average key comparisons, i.e., finding the m that minimizes $\frac{n}{m} + m$.

Sketch: Take the derivative of $C_{avg}(n)$, set it to 0, and solve the equation for m:

$$\frac{d}{dm} C_{avg}(n) = 1 - \frac{n}{m^2} = 0 \quad \Rightarrow \quad m^2 = n \quad \Rightarrow \quad m = \sqrt{n}$$

$\Rightarrow$ Optimal jump length $m = \lfloor \sqrt{n} \rfloor$; then complexity is in $O(\sqrt{n})$ [better than linear search!]

Jump seach is better than linear search but worse than binary search.

Why not use binary search instead?

If your data is too large for main memory, with jump search, only parts (blocks) need to be loaded into main memory!

* For more information, see Shneiderman, B. (1978) "Jump searching: a fast sequential search technique" Communications of the ACM, 21(10), pp.831-834.

Part 3: Searching in Arrays - Jump search

Assuming that all keys in L are randomly distributed and n is length of L and jumps and comparison can be done in constant time:
Average Search Costs*:

$$C_{avg}(n) \in O(\frac{n}{m} + m)$$

In total, at most $\frac{n}{m}$ jumps are possible
and we need to check one of the blocks of size m to check if x exists via linear search

Question: Is there an optimal jump-width?

Idea: One could attempt to optimize the average key comparisons, i.e., finding the m that minimizes $\frac{n}{m} + m$.

Sketch: Take the derivative of $C_{avg}(n)$, set it to 0, and solve the equation for m:

$$\frac{d}{dm} C_{avg}(n) = 1 - \frac{n}{m^2} = 0 \quad \Rightarrow \quad m^2 = n \quad \Rightarrow \quad m = \sqrt{n}$$

$\Rightarrow$ Optimal jump length $m = \lfloor \sqrt{n} \rfloor$; then complexity is in $O(\sqrt{n})$ [better than linear search!]

Jump seach is better than linear search but worse than binary search.

Why not use binary search instead?

If your data is too large for main memory, with jump search, only parts (blocks) need to be loaded into main memory!

* For more information, see Shneiderman, B. (1978) "Jump searching: a fast sequential search technique" Communications of the ACM, 21(10), pp.831-834.

Part 3: Searching in Arrays - Jump search

Assuming that all keys in L are randomly distributed and n is length of L and jumps and comparison can be done in constant time:
Average Search Costs*:

$$C_{avg}(n) \in O(\frac{n}{m} + m)$$

In total, at most $\frac{n}{m}$ jumps are possible

and we need to check one of the blocks of size m to check if x exists via linear search

Question: Is there an optimal jump-width?

Idea: One could attempt to optimize the average key comparisons, i.e., finding the m that minimizes $\frac{n}{m} + m$.

Sketch: Take the derivative of $C_{avg}(n)$, set it to 0, and solve the equation for m:

$$\frac{d}{dm} C_{avg}(n) = 1 - \frac{n}{m^2} = 0 \quad \Rightarrow \quad m^2 = n \quad \Rightarrow \quad m = \sqrt{n}$$

$\Rightarrow$ Optimal jump length $m = \lfloor \sqrt{n} \rfloor$; then complexity is in $O(\sqrt{n})$ [better than linear search!]

Jump seach is better than linear search but worse than binary search.

Why not use binary search instead?

If your data is too large for main memory, with jump search, only parts (blocks) need to be loaded into main memory!

* For more information, see Shneiderman, B. (1978) "Jump searching: a fast sequential search technique" Communications of the ACM, 21(10), pp.831-834.

Part 3: Searching in Arrays - Jump search

Assuming that all keys in L are randomly distributed and n is length of L and jumps and comparison can be done in constant time:
Average Search Costs*:

$$C_{avg}(n) \in O(\frac{n}{m} + m)$$

In total, at most $\frac{n}{m}$ jumps are possible

and we need to check one of the blocks of size m to check if x exists via linear search

Question: Is there an optimal jump-width?

Idea: One could attempt to optimize the average key comparisons, i.e., finding the m that minimizes $\frac{n}{m} + m$.

Sketch: Take the derivative of $C_{avg}(n)$, set it to 0, and solve the equation for m:

$$\frac{d}{dm} C_{avg}(n) = 1 - \frac{n}{m^2} = 0 \quad \Rightarrow \quad m^2 = n \quad \Rightarrow \quad m = \sqrt{n}$$

$\Rightarrow$ Optimal jump length $m = \lfloor \sqrt{n} \rfloor$; then complexity is in $O(\sqrt{n})$ [better than linear search!]

Jump seach is better than linear search but worse than binary search.

Why not use binary search instead?

If your data is too large for main memory, with jump search, only parts (blocks) need to be loaded into main memory!

* For more information, see Shneiderman, B. (1978) "Jump searching: a fast sequential search technique" Communications of the ACM, 21(10), pp.831-834.

Part 3: Searching in Arrays - Jump search

Assuming that all keys in L are randomly distributed and n is length of L and jumps and comparison can be done in constant time:
Average Search Costs*:

$$C_{avg}(n) \in O(\frac{n}{m} + m)$$

In total, at most $\frac{n}{m}$ jumps are possible

and we need to check one of the blocks of size m to check if x exists via linear search

Question: Is there an optimal jump-width?

Idea: One could attempt to optimize the average key comparisons, i.e., finding the m that minimizes $\frac{n}{m} + m$.

Sketch: Take the derivative of $C_{avg}(n)$, set it to 0, and solve the equation for m:

$$\frac{d}{dm} C_{avg}(n) = 1 - \frac{n}{m^2} = 0 \quad \Rightarrow \quad m^2 = n \quad \Rightarrow \quad m = \sqrt{n}$$

$\Rightarrow$ Optimal jump length $m = \lfloor \sqrt{n} \rfloor$; then complexity is in $O(\sqrt{n})$ [better than linear search!]

Jump seach is better than linear search but worse than binary search.

Why not use binary search instead?

If your data is too large for main memory, with jump search, only parts (blocks) need to be loaded into main memory!

* For more information, see Shneiderman, B. (1978) "Jump searching: a fast sequential search technique" Communications of the ACM, 21(10), pp.831-834.

Part 3: Searching in Arrays - Jump search

Assuming that all keys in L are randomly distributed and n is length of L and jumps and comparison can be done in constant time:
Average Search Costs*:

$$C_{avg}(n) \in O(\frac{n}{m} + m)$$

In total, at most $\frac{n}{m}$ jumps are possible

and we need to check one of the blocks of size m to check if x exists via linear search

Question: Is there an optimal jump-width?

Idea: One could attempt to optimize the average key comparisons, i.e., finding the m that minimizes $\frac{n}{m} + m$.

Sketch: Take the derivative of $C_{avg}(n)$, set it to 0, and solve the equation for m:

$$\frac{d}{dm} C_{avg}(n) = 1 - \frac{n}{m^2} = 0 \quad \Rightarrow \quad m^2 = n \quad \Rightarrow \quad m = \sqrt{n}$$

$\Rightarrow$ Optimal jump length $m = \lfloor \sqrt{n} \rfloor$; then complexity is in $O(\sqrt{n})$ [better than linear search!]

Jump seach is better than linear search but worse than binary search.

Why not use binary search instead?

If your data is too large for main memory, with jump search, only parts (blocks) need to be loaded into main memory!

* For more information, see Shneiderman, B. (1978) "Jump searching: a fast sequential search technique" Communications of the ACM, 21(10), pp.831-834.

Part 3: Searching in Arrays - Jump search

Assuming that all keys in L are randomly distributed and n is length of L and jumps and comparison can be done in constant time:
Average Search Costs*:

$$C_{avg}(n) \in O(\frac{n}{m} + m)$$

In total, at most $\frac{n}{m}$ jumps are possible

and we need to check one of the blocks of size m to check if x exists via linear search

Question: Is there an optimal jump-width?

Idea: One could attempt to optimize the average key comparisons, i.e., finding the m that minimizes $\frac{n}{m} + m$.

Sketch: Take the derivative of $C_{avg}(n)$, set it to 0, and solve the equation for m:

$$\frac{d}{dm} C_{avg}(n) = 1 - \frac{n}{m^2} = 0 \quad \Rightarrow \quad m^2 = n \quad \Rightarrow \quad m = \sqrt{n}$$

$\Rightarrow$ Optimal jump length $m = \lfloor \sqrt{n} \rfloor$; then complexity is in $O(\sqrt{n})$ [better than linear search!]

Jump seach is better than linear search but worse than binary search.

Why not use binary search instead?

If your data is too large for main memory, with jump search, only parts (blocks) need to be loaded into main memory!

---

* For more information, see Shneiderman, B. (1978) "Jump searching: a fast sequential search technique" Communications of the ACM, 21(10), pp.831-834.

Part 3: Searching in Arrays - Exponential Search

Question: How can we search when the length of a search range is initially unknown?

Binary search and or jump search with optimal jump-width assume that one knows the length of the range to be searched before starting the search. However, there may be cases where the search range is finite but “practically” unlimited. In such a case, it is reasonable to first determine an upper limit for the range to be searched, within which an element with key k must lie if such an element exists at all.

Idea: Determine, in exponentially growing steps, a range in which the search key must lie.

Principle of Exponential Search for x in increasingly sorted L (first entry $L[1]$):

1. Test $L[1]$, $L[2]$, $L[4]$, $L[8]$, ..., $L[2^j]$, ...
2. At the smallest j such that $x \leq L[2^j]$: Either x is in $L[(2^{j-1}+1)\ldots 2^j]$ or x is not in L.
3. Search within $[L[2^{j-1}+1], \ldots, L[2^j]]$ using any search method.

```
Exponential Search(L (sorted increasingly), x)
IF (x = L[1]) RETURN 1
i := 2
WHILE (x > L[i]) DO //Determine boundaries of search space
    i := 2 · i
FOR (j = i/2 + 1, i/2 + 2, ..., i) DO
    IF (L[j] = x) THEN RETURN j
RETURN −1//x not in L
```

Missing detail: Must be careful here, to avoid that program crashes when i in while-loop is out of range of L.

In the pseudo-code, we simply assume that it terminates when $L[i] = NIL$, i.e., i is out of range of L.

Part 3: Searching in Arrays - Exponential Search

Question: How can we search when the length of a search range is initially unknown?

Binary search and or jump search with optimal jump-width assume that one knows the length of the range to be searched before starting the search. However, there may be cases where the search range is finite but “practically” unlimited. In such a case, it is reasonable to first determine an upper limit for the range to be searched, within which an element with key k must lie if such an element exists at all.

Idea: Determine, in exponentially growing steps, a range in which the search key must lie.

Principle of Exponential Search for x in increasingly sorted L (first entry $L[1]$):

1. Test $L[1]$, $L[2]$, $L[4]$, $L[8]$, ..., $L[2^j]$, ...
2. At the smallest j such that $x \leq L[2^j]$: Either x is in $L[(2^{j-1}+1)\ldots 2^j]$ or x is not in L.
3. Search within $[L[2^{j-1}+1], \ldots, L[2^j]]$ using any search method.

```
Exponential Search(L (sorted increasingly), x)
IF (x = L[1]) RETURN 1
i := 2
WHILE (x > L[i]) DO //Determine boundaries of search space
    i := 2 · i
FOR (j = i/2 + 1, i/2 + 2, ..., i) DO
    IF (L[j] = x) THEN RETURN j
RETURN −1//x not in L
```

Missing detail: Must be careful here, to avoid that program crashes when i in while-loop is out of range of L.

In the pseudo-code, we simply assume that it terminates when $L[i] = NIL$, i.e., i is out of range of L.

Part 3: Searching in Arrays - Exponential Search

Question: How can we search when the length of a search range is initially unknown?

Binary search and or jump search with optimal jump-width assume that one knows the length of the range to be searched before starting the search. However, there may be cases where the search range is finite but “practically” unlimited. In such a case, it is reasonable to first determine an upper limit for the range to be searched, within which an element with key k must lie if such an element exists at all.

Idea: Determine, in exponentially growing steps, a range in which the search key must lie.

Principle of Exponential Search for x in increasingly sorted L (first entry $L[1]$):

1. Test $L[1]$, $L[2]$, $L[4]$, $L[8]$, ..., $L[2^j]$, ...
2. At the smallest j such that $x \leq L[2^j]$: Either x is in $L[(2^{j-1}+1)\dots 2^j]$ or x is not in L.
3. Search within $[L[2^{j-1}+1], \dots, L[2^j]]$ using any search method.

```
Exponential Search(L (sorted increasingly), x)
  IF (x = L[1]) RETURN 1
  i := 2
  WHILE (x > L[i]) DO //Determine boundaries of search space
      i := 2 · i
  FOR (j = i/2 + 1, i/2 + 2, ..., i) DO
      IF (L[j] = x) THEN RETURN j
  RETURN −1//x not in L
```

Missing detail: Must be careful here, to avoid that program crashes when i in while-loop is out of range of L.

In the pseudo-code, we simply assume that it terminates when $L[i] = NIL$, i.e., i is out of range of L.

Part 3: Searching in Arrays - Exponential Search

Question: How can we search when the length of a search range is initially unknown?

Binary search and or jump search with optimal jump-width assume that one knows the length of the range to be searched before starting the search. However, there may be cases where the search range is finite but “practically” unlimited. In such a case, it is reasonable to first determine an upper limit for the range to be searched, within which an element with key k must lie if such an element exists at all.

Idea: Determine, in exponentially growing steps, a range in which the search key must lie.

Principle of Exponential Search for x in increasingly sorted L (first entry $L[1]$):

1. Test $L[1]$, $L[2]$, $L[4]$, $L[8]$, ..., $L[2^j]$, ...
2. At the smallest j such that $x \leq L[2^j]$: Either x is in $L[(2^{j-1}+1)\ldots 2^j]$ or x is not in L.
3. Search within $[L[2^{j-1}+1], \ldots, L[2^j]]$ using any search method.

```
Exponential Search(L (sorted increasingly), x)
IF (x = L[1]) RETURN 1
i := 2
WHILE (x > L[i]) DO //Determine boundaries of search space
    i := 2 · i
FOR (j = i/2 + 1, i/2 + 2, . . . , i) DO
    IF (L[j] = x) THEN RETURN j
RETURN −1//x not in L
```

Missing detail: Must be careful here, to avoid that program crashes when i in while-loop is out of range of L.

In the pseudo-code, we simply assume that it terminates when $L[i] = NIL$, i.e., i is out of range of L.

Example.

$$L = [\ 3\ \ 5\ \ 7\ \ 11\ \ 13\ \ 17\ \ 19\ \ 23\ ]$$

Find $x = 13$ in L.

Part 3: Searching in Arrays - Exponential Search

Example.

L = [3 5 7 11 13 17 19 23]

i = 1

Since $L[1] = 3 < 13$, we have $i := 2 \cdot 1 = 2$. Check now $L[2]$.

Example.

$$L = [\ 3\ \ 5\ \ 7\ \ 11\ \ 13\ \ 17\ \ 19\ \ 23\ ]$$

i = 2.

Note: In red part, we cannot find x since L is sorted.

Since $L[2] = 5 < 13$, we have $i := 2 \cdot 2 = 4$. Check now $L[4]$.

Part 3: Searching in Arrays - Exponential Search

Example.

$$L = [\ 3\ \ 5\ \ 7\ \ 11\ \ 13\ \ 17\ \ 19\ \ 23\ ]$$

i = 4.

Note: In red part, we cannot find x since L is sorted.

Since $L[4] = 11 < 13$, we have $i := 2 \cdot 4 = 8$. Check now $L[8]$.

Part 3: Searching in Arrays - Exponential Search

Example.

L = [3 5 7 11 13 17 19 23]

i = 8.

Note: In red part, we cannot find x since L is sorted.

Since $L[8] = 23 > 13$, it follows that 13 is in $L[5..8]$ (if x in L at all).

Part 3: Searching in Arrays - Exponential Search

Example.

$$L = [\ 3\ \ 5\ \ 7\ \ 11\ \boxed{13\ \ 17\ \ 19\ \ 23}\ ]$$

Check existence of $x = 13$ via linear search in $L[5..8]$:
In the first step of the linear search we found 13. Return position of 13.

Part 3: Searching in Arrays - Exponential Search

Costs.

If L contains only pairwise-disinct keys from $\mathbb{N}$, then exponential search runs in $O(\log_2 x)$ time

Note, $\lfloor \log_2 x \rfloor + 1$ = number of bits in binary representation of the key x [x not size of array!]

Hence, if $x \leq 2^k$ for some constant k, then $\log_2 x \leq \log_2 2^k = k$ and thus, exponential search runs in constant time

Proof-sketch (runtime in $O(\log_2 x)$).

Since L contains only pairwise-distinct keys from $\mathbb{N}$

$\Longrightarrow$ Key values x grow at least as fast as the element indices, i.e., $L[k] \geq k \; \forall k$.

$\Longrightarrow$ i is doubled at most $\log_2 x$ times because $i \geq \log_2(x)$ implies $L[2^i] \geq 2^i \geq 2^{\log_2(x)} = x$. (gives stop criterion!)

$\Longrightarrow$ Determine the correct interval: in $\leq j \leq \log_2 x$ comparisons with $i = 2^j$.

Thus, $j \in O(\log_2 x)$

Length of this interval: $2^j - 2^{j-1} = 2^{j-1}(2-1) = 2^{j-1}$ where $j \in O(\log_2 x)$.

This means that the length of the interval being searched is in $O(2^{\log_2(x)-1})$.

Search within this intervall (e.g., binary search): $O(\log_2(2^{\log_2(x)-1})) = O(\log_2(x) - 1) = O(\log_2 x)$.

$\Longrightarrow$ Overall effort $O(\log_2 x)$ $\square$

Part 3: Searching in Arrays - Exponential Search

Costs.

If L contains only pairwise-disinct keys from $\mathbb{N}$, then exponential search runs in $O(\log_2 x)$ time

Note, $\lfloor \log_2 x \rfloor + 1$ = number of bits in binary representation of the key x [x not size of array!]

Hence, if $x \leq 2^k$ for some constant k, then $\log_2 x \leq \log_2 2^k = k$ and thus, exponential search runs in constant time

Proof-sketch (runtime in $O(\log_2 x)$).

Since L contains only pairwise-distinct keys from $\mathbb{N}$

$\Longrightarrow$ Key values x grow at least as fast as the element indices, i.e., $L[k] \geq k \; \forall k$.

$\Longrightarrow$ i is doubled at most $\log_2 x$ times because $i \geq \log_2(x)$ implies $L[2^i] \geq 2^i \geq 2^{\log_2(x)} = x$. (gives stop criterion!)

$\Longrightarrow$ Determine the correct interval: in $\leq j \leq \log_2 x$ comparisons with $i = 2^j$.

Thus, $j \in O(\log_2 x)$

Length of this interval: $2^j - 2^{j-1} = 2^{j-1}(2-1) = 2^{j-1}$ where $j \in O(\log_2 x)$.

This means that the length of the interval being searched is in $O(2^{\log_2(x)-1})$.

Search within this intervall (e.g., binary search): $O(\log_2(2^{\log_2(x)-1})) = O(\log_2(x) - 1) = O(\log_2 x)$.

$\Longrightarrow$ Overall effort $O(\log_2 x)$ $\square$

Part 3: Searching in Arrays - Exponential Search

Costs.

If L contains only pairwise-disinct keys from $\mathbb{N}$, then exponential search runs in $O(\log_2 x)$ time

Note, $\lfloor \log_2 x \rfloor + 1$ = number of bits in binary representation of the key x [x not size of array!]

Hence, if $x \leq 2^k$ for some constant k, then $\log_2 x \leq \log_2 2^k = k$ and thus, exponential search runs in constant time

Proof-sketch (runtime in $O(\log_2 x)$).

Since L contains only pairwise-distinct keys from $\mathbb{N}$

$\implies$ Key values x grow at least as fast as the element indices, i.e., $L[k] \geq k \ \forall k$.

$\implies$ i is doubled at most $\log_2 x$ times because $i \geq \log_2(x)$ implies $L[2^i] \geq 2^i \geq 2^{\log_2(x)} = x$. (gives stop criterion!)

$\implies$ Determine the correct interval: in $\leq j \leq \log_2 x$ comparisons with $i = 2^j$.

Thus, $j \in O(\log_2 x)$

Length of this interval: $2^j - 2^{j-1} = 2^{j-1}(2-1) = 2^{j-1}$ where $j \in O(\log_2 x)$.

This means that the length of the interval being searched is in $O(2^{\log_2(x)-1})$.

Search within this intervall (e.g., binary search): $O(\log_2(2^{\log_2(x)-1})) = O(\log_2(x) - 1) = O(\log_2 x)$.

$\implies$ Overall effort $O(\log_2 x)$ $\square$

Part 3: Searching in Arrays - Exponential Search

Costs.

If L contains only pairwise-disinct keys from $\mathbb{N}$, then exponential search runs in $O(\log_2 x)$ time

Note, $\lfloor \log_2 x \rfloor + 1$ = number of bits in binary representation of the key x [x not size of array!]

Hence, if $x \leq 2^k$ for some constant k, then $\log_2 x \leq \log_2 2^k = k$ and thus, exponential search runs in constant time

Proof-sketch (runtime in $O(\log_2 x)$).

Since L contains only pairwise-distinct keys from $\mathbb{N}$

$\implies$ Key values x grow at least as fast as the element indices, i.e., $L[k] \geq k \; \forall k$.

$\implies$ i is doubled at most $\log_2 x$ times because $i \geq \log_2(x)$ implies $L[2^i] \geq 2^i \geq 2^{\log_2(x)} = x$. (gives stop criterion!)

$\implies$ Determine the correct interval: in $\leq j \leq \log_2 x$ comparisons with $i = 2^j$.

Thus, $j \in O(\log_2 x)$

Length of this interval: $2^j - 2^{j-1} = 2^{j-1}(2-1) = 2^{j-1}$ where $j \in O(\log_2 x)$.

This means that the length of the interval being searched is in $O(2^{\log_2(x)-1})$.

Search within this intervall (e.g., binary search): $O(\log_2(2^{\log_2(x)-1})) = O(\log_2(x) - 1) = O(\log_2 x)$.

$\implies$ Overall effort $O(\log_2 x)$ $\square$

Part 3: Searching in Arrays - Exponential Search

Costs.

If L contains only pairwise-disinct keys from $\mathbb{N}$, then exponential search runs in $O(\log_2 x)$ time

Note, $\lfloor \log_2 x \rfloor + 1$ = number of bits in binary representation of the key x [x not size of array!]

Hence, if $x \leq 2^k$ for some constant k, then $\log_2 x \leq \log_2 2^k = k$ and thus, exponential search runs in constant time

Proof-sketch (runtime in $O(\log_2 x)$).

Since L contains only pairwise-distinct keys from $\mathbb{N}$

$\Longrightarrow$ Key values x grow at least as fast as the element indices, i.e., $L[k] \geq k \ \forall k$.

$\Longrightarrow$ i is doubled at most $\log_2 x$ times because $i \geq \log_2(x)$ implies $L[2^i] \geq 2^i \geq 2^{\log_2(x)} = x$. (gives stop criterion!)

$\Longrightarrow$ Determine the correct interval: in $\leq j \leq \log_2 x$ comparisons with $i = 2^j$.

Thus, $j \in O(\log_2 x)$

Length of this interval: $2^j - 2^{j-1} = 2^{j-1}(2-1) = 2^{j-1}$ where $j \in O(\log_2 x)$.

This means that the length of the interval being searched is in $O(2^{\log_2(x)-1})$.

Search within this intervall (e.g., binary search): $O(\log_2(2^{\log_2(x)-1})) = O(\log_2(x) - 1) = O(\log_2 x)$.

$\Longrightarrow$ Overall effort $O(\log_2 x)$ $\square$

Part 3: Searching in Arrays - Exponential Search

Costs.

If L contains only pairwise-disinct keys from $\mathbb{N}$, then exponential search runs in $O(\log_2 x)$ time

Note, $\lfloor \log_2 x \rfloor + 1$ = number of bits in binary representation of the key x [x not size of array!]

Hence, if $x \leq 2^k$ for some constant k, then $\log_2 x \leq \log_2 2^k = k$ and thus, exponential search runs in constant time

Proof-sketch (runtime in $O(\log_2 x)$).

Since L contains only pairwise-distinct keys from $\mathbb{N}$

$\implies$ Key values x grow at least as fast as the element indices, i.e., $L[k] \geq k \; \forall k$.

$\implies$ i is doubled at most $\log_2 x$ times because $i \geq \log_2(x)$ implies $L[2^i] \geq 2^i \geq 2^{\log_2(x)} = x$. (gives stop criterion!)

$\implies$ Determine the correct interval: in $\leq j \leq \log_2 x$ comparisons with $i = 2^j$.

Thus, $j \in O(\log_2 x)$

Length of this interval: $2^j - 2^{j-1} = 2^{j-1}(2-1) = 2^{j-1}$ where $j \in O(\log_2 x)$.

This means that the length of the interval being searched is in $O(2^{\log_2(x)-1})$.

Search within this intervall (e.g., binary search): $O(\log_2(2^{\log_2(x)-1})) = O(\log_2(x) - 1) = O(\log_2 x)$.

$\implies$ Overall effort $O(\log_2 x)$ $\square$

Part 3: Searching in Arrays - Exponential Search

Costs.

If L contains only pairwise-disinct keys from $\mathbb{N}$, then exponential search runs in $O(\log_2 x)$ time

Note, $\lfloor \log_2 x \rfloor + 1$ = number of bits in binary representation of the key x [x not size of array!]

Hence, if $x \leq 2^k$ for some constant k, then $\log_2 x \leq \log_2 2^k = k$ and thus, exponential search runs in constant time

Proof-sketch (runtime in $O(\log_2 x)$).

Since L contains only pairwise-distinct keys from $\mathbb{N}$

$\Longrightarrow$ Key values x grow at least as fast as the element indices, i.e., $L[k] \geq k \; \forall k$.

$\Longrightarrow$ i is doubled at most $\log_2 x$ times because $i \geq \log_2(x)$ implies $L[2^i] \geq 2^i \geq 2^{\log_2(x)} = x$. (gives stop criterion!)

$\Longrightarrow$ Determine the correct interval: in $\leq j \leq \log_2 x$ comparisons with $i = 2^j$.

Thus, $j \in O(\log_2 x)$

Length of this interval: $2^j - 2^{j-1} = 2^{j-1}(2-1) = 2^{j-1}$ where $j \in O(\log_2 x)$.

This means that the length of the interval being searched is in $O(2^{\log_2(x)-1})$.

Search within this intervall (e.g., binary search): $O(\log_2(2^{\log_2(x)-1})) = O(\log_2(x) - 1) = O(\log_2 x)$.

$\Longrightarrow$ Overall effort $O(\log_2 x)$ $\square$

Part 3: Searching in Arrays - Exponential Search

Costs.

If L contains only pairwise-disinct keys from $\mathbb{N}$, then exponential search runs in $O(\log_2 x)$ time

Note, $\lfloor \log_2 x \rfloor + 1$ = number of bits in binary representation of the key x [x not size of array!]

Hence, if $x \leq 2^k$ for some constant k, then $\log_2 x \leq \log_2 2^k = k$ and thus, exponential search runs in constant time

Proof-sketch (runtime in $O(\log_2 x)$).

Since L contains only pairwise-distinct keys from $\mathbb{N}$

$\Longrightarrow$ Key values x grow at least as fast as the element indices, i.e., $L[k] \geq k \; \forall k$.

$\Longrightarrow$ i is doubled at most $\log_2 x$ times because $i \geq \log_2(x)$ implies $L[2^i] \geq 2^i \geq 2^{\log_2(x)} = x$. (gives stop criterion!)

$\Longrightarrow$ Determine the correct interval: in $\leq j \leq \log_2 x$ comparisons with $i = 2^j$.

Thus, $j \in O(\log_2 x)$

Length of this interval: $2^j - 2^{j-1} = 2^{j-1}(2-1) = 2^{j-1}$ where $j \in O(\log_2 x)$.

This means that the length of the interval being searched is in $O(2^{\log_2(x)-1})$.

Search within this intervall (e.g., binary search): $O(\log_2(2^{\log_2(x)-1})) = O(\log_2(x) - 1) = O(\log_2 x)$.

$\Longrightarrow$ Overall effort $O(\log_2 x)$ $\square$

Part 3: Searching in Arrays - Exponential Search

Costs.

If L contains only pairwise-disinct keys from $\mathbb{N}$, then exponential search runs in $O(\log_2 x)$ time

Note, $\lfloor \log_2 x \rfloor + 1$ = number of bits in binary representation of the key x [x not size of array!]

Hence, if $x \leq 2^k$ for some constant k, then $\log_2 x \leq \log_2 2^k = k$ and thus, exponential search runs in constant time

Proof-sketch (runtime in $O(\log_2 x)$).

Since L contains only pairwise-distinct keys from $\mathbb{N}$

$\implies$ Key values x grow at least as fast as the element indices, i.e., $L[k] \geq k \; \forall k$.

$\implies$ i is doubled at most $\log_2 x$ times because $i \geq \log_2(x)$ implies $L[2^i] \geq 2^i \geq 2^{\log_2(x)} = x$. (gives stop criterion!)

$\implies$ Determine the correct interval: in $\leq j \leq \log_2 x$ comparisons with $i = 2^j$.

Thus, $j \in O(\log_2 x)$

Length of this interval: $2^j - 2^{j-1} = 2^{j-1}(2-1) = 2^{j-1}$ where $j \in O(\log_2 x)$.

This means that the length of the interval being searched is in $O(2^{\log_2(x)-1})$.

Search within this intervall (e.g., binary search): $O(\log_2(2^{\log_2(x)-1})) = O(\log_2(x) - 1) = O(\log_2 x)$.

$\implies$ Overall effort $O(\log_2 x)$ $\square$

Part 3: Searching in Arrays - Exponential Search

Costs.

If L contains only pairwise-disinct keys from $\mathbb{N}$, then exponential search runs in $O(\log_2 x)$ time
Note, $\lfloor \log_2 x \rfloor + 1$ = number of bits in binary representation of the key x [x not size of array!]
Hence, if $x \le 2^k$ for some constant k, then $\log_2 x \le \log_2 2^k = k$ and thus, exponential search runs in constant time

Proof-sketch (runtime in $O(\log_2 x)$).

Since L contains only pairwise-distinct keys from $\mathbb{N}$

$\Longrightarrow$ Key values x grow at least as fast as the element indices, i.e., $L[k] \ge k\ \forall k$.

$\Longrightarrow$ i is doubled at most $\log_2 x$ times because $i \ge \log_2(x)$ implies
$L[2^i] \ge 2^i \ge 2^{\log_2(x)} = x$. (gives stop criterion!)

$\Longrightarrow$ Determine the correct interval: in $\le j \le \log_2 x$ comparisons with $i = 2^j$.
Thus, $j \in O(\log_2 x)$

Length of this interval: $2^j - 2^{j-1} = 2^{j-1}(2-1) = 2^{j-1}$ where $j \in O(\log_2 x)$.
This means that the length of the interval being searched is in $O(2^{\log_2(x)-1})$.

Search within this intervall (e.g., binary search): $O(\log_2(2^{\log_2(x)-1})) = O(\log_2(x) - 1) = O(\log_2 x)$.

$\Longrightarrow$ Overall effort $O(\log_2 x)$ $\square$

Part 3: Searching in Arrays

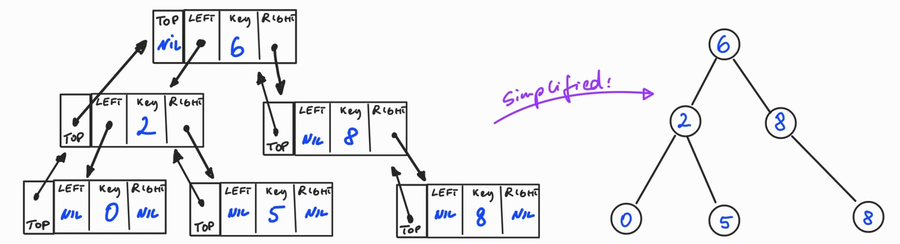
While searching in arrays is reasonable fast once they are sorted, modication of arrays (removing or adding elements) is a somewhat teadious task.

To support dynamic-set operations (including search, find_minimum, find_maximum, but also insert or delete) a tree data structure is more suitable.

Part 3: Binary Search Trees

Part 3: Binary Search Trees

A **binary** tree is a rooted tree for which each vertex has *at most* two children.



In a tree data structure, each vertex x is an object with

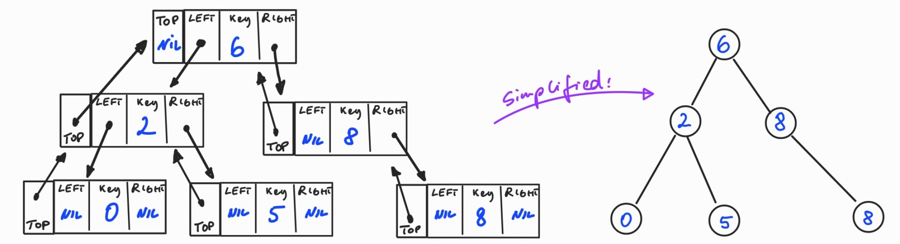
$x.key$ some value to be stored (maybe also some extra satellite data)

$x.left$, $x.right$, $x.top$ are pointers refering to the address of the left child, right child and parent of x, respectively

If a child or the parent is missing, the appropriate attribute contains the value *NIL* .

Part 3: Binary Search Trees

A **binary** tree is a rooted tree for which each vertex has *at most* two children.



In a tree data structure, each vertex x is an object with

- $x.key$ some value to be stored (maybe also some extra satellite data)
- $x.left$, $x.right$, $x.top$ are pointers refering to the address of the left child, right child and parent of x, respectively

  If a child or the parent is missing, the appropriate attribute contains the value *NIL* .

Part 3: Binary Search Trees

A **Binary Search Trees** is a binary tree in which the keys are always stored in such a way that they satisfy the binary-search-tree property:

Let x be a node in a binary search tree.

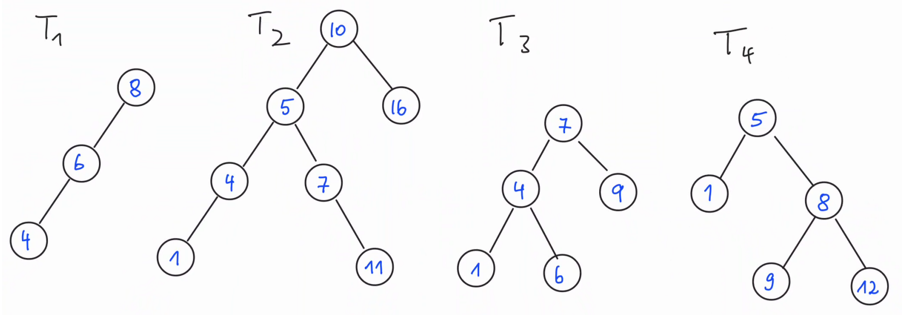
If y is a node in the left subtree of x, then $y.key \leq x.key$.

If y is a node in the right subtree of x, then $y.key \geq x.key$.

Part 3: Binary Search Trees

A **Binary Search Trees** is a binary tree in which the keys are always stored in such a way that they satisfy the binary-search-tree property:

> Let x be a node in a binary search tree.
>
> If y is a node in the left subtree of x, then $y.key \leq x.key$.
>
> If y is a node in the right subtree of x, then $y.key \geq x.key$.

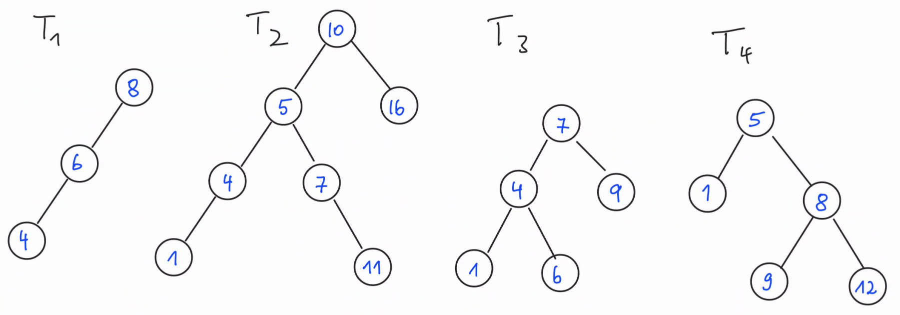


Which of them are search-trees?

Part 3: Binary Search Trees

A **Binary Search Trees** is a binary tree in which the keys are always stored in such a way that they satisfy the binary-search-tree property:

> Let x be a node in a binary search tree.
>
> If y is a node in the left subtree of x, then $y.key \leq x.key$.
>
> If y is a node in the right subtree of x, then $y.key \geq x.key$.



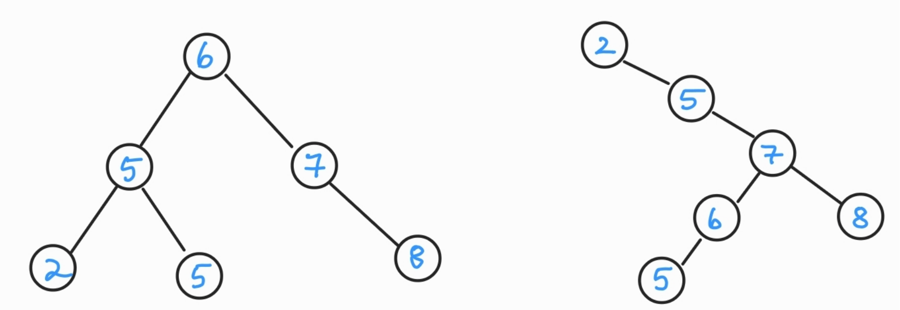
Which of them are search-trees?

Answer: Only T_1 and T_3 (T_2: 10/11, T_4: 8/9)

A **Binary Search Trees** is a binary tree in which the keys are always stored in such a way that they satisfy the binary-search-tree property:

> Let x be a node in a binary search tree.
>
> If y is a node in the left subtree of x, then $y.key \leq x.key$.
>
> If y is a node in the right subtree of x, then $y.key \geq x.key$.

Binary Search Trees are not necessarily uniquely determined:



Part 3: Binary Search Trees (Traversing)

Preorder:

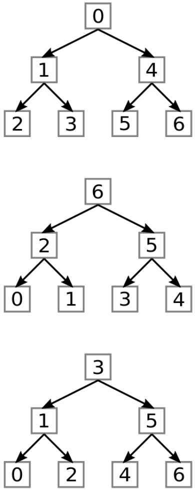
1. visit current vertex
2. recursively traverse left subtree
3. recursively traverse right subtree

Postorder:

1. recursively traverse left subtree
2. recursively traverse right subtree
3. visit current vertex

Inorder:

1. recursively traverse left subtree
2. visit current vertex
3. recursively traverse right subtree



numbers in squares =
order in which nodes are visited

Part 3: Binary Search Trees (Traversing)

Preorder:

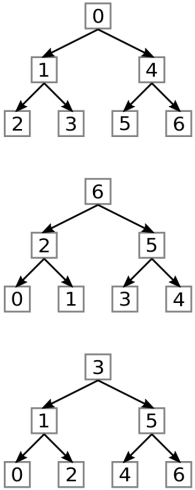
1. visit current vertex
2. recursively traverse left subtree
3. recursively traverse right subtree

Postorder:

1. recursively traverse left subtree
2. recursively traverse right subtree
3. visit current vertex

Inorder:

1. recursively traverse left subtree
2. visit current vertex
3. recursively traverse right subtree

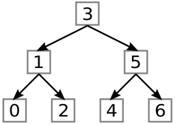


numbers in squares =
order in which nodes are visited

Part 3: Binary Search Trees (Inorder)

Inorder:

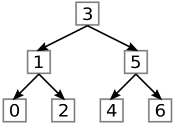
1. recursively traverse left subtree
2. visit current vertex
3. recursively traverse right subtree



Part 3: Binary Search Trees (Inorder)

Inorder:

1. recursively traverse left subtree
2. visit current vertex
3. recursively traverse right subtree

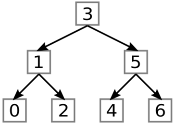



```
INORDER-TREE-WALK(x)
IF (x ≠ NIL) THEN
    INORDER-TREE-WALK(x.left)
    PRINT x.key
    INORDER-TREE-WALK(x.right)
```

Part 3: Binary Search Trees (Inorder)

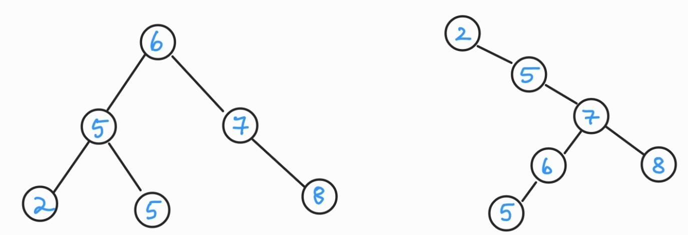
Inorder:

1. recursively traverse left subtree
2. visit current vertex
3. recursively traverse right subtree




```
INORDER-TREE-WALK(x)
  IF (x ≠ NIL) THEN
      INORDER-TREE-WALK(x.left)
      PRINT x.key
      INORDER-TREE-WALK(x.right)
```

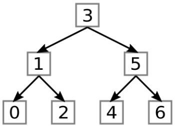
Execute `INORDER-TREE-WALK`(x) in:



Part 3: Binary Search Trees (Inorder)

Inorder:

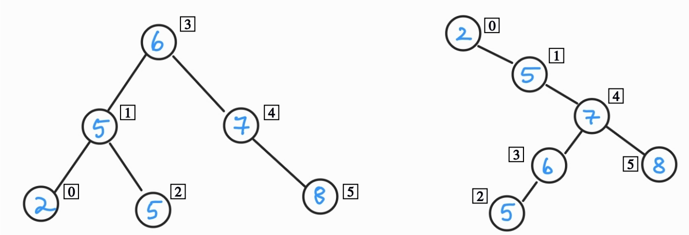
1. recursively traverse left subtree
2. visit current vertex
3. recursively traverse right subtree



INORDER-TREE-WALK(x)

```
IF (x ≠ NIL) THEN
    INORDER-TREE-WALK(x.left)
    PRINT x.key
    INORDER-TREE-WALK(x.right)
```

Execute INORDER-TREE-WALK(x) in:

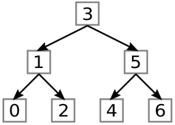


PRINT $x.key$: 2,5,5,6,7,8

Part 3: Binary Search Trees (Inorder)

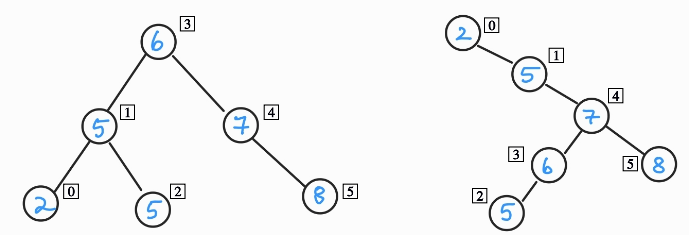
Inorder:

1. recursively traverse left subtree
2. visit current vertex
3. recursively traverse right subtree




```
INORDER-TREE-WALK(x)
  IF (x ≠ NIL) THEN
      INORDER-TREE-WALK(x.left)
      PRINT x.key
      INORDER-TREE-WALK(x.right)
```

Execute INORDER-TREE-WALK(x) in:



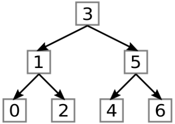
PRINT $x.key$: 2,5,5,6,7,8

Inorder traversal allows us to print all elements in a search tree in sorted order.

Part 3: Binary Search Trees (Inorder)

Inorder:

1. recursively traverse left subtree
2. visit current vertex
3. recursively traverse right subtree



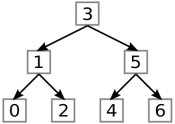

```
INORDER-TREE-WALK(x)
IF (x ≠ NIL) THEN
    INORDER-TREE-WALK(x.left)
    PRINT x.key
    INORDER-TREE-WALK(x.right)
```

Theorem. *If x is a root of an n-vertex binary search tree, then* INORDER-TREE-WALK(x) *prints all elements in sorted order in $\Theta(n)$ time*

Part 3: Binary Search Trees (Inorder)

Inorder:

1. recursively traverse left subtree
2. visit current vertex
3. recursively traverse right subtree




```
INORDER-TREE-WALK(x)
  IF (x ≠ NIL) THEN
      INORDER-TREE-WALK(x.left)
      PRINT x.key
      INORDER-TREE-WALK(x.right)
```

Theorem. *If x is a root of an n-vertex binary search tree, then* INORDER-TREE-WALK(x) *prints all elements in sorted order in $\Theta(n)$ time*

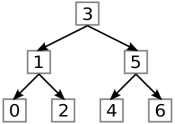
Proof. correct: due to binary-search-tree property:

$y.key \leq x.key$ if y in left subtree and $x.key \leq y.key$ if y in right subtree

Part 3: Binary Search Trees (Inorder)

Inorder:

1. recursively traverse left subtree
2. visit current vertex
3. recursively traverse right subtree




```
INORDER-TREE-WALK(x)
  IF (x ≠ NIL) THEN
      INORDER-TREE-WALK(x.left)
      PRINT x.key
      INORDER-TREE-WALK(x.right)
```

Theorem. *If x is a root of an n-vertex binary search tree, then* INORDER-TREE-WALK(x) *prints all elements in sorted order in* $\Theta(n)$ *time*

Proof. correct: due to binary-search-tree property:

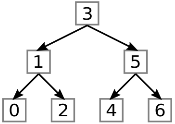
$y.key \leq x.key$ if y in left subtree and $x.key \leq y.key$ if y in right subtree

runtime: INORDER-TREE-WALK(x) visits all n nodes of the subtree $\implies T(n) = \Omega(n)$

Part 3: Binary Search Trees (Inorder)

Inorder:

1. recursively traverse left subtree
2. visit current vertex
3. recursively traverse right subtree




```
INORDER-TREE-WALK(x)
  IF (x ≠ NIL) THEN
      INORDER-TREE-WALK(x.left)
      PRINT x.key
      INORDER-TREE-WALK(x.right)
```

Theorem. *If x is a root of an n-vertex binary search tree, then* INORDER-TREE-WALK(x) *prints all elements in sorted order in $\Theta(n)$ time*

Proof. correct: due to binary-search-tree property:
$y.key \leq x.key$ if y in left subtree and $x.key \leq y.key$ if y in right subtree

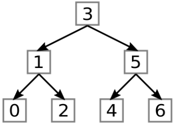
runtime: INORDER-TREE-WALK(x) visits all n nodes of the subtree $\implies T(n) = \Omega(n)$

left subtree $k \geq 0$ nodes and right subtree has $n - k - 1$ nodes
$\implies T(n) = T(k) + T(n - k - 1) + d$ where $T(0) = c$ (c, d constants)

Part 3: Binary Search Trees (Inorder)

Inorder:

1. recursively traverse left subtree
2. visit current vertex
3. recursively traverse right subtree




```
INORDER-TREE-WALK(x)
IF (x ≠ NIL) THEN
    INORDER-TREE-WALK(x.left)
    PRINT x.key
    INORDER-TREE-WALK(x.right)
```

Theorem. *If x is a root of an n-vertex binary search tree, then* INORDER-TREE-WALK(x) *prints all elements in sorted order in $\Theta(n)$ time*

Proof. correct: due to binary-search-tree property:

$y.key \leq x.key$ if y in left subtree and $x.key \leq y.key$ if y in right subtree

runtime: INORDER-TREE-WALK(x) visits all n nodes of the subtree $\implies T(n) = \Omega(n)$

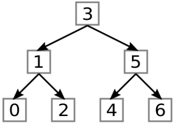
left subtree $k \geq 0$ nodes and right subtree has $n - k - 1$ nodes
$\implies T(n) = T(k) + T(n-k-1) + d$ where $T(0) = c$ (c, d constants)

Show, by induction, $T(n) = (c+d)n + c$.

Part 3: Binary Search Trees (Inorder)

Inorder:

1. recursively traverse left subtree
2. visit current vertex
3. recursively traverse right subtree




```
INORDER-TREE-WALK(x)
IF (x ≠ NIL) THEN
    INORDER-TREE-WALK(x.left)
    PRINT x.key
    INORDER-TREE-WALK(x.right)
```

Theorem. *If x is a root of an n-vertex binary search tree, then* INORDER-TREE-WALK(x) *prints all elements in sorted order in $\Theta(n)$ time*

Proof. correct: due to binary-search-tree property:

$y.key \leq x.key$ if y in left subtree and $x.key \leq y.key$ if y in right subtree

runtime: INORDER-TREE-WALK(x) visits all n nodes of the subtree $\implies T(n) = \Omega(n)$

left subtree $k \geq 0$ nodes and right subtree has $n - k - 1$ nodes
$\implies T(n) = T(k) + T(n - k - 1) + d$ where $T(0) = c$ (c, d constants)

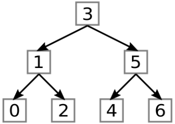
Show, by induction, $T(n) = (c + d)n + c$.

Base case $\ell = 0$: $T(0) = (c + d) \cdot 0 + c = c$ correct

Part 3: Binary Search Trees (Inorder)

Inorder:

1. recursively traverse left subtree
2. visit current vertex
3. recursively traverse right subtree



INORDER-TREE-WALK(x)
IF $(x \neq NIL)$ THEN
    INORDER-TREE-WALK$(x.left)$
    PRINT $x.key$
    INORDER-TREE-WALK$(x.right)$

Theorem. *If x is a root of an n-vertex binary search tree, then* INORDER-TREE-WALK(x) *prints all elements in sorted order in $\Theta(n)$ time*

Proof. correct: due to binary-search-tree property:

$y.key \leq x.key$ if y in left subtree and $x.key \leq y.key$ if y in right subtree

runtime: INORDER-TREE-WALK(x) visits all n nodes of the subtree $\implies T(n) = \Omega(n)$

left subtree $k \geq 0$ nodes and right subtree has $n - k - 1$ nodes
$\implies T(n) = T(k) + T(n - k - 1) + d$ where $T(0) = c$ (c, d constants)

Show, by induction, $T(n) = (c + d)n + c$.

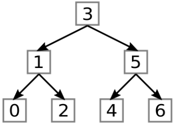
Base case $\ell = 0$: $T(0) = (c + d) \cdot 0 + c = c$ correct

Assume $T(\ell) = (c + d)\ell + c$ true for all $\ell < n$.

Part 3: Binary Search Trees (Inorder)

Inorder:

1. recursively traverse left subtree
2. visit current vertex
3. recursively traverse right subtree




```
INORDER-TREE-WALK(x)
  IF (x ≠ NIL) THEN
     INORDER-TREE-WALK(x.left)
     PRINT x.key
     INORDER-TREE-WALK(x.right)
```

Theorem. *If x is a root of an n-vertex binary search tree, then* INORDER-TREE-WALK(x) *prints all elements in sorted order in $\Theta(n)$ time*

Proof. correct: due to binary-search-tree property:

$y.key \leq x.key$ if y in left subtree and $x.key \leq y.key$ if y in right subtree

runtime: INORDER-TREE-WALK(x) visits all n nodes of the subtree $\implies T(n) = \Omega(n)$

left subtree $k \geq 0$ nodes and right subtree has $n - k - 1$ nodes
$\implies T(n) = T(k) + T(n-k-1) + d$ where $T(0) = c$ (c, d constants)

Show, by induction, $T(n) = (c + d)n + c$.

Base case $\ell = 0$: $T(0) = (c + d) \cdot 0 + c = c$ correct

Assume $T(\ell) = (c + d)\ell + c$ true for all $\ell < n$.

$$T(n) = T(k) + T(n-k-1) + d =$$
$$= [(c+d)k + c] + [(c+d)(n-k-1) + c] + d = (c+d)n + c = O(n) \qquad \square$$

Part 3: Binary Search Trees (Querying)

As discussed next, binary search trees support the queries search, find_minimum, find_maximum, ...

Each query can be done on $O(h)$ time on any binary search tree of height h.

Recall: height of tree T is $h(T) =$ #edges along longest simple path from ρ_T to a leaf.

Part 3: Binary Search Trees (Querying: Search)

```
//find value k in subtree rooted at x
Tree-Search(x, k)
IF (x = NIL or k = x.key) THEN
    RETURN x
IF (k < x.key) THEN
    RETURN Tree-Search(x.left, k)
ELSE RETURN Tree-Search(x.right, k)
```

The Tree-Search procedure begins its search at the root and traces a simple “downward” path

If $k = x.key$, search terminates (k found). If $x = NIL$, search terminates (k not found).

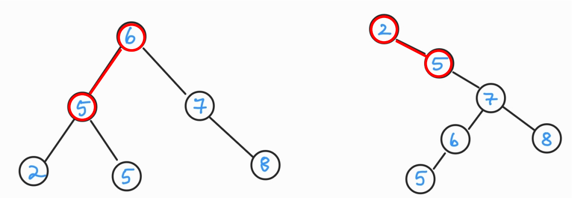
Hence, $k \neq x.key$ and $x \neq NIL$ implies either $k < x.key$ (continue left) or $k > x.key$ (continue right).

Due to binary-search-tree property, Tree-Search is correct.

The nodes encountered during the recursion form a simple path downward from the root of the tree, and thus the running time of Tree-Search is $O(h)$ where h is the height of the tree.

Part 3: Binary Search Trees (Querying: Search)

```
//find value k in subtree rooted at x
Tree-Search(x, k)
  IF (x = NIL or k = x.key) THEN
       RETURN x
  IF (k < x.key) THEN
       RETURN Tree-Search(x.left, k)
  ELSE RETURN Tree-Search(x.right, k)
```



`Tree-Search`$(T.root, 5)$

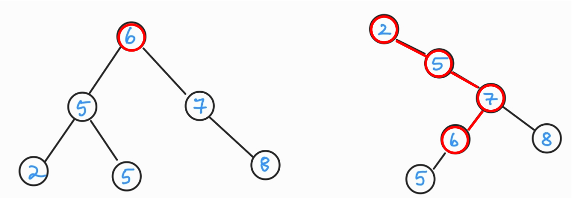
The `Tree-Search` procedure begins its search at the root and traces a simple "downward" path

If $k = x.key$, search terminates (k found). If $x = NIL$, search terminates (k not found).
Hence, $k \neq x.key$ and $x \neq NIL$ implies either $k < x.key$ (continue left) or $k > x.key$ (continue right).
Due to binary-search-tree property, `Tree-Search` is correct.

The nodes encountered during the recursion form a simple path downward from the root of the tree, and thus the running time of `Tree-Search` is $O(h)$ where h is the height of the tree.

Part 3: Binary Search Trees (Querying: Search)

```
//find value k in subtree rooted at x
Tree-Search(x, k)
  IF (x = NIL or k = x.key) THEN
      RETURN x
  IF (k < x.key) THEN
      RETURN Tree-Search(x.left, k)
  ELSE RETURN Tree-Search(x.right, k)
```



`Tree-Search`$(T.root, 6)$

The `Tree-Search` procedure begins its search at the root and traces a simple "downward" path

If $k = x.key$, search terminates (k found). If $x = NIL$, search terminates (k not found).

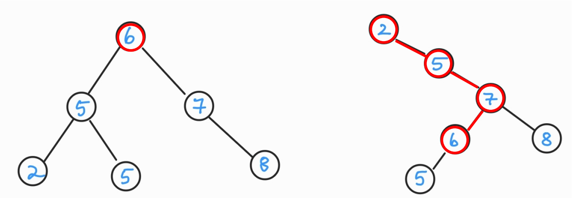
Hence, $k \neq x.key$ and $x \neq NIL$ implies either $k < x.key$ (continue left) or $k > x.key$ (continue right).

Due to binary-search-tree property, `Tree-Search` is correct.

The nodes encountered during the recursion form a simple path downward from the root of the tree, and thus the running time of `Tree-Search` is $O(h)$ where h is the height of the tree.

Part 3: Binary Search Trees (Querying: Search)

```
//find value k in subtree rooted at x
Tree-Search(x, k)
  IF (x = NIL or k = x.key) THEN
      RETURN x
  IF (k < x.key) THEN
      RETURN Tree-Search(x.left, k)
  ELSE RETURN Tree-Search(x.right, k)
```



Tree-Search($T.root$, 6)

The Tree-Search procedure begins its search at the root and traces a simple “downward” path

If $k = x.key$, search terminates (k found). If $x = NIL$, search terminates (k not found).

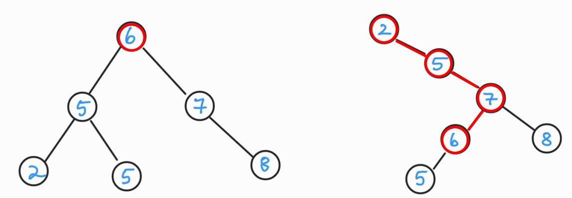
Hence, $k \neq x.key$ and $x \neq NIL$ implies either $k < x.key$ (continue left) or $k > x.key$ (continue right).

Due to binary-search-tree property, Tree-Search is correct.

The nodes encountered during the recursion form a simple path downward from the root of the tree, and thus the running time of Tree-Search is $O(h)$ where h is the height of the tree.

Part 3: Binary Search Trees (Querying: Search)

```
//find value k in subtree rooted at x
Tree-Search(x, k)
  IF (x = NIL or k = x.key) THEN
      RETURN x
  IF (k < x.key) THEN
      RETURN Tree-Search(x.left, k)
  ELSE RETURN Tree-Search(x.right, k)
```



`Tree-Search`$(T.root, 6)$

The `Tree-Search` procedure begins its search at the root and traces a simple "downward" path

If $k = x.key$, search terminates (k found). If $x = NIL$, search terminates (k not found).

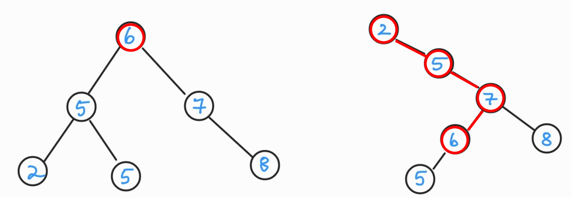
Hence, $k \neq x.key$ and $x \neq NIL$ implies either $k < x.key$ (continue left) or $k > x.key$ (continue right).

Due to binary-search-tree property, `Tree-Search` is correct.

The nodes encountered during the recursion form a simple path downward from the root of the tree, and thus the running time of `Tree-Search` is $O(h)$ where h is the height of the tree.

Part 3: Binary Search Trees (Querying: Search)

```
//find value k in subtree rooted at x
Tree-Search(x, k)
  IF (x = NIL or k = x.key) THEN
      RETURN x
  IF (k < x.key) THEN
      RETURN Tree-Search(x.left, k)
  ELSE RETURN Tree-Search(x.right, k)
```



Tree-Search($T.root$, 6)

The Tree-Search procedure begins its search at the root and traces a simple “downward” path

If $k = x.key$, search terminates (k found). If $x = NIL$, search terminates (k not found).

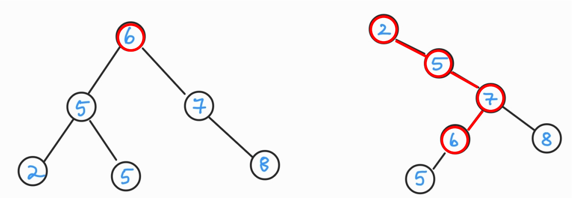
Hence, $k \neq x.key$ and $x \neq NIL$ implies either $k < x.key$ (continue left) or $k > x.key$ (continue right).

Due to binary-search-tree property, Tree-Search is correct.

The nodes encountered during the recursion form a simple path downward from the root of the tree, and thus the running time of Tree-Search is $O(h)$ where h is the height of the tree.

Part 3: Binary Search Trees (Querying: Search)

```
//find value k in subtree rooted at x
Tree-Search(x, k)
  IF (x = NIL or k = x.key) THEN
      RETURN x
  IF (k < x.key) THEN
      RETURN Tree-Search(x.left, k)
  ELSE RETURN Tree-Search(x.right, k)
```



`Tree-Search`$(T.root, 6)$

The `Tree-Search` procedure begins its search at the root and traces a simple "downward" path

If $k = x.key$, search terminates (k found). If $x = NIL$, search terminates (k not found).

Hence, $k \neq x.key$ and $x \neq NIL$ implies either $k < x.key$ (continue left) or $k > x.key$ (continue right).

Due to binary-search-tree property, `Tree-Search` is correct.

The nodes encountered during the recursion form a simple path downward from the root of the tree, and thus the running time of `Tree-Search` is $O(h)$ where h is the height of the tree.

Part 3: Binary Search Trees (Querying: Find min / max)

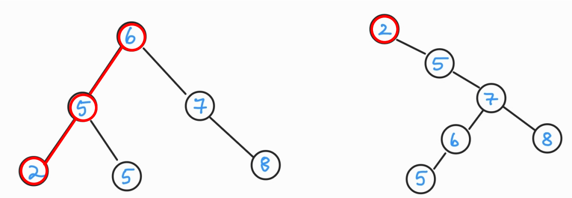
```
//Find minimum key in T(x) assuming
x ≠ NIL
Tree-Min(x)
WHILE (x.left ≠ NIL) DO
      x := x.left
RETURN x
```

Similar arguments as before show that `Tree-Min`(x), resp., `Tree-Max`(x) correctly determines the max, resp., min element in the subtree rooted at x in $O(h)$ time where h is the height of the tree.

Part 3: Binary Search Trees (Querying: Find min / max)

//Find minimum key in $T(x)$ assuming $x \neq NIL$

```
Tree-Min(x)
   WHILE (x.left ≠ NIL) DO
        x := x.left
   RETURN x
```



Tree-Min($T.root$)

Similar arguments as before show that Tree-Min(x), resp., Tree-Max(x) correctly determines the max, resp., min element in the subtree rooted at x in $O(h)$ time where h is the height of the tree.

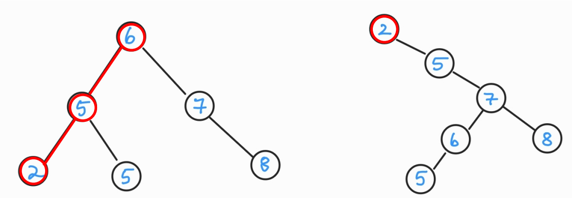
Part 3: Binary Search Trees (Querying: Find min / max)

//Find minimum key in $T(x)$ assuming $x \neq NIL$

```
Tree-Min(x)
  WHILE (x.left ≠ NIL) DO
      x := x.left
  RETURN x
```

//Find maximum key in $T(x)$ assuming $x \neq NIL$

```
Tree-Max(x)
  WHILE (x.right ≠ NIL) DO
      x := x.right
  RETURN x
```



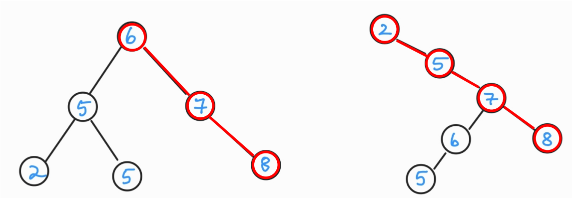
`Tree-Min`$(T.root)$

Similar arguments as before show that `Tree-Min`(x), resp., `Tree-Max`(x) correctly determines the max, resp., min element in the subtree rooted at x in $O(h)$ time where h is the height of the tree.

Part 3: Binary Search Trees (Querying: Find min / max)

```
//Find minimum key in T(x) assuming x ≠ NIL
Tree-Min(x)
    WHILE (x.left ≠ NIL) DO
        x := x.left
    RETURN x
```

```
//Find maximum key in T(x) assuming x ≠ NIL
Tree-Max(x)
    WHILE (x.right ≠ NIL) DO
        x := x.right
    RETURN x
```



`Tree-Max`($T.root$)

Similar arguments as before show that `Tree-Min`(x), resp., `Tree-Max`(x) correctly determines the max, resp., min element in the subtree rooted at x in $O(h)$ time where h is the height of the tree.

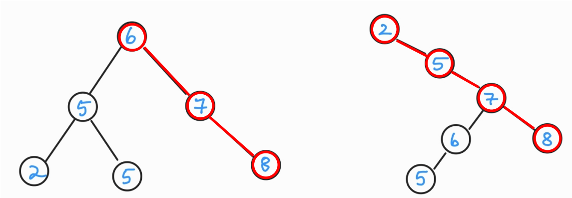
Part 3: Binary Search Trees (Querying: Find min / max)

//Find minimum key in $T(x)$ assuming $x \neq NIL$

```
Tree-Min(x)
    WHILE (x.left ≠ NIL) DO
        x := x.left
    RETURN x
```

//Find maximum key in $T(x)$ assuming $x \neq NIL$

```
Tree-Max(x)
    WHILE (x.right ≠ NIL) DO
        x := x.right
    RETURN x
```



Tree-Max($T.root$)

Similar arguments as before show that Tree-Min(x), resp., Tree-Max(x) correctly determines the max, resp., min element in the subtree rooted at x in $O(h)$ time where h is the height of the tree.

Part 3: Binary Search Trees (Insert)

```
Tree-Insert(T, z)
  x := T.root //node being compared with z
  y := NIL //y will be parent of z
  WHILE (x ≠ NIL)
  //descend until reaching a leaf
      y := x
      IF (z.key < x.key) THEN x := x.left
      ELSE x := x.right
  z.top := y
  //found the location - insert z with parent y
  IF (y = NIL) THEN T.root := z //T was empty
  ELSEIF (z.key < y.key) THEN y.left := z
  ELSE y.right := z
```

Example board

get left tree by insertion in order e.g. $6, 5, 5, 2, 7, 8$
get right tree by insertion in order e.g. $2, 5, 7, 6, 5, 8$

Part 3: Binary Search Trees (Insert)

```
Tree-Insert(T, z)
x := T.root //node being compared with z
y := NIL //y will be parent of z
WHILE (x ≠ NIL)
//descend until reaching a leaf
    y := x
    IF (z.key < x.key) THEN x := x.left
    ELSE x := x.right
z.top := y
//found the location - insert z with parent y
IF (y = NIL) THEN T.root := z //T was empty
ELSEIF (z.key < y.key) THEN y.left := z
ELSE y.right := z
```

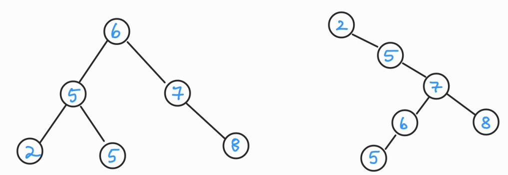
Example board

get left tree by insertion in order e.g. 6, 5, 5, 2, 7, 8
get right tree by insertion in order e.g. 2, 5, 7, 6, 5, 8

Part 3: Binary Search Trees (Insert)

```
Tree-Insert(T, z)
x := T.root //node being compared with z
y := NIL //y will be parent of z
WHILE (x ≠ NIL)
//descend until reaching a leaf
    y := x
    IF (z.key < x.key) THEN x := x.left
    ELSE x := x.right
z.top := y
//found the location - insert z with parent y
IF (y = NIL) THEN T.root := z //T was empty
ELSEIF (z.key < y.key) THEN y.left := z
ELSE y.right := z
```

Example board



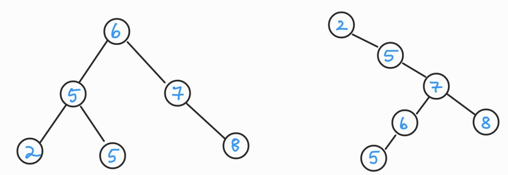
get left tree by insertion in order e.g. 6, 5, 5, 2, 7, 8
get right tree by insertion in order e.g. 2, 5, 7, 6, 5, 8

Part 3: Binary Search Trees (Insert)

Tree-Insert(T, z)

$x := T.root$ //node being compared with z
$y := NIL$ //y will be parent of z
WHILE $(x \neq NIL)$
//descend until reaching a leaf
    $y := x$
    IF $(z.key < x.key)$ THEN $x := x.left$
    ELSE $x := x.right$
$z.top := y$
//found the location - insert z with parent y
IF $(y = NIL)$ THEN $T.root := z$ //T was empty
ELSEIF $(z.key < y.key)$ THEN $y.left := z$
ELSE $y.right := z$

Example board



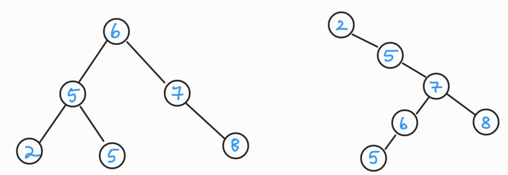
get left tree by insertion in order e.g. 6, 5, 5, 2, 7, 8
get right tree by insertion in order e.g. 2, 5, 7, 6, 5, 8

Tree-Insert(T, z) is correct and runs in $O(h)$ time where h = height of tree [exercise]

Part 3: Binary Search Trees (Insert)

Tree-Insert(T, z)
$x := T.root$ //node being compared with z
$y := NIL$ //y will be parent of z
WHILE $(x \neq NIL)$
//descend until reaching a leaf
  $y := x$
  IF $(z.key < x.key)$ THEN $x := x.left$
  ELSE $x := x.right$
$z.top := y$
//found the location - insert z with parent y
IF $(y = NIL)$ THEN $T.root := z$ //T was empty
ELSEIF $(z.key < y.key)$ THEN $y.left := z$
ELSE $y.right := z$

Example board



get left tree by insertion in order e.g. 6, 5, 5, 2, 7, 8
get right tree by insertion in order e.g. 2, 5, 7, 6, 5, 8

Tree-Insert(T, z) is correct and runs in $O(h)$ time where h = height of tree [exercise]

deletion of vertices is more involved in case we want to keep the binary-search-tree property but works also in $O(h)$ time

[left tree: deleting 8 or 7 is easy, deleting inner 5 is more involved]

Part 3: Binary Search Trees (delete)

`Tree-Delete`(T, z)

CASE z has no child:
just delete z

CASE z has one child:
Delete z and make child x of z ..
.. the right child of $parent(z) = v$ in case z is right child of v
.. the left child of $parent(z) = v$ in case z is left child of v
to get tree T'

CASE z has two children:
Find y in $T(r)$ with min $y.key$ and such that y has no left child $\Rightarrow$ y has 0 or 1 child
[latter needed if $y'.key = y.key$ for some y']

Delete y from T [see cases above]
Replace z by y [$y = r$ is possible] and we get tree T'

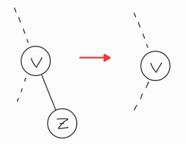
Delete vertex w means remove w from $V(T)$ and all edges from $E(T)$ that contain w !! [Example Board]

`Tree-Delete`(T, z) yields a BST

Part 3: Binary Search Trees (delete)

`Tree-Delete`(T, z)

CASE z has no child:
just delete z



CASE z has one child:
Delete z and make child x of z ..
.. the right child of $parent(z) = v$ in case z is right child of v
.. the left child of $parent(z) = v$ in case z is left child of v
to get tree T'

CASE z has two children:
Find y in $T(r)$ with min $y.key$ and such that y has no left child $\Rightarrow$ y has 0 or 1 child
[latter needed if $y'.key = y.key$ for some y']

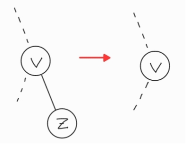
Delete y from T [see cases above]
Replace z by y [$y = r$ is possible] and we get tree T'

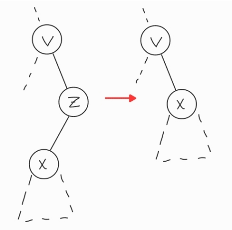
Delete vertex w means remove w from $V(T)$ and all edges from $E(T)$ that contain w !! [Example Board]

`Tree-Delete`(T, z) yields a BST

`Tree-Delete`(T, z)

CASE z has no child:
just delete z





CASE z has one child:
Delete z and make child x of z ..
.. the right child of $parent(z) = v$ in case z is right child of v
.. the left child of $parent(z) = v$ in case z is left child of v
to get tree T'

CASE z has two children:
Find y in $T(r)$ with min $y.key$ and such that y has no left child $\Rightarrow$ y has 0 or 1 child
[latter needed if $y'.key = y.key$ for some y']

Delete y from T [see cases above]
Replace z by y [$y = r$ is possible] and we get tree T'

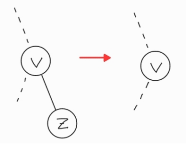
Delete vertex w means remove w from $V(T)$ and all edges from $E(T)$ that contain w !! [Example Board]

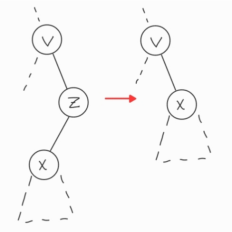
`Tree-Delete`(T, z) yields a BST

Part 3: Binary Search Trees (delete)

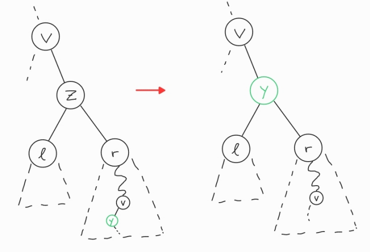
`Tree-Delete`(T, z)

CASE z has no child:
just delete z





CASE z has one child:
Delete z and make child x of z ..
.. the right child of $parent(z) = v$ in case z is right child of v
.. the left child of $parent(z) = v$ in case z is left child of v
to get tree T'



CASE z has two children:
Find y in $T(r)$ with min $y.key$ and such that y has no left child $\Rightarrow$ y has 0 or 1 child
[latter needed if $y'.key = y.key$ for some y']

Delete y from T [see cases above]
Replace z by y [$y = r$ is possible] and we get tree T'

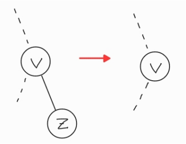
Delete vertex w means remove w from $V(T)$ and all edges from $E(T)$ that contain w !! [Example Board]

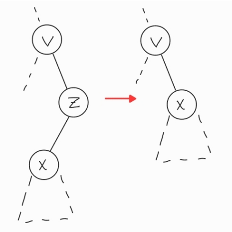
`Tree-Delete`(T, z) yields a BST

Part 3: Binary Search Trees (delete)

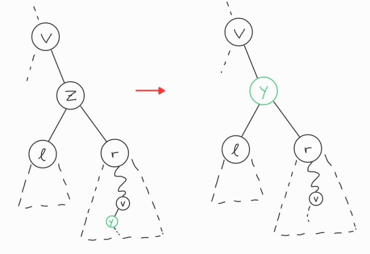
`Tree-Delete`(T, z)

CASE z has no child:
just delete z





CASE z has one child:
Delete z and make child x of z ..
.. the right child of $parent(z) = v$ in case z is right child of v
.. the left child of $parent(z) = v$ in case z is left child of v
to get tree T'



CASE z has two children:
Find y in $T(r)$ with min $y.key$ and such that y has no left child $\Rightarrow$ y has 0 or 1 child
[latter needed if $y'.key = y.key$ for some y']

Delete y from T [see cases above]
Replace z by y [$y = r$ is possible] and we get tree T'

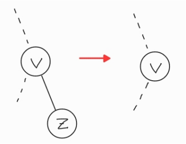
Delete vertex w means remove w from $V(T)$ and all edges from $E(T)$ that contain w !! [Example Board]

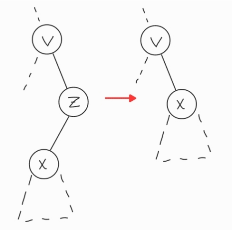
`Tree-Delete`(T, z) yields a BST

Part 3: Binary Search Trees (delete)

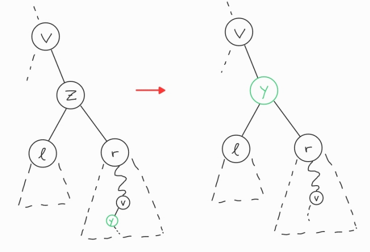
`Tree-Delete`(T, z)

CASE z has no child:
just delete z





CASE z has one child:
Delete z and make child x of z ..
.. the right child of $parent(z) = v$ in case z is right child of v
.. the left child of $parent(z) = v$ in case z is left child of v
to get tree T'



CASE z has two children:
Find y in $T(r)$ with min $y.key$ and such that y has no left child $\Rightarrow$ y has 0 or 1 child
[latter needed if $y'.key = y.key$ for some y']

Delete y from T [see cases above]
Replace z by y [$y = r$ is possible] and we get tree T'

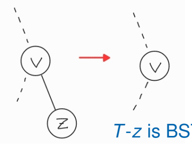
Delete vertex w means remove w from $V(T)$ and all edges from $E(T)$ that contain w !! [Example Board]

`Tree-Delete`(T, z) yields a BST

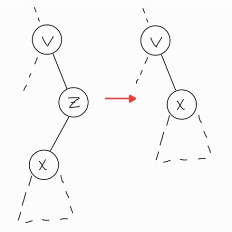
Part 3: Binary Search Trees (delete)

`Tree-Delete`(T, z)

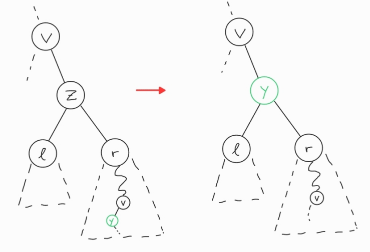
CASE z has no child:
just delete z



T-z is BST



CASE z has one child:
Delete z and make child x of z ..
.. the right child of $parent(z) = v$ in case z is right child of v
.. the left child of $parent(z) = v$ in case z is left child of v
to get tree T'



CASE z has two children:
Find y in $T(r)$ with min $y.key$ and such that y has no left child $\Rightarrow$ y has 0 or 1 child
[latter needed if $y'.key = y.key$ for some y']

Delete y from T [see cases above]
Replace z by y [$y = r$ is possible] and we get tree T'

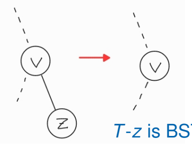
Delete vertex w means remove w from $V(T)$ and all edges from $E(T)$ that contain w !! [Example Board]

`Tree-Delete`(T, z) yields a BST

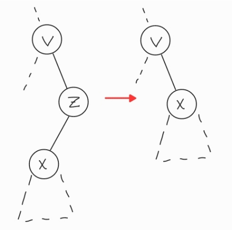
Part 3: Binary Search Trees (delete)

$\texttt{Tree-Delete}(T, z)$

CASE z has no child:
just delete z

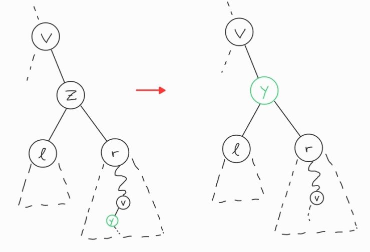


T-z is BST



CASE z has one child:
Delete z and make child x of z ..
.. the right child of $parent(z) = v$ in case z is right child of v
.. the left child of $parent(z) = v$ in case z is left child of v
to get tree T'

Case: z is right child of v:
T is $BST \Rightarrow \forall\ w$ in $T(z)$: $v.key \leq w.key$



CASE z has two children:
Find y in $T(r)$ with min $y.key$ and such that y has no left child $\Rightarrow$ y has 0 or 1 child
[latter needed if $y'.key = y.key$ for some y']

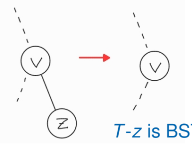
Delete y from T [see cases above]
Replace z by y [$y = r$ is possible] and we get tree T'

Delete vertex w means remove w from $V(T)$ and all edges from $E(T)$ that contain w !! [Example Board]

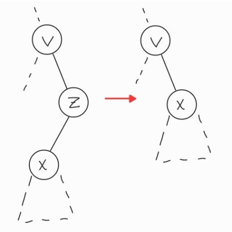
$\texttt{Tree-Delete}(T, z)$ yields a BST

`Tree-Delete`(T, z)

CASE z **has no child:**
just delete z

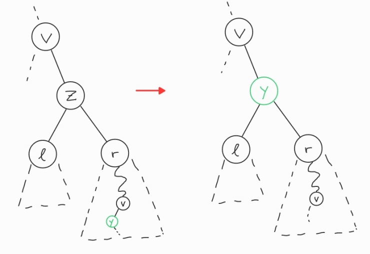


T-z is BST



CASE z **has one child:**
Delete z and make child x of z ..
.. the right child of $parent(z) = v$ in case z is right child of v
.. the left child of $parent(z) = v$ in case z is left child of v
to get tree T'

Case: z is right child of v:
T is $BST \Rightarrow \forall\ w$ in $T(z)$: $v.key \leq w.key$
$\Rightarrow \forall\ w$ in $T(x)$: $v.key \leq w.key$



CASE z **has two children:**
Find y in $T(r)$ with min $y.key$ and such that y has no left child $\Rightarrow$ y has 0 or 1 child
[latter needed if $y'.key = y.key$ for some y']

Delete y from T [see cases above]
Replace z by y [$y = r$ is possible] and we get tree T'

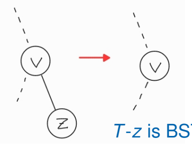
Delete vertex w **means remove** w **from** $V(T)$ **and all edges from** $E(T)$ **that contain** w **!!** [Example Board]

`Tree-Delete`(T, z) yields a BST

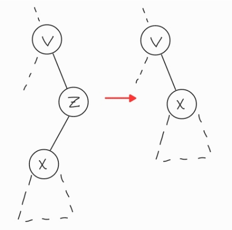
Part 3: Binary Search Trees (delete)

`Tree-Delete`(T, z)

CASE z has no child:
just delete z

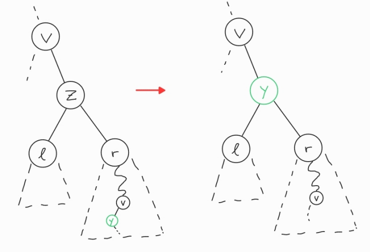


T-z is BST



CASE z has one child:
Delete z and make child x of z ..
.. the right child of $parent(z) = v$ in case z is right child of v
.. the left child of $parent(z) = v$ in case z is left child of v
to get tree T'

Case: z is right child of v:
T is $BST \Rightarrow \forall$ w in $T(z)$: $v.key \leq w.key$
$\Rightarrow \forall$ w in $T(x)$: $v.key \leq w.key$
$\Rightarrow \forall$ w in $T'(x)$: $v.key \leq w.key$



CASE z has two children:
Find y in $T(r)$ with min $y.key$ and such that y has no left child $\Rightarrow$ y has 0 or 1 child
[latter needed if $y'.key = y.key$ for some y']

Delete y from T [see cases above]
Replace z by y [$y = r$ is possible] and we get tree T'

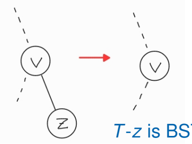
Delete vertex w means remove w from $V(T)$ and all edges from $E(T)$ that contain w !! [Example Board]

`Tree-Delete`(T, z) yields a BST

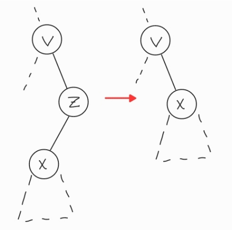
Part 3: Binary Search Trees (delete)

`Tree-Delete`(T, z)

CASE z has no child:
just delete z

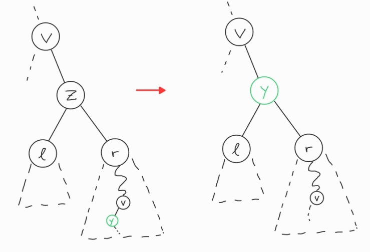


T-z is BST



CASE z has one child:
Delete z and make child x of z ..
.. the right child of *parent*$(z) = v$ in case z is right child of v
.. the left child of *parent*$(z) = v$ in case z is left child of v
to get tree T'

Case: z is right child of v:
T is $BST \Rightarrow \forall$ w in $T(z)$: $v.key \leq w.key$
$\Rightarrow \forall$ w in $T(x)$: $v.key \leq w.key$
$\Rightarrow \forall$ w in $T'(x)$: $v.key \leq w.key$
and other key "posititions" unchanged in T'



CASE z has two children:
Find y in $T(r)$ with min $y.key$ and such that y has no left child $\Rightarrow$ y has 0 or 1 child
[latter needed if $y'.key = y.key$ for some y']
Delete y from T [see cases above]
Replace z by y [$y = r$ is possible] and we get tree T'

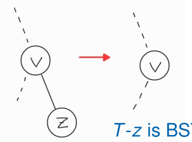
Delete vertex w means remove w from $V(T)$ and all edges from $E(T)$ that contain w !! [Example Board]

`Tree-Delete`(T, z) yields a BST

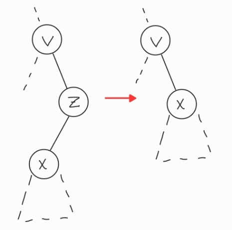
Part 3: Binary Search Trees (delete)

`Tree-Delete`(T, z)

CASE z has no child:
just delete z

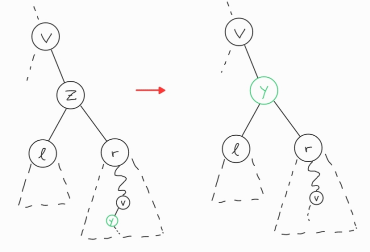


T-z is BST



CASE z has one child:
Delete z and make child x of z ..
.. the right child of $parent(z) = v$ in case z is right child of v
.. the left child of $parent(z) = v$ in case z is left child of v
to get tree T'

Case: z is right child of v:
T is $BST \Rightarrow \forall$ w in $T(z)$: $v.key \leq w.key$
$\Rightarrow \forall$ w in $T(x)$: $v.key \leq w.key$
$\Rightarrow \forall$ w in $T'(x)$: $v.key \leq w.key$
and other key "posititions" unchanged in T'
$\Rightarrow$ T' is BST. (for case z is left child of v replace $\leq$ by $\geq$)



CASE z has two children:
Find y in $T(r)$ with min $y.key$ and such that y has no left child $\Rightarrow$ y has 0 or 1 child
[latter needed if $y'.key = y.key$ for some y']

Delete y from T [see cases above]
Replace z by y [$y = r$ is possible] and we get tree T'

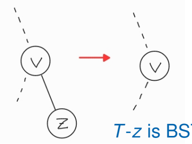
Delete vertex w means remove w from $V(T)$ and all edges from $E(T)$ that contain w !! [Example Board]

`Tree-Delete`(T, z) yields a BST

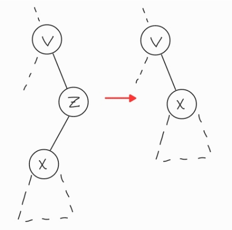
Part 3: Binary Search Trees (delete)

Tree-Delete(T, z)

CASE z has no child:
just delete z

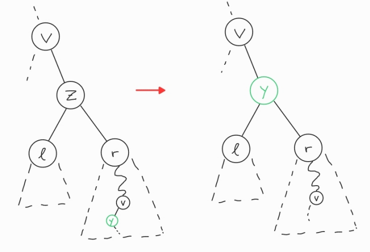


T-z is BST



CASE z has one child:
Delete z and make child x of z ..
.. the right child of $parent(z) = v$ in case z is right child of v
.. the left child of $parent(z) = v$ in case z is left child of v
to get tree T'

Case: z is right child of v:
T is $BST \Rightarrow \forall$ w in $T(z)$: $v.key \leq w.key$
$\Rightarrow \forall$ w in $T(x)$: $v.key \leq w.key$
$\Rightarrow \forall$ w in $T'(x)$: $v.key \leq w.key$
and other *key* "posititions" unchanged in T'
$\Rightarrow$ T' is BST. (for case z is left child of v replace $\leq$ by $\geq$)



CASE z has two children:
Find y in $T(r)$ with min $y.key$ and such that y has no left child $\Rightarrow$ y has 0 or 1 child
[latter needed if $y'.key = y.key$ for some y']

Delete y from T [see cases above]
Replace z by y [$y = r$ is possible] and we get tree T'

By choice of y: $\forall w \in T(r) : w.key \leq y.key$

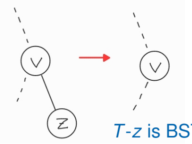
Delete vertex w means remove w from $V(T)$ and all edges from $E(T)$ that contain w !! [Example Board]

Tree-Delete(T, z) yields a BST

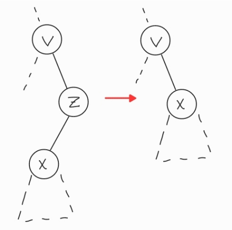
Part 3: Binary Search Trees (delete)

`Tree-Delete`(T, z)

CASE z has no child:
just delete z

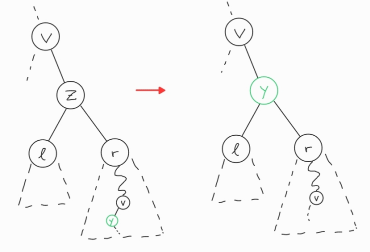


T-z is BST



CASE z has one child:
Delete z and make child x of z ..
.. the right child of $parent(z) = v$ in case z is right child of v
.. the left child of $parent(z) = v$ in case z is left child of v
to get tree T'

Case: z is right child of v:
T is $BST \Rightarrow \forall$ w in $T(z)$: $v.key \leq w.key$
$\Rightarrow \forall$ w in $T(x)$: $v.key \leq w.key$
$\Rightarrow \forall$ w in $T'(x)$: $v.key \leq w.key$
and other key "posititions" unchanged in T'
$\Rightarrow$ T' is BST. (for case z is left child of v replace $\leq$ by $\geq$)



CASE z has two children:
Find y in $T(r)$ with min $y.key$ and such that y has no left child $\Rightarrow$ y has 0 or 1 child
[latter needed if $y'.key = y.key$ for some y']

Delete y from T [see cases above]
Replace z by y [$y = r$ is possible] and we get tree T'

By choice of y: $\forall w \in T(r) : w.key \leq y.key$
Since T is BST: $\forall w' \in T(\ell) : w'.key \leq z.key \leq y.key$

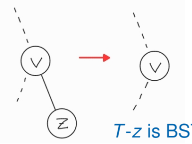
Delete vertex w means remove w from $V(T)$ and all edges from $E(T)$ that contain w !! [Example Board]

`Tree-Delete`(T, z) yields a BST

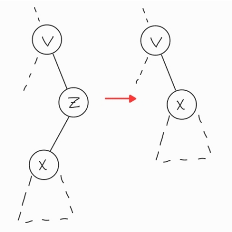
Part 3: Binary Search Trees (delete)

`Tree-Delete`(T, z)

CASE z has no child:
just delete z

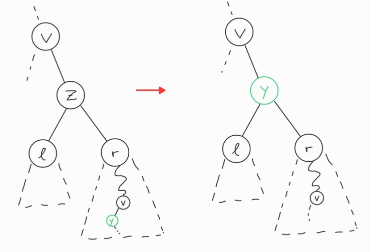


T-z is BST



CASE z has one child:
Delete z and make child x of z ..
.. the right child of $parent(z) = v$ in case z is right child of v
.. the left child of $parent(z) = v$ in case z is left child of v
to get tree T'

Case: z is right child of v:
T is $BST \Rightarrow \forall\ w$ in $T(z)$: $v.key \leq w.key$
$\Rightarrow \forall\ w$ in $T(x)$: $v.key \leq w.key$
$\Rightarrow \forall\ w$ in $T'(x)$: $v.key \leq w.key$
and other key "posititions" unchanged in T'
$\Rightarrow$ T' is BST. (for case z is left child of v replace $\leq$ by $\geq$)



CASE z has two children:
Find y in $T(r)$ with min $y.key$ and such that y has no left child $\Rightarrow$ y has 0 or 1 child
[latter needed if $y'.key = y.key$ for some y']
Delete y from T [see cases above]
Replace z by y [$y = r$ is possible] and we get tree T'

By choice of y: $\forall w \in T(r) : w.key \leq y.key$
Since T is BST: $\forall w' \in T(\ell) : w'.key \leq z.key \leq y.key$
$\Rightarrow \forall w \in T'(r) : y.key \leq w.key$ and $\forall w' \in T(\ell) : w'.key \leq y.key$

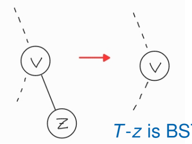
Delete vertex w means remove w from $V(T)$ and all edges from $E(T)$ that contain w !! [Example Board]

`Tree-Delete`(T, z) yields a BST

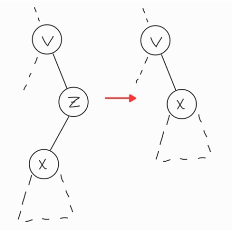
Part 3: Binary Search Trees (delete)

`Tree-Delete`(T, z)

CASE z has no child:
just delete z

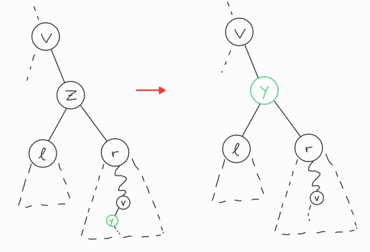


T-z is BST



CASE z has one child:
Delete z and make child x of z ..
.. the right child of $parent(z) = v$ in case z is right child of v
.. the left child of $parent(z) = v$ in case z is left child of v
to get tree T'

Case: z is right child of v:
T is $BST \Rightarrow \forall$ w in $T(z)$: $v.key \leq w.key$
$\Rightarrow \forall$ w in $T(x)$: $v.key \leq w.key$
$\Rightarrow \forall$ w in $T'(x)$: $v.key \leq w.key$
and other key "posititions" unchanged in T'
$\Rightarrow$ T' is BST. (for case z is left child of v replace $\leq$ by $\geq$)



CASE z has two children:
Find y in $T(r)$ with min $y.key$ and such that y has no left child $\Rightarrow$ y has 0 or 1 child
[latter needed if $y'.key = y.key$ for some y']
Delete y from T [see cases above]
Replace z by y [$y = r$ is possible] and we get tree T'

By choice of y: $\forall w \in T(r) : w.key \leq y.key$
Since T is BST: $\forall w' \in T(\ell) : w'.key \leq z.key \leq y.key$
$\Rightarrow \forall w \in T'(r) : y.key \leq w.key$ and $\forall w' \in T(\ell) : w'.key \leq y.key$
and other key "posititions" unchanged in T'

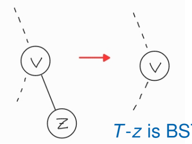
Delete vertex w means remove w from $V(T)$ and all edges from $E(T)$ that contain w !! [Example Board]

`Tree-Delete`(T, z) yields a BST

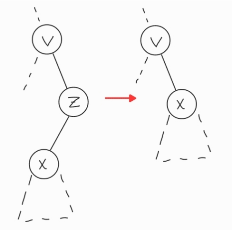
Part 3: Binary Search Trees (delete)

`Tree-Delete`(T, z)

CASE z has no child:
just delete z

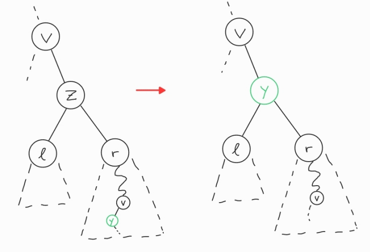


T-z is BST



CASE z has one child:
Delete z and make child x of z ..
.. the right child of $parent(z) = v$ in case z is right child of v
.. the left child of $parent(z) = v$ in case z is left child of v
to get tree T'

Case: z is right child of v:
T is $BST \Rightarrow \forall$ w in $T(z)$: $v.key \leq w.key$
$\Rightarrow \forall$ w in $T(x)$: $v.key \leq w.key$
$\Rightarrow \forall$ w in $T'(x)$: $v.key \leq w.key$
and other key "posititions" unchanged in T'
$\Rightarrow$ T' is BST. (for case z is left child of v replace $\leq$ by $\geq$)



CASE z has two children:
Find y in $T(r)$ with min $y.key$ and such that y has no left child $\Rightarrow$ y has 0 or 1 child
[latter needed if $y'.key = y.key$ for some y']
Delete y from T [see cases above]
Replace z by y [$y = r$ is possible] and we get tree T'

By choice of y: $\forall w \in T(r) : w.key \leq y.key$
Since T is BST: $\forall w' \in T(\ell) : w'.key \leq z.key \leq y.key$
$\Rightarrow \forall w \in T'(r) : y.key \leq w.key$ and $\forall w' \in T(\ell) : w'.key \leq y.key$
and other key "posititions" unchanged in T'
$\Rightarrow$ T' is BST

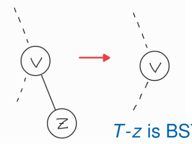
Delete vertex w means remove w from $V(T)$ and all edges from $E(T)$ that contain w !! [Example Board]

`Tree-Delete`(T, z) yields a BST

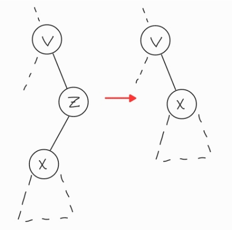
Part 3: Binary Search Trees (delete)

`Tree-Delete`(T, z)

CASE z has no child:
just delete z

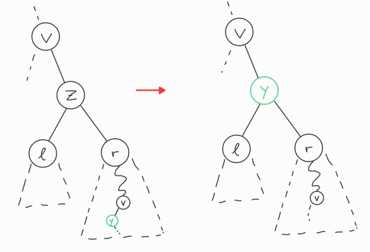


T-z is BST



CASE z has one child:
Delete z and make child x of z ..
.. the right child of $parent(z) = v$ in case z is right child of v
.. the left child of $parent(z) = v$ in case z is left child of v
to get tree T'

Case: z is right child of v:
T is $BST \Rightarrow \forall$ w in $T(z)$: $v.key \leq w.key$
$\Rightarrow \forall$ w in $T(x)$: $v.key \leq w.key$
$\Rightarrow \forall$ w in $T'(x)$: $v.key \leq w.key$
and other key "posititions" unchanged in T'
$\Rightarrow$ T' is BST. (for case z is left child of v replace $\leq$ by $\geq$)



CASE z has two children:
Find y in $T(r)$ with min $y.key$ and such that y has no left child $\Rightarrow$ y has 0 or 1 child
[latter needed if $y'.key = y.key$ for some y']

Delete y from T [see cases above]
Replace z by y [$y = r$ is possible] and we get tree T'

By choice of y: $\forall w \in T(r) : w.key \leq y.key$
Since T is BST: $\forall w' \in T(\ell) : w'.key \leq z.key \leq y.key$
$\Rightarrow \forall w \in T'(r) : y.key \leq w.key$ and $\forall w' \in T(\ell) : w'.key \leq y.key$
and other key "posititions" unchanged in T'
$\Rightarrow$ T' is BST

Delete vertex w means remove w from $V(T)$ and all edges from $E(T)$ that contain w !! [Example Board]

`Tree-Delete`(T, z) yields a BST and can be implemented to run in $O(h)$ time where h = height of tree [exercise]

To summarize at this point:

queries as search, find_minimum, find_maximum as well as insertion, deletion can be done in $O(h)$ time in binary search trees where h = height of tree.

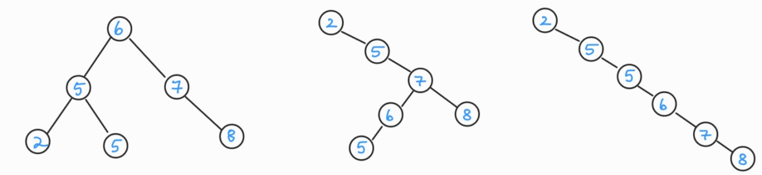
Problem: $O(h) = O(n)$ where n =number of vertices [can we control the height?]

We consider now AVL trees and Red-Black trees that are one of *many* search-tree schemes that are "balanced" in order to guarantee that basic dynamic-set operations take $O(\log n)$ time in the worst case.

To summarize at this point:

queries as search, find_minimum, find_maximum as well as insertion, deletion can be done in $O(h)$ time in binary search trees where h = height of tree.

Problem: $O(h) = O(n)$ where n =number of vertices [can we control the height?]

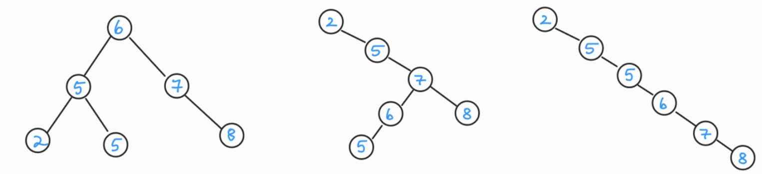


We consider now AVL trees and Red-Black trees that are one of *many* search-tree schemes that are “balanced” in order to guarantee that basic dynamic-set operations take $O(\log n)$ time in the worst case.

To summarize at this point:

queries as search, find_minimum, find_maximum as well as insertion, deletion can be done in $O(h)$ time in binary search trees where h = height of tree.

Problem: $O(h) = O(n)$ where n =number of vertices [can we control the height?]



We consider now AVL trees and Red-Black trees that are one of *many* search-tree schemes that are “balanced” in order to guarantee that basic dynamic-set operations take $O(\log n)$ time in the worst case.

Part 3: AVL Trees

Recall

height of x in T: $h(x) :=$ #edges along longest simple path from x to a leaf $l \preceq_T x$

height of T: $h(T) = h(\rho_T)$, i.e., height of root ρ_T of T

We define the height of an empty tree as $h(\emptyset) = -1$.

Define the **balance factor of** x in a binary tree T as $BF(x) = h(T(x.right)) - h(T(x.left))$.

A binary tree is **balanced** if $BF(x) \in \{1, 0, -1\}$ for all nodes x in T.

A balanced BST T is called **AVL tree**.

Part 3: AVL Trees

Recall

height of x in T: $h(x) :=$ #edges along longest simple path from x to a leaf $l \preceq_T x$

height of T: $h(T) = h(\rho_T)$, i.e., height of root ρ_T of T

We define the height of an empty tree as $h(\emptyset) = -1$.

Define the **balance factor of** x in a binary tree T as $BF(x) = h(T(x.right)) - h(T(x.left))$.

A binary tree is **balanced** if $BF(x) \in \{1, 0, -1\}$ for all nodes x in T.

A balanced BST T is called **AVL tree**.

Part 3: AVL Trees

Recall

height of x in T: $h(x) :=$ #edges along longest simple path from x to a leaf $l \preceq_T x$

height of T: $h(T) = h(\rho_T)$, i.e., height of root ρ_T of T

We define the height of an empty tree as $h(\emptyset) = -1$.

Define the **balance factor of** x in a binary tree T as $BF(x) = h(T(x.right)) - h(T(x.left))$.

A binary tree is **balanced** if $BF(x) \in \{1, 0, -1\}$ for all nodes x in T.

A balanced BST T is called **AVL tree**.

Part 3: AVL Trees

Recall

height of x in T: $h(x) :=$ #edges along longest simple path from x to a leaf $l \preceq_T x$

height of T: $h(T) = h(\rho_T)$, i.e., height of root ρ_T of T

We define the height of an empty tree as $h(\emptyset) = -1$.

Define the **balance factor of** x in a binary tree T as $BF(x) = h(T(x.right)) - h(T(x.left))$.

A binary tree is **balanced** if $BF(x) \in \{1, 0, -1\}$ for all nodes x in T.

A balanced BST T is called **AVL tree**.

Part 3: AVL Trees

Recall

height of x in T: $h(x) :=$ #edges along longest simple path from x to a leaf $l \preceq_T x$

height of T: $h(T) = h(\rho_T)$, i.e., height of root ρ_T of T

We define the height of an empty tree as $h(\emptyset) = -1$.

AVL tree = binary search tree (BST) in which the height of the two subtree rooted at children of any vertex differ by at most 1.

Define the **balance factor of** x in a binary tree T as $BF(x) = h(T(x.right)) - h(T(x.left))$.

A binary tree is **balanced** if $BF(x) \in \{1, 0, -1\}$ for all nodes x in T.

A balanced BST T is called **AVL tree**.

Part 3: AVL Trees

Recall

height of x in T: $h(x) :=$ #edges along longest simple path from x to a leaf $l \preceq_T x$

height of T: $h(T) = h(\rho_T)$, i.e., height of root ρ_T of T

We define the height of an empty tree as $h(\emptyset) = -1$.

AVL tree = binary search tree (BST) in which the height of the two subtree rooted at children of any vertex differ by at most 1.

More formal:

Define the **balance factor of x** in a binary tree T as $BF(x) = h(T(x.right)) - h(T(x.left))$.

A binary tree is **balanced** if $BF(x) \in \{1, 0, -1\}$ for all nodes x in T.

A balanced BST T is called **AVL tree**.

Part 3: AVL Trees

Recall

height of x in T: $h(x) :=$ #edges along longest simple path from x to a leaf $l \preceq_T x$

height of T: $h(T) = h(\rho_T)$, i.e., height of root ρ_T of T

We define the height of an empty tree as $h(\emptyset) = -1$.

AVL tree = binary search tree (BST) in which the height of the two subtree rooted at children of any vertex differ by at most 1.

More formal:

Define the **balance factor of** x in a binary tree T as $BF(x) = h(T(x.right)) - h(T(x.left))$.

A binary tree is **balanced** if $BF(x) \in \{1, 0, -1\}$ for all nodes x in T.

A balanced BST T is called **AVL tree**.

Part 3: AVL Trees

Recall

height of x in T: $h(x) :=$ #edges along longest simple path from x to a leaf $l \preceq_T x$

height of T: $h(T) = h(\rho_T)$, i.e., height of root ρ_T of T

We define the height of an empty tree as $h(\emptyset) = -1$.

AVL tree = binary search tree (BST) in which the height of the two subtree rooted at children of any vertex differ by at most 1.

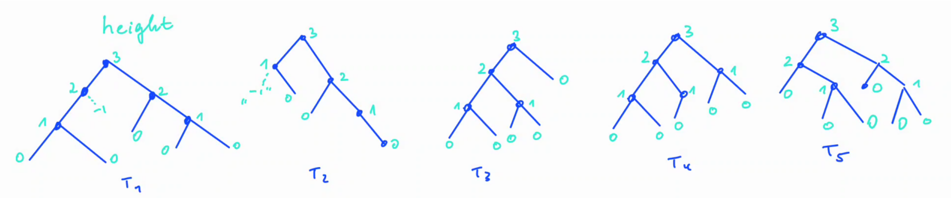
More formal:

Define the **balance factor of** x in a binary tree T as $BF(x) = h(T(x.right)) - h(T(x.left))$.

A binary tree is **balanced** if $BF(x) \in \{1, 0, -1\}$ for all nodes x in T.

A balanced BST T is called **AVL tree**.

Which of the trees is balanced?



Part 3: AVL Trees

Recall

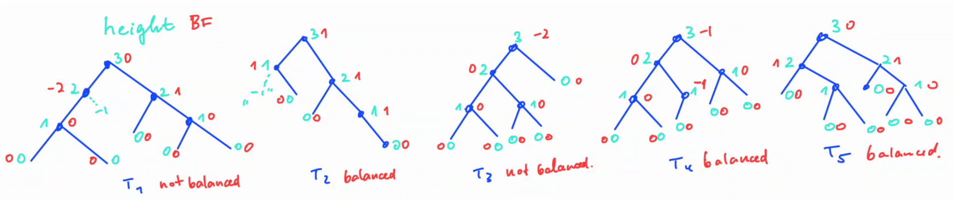
- height of x in T: $h(x) :=$ #edges along longest simple path from x to a leaf $l \preceq_T x$
- height of T: $h(T) = h(\rho_T)$, i.e., height of root ρ_T of T
- We define the height of an empty tree as $h(\emptyset) = -1$.

AVL tree = binary search tree (BST) in which the height of the two subtree rooted at children of any vertex differ by at most 1.

More formal:

- Define the **balance factor of** x in a binary tree T as $BF(x) = h(T(x.right)) - h(T(x.left))$.
- A binary tree is **balanced** if $BF(x) \in \{1, 0, -1\}$ for all nodes x in T.
- A balanced BST T is called **AVL tree**.

Which of the tree is balanced?



Part 3: AVL Trees

Recall

height of x in T: $h(x) :=$ #edges along longest simple path from x to a leaf $l \preceq_T x$

height of T: $h(T) = h(\rho_T)$, i.e., height of root ρ_T of T

We define the height of an empty tree as $h(\emptyset) = -1$.

AVL tree = binary search tree (BST) in which the height of the two subtree rooted at children of any vertex differ by at most 1.

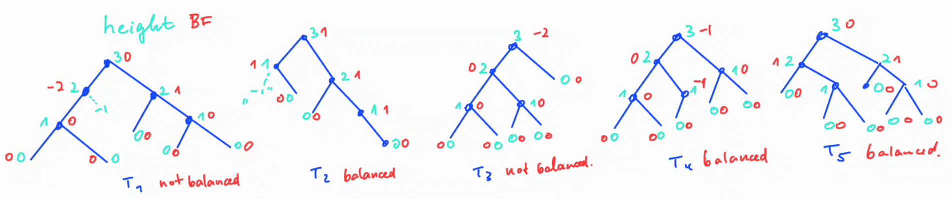
More formal:

Define the **balance factor of x** in a binary tree T as $BF(x) = h(T(x.right)) - h(T(x.left))$.

A binary tree is **balanced** if $BF(x) \in \{1, 0, -1\}$ for all nodes x in T.

A balanced BST T is called **AVL tree**.

Which of the tree is balanced?



Part 3: AVL Trees

Recall

height of x in T: $h(x) :=$ #edges along longest simple path from x to a leaf $l \preceq_T x$

height of T: $h(T) = h(\rho_T)$, i.e., height of root ρ_T of T

We define the height of an empty tree as $h(\emptyset) = -1$.

AVL tree = binary search tree (BST) in which the height of the two subtree rooted at children of any vertex differ by at most 1.

More formal:

Define the **balance factor of x** in a binary tree T as $BF(x) = h(T(x.right)) - h(T(x.left))$.

A binary tree is **balanced** if $BF(x) \in \{1, 0, -1\}$ for all nodes x in T.

A balanced BST T is called **AVL tree**.

Part 3: AVL Trees

T is **AVL tree** = if T is a balanced BST, i.e., $BF(x) \in \{1, 0, -1\}$ for all nodes x in T with $BF(x) = h(T(x.\mathit{right})) - h(T(x.\mathit{left}))$.

[named after inventors **A**delson-**V**elsky and **L**andis]

Part 3: AVL Trees

T is **AVL tree** = if T is a balanced BST, i.e., $BF(x) \in \{1, 0, -1\}$ for all nodes x in T with $BF(x) = h(T(x.right)) - h(T(x.left))$.
[named after inventors **A**delson-**V**elsky and **L**andis]

Lemma. *Any AVL tree with n nodes has height $O(\log n)$.*

Part 3: AVL Trees

T is **AVL tree** = if T is a balanced BST, i.e., $BF(x) \in \{1, 0, -1\}$ for all nodes x in T with $BF(x) = h(T(x.right)) - h(T(x.left))$.
[named after inventors **A**delson-**V**elsky and **L**andis]

Lemma. *Any AVL tree with n nodes has height $O(\log n)$.*

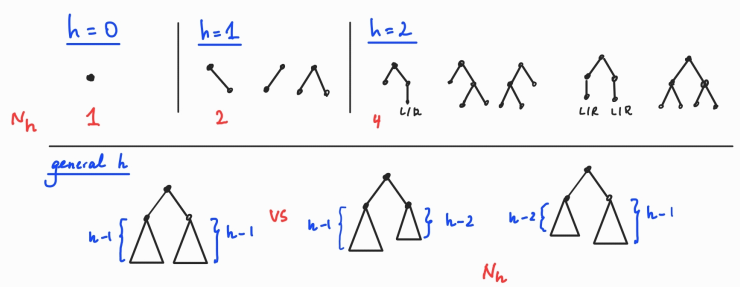
Proof: Let N_h = min number of vertices in AVL tree of height h

Part 3: AVL Trees

T is **AVL tree** = if T is a balanced BST, i.e., $BF(x) \in \{1, 0, -1\}$ for all nodes x in T with $BF(x) = h(T(x.right)) - h(T(x.left))$.
[named after inventors **A**delson-**V**elsky and **L**andis]

Lemma. *Any AVL tree with n nodes has height $O(\log n)$.*

Proof: Let N_h = min number of vertices in AVL tree of height h

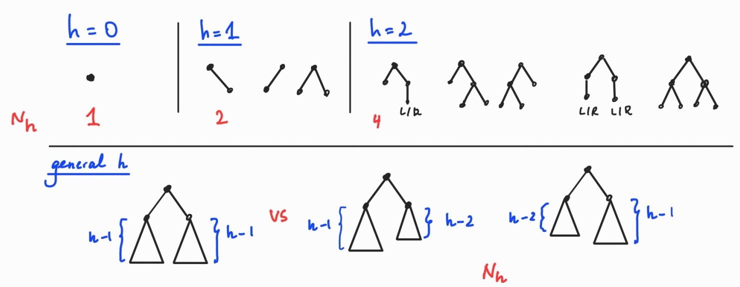


Part 3: AVL Trees

T is **AVL tree** = if T is a balanced BST, i.e., $BF(x) \in \{1, 0, -1\}$ for all nodes x in T with $BF(x) = h(T(x.right)) - h(T(x.left))$.
[named after inventors **A**delson-**V**elsky and **L**andis]

Lemma. *Any AVL tree with n nodes has height $O(\log n)$.*

Proof: Let N_h = min number of vertices in AVL tree of height h $\implies$ $N_h = 1 + N_{h-1} + N_{h-2}$



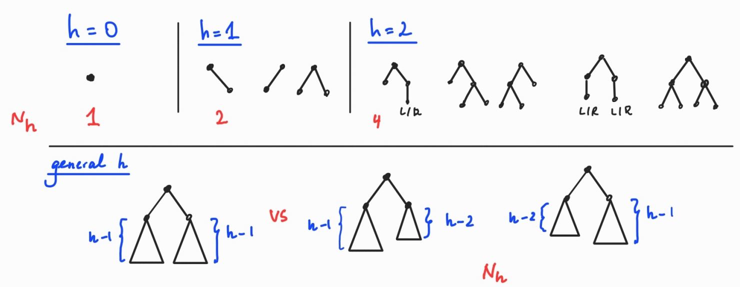
Part 3: AVL Trees

T is **AVL tree** = if T is a balanced BST, i.e., $BF(x) \in \{1, 0, -1\}$ for all nodes x in T with $BF(x) = h(T(x.right)) - h(T(x.left))$.
[named after inventors **A**delson-**V**elsky and **L**andis]

Lemma. *Any AVL tree with n nodes has height $O(\log n)$.*

Proof: Let N_h = min number of vertices in AVL tree of height h $\implies N_h = 1 + N_{h-1} + N_{h-2}$

We first show, by induction on h, that $N_h > 2^{\frac{h}{2}-1}$ for all $h \geq 0$



Part 3: AVL Trees

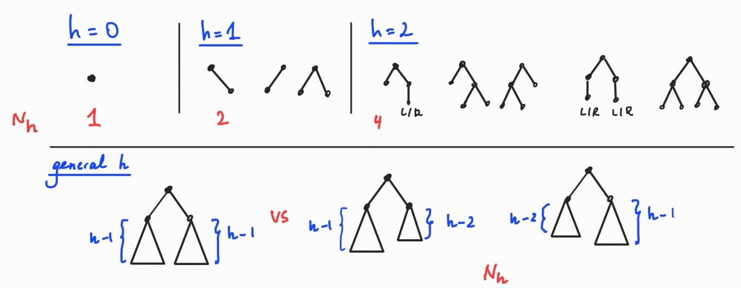
T is **AVL tree** = if T is a balanced BST, i.e., $BF(x) \in \{1, 0, -1\}$ for all nodes x in T with $BF(x) = h(T(x.right)) - h(T(x.left))$.
[named after inventors **A**delson-**V**elsky and **L**andis]

Lemma. *Any AVL tree with n nodes has height $O(\log n)$.*

Proof: Let N_h = min number of vertices in AVL tree of height h $\implies$ $N_h = 1 + N_{h-1} + N_{h-2}$

We first show, by induction on h, that $N_h > 2^{\frac{h}{2}-1}$ for all $h \geq 0$

Base cases: $h = 0 \implies N_h = 1 > 2^{\frac{0}{2}-1} = 0.5$ and $h = 1 \implies N_h = 2 > 2^{\frac{1}{2}-1} \simeq 0.71$



Part 3: AVL Trees

T is **AVL tree** = if T is a balanced BST, i.e., $BF(x) \in \{1, 0, -1\}$ for all nodes x in T with $BF(x) = h(T(x.right)) - h(T(x.left))$.
[named after inventors **A**delson-**V**elsky and **L**andis]

Lemma. *Any AVL tree with n nodes has height $O(\log n)$.*

Proof: Let N_h = min number of vertices in AVL tree of height h $\implies$ $N_h = 1 + N_{h-1} + N_{h-2}$

We first show, by induction on h, that $N_h > 2^{\frac{h}{2}-1}$ for all $h \geq 0$

Base cases: $h = 0 \implies N_h = 1 > 2^{\frac{0}{2}-1} = 0.5$ and $h = 1 \implies N_h = 2 > 2^{\frac{1}{2}-1} \simeq 0.71$

Assume now that $N_k > 2^{\frac{k}{2}-1}$ for all $k \in \{0, 1, \ldots, h-1\}$.

Let T be an AVL tree of height h with N_h vertices.

Hence, $N_h = 1 + N_{h-1} + N_{h-2} \geq 1 + 2N_{h-2} > 2N_{h-2} \overset{Ind.hyp.}{>} 2(2^{\frac{h-2}{2}-1}) = 2^{\frac{h-2}{2}} = 2^{\frac{h}{2}-1}$ (which completes induct.-proof)

Part 3: AVL Trees

T is **AVL tree** = if T is a balanced BST, i.e., $BF(x) \in \{1, 0, -1\}$ for all nodes x in T with $BF(x) = h(T(x.right)) - h(T(x.left))$.
[named after inventors **A**delson-**V**elsky and **L**andis]

Lemma. *Any AVL tree with n nodes has height $O(\log n)$.*

Proof: Let N_h = min number of vertices in AVL tree of height $h \implies N_h = 1 + N_{h-1} + N_{h-2}$

We first show, by induction on h, that $N_h > 2^{\frac{h}{2}-1}$ for all $h \geq 0$

Base cases: $h = 0 \implies N_h = 1 > 2^{\frac{0}{2}-1} = 0.5$ and $h = 1 \implies N_h = 2 > 2^{\frac{1}{2}-1} \simeq 0.71$

Assume now that $N_k > 2^{\frac{k}{2}-1}$ for all $k \in \{0, 1, \ldots, h-1\}$.

Let T be an AVL tree of height h with N_h vertices.

Hence, $N_h = 1 + N_{h-1} + N_{h-2} \geq 1 + 2N_{h-2} > 2N_{h-2} \overset{Ind.hyp.}{>} 2(2^{\frac{h-2}{2}-1}) = 2^{\frac{h-2}{2}} = 2^{\frac{h}{2}-1}$ (which completes induct.-proof)

Part 3: AVL Trees

T is **AVL tree** = if T is a balanced BST, i.e., $BF(x) \in \{1, 0, -1\}$ for all nodes x in T with $BF(x) = h(T(x.right)) - h(T(x.left))$.
[named after inventors **A**delson-**V**elsky and **L**andis]

Lemma. *Any AVL tree with n nodes has height $O(\log n)$.*

Proof: Let N_h = min number of vertices in AVL tree of height h $\implies$ $N_h = 1 + N_{h-1} + N_{h-2}$

We first show, by induction on h, that $N_h > 2^{\frac{h}{2}-1}$ for all $h \geq 0$

Base cases: $h = 0 \implies N_h = 1 > 2^{\frac{0}{2}-1} = 0.5$ and $h = 1 \implies N_h = 2 > 2^{\frac{1}{2}-1} \simeq 0.71$

Assume now that $N_k > 2^{\frac{k}{2}-1}$ for all $k \in \{0, 1, \ldots, h-1\}$.

Let T be an AVL tree of height h with N_h vertices.

Hence, $N_h = 1 + N_{h-1} + N_{h-2} \geq 1 + 2N_{h-2} > 2N_{h-2} \overset{Ind.hyp.}{>} 2(2^{\frac{h-2}{2}-1}) = 2^{\frac{h-2}{2}} = 2^{\frac{h}{2}-1}$ (which completes induct.-proof)

Part 3: AVL Trees

T is **AVL tree** = if T is a balanced BST, i.e., $BF(x) \in \{1, 0, -1\}$ for all nodes x in T with $BF(x) = h(T(x.right)) - h(T(x.left))$.
[named after inventors **A**delson-**V**elsky and **L**andis]

Lemma. *Any AVL tree with n nodes has height $O(\log n)$.*

Proof: Let N_h = min number of vertices in AVL tree of height h $\implies N_h = 1 + N_{h-1} + N_{h-2}$

We first show, by induction on h, that $N_h > 2^{\frac{h}{2}-1}$ for all $h \geq 0$

Base cases: $h = 0 \implies N_h = 1 > 2^{\frac{0}{2}-1} = 0.5$ and $h = 1 \implies N_h = 2 > 2^{\frac{1}{2}-1} \simeq 0.71$

Assume now that $N_k > 2^{\frac{k}{2}-1}$ for all $k \in \{0, 1, \ldots, h-1\}$.

Let T be an AVL tree of height h with N_h vertices.

Hence, $N_h = 1 + N_{h-1} + N_{h-2} \geq 1 + 2N_{h-2} > 2N_{h-2} \overset{Ind.hyp.}{>} 2(2^{\frac{h-2}{2}-1}) = 2^{\frac{h-2}{2}} = 2^{\frac{h}{2}-1}$ (which completes induct.-proof)

$\implies N_h > 2^{\frac{h}{2}-1}$

Part 3: AVL Trees

T is **AVL tree** = if T is a balanced BST, i.e., $BF(x) \in \{1, 0, -1\}$ for all nodes x in T with $BF(x) = h(T(x.right)) - h(T(x.left))$.
[named after inventors **A**delson-**V**elsky and **L**andis]

Lemma. *Any AVL tree with n nodes has height $O(\log n)$.*

Proof: Let N_h = min number of vertices in AVL tree of height h $\implies$ $N_h = 1 + N_{h-1} + N_{h-2}$

We first show, by induction on h, that $N_h > 2^{\frac{h}{2}-1}$ for all $h \geq 0$

Base cases: $h = 0 \implies N_h = 1 > 2^{\frac{0}{2}-1} = 0.5$ and $h = 1 \implies N_h = 2 > 2^{\frac{1}{2}-1} \simeq 0.71$

Assume now that $N_k > 2^{\frac{k}{2}-1}$ for all $k \in \{0, 1, \ldots, h-1\}$.

Let T be an AVL tree of height h with N_h vertices.

Hence, $N_h = 1 + N_{h-1} + N_{h-2} \geq 1 + 2N_{h-2} > 2N_{h-2} \overset{Ind.hyp.}{>} 2(2^{\frac{h-2}{2}-1}) = 2^{\frac{h-2}{2}} = 2^{\frac{h}{2}-1}$ (which completes induct.-proof)

$\implies N_h > 2^{\frac{h}{2}-1} \iff \log_2(N_h) > \frac{h}{2} - 1$

Part 3: AVL Trees

T is **AVL tree** = if T is a balanced BST, i.e., $BF(x) \in \{1, 0, -1\}$ for all nodes x in T with $BF(x) = h(T(x.right)) - h(T(x.left))$.
[named after inventors **A**delson-**V**elsky and **L**andis]

Lemma. *Any AVL tree with n nodes has height $O(\log n)$.*

Proof: Let N_h = min number of vertices in AVL tree of height $h \implies N_h = 1 + N_{h-1} + N_{h-2}$

We first show, by induction on h, that $N_h > 2^{\frac{h}{2}-1}$ for all $h \geq 0$

Base cases: $h = 0 \implies N_h = 1 > 2^{\frac{0}{2}-1} = 0.5$ and $h = 1 \implies N_h = 2 > 2^{\frac{1}{2}-1} \simeq 0.71$

Assume now that $N_k > 2^{\frac{k}{2}-1}$ for all $k \in \{0, 1, \ldots, h-1\}$.

Let T be an AVL tree of height h with N_h vertices.

Hence, $N_h = 1 + N_{h-1} + N_{h-2} \geq 1 + 2N_{h-2} > 2N_{h-2} \overset{Ind.hyp.}{>} 2(2^{\frac{h-2}{2}-1}) = 2^{\frac{h-2}{2}} = 2^{\frac{h}{2}-1}$ (which completes induct.-proof)

$\implies N_h > 2^{\frac{h}{2}-1} \iff \log_2(N_h) > \frac{h}{2} - 1 \iff 2\log_2(N_h) + 2 > h$

Part 3: AVL Trees

T is **AVL tree** = if T is a balanced BST, i.e., $BF(x) \in \{1, 0, -1\}$ for all nodes x in T with $BF(x) = h(T(x.right)) - h(T(x.left))$.
[named after inventors **A**delson-**V**elsky and **L**andis]

Lemma. *Any AVL tree with n nodes has height $O(\log n)$.*

Proof: Let N_h = min number of vertices in AVL tree of height h $\implies$ $N_h = 1 + N_{h-1} + N_{h-2}$

We first show, by induction on h, that $N_h > 2^{\frac{h}{2}-1}$ for all $h \geq 0$

Base cases: $h = 0 \implies N_h = 1 > 2^{\frac{0}{2}-1} = 0.5$ and $h = 1 \implies N_h = 2 > 2^{\frac{1}{2}-1} \simeq 0.71$

Assume now that $N_k > 2^{\frac{k}{2}-1}$ for all $k \in \{0, 1, \ldots, h-1\}$.

Let T be an AVL tree of height h with N_h vertices.

Hence, $N_h = 1 + N_{h-1} + N_{h-2} \geq 1 + 2N_{h-2} > 2N_{h-2} \overset{Ind.hyp.}{>} 2(2^{\frac{h-2}{2}-1}) = 2^{\frac{h-2}{2}} = 2^{\frac{h}{2}-1}$ (which completes induct.-proof)

$\implies N_h > 2^{\frac{h}{2}-1} \iff \log_2(N_h) > \frac{h}{2} - 1 \iff 2\log_2(N_h) + 2 > h$

For general AVL tree T with height h it holds that $|V(T)| \geq N_h$

Part 3: AVL Trees

T is **AVL tree** = if T is a balanced BST, i.e., $BF(x) \in \{1, 0, -1\}$ for all nodes x in T with $BF(x) = h(T(x.right)) - h(T(x.left))$.
[named after inventors **A**delson-**V**elsky and **L**andis]

Lemma. *Any AVL tree with n nodes has height $O(\log n)$.*

Proof: Let N_h = min number of vertices in AVL tree of height $h \implies N_h = 1 + N_{h-1} + N_{h-2}$

We first show, by induction on h, that $N_h > 2^{\frac{h}{2}-1}$ for all $h \geq 0$

Base cases: $h = 0 \implies N_h = 1 > 2^{\frac{0}{2}-1} = 0.5$ and $h = 1 \implies N_h = 2 > 2^{\frac{1}{2}-1} \simeq 0.71$

Assume now that $N_k > 2^{\frac{k}{2}-1}$ for all $k \in \{0, 1, \ldots, h-1\}$.

Let T be an AVL tree of height h with N_h vertices.

Hence, $N_h = 1 + N_{h-1} + N_{h-2} \geq 1 + 2N_{h-2} > 2N_{h-2} \overset{Ind.hyp.}{>} 2(2^{\frac{h-2}{2}-1}) = 2^{\frac{h-2}{2}} = 2^{\frac{h}{2}-1}$ (which completes induct.-proof)

$\implies N_h > 2^{\frac{h}{2}-1} \iff \log_2(N_h) > \frac{h}{2} - 1 \iff 2\log_2(N_h) + 2 > h$

For general AVL tree T with height h it holds that $|V(T)| \geq N_h$

$\implies h < 2\log_2(N_h) + 2 \leq 2\log_2(|V(T)|) + 2$ and thus, $h \in O(\log_2(|V(T)|))$ $\square$

Part 3: AVL Trees

T is **AVL tree** = if T is a balanced BST, i.e., $BF(x) \in \{1, 0, -1\}$ for all nodes x in T with $BF(x) = h(T(x.right)) - h(T(x.left))$.
[named after inventors **A**delson-**V**elsky and **L**andis]

Lemma. *Any AVL tree with n nodes has height* $O(\log n)$.

Part 3: AVL Trees

T is **AVL tree** = if T is a balanced BST, i.e., $BF(x) \in \{1, 0, -1\}$ for all nodes x in T with $BF(x) = h(T(x.right)) - h(T(x.left))$.
[named after inventors **A**delson-**V**elsky and **L**andis]

Lemma. *Any AVL tree with n nodes has height* $O(\log n)$.

Thus, queries as search, find_minimum, find_maximum as well as insert and delete can be done in $h \in O(\log n)$ time in AVL trees.

Part 3: AVL Trees

T is **AVL tree** = if T is a balanced BST, i.e., $BF(x) \in \{1, 0, -1\}$ for all nodes x in T with $BF(x) = h(T(x.right)) - h(T(x.left))$.
[named after inventors **A**delson-**V**elsky and **L**andis]

Lemma. *Any AVL tree with n nodes has height $O(\log n)$.*

Thus, queries as search, find_minimum, find_maximum as well as insert and delete can be done in $h \in O(\log n)$ time in AVL trees.

However, after a single use of the operations insert and delete it is not ensured that the resulting tree is still balanced, that is, the BST is possibly not an AVL tree.

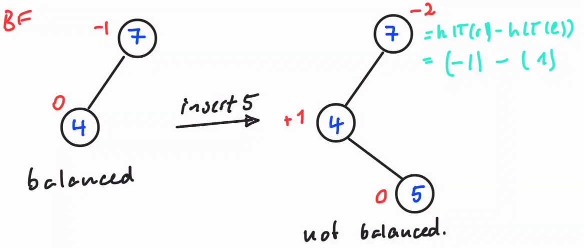
Part 3: AVL Trees

T is **AVL tree** = if T is a balanced BST, i.e., $BF(x) \in \{1, 0, -1\}$ for all nodes x in T with $BF(x) = h(T(x.right)) - h(T(x.left))$.
[named after inventors **A**delson-**V**elsky and **L**andis]

Lemma. *Any AVL tree with n nodes has height $O(\log n)$.*

Thus, queries as search, find_minimum, find_maximum as well as insert and delete can be done in $h \in O(\log n)$ time in AVL trees.

However, after a single use of the operations insert and delete it is not ensured that the resulting tree is still balanced, that is, the BST is possibly not an AVL tree.



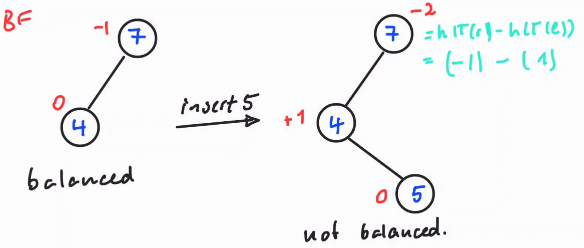
Part 3: AVL Trees

T is **AVL tree** = if T is a balanced BST, i.e., $BF(x) \in \{1, 0, -1\}$ for all nodes x in T with $BF(x) = h(T(x.right)) - h(T(x.left))$.
[named after inventors **A**delson-**V**elsky and **L**andis]

Lemma. *Any AVL tree with n nodes has height $O(\log n)$.*

Thus, queries as search, find_minimum, find_maximum as well as insert and delete can be done in $h \in O(\log n)$ time in AVL trees.

However, after a single use of the operations insert and delete it is not ensured that the resulting tree is still balanced, that is, the BST is possibly not an AVL tree.



Hence, it is not ensured that the latter operations still run in $O(\log n)$ time $\Longrightarrow$ must correct trees to obtain AVL trees

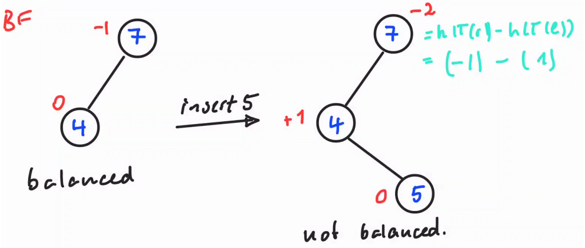
Part 3: AVL Trees

T is **AVL tree** = if T is a balanced BST, i.e., $BF(x) \in \{1, 0, -1\}$ for all nodes x in T with $BF(x) = h(T(x.right)) - h(T(x.left))$.
[named after inventors **A**delson-**V**elsky and **L**andis]

Lemma. *Any AVL tree with n nodes has height $O(\log n)$.*

Thus, queries as search, find_minimum, find_maximum as well as insert and delete can be done in $h \in O(\log n)$ time in AVL trees.

However, after a single use of the operations insert and delete it is not ensured that the resulting tree is still balanced, that is, the BST is possibly not an AVL tree.



Hence, it is not ensured that the latter operations still run in $O(\log n)$ time $\Longrightarrow$ must correct trees to obtain AVL trees

These corrections should run in $O(\log n)$ time to ensure that the overall time complexity together with the operations as above remains in $O(\log n)$

Part 3: AVL Trees (rotation)

An important role for obtaining a AVL tree after insertion/deletion are **rotations**:

[Some Examples on Board]

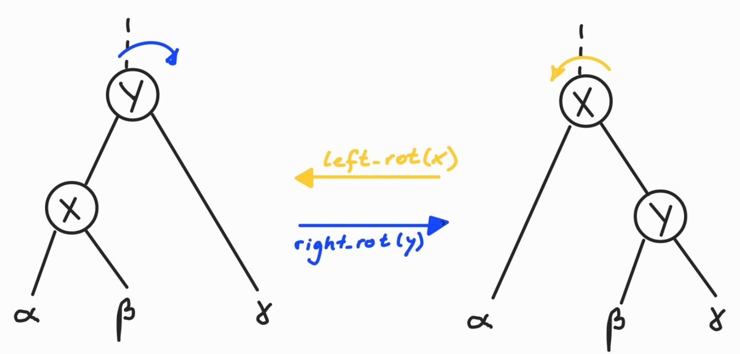
Rotations in BST preserve binary-search tree properies: "α.*keys*" $\leq x$.*key* $\leq$ "β.*keys*" $\leq y$.*key* $\leq$ "γ.*keys*"

Note, a rotation is just a "rearrangement" of a constant nr. of pointers and thus runs in $\Theta(1)$ time

[pseudocode = exercise (don't forget cases as *y* is right/left of parents_*y*, *x* or *y* is root, ...)]

Part 3: AVL Trees (rotation)

An important role for obtaining a AVL tree after insertion/deletion are **rotations**:



[Some Examples on Board]

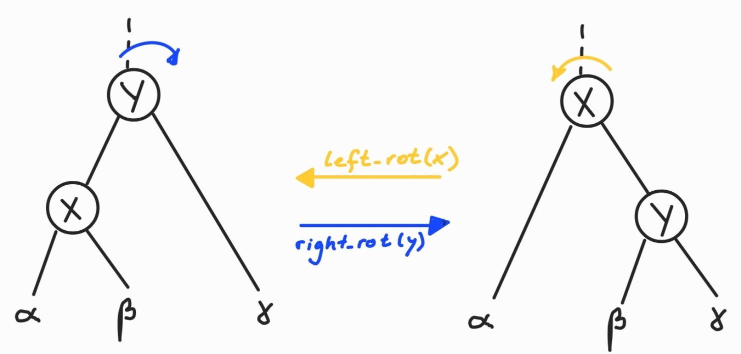
Rotations in BST preserve binary-search tree properies: "α.*keys*" $\leq x.key \leq$ "β.*keys*" $\leq y.key \leq$ "γ.*keys*"

Note, a rotation is just a "rearrangement" of a constant nr. of pointers and thus runs in $\Theta(1)$ time

[pseudocode = exercise (don't forget cases as y is right/left of parents_y, x or y is root, ...)]

Part 3: AVL Trees (rotation)

An important role for obtaining a AVL tree after insertion/deletion are **rotations**:



[Some Examples on Board]

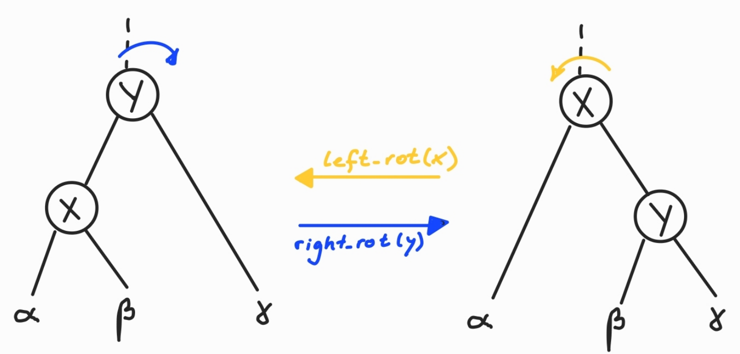
Rotations in BST preserve binary-search tree properies: "$\alpha.keys$" $\leq x.key \leq$ "$\beta.keys$" $\leq y.key \leq$ "$\gamma.keys$"

Note, a rotation is just a "rearrangement" of a constant nr. of pointers and thus runs in $\Theta(1)$ time

[pseudocode = exercise (don't forget cases as y is right/left of parents_y, x or y is root, ...)]

Part 3: AVL Trees (rotation)

An important role for obtaining a AVL tree after insertion/deletion are **rotations**:



[Some Examples on Board]

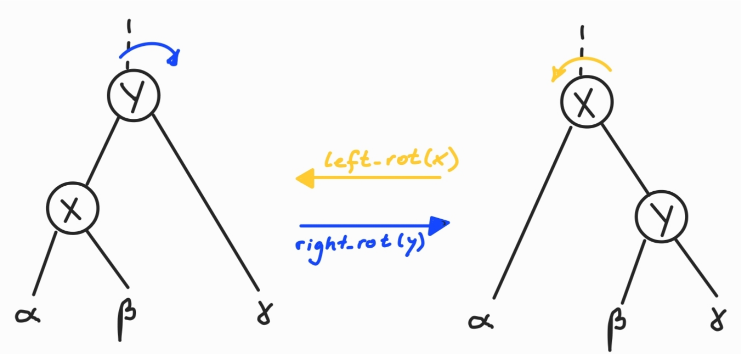
Rotations in BST preserve binary-search tree properies: "$\alpha.keys$" $\leq x.key \leq$ "$\beta.keys$" $\leq y.key \leq$ "$\gamma.keys$"

Note, a rotation is just a "rearrangement" of a constant nr. of pointers and thus runs in $\Theta(1)$ time

[pseudocode = exercise (don't forget cases as y is right/left of parents_y, x or y is root, . . .)]

Part 3: AVL Trees (rotation)

An important role for obtaining a AVL tree after insertion/deletion are **rotations**:



[Some Examples on Board]

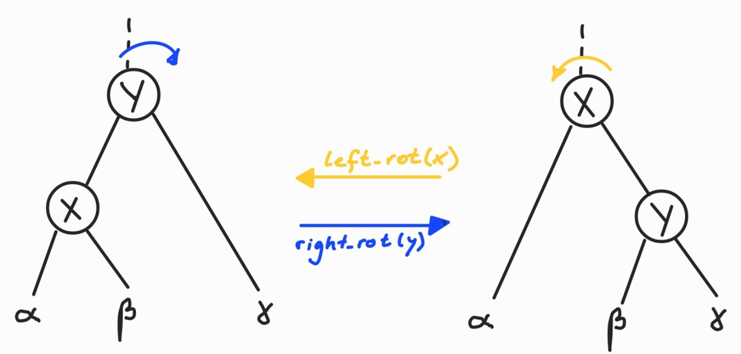
Rotations in BST preserve binary-search tree properies: "α.*keys*" $\leq$ x.*key* $\leq$ "β.*keys*" $\leq$ y.*key* $\leq$ "γ.*keys*"

Note, a rotation is just a "rearrangement" of a constant nr. of pointers and thus runs in $\Theta(1)$ time

[pseudocode = exercise (don't forget cases as y is right/left of parents_y, x or y is root, ...)]

Part 3: AVL Trees (rotation)

An important role for obtaining a AVL tree after insertion/deletion are **rotations**:



[Some Examples on Board]

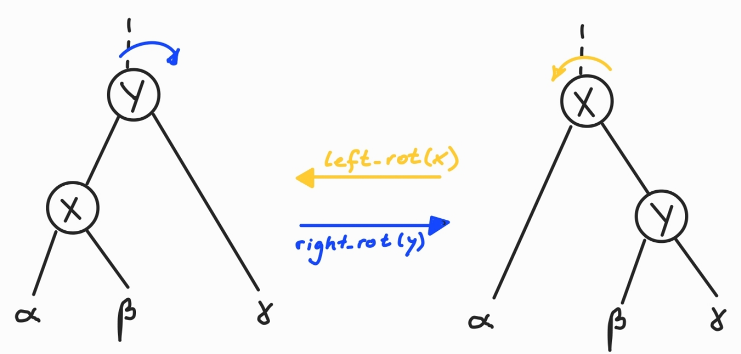
Rotations in BST preserve binary-search tree properies: "$\alpha.keys$" $\leq x.key \leq$ "$\beta.keys$" $\leq y.key \leq$ "$\gamma.keys$"

Note, a rotation is just a "rearrangement" of a constant nr. of pointers and thus runs in $\Theta(1)$ time

[pseudocode = exercise (don't forget cases as y is right/left of parents_y, x or y is root, ...)]

Part 3: AVL Trees (rotation)

An important role for obtaining a AVL tree after insertion/deletion are **rotations**:



[Some Examples on Board]

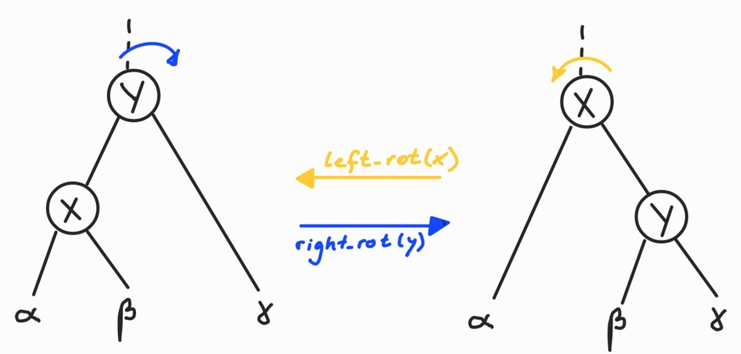
Rotations in BST preserve binary-search tree properies: "α.*keys*" $\leq$ x.*key* $\leq$ "β.*keys*" $\leq$ y.*key* $\leq$ "γ.*keys*"

Note, a rotation is just a "rearrangement" of a constant nr. of pointers and thus runs in $\Theta(1)$ time

[pseudocode = exercise (don't forget cases as y is right/left of parents_y, x or y is root, ...)]

Part 3: AVL Trees (rotation)

An important role for obtaining a AVL tree after insertion/deletion are **rotations**:



[Some Examples on Board]

Rotations in BST preserve binary-search tree properies: "α.*keys*" $\leq x.key \leq$ "β.*keys*" $\leq y.key \leq$ "γ.*keys*"

Note, a rotation is just a "rearrangement" of a constant nr. of pointers and thus runs in $\Theta(1)$ time

[pseudocode = exercise (don't forget cases as y is right/left of parents_y, x or y is root, ...)]

Part 3: AVL Trees (inserting x)

Inserting a vertex x to T is done as in case for BST $\implies$ we get BST $T + x$ which might be imbalanced (corrections!).

Corrections of tree $T + x$ are based on the following cases for parent p of x in $T + x$.

We denote with $BF_{T'}(v)$ the balance factor of v in tree T'

Since p is parent of x in $T + x$ it can have at most one child (since $T + x$ is binary search tree)

Part 3: AVL Trees (inserting x)

Inserting a vertex x to T is done as in case for BST $\implies$ we get BST $T + x$ which might be imbalanced (corrections!).

Corrections of tree $T + x$ are based on the following cases for parent p of x in $T + x$.

We denote with $BF_{T'}(v)$ the balance factor of v in tree T'

Since p is parent of x in $T + x$ it can have at most one child (since $T + x$ is binary search tree)

Part 3: AVL Trees (inserting x)

Inserting a vertex x to T is done as in case for BST $\implies$ we get BST $T + x$ which might be imbalanced (corrections!).

Corrections of tree $T + x$ are based on the following cases for parent p of x in $T + x$.

We denote with $BF_{T'}(v)$ the balance factor of v in tree T'

Since p is parent of x in $T + x$ it can have at most one child (since $T + x$ is binary search tree)

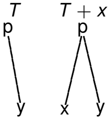
Part 3: AVL Trees (inserting x)

Inserting a vertex x to T is done as in case for BST $\implies$ we get BST $T + x$ which might be imbalanced (corrections!).
Corrections of tree $T + x$ are based on the following cases for parent p of x in $T + x$.
We denote with $BF_{T'}(v)$ the balance factor of v in tree T'
Since p is parent of x in $T + x$ it can have at most one child (since $T + x$ is <u>binary</u> search tree)

Part 3: AVL Trees (inserting x)

Inserting a vertex x to T is done as in case for BST $\implies$ we get BST $T+x$ which might be imbalanced (corrections!).
Corrections of tree $T+x$ are based on the following cases for parent p of x in $T+x$.
We denote with $BF_{T'}(v)$ the balance factor of v in tree T'
Since p is parent of x in $T+x$ it can have at most one child (since $T+x$ is binary search tree)

[1] p has right child y but no left child:

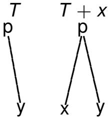


Part 3: AVL Trees (inserting x)

Inserting a vertex x to T is done as in case for BST $\implies$ we get BST $T+x$ which might be imbalanced (corrections!).
Corrections of tree $T+x$ are based on the following cases for parent p of x in $T+x$.
We denote with $BF_{T'}(v)$ the balance factor of v in tree T'
Since p is parent of x in $T+x$ it can have at most one child (since $T+x$ is binary search tree)

[1] p has right child y but no left child:

Since T is balanced, y must be a leaf and $BF_T(p) = 0 - (-1) = +1$



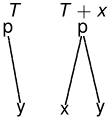
Part 3: AVL Trees (inserting x)

Inserting a vertex x to T is done as in case for BST $\implies$ we get BST $T + x$ which might be imbalanced (corrections!).
Corrections of tree $T + x$ are based on the following cases for parent p of x in $T + x$.
We denote with $BF_{T'}(v)$ the balance factor of v in tree T'
Since p is parent of x in $T + x$ it can have at most one child (since $T + x$ is binary search tree)

[1] p has right child y but no left child:

Since T is balanced, y must be a leaf and $BF_T(p) = 0 - (-1) = +1$

After inserting x: $BF_{T+x}(p) = 0$ and $BF_T(v) = BF_{T+x}(v)$ for all $v \in V(T)$
(since height of $T(v)$ remains unchanged for all $v \in V(T)$)



Part 3: AVL Trees (inserting x)

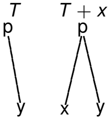
Inserting a vertex x to T is done as in case for BST $\implies$ we get BST $T + x$ which might be imbalanced (corrections!).
Corrections of tree $T + x$ are based on the following cases for parent p of x in $T + x$.
We denote with $BF_{T'}(v)$ the balance factor of v in tree T'
Since p is parent of x in $T + x$ it can have at most one child (since $T + x$ is binary search tree)

[1] p has right child y but no left child:

Since T is balanced, y must be a leaf and $BF_T(p) = 0 - (-1) = +1$

After inserting x: $BF_{T+x}(p) = 0$ and $BF_T(v) = BF_{T+x}(v)$ for all $v \in V(T)$
(since height of $T(v)$ remains unchanged for all $v \in V(T)$)

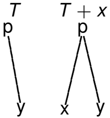
$\implies$ $T + x$ is an AVL tree. [no correction needed]



Part 3: AVL Trees (inserting x)

Inserting a vertex x to T is done as in case for BST $\implies$ we get BST $T + x$ which might be imbalanced (corrections!).
Corrections of tree $T + x$ are based on the following cases for parent p of x in $T + x$.
We denote with $BF_{T'}(v)$ the balance factor of v in tree T'
Since p is parent of x in $T + x$ it can have at most one child (since $T + x$ is binary search tree)

[1] p has right child y but no left child:

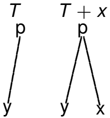


Since T is balanced, y must be a leaf and $BF_T(p) = 0 - (-1) = +1$

After inserting x: $BF_{T+x}(p) = 0$ and $BF_T(v) = BF_{T+x}(v)$ for all $v \in V(T)$
(since height of $T(v)$ remains unchanged for all $v \in V(T)$)

$\implies$ $T + x$ is an AVL tree. [no correction needed]

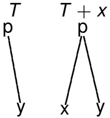
[2] p has left child y but no right child:



Part 3: AVL Trees (inserting x)

Inserting a vertex x to T is done as in case for BST $\implies$ we get BST $T + x$ which might be imbalanced (corrections!).
Corrections of tree $T + x$ are based on the following cases for parent p of x in $T + x$.
We denote with $BF_{T'}(v)$ the balance factor of v in tree T'
Since p is parent of x in $T + x$ it can have at most one child (since $T + x$ is binary search tree)

[1] p has right child y but no left child:

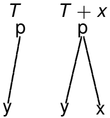


Since T is balanced, y must be a leaf and $BF_T(p) = 0 - (-1) = +1$

After inserting x: $BF_{T+x}(p) = 0$ and $BF_T(v) = BF_{T+x}(v)$ for all $v \in V(T)$
(since height of $T(v)$ remains unchanged for all $v \in V(T)$)

$\implies$ $T + x$ is an AVL tree. [no correction needed]

[2] p has left child y but no right child:

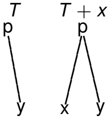


Since T is balanced, y must be a leaf and $BF_T(p) = -1 - 0 = -1$

Part 3: AVL Trees (inserting x)

Inserting a vertex x to T is done as in case for BST $\implies$ we get BST $T + x$ which might be imbalanced (corrections!).
Corrections of tree $T + x$ are based on the following cases for parent p of x in $T + x$.
We denote with $BF_{T'}(v)$ the balance factor of v in tree T'
Since p is parent of x in $T + x$ it can have at most one child (since $T + x$ is binary search tree)

[1] p has right child y but no left child:

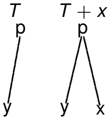


Since T is balanced, y must be a leaf and $BF_T(p) = 0 - (-1) = +1$

After inserting x: $BF_{T+x}(p) = 0$ and $BF_T(v) = BF_{T+x}(v)$ for all $v \in V(T)$
(since height of $T(v)$ remains unchanged for all $v \in V(T)$)

$\implies$ $T + x$ is an AVL tree. [no correction needed]

[2] p has left child y but no right child:



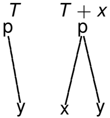
Since T is balanced, y must be a leaf and $BF_T(p) = -1 - 0 = -1$

After inserting x: $BF_{T+x}(p) = 0$ and $BF_T(v) = BF_{T+x}(v)$ for all $v \in V(T)$
(since height of $T(v)$ remains unchanged for all $v \in V(T)$)

Part 3: AVL Trees (inserting x)

Inserting a vertex x to T is done as in case for BST $\implies$ we get BST $T + x$ which might be imbalanced (corrections!).
Corrections of tree $T + x$ are based on the following cases for parent p of x in $T + x$.
We denote with $BF_{T'}(v)$ the balance factor of v in tree T'
Since p is parent of x in $T + x$ it can have at most one child (since $T + x$ is binary search tree)

[1] p has right child y but no left child:

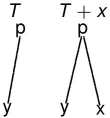


Since T is balanced, y must be a leaf and $BF_T(p) = 0 - (-1) = +1$

After inserting x: $BF_{T+x}(p) = 0$ and $BF_T(v) = BF_{T+x}(v)$ for all $v \in V(T)$
(since height of $T(v)$ remains unchanged for all $v \in V(T)$)

$\implies$ $T + x$ is an AVL tree. [no correction needed]

[2] p has left child y but no right child:



Since T is balanced, y must be a leaf and $BF_T(p) = -1 - 0 = -1$

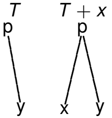
After inserting x: $BF_{T+x}(p) = 0$ and $BF_T(v) = BF_{T+x}(v)$ for all $v \in V(T)$
(since height of $T(v)$ remains unchanged for all $v \in V(T)$)

$\implies$ $T + x$ is an AVL tree. [no correct needed]

Part 3: AVL Trees (inserting x)

Inserting a vertex x to T is done as in case for BST $\implies$ we get BST $T+x$ which might be imbalanced (corrections!).
Corrections of tree $T+x$ are based on the following cases for parent p of x in $T+x$.
We denote with $BF_{T'}(v)$ the balance factor of v in tree T'
Since p is parent of x in $T+x$ it can have at most one child (since $T+x$ is <u>binary</u> search tree)

[1] p has right child y but no left child:

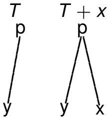


Since T is balanced, y must be a leaf and $BF_T(p) = 0 - (-1) = +1$

After inserting x: $BF_{T+x}(p) = 0$ and $BF_T(v) = BF_{T+x}(v)$ for all $v \in V(T)$
(since height of $T(v)$ remains unchanged for all $v \in V(T)$)

$\implies$ $T+x$ is an AVL tree. [no correction needed]

[2] p has left child y but no right child:

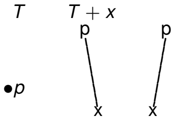


Since T is balanced, y must be a leaf and $BF_T(p) = -1 - 0 = -1$

After inserting x: $BF_{T+x}(p) = 0$ and $BF_T(v) = BF_{T+x}(v)$ for all $v \in V(T)$
(since height of $T(v)$ remains unchanged for all $v \in V(T)$)

$\implies$ $T+x$ is an AVL tree. [no correct needed]

[3] p has no child:



Part 3: AVL Trees (inserting x)

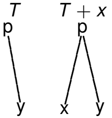
Inserting a vertex x to T is done as in case for BST $\implies$ we get BST $T+x$ which might be imbalanced (corrections!).
Corrections of tree $T+x$ are based on the following cases for parent p of x in $T+x$.
We denote with $BF_{T'}(v)$ the balance factor of v in tree T'
Since p is parent of x in $T+x$ it can have at most one child (since $T+x$ is binary search tree)

[1] p has right child y but no left child:

Since T is balanced, y must be a leaf and $BF_T(p) = 0 - (-1) = +1$

After inserting x: $BF_{T+x}(p) = 0$ and $BF_T(v) = BF_{T+x}(v)$ for all $v \in V(T)$
(since height of $T(v)$ remains unchanged for all $v \in V(T)$)

$\implies$ $T+x$ is an AVL tree. [no correction needed]

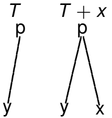


[2] p has left child y but no right child:

Since T is balanced, y must be a leaf and $BF_T(p) = -1 - 0 = -1$

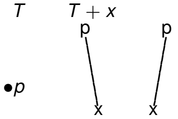
After inserting x: $BF_{T+x}(p) = 0$ and $BF_T(v) = BF_{T+x}(v)$ for all $v \in V(T)$
(since height of $T(v)$ remains unchanged for all $v \in V(T)$)

$\implies$ $T+x$ is an AVL tree. [no correct needed]



[3] p has no child:

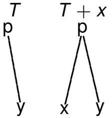
Since p is a leaf, $BF_T(p) = -1 - (-1) = 0$



Part 3: AVL Trees (inserting x)

Inserting a vertex x to T is done as in case for BST $\implies$ we get BST $T + x$ which might be imbalanced (corrections!).
Corrections of tree $T + x$ are based on the following cases for parent p of x in $T + x$.
We denote with $BF_{T'}(v)$ the balance factor of v in tree T'
Since p is parent of x in $T + x$ it can have at most one child (since $T + x$ is binary search tree)

[1] p has right child y but no left child:

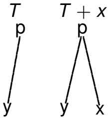


Since T is balanced, y must be a leaf and $BF_T(p) = 0 - (-1) = +1$

After inserting x: $BF_{T+x}(p) = 0$ and $BF_T(v) = BF_{T+x}(v)$ for all $v \in V(T)$
(since height of $T(v)$ remains unchanged for all $v \in V(T)$)

$\implies$ $T + x$ is an AVL tree. [no correction needed]

[2] p has left child y but no right child:

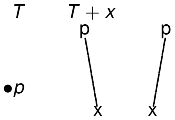


Since T is balanced, y must be a leaf and $BF_T(p) = -1 - 0 = -1$

After inserting x: $BF_{T+x}(p) = 0$ and $BF_T(v) = BF_{T+x}(v)$ for all $v \in V(T)$
(since height of $T(v)$ remains unchanged for all $v \in V(T)$)

$\implies$ $T + x$ is an AVL tree. [no correct needed]

[3] p has no child:



Since p is a leaf, $BF_T(p) = -1 - (-1) = 0$

After inserting x: $BF_{T+x}(p) \in \{+1, -1\}$ and $BF_T(v) \neq BF_{T+x}(v)$ might be possible for $v \in V(T)$

Part 3: AVL Trees (inserting x)

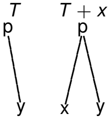
Inserting a vertex x to T is done as in case for BST $\Longrightarrow$ we get BST $T + x$ which might be imbalanced (corrections!).
Corrections of tree $T + x$ are based on the following cases for parent p of x in $T + x$.
We denote with $BF_{T'}(v)$ the balance factor of v in tree T'
Since p is parent of x in $T + x$ it can have at most one child (since $T + x$ is binary search tree)

[1] p has right child y but no left child:

Since T is balanced, y must be a leaf and $BF_T(p) = 0 - (-1) = +1$

After inserting x: $BF_{T+x}(p) = 0$ and $BF_T(v) = BF_{T+x}(v)$ for all $v \in V(T)$
(since height of $T(v)$ remains unchanged for all $v \in V(T)$)

$\Longrightarrow$ $T + x$ is an AVL tree. [no correction needed]

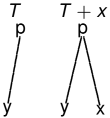


[2] p has left child y but no right child:

Since T is balanced, y must be a leaf and $BF_T(p) = -1 - 0 = -1$

After inserting x: $BF_{T+x}(p) = 0$ and $BF_T(v) = BF_{T+x}(v)$ for all $v \in V(T)$
(since height of $T(v)$ remains unchanged for all $v \in V(T)$)

$\Longrightarrow$ $T + x$ is an AVL tree. [no correct needed]

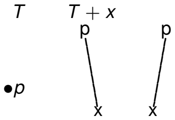


[3] p has no child:

Since p is a leaf, $BF_T(p) = -1 - (-1) = 0$

After inserting x: $BF_{T+x}(p) \in \{+1, -1\}$ and $BF_T(v) \neq BF_{T+x}(v)$ might be possible for $v \in V(T)$

Note, if $BF_T(v) \neq BF_{T+x}(v)$, then v is located on path from ρ to p, i.e.,
at most $h = O(\log(n))$ vertices could be affected [possible correction needed]



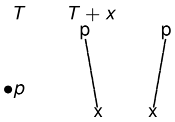
Inserting a vertex x to T is done as in case for BST $\implies$ we get BST $T + x$ which might be imbalanced (corrections!).
Corrections of tree $T + x$ are based on the following cases for parent p of x in $T + x$.
We denote with $BF_{T'}(v)$ the balance factor of v in tree T'
Since p is parent of x in $T + x$ it can have at most one child (since $T + x$ is binary search tree)

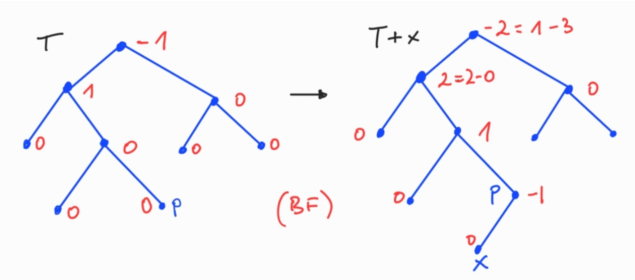
Focus now on **Case [3] p has no child:**

Since p is a leaf, $BF_T(p) = -1 - (-1) = 0$

After inserting x: $BF_{T+x}(p) \in \{+1, -1\}$ and $BF_T(v) \neq BF_{T+x}(v)$ might be possible for $v \in V(T)$

Note, if $BF_T(v) \neq BF_{T+x}(v)$, then v is located on path from ρ to p, i.e., at most $h = O(\log(n))$ vertices could be affected [possible correction needed]





In $BF_{T+x}(v) \in \{-2, -1, 0, +1, +2\}$ since height of T is increased by at most 1 in $T + x$

Part 3: AVL Trees (type of rotations and rebalancing)

Let u be a vertex with child v in T where v is located in the subtree with greater height.

4 cases that yield different "types of rotations":

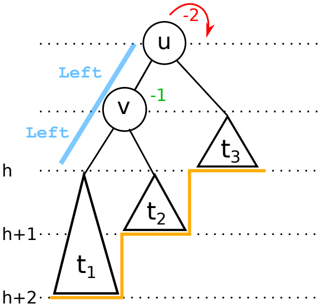
1. $BF_{T'}(u) = -2$, $BF_{T'}(v) \in \{0, -1\}$: **single rotation** *right_rot*(u) `Left-Left-Case`
2. $BF_{T'}(u) = +2$, $BF_{T'}(v) \in \{0, +1\}$: **single rotation** *left_rot*(u) `Right-Right-Case`
3. $BF_{T'}(u) = -2$, $BF_{T'}(v) = +1$: **double rotation** *left_rot*(v) + *right_rot*(u) `Left-Right-Case`
4. $BF_{T'}(u) = +2$, $BF_{T'}(v) = -1$: **double rotation** *right_rot*(v) + *left_rot*(u) `Right-Left-Case`

Part 3: AVL Trees (type of rotations and rebalancing)

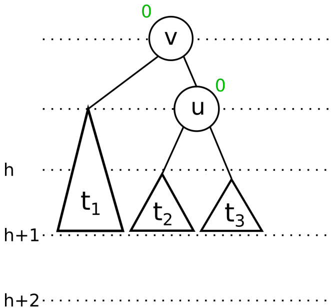
Let u be a vertex with child v in T where v is located in the subtree with greater height.

4 cases that yield different "types of rotations":

1. $BF_{T'}(u) = -2$, $BF_{T'}(v) \in \{0, -1\}$: **single rotation** *right_rot*(u) `Left-Left-Case`
2. $BF_{T'}(u) = +2$, $BF_{T'}(v) \in \{0, +1\}$: **single rotation** *left_rot*(u) `Right-Right-Case`
3. $BF_{T'}(u) = -2$, $BF_{T'}(v) = +1$: **double rotation** *left_rot*(v) + *right_rot*(u) `Left-Right-Case`
4. $BF_{T'}(u) = +2$, $BF_{T'}(v) = -1$: **double rotation** *right_rot*(v) + *left_rot*(u) `Right-Left-Case`



u — Left — v — Left



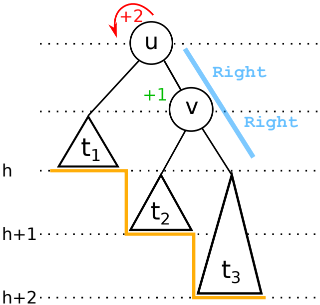
right_rot(u)

Part 3: AVL Trees (type of rotations and rebalancing)

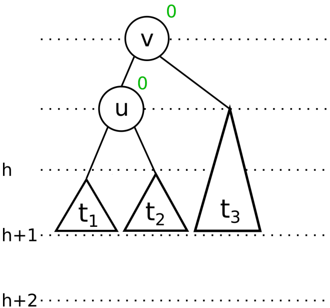
Let u be a vertex with child v in T where v is located in the subtree with greater height.

4 cases that yield different "types of rotations":

1. $BF_{T'}(u) = -2$, $BF_{T'}(v) \in \{0, -1\}$: **single rotation** *right_rot*(u) `Left-Left-Case`
2. $BF_{T'}(u) = +2$, $BF_{T'}(v) \in \{0, +1\}$: **single rotation** *left_rot*(u) `Right-Right-Case`
3. $BF_{T'}(u) = -2$, $BF_{T'}(v) = +1$: **double rotation** *left_rot*(v) + *right_rot*(u) `Left-Right-Case`
4. $BF_{T'}(u) = +2$, $BF_{T'}(v) = -1$: **double rotation** *right_rot*(v) + *left_rot*(u) `Right-Left-Case`



u — Right — v — Right



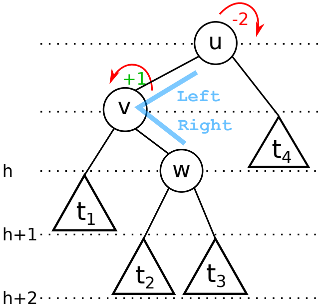
left_rot(u)

Part 3: AVL Trees (type of rotations and rebalancing)

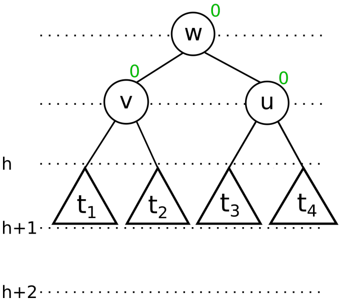
Let u be a vertex with child v in T where v is located in the subtree with greater height.

4 cases that yield different "types of rotations":

1. $BF_{T'}(u) = -2$, $BF_{T'}(v) \in \{0, -1\}$: **single rotation** *right_rot*(u) — `Left-Left-Case`
2. $BF_{T'}(u) = +2$, $BF_{T'}(v) \in \{0, +1\}$: **single rotation** *left_rot*(u) — `Right-Right-Case`
3. $BF_{T'}(u) = -2$, $BF_{T'}(v) = +1$: **double rotation** *left_rot*(v) + *right_rot*(u) — `Left-Right-Case`
4. $BF_{T'}(u) = +2$, $BF_{T'}(v) = -1$: **double rotation** *right_rot*(v) + *left_rot*(u) — `Right-Left-Case`



u — `Left` — v — `Right`



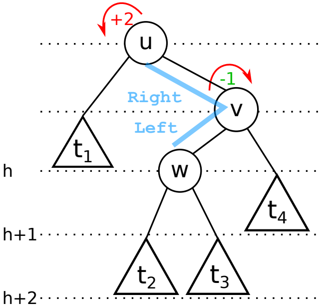
left_rot(v) + *right_rot*(u)

Part 3: AVL Trees (type of rotations and rebalancing)

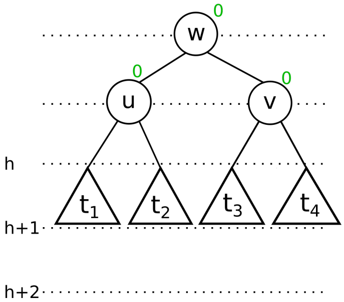
Let u be a vertex with child v in T where v is located in the subtree with greater height.

4 cases that yield different "types of rotations":

1. $BF_{T'}(u) = -2$, $BF_{T'}(v) \in \{0, -1\}$: **single rotation** *right_rot*(u) — `Left-Left-Case`
2. $BF_{T'}(u) = +2$, $BF_{T'}(v) \in \{0, +1\}$: **single rotation** *left_rot*(u) — `Right-Right-Case`
3. $BF_{T'}(u) = -2$, $BF_{T'}(v) = +1$: **double rotation** *left_rot*(v) + *right_rot*(u) — `Left-Right-Case`
4. $BF_{T'}(u) = +2$, $BF_{T'}(v) = -1$: **double rotation** *right_rot*(v) + *left_rot*(u) — `Right-Left-Case`



u — `Right` — v — `Left`



right_rot(v) + *left_rot*(u)

Part 3: AVL Trees (type of rotations and rebalancing)

Let u be a vertex with child v in T where v is located in the subtree with greater height.

4 cases that yield different "types of rotations":

1. $BF_{T'}(u) = -2$, $BF_{T'}(v) \in \{0, -1\}$: **single rotation** $right\_rot(u)$ `Left-Left-Case`
2. $BF_{T'}(u) = +2$, $BF_{T'}(v) \in \{0, +1\}$: **single rotation** $left\_rot(u)$ `Right-Right-Case`
3. $BF_{T'}(u) = -2$, $BF_{T'}(v) = +1$: **double rotation** $left\_rot(v) + right\_rot(u)$ `Left-Right-Case`
4. $BF_{T'}(u) = +2$, $BF_{T'}(v) = -1$: **double rotation** $right\_rot(v) + left\_rot(u)$ `Right-Left-Case`

Let $T + x$ be BST obtained from AVL tree after inserting x.

Next "rebalancing" algorithm applied on $T' = T + x$ ensures that the resulting tree is an AVL tree.

pseudocode - sketch `ReBalance`:

`FOR` all vertices u on path from x to root (in this order) `DO`

`IF` $BF_{T'}(u) = -2$ `THEN` consider left child v of u. `//Left`

`IF` $BF_{T'}(v) \leq 0$ `THEN` $right\_rot(u)$. `//Left-Left`

`ELSE` $left\_rot(v)$ and $right\_rot(u)$. `//Left-Right`

`IF` $BF_{T'}(u) = +2$ `THEN` consider right child v of u. `//Right`

`IF` $BF_{T'}(v) \geq 0$ `THEN` $left\_rot(u)$. `//Right-Right`

`ELSE` $right\_rot(v)$ and $left\_rot(u)$. `//Right-Left`

Part 3: AVL Trees (type of rotations and rebalancing)

Let u be a vertex with child v in T where v is located in the subtree with greater height.

4 cases that yield different "types of rotations":

1. $BF_{T'}(u) = -2$, $BF_{T'}(v) \in \{0, -1\}$: **single rotation** *right_rot*(u) `Left-Left-Case`
2. $BF_{T'}(u) = +2$, $BF_{T'}(v) \in \{0, +1\}$: **single rotation** *left_rot*(u) `Right-Right-Case`
3. $BF_{T'}(u) = -2$, $BF_{T'}(v) = +1$: **double rotation** *left_rot*(v) + *right_rot*(u) `Left-Right-Case`
4. $BF_{T'}(u) = +2$, $BF_{T'}(v) = -1$: **double rotation** *right_rot*(v) + *left_rot*(u) `Right-Left-Case`

Let $T + x$ be BST obtained from AVL tree after inserting x.

Next "rebalancing" algorithm applied on $T' = T + x$ ensures that the resulting tree is an AVL tree.

pseudocode - sketch `ReBalance`:

`FOR` all vertices u on path from x to root (in this order) `DO`

  `IF` $BF_{T'}(u) = -2$ `THEN` consider left child v of u. //Left

    `IF` $BF_{T'}(v) \leq 0$ `THEN` *right_rot*(u). //Left-Left

    `ELSE` *left_rot*(v) and *right_rot*(u). //Left-Right

  `IF` $BF_{T'}(u) = +2$ `THEN` consider right child v of u. //Right

    `IF` $BF_{T'}(v) \geq 0$ `THEN` *left_rot*(u). //Right-Right

    `ELSE` *right_rot*(v) and *left_rot*(u). //Right-Left

[Example Board]

Part 3: AVL Trees (type of rotations and rebalancing)

Let u be a vertex with child v in T where v is located in the subtree with greater height.

4 cases that yield different "types of rotations":

1. $BF_{T'}(u) = -2$, $BF_{T'}(v) \in \{0, -1\}$: **single rotation** *right_rot*(u) `Left-Left-Case`
2. $BF_{T'}(u) = +2$, $BF_{T'}(v) \in \{0, +1\}$: **single rotation** *left_rot*(u) `Right-Right-Case`
3. $BF_{T'}(u) = -2$, $BF_{T'}(v) = +1$: **double rotation** *left_rot*(v) + *right_rot*(u) `Left-Right-Case`
4. $BF_{T'}(u) = +2$, $BF_{T'}(v) = -1$: **double rotation** *right_rot*(v) + *left_rot*(u) `Right-Left-Case`

RUNTIME-sketch (insert incl. rebalancing):

Costs for inserting a new vertex: $O(\log(n))$ time.

Costs for single/double rotation: $O(1)$ time.

there are at most $h = \log(n)$ vertices along path from x to root.

determining BFs in $T + x$ can be done in $O(1)$ time

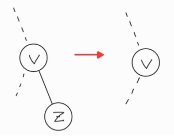
(since it is determined by the BFs in T [advanced exercise])

Total time for insert (incl. rebalancing): $O(\log(n))$

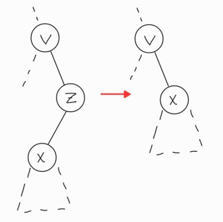
Part 3: AVL-Trees (delete)

The same as in BST (delete).

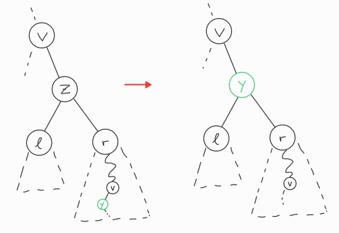
`Tree-Delete`(T, z)



CASE z has no child:
just delete z



CASE z has one child:
Delete z and make child x of z ..
.. the right child of $parent(z) = v$ in case z is right child of v
.. the left child of $parent(z) = v$ in case z is left child of v



CASE z has two children:
Find y in $T(r)$ with min $y.key$ and such that y has no left child
Delete y from T [see previous cases]
Replace z by y

Now apply pseudocode - sketch `ReBalance`:

$\Longrightarrow$ Total time for delete in AVL tree (incl. rebalancing): $O(\log(n))$

[Example Board]

Part 3: Red-Black Trees

To summarize at this point:

queries as search, find_minimum, find_maximum as well as insertion, deletion can be done in $O(h)$ time in binary search trees where h = height of tree.

Problem: $O(h) = O(n)$ where n =number of vertices [can we control the height?]

Answer: In AVL tree we can control height $h \in O(\log n)$

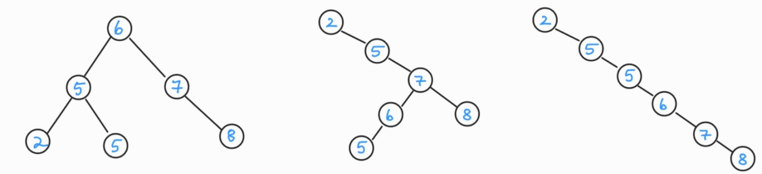
We continue now with Red-Black trees that are one of *many* search-tree schemes that are “balanced” in order to guarantee that basic dynamic-set operations take $O(\log n)$ time in the worst case.

Part 3: Red-Black Trees

To summarize at this point:

queries as search, find_minimum, find_maximum as well as insertion, deletion can be done in $O(h)$ time in binary search trees where h = height of tree.

Problem: $O(h) = O(n)$ where n =number of vertices [can we control the height?]



Answer: In AVL tree we can control height $h \in O(\log n)$

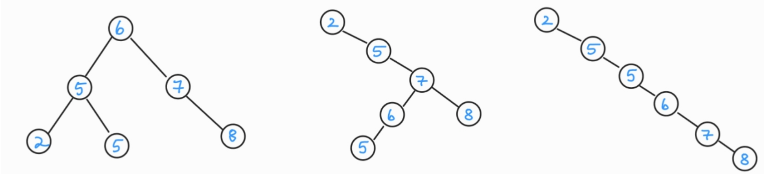
We continue now with Red-Black trees that are one of *many* search-tree schemes that are "balanced" in order to guarantee that basic dynamic-set operations take $O(\log n)$ time in the worst case.

Part 3: Red-Black Trees

To summarize at this point:

queries as search, find_minimum, find_maximum as well as insertion, deletion can be done in $O(h)$ time in binary search trees where h = height of tree.

Problem: $O(h) = O(n)$ where n =number of vertices [can we control the height?]



Answer: In AVL tree we can control height $h \in O(\log n)$

We continue now with Red-Black trees that are one of *many* search-tree schemes that are "balanced" in order to guarantee that basic dynamic-set operations take $O(\log n)$ time in the worst case.

Part 3: Red-Black Trees

In Red-Black trees we have one extra bit of storage per node: its color, either RED or BLACK.

A Red-Black tree is a binary search tree that satisfies the following Red-Black properties:

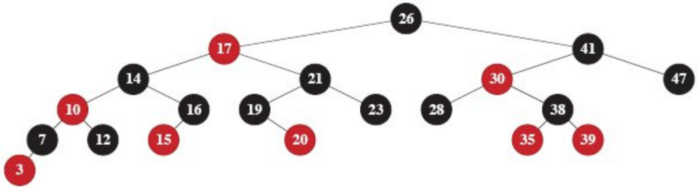
I. (a) Every node is either RED or BLACK. (b) The root is BLACK.
II. If a node is RED, then each of its children must be BLACK
III. For each node x, all simple paths from x to descendant leaves contain the same number of BLACK nodes.

Part 3: Red-Black Trees

In Red-Black trees we have one extra bit of storage per node: its color, either RED or BLACK.

A Red-Black tree is a binary search tree that satisfies the following Red-Black properties:

I. (a) Every node is either RED or BLACK. (b) The root is BLACK.
II. If a node is RED, then each of its children must be BLACK
III. For each node x, all simple paths from x to descendant leaves contain the same number of BLACK nodes.

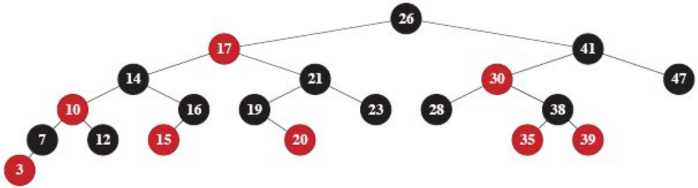


Part 3: Red-Black Trees

In Red-Black trees we have one extra bit of storage per node: its color, either RED or BLACK.

A Red-Black tree is a binary search tree that satisfies the following Red-Black properties:

I. (a) Every node is either RED or BLACK. (b) The root is BLACK.
II. If a node is RED, then each of its children must be BLACK
III. For each node x, all simple paths from x to descendant leaves contain the same number of BLACK nodes.



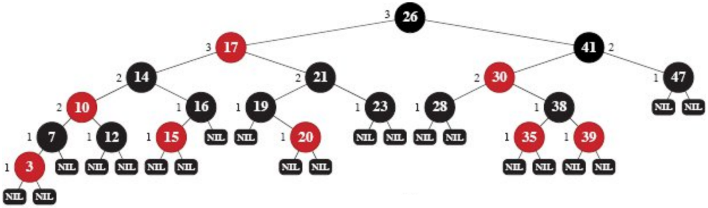
If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value *NIL*: Think of these *NIL*s as pointers to leaves (external nodes) of the binary search tree and the normal, key-bearing nodes as internal nodes of the tree. Those NIL-leaves are always supposed to be BLACK.

Part 3: Red-Black Trees

In Red-Black trees we have one extra bit of storage per node: its color, either RED or BLACK.

A Red-Black tree is a binary search tree that satisfies the following Red-Black properties:

I. (a) Every node is either RED or BLACK. (b) The root is BLACK. (c) Every leaf (*T.nil*) is BLACK.
II. If a node is RED, then each of its children must be BLACK
III. For each node *x*, all simple paths from *x* to descendant leaves contain the same number of BLACK nodes.



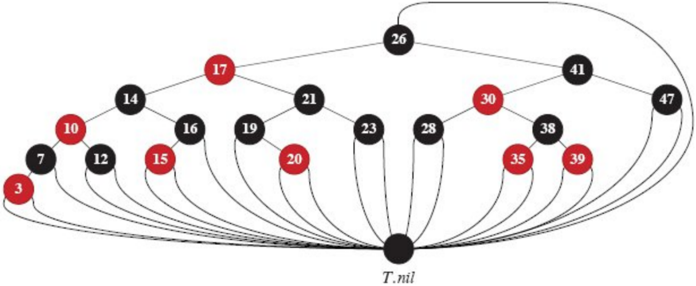
If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value *NIL*: Think of these *NIL*s as pointers to leaves (external nodes) of the binary search tree and the normal, key-bearing nodes as internal nodes of the tree. Those NIL-leaves are always supposed to be BLACK.

Part 3: Red-Black Trees

In Red-Black trees we have one extra bit of storage per node: its color, either RED or BLACK.

A Red-Black tree is a binary search tree that satisfies the following Red-Black properties:

I. (a) Every node is either RED or BLACK. (b) The root is BLACK. (c) Every leaf ($T.nil$) is BLACK.
II. If a node is RED, then each of its children must be BLACK
III. For each node x, all simple paths from x to descendant leaves contain the same number of BLACK nodes.



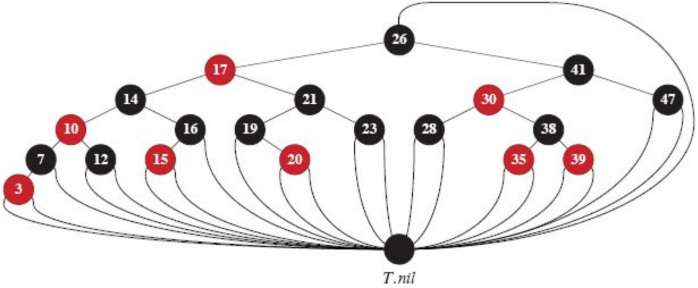
As a matter of convenience in dealing with boundary conditions and for having simpler code, we use a single **sentinel** *T.nil* (*sv: väktare*) to represent NIL and that is always of color BLACK and we don't care about *T.nils*' other attributes $\Longrightarrow$ Red-Black tree is fully binary

Part 3: Red-Black Trees

In Red-Black trees we have one extra bit of storage per node: its color, either RED or BLACK.

A Red-Black tree is a binary search tree that satisfies the following Red-Black properties:

I. (a) Every node is either RED or BLACK. (b) The root is BLACK. (c) Every leaf ($T.nil$) is BLACK.
II. If a node is RED, then each of its children must be BLACK
III. For each node x, all simple paths from x to descendant leaves contain the same number of BLACK nodes.



As a matter of convenience in dealing with boundary conditions and for having simpler code, we use a single **sentinel** *T.nil* (*sv: väktare*) to represent NIL and that is always of color BLACK and we don't care about *T.nils*' other attributes $\Longrightarrow$ Red-Black tree is fully binary

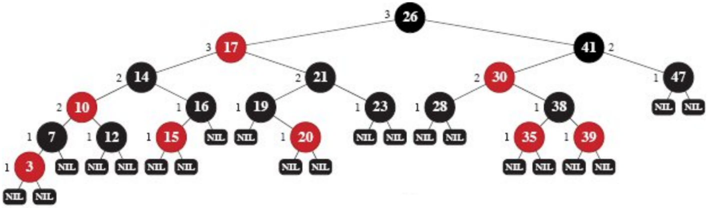
black-height $bh(x)$ = number of BLACK nodes on any simple path from, but not including, node x down to a leaf ($T.nil$)

Part 3: Red-Black Trees

In Red-Black trees we have one extra bit of storage per node: its color, either RED or BLACK.

A Red-Black tree is a binary search tree that satisfies the following Red-Black properties:

I. (a) Every node is either RED or BLACK. (b) The root is BLACK. (c) Every leaf ($T.nil$) is BLACK.
II. If a node is RED, then each of its children must be BLACK
III. For each node x, all simple paths from x to descendant leaves contain the same number of BLACK nodes.



As a matter of convenience in dealing with boundary conditions and for having simpler code, we use a single **sentinel** $T.nil$ (*sv: väktare*) to represent NIL and that is always of color BLACK and we don't care about $T.nil$s' other attributes $\Longrightarrow$ Red-Black tree is fully binary

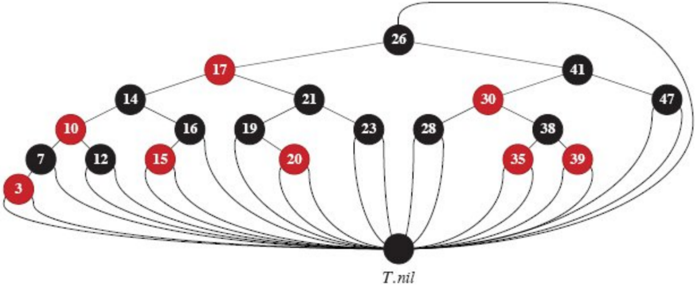
black-height $bh(x)$ = number of BLACK nodes on any simple path from, but not including, node x down to a leaf ($T.nil$)

Part 3: Red-Black Trees

In Red-Black trees we have one extra bit of storage per node: its color, either RED or BLACK.

A Red-Black tree is a binary search tree that satisfies the following Red-Black properties:

I. (a) Every node is either RED or BLACK. (b) The root is BLACK. (c) Every leaf ($T.nil$) is BLACK.
II. If a node is RED, then each of its children must be BLACK
III. For each node x, all simple paths from x to descendant leaves contain the same number of BLACK nodes.



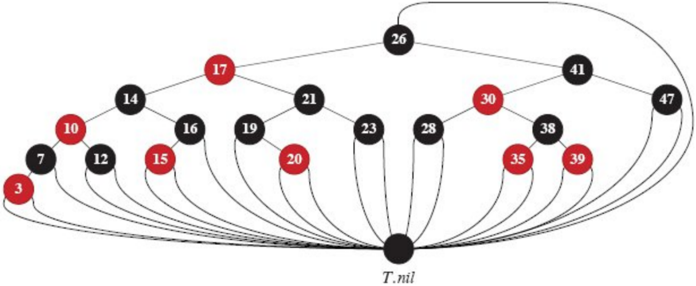
Lemma. *Any Red-Black tree with n internal nodes (=vertices x with x.key $\neq$ NIL) has height* $O(\log n)$ [proof board]

Part 3: Red-Black Trees

In Red-Black trees we have one extra bit of storage per node: its color, either RED or BLACK.

A Red-Black tree is a binary search tree that satisfies the following Red-Black properties:

I. (a) Every node is either RED or BLACK. (b) The root is BLACK. (c) Every leaf ($T.nil$) is BLACK.
II. If a node is RED, then each of its children must be BLACK
III. For each node x, all simple paths from x to descendant leaves contain the same number of BLACK nodes.



Lemma. *Any Red-Black tree with n internal nodes (=vertices x with x.key $\neq$ NIL) has height* $O(\log n)$ [proof board]

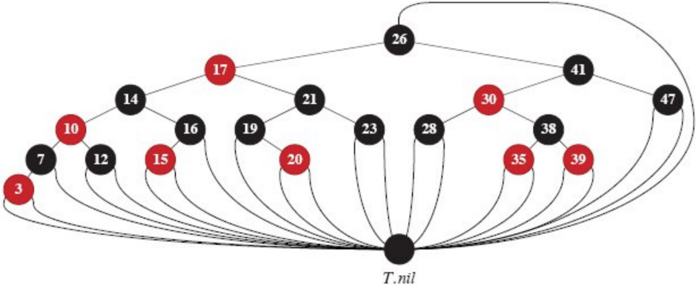
Thus, queries as search, find_minimum, find_maximum as well as insert and delete can be done in $O(h) = O(\log n)$ time in Red-Black trees.

Part 3: Red-Black Trees

In Red-Black trees we have one extra bit of storage per node: its color, either RED or BLACK.

A Red-Black tree is a binary search tree that satisfies the following Red-Black properties:

I. (a) Every node is either RED or BLACK. (b) The root is BLACK. (c) Every leaf ($T.nil$) is BLACK.
II. If a node is RED, then each of its children must be BLACK
III. For each node x, all simple paths from x to descendant leaves contain the same number of BLACK nodes.



Lemma. *Any Red-Black tree with n internal nodes (=vertices x with x.key $\neq$ NIL) has height* $O(\log n)$ [proof board]

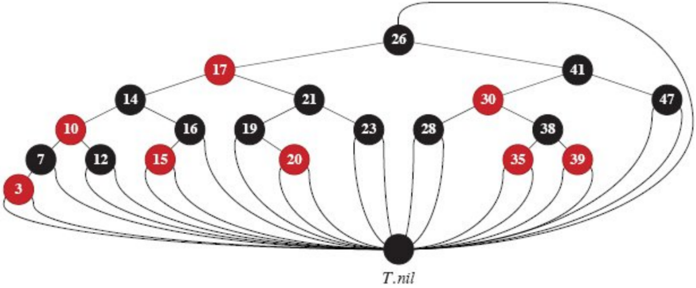
Thus, queries as search, find_minimum, find_maximum as well as insert and delete can be done in $O(h) = O(\log n)$ time in Red-Black trees. However, after a single of the operations insert and delete it is not ensured that the resulting tree is a Red-Black tree and so, $h > \log(n)$ might be possible.

Part 3: Red-Black Trees

In Red-Black trees we have one extra bit of storage per node: its color, either RED or BLACK.

A Red-Black tree is a binary search tree that satisfies the following Red-Black properties:

I. (a) Every node is either RED or BLACK. (b) The root is BLACK. (c) Every leaf ($T.nil$) is BLACK.
II. If a node is RED, then each of its children must be BLACK
III. For each node x, all simple paths from x to descendant leaves contain the same number of BLACK nodes.



Lemma. *Any Red-Black tree with n internal nodes (=vertices x with $x.key \neq$ NIL) has height $O(\log n)$* [proof board]

Thus, queries as search, find_minimum, find_maximum as well as insert and delete can be done in $O(h) = O(\log n)$ time in Red-Black trees. However, after a single of the operations insert and delete it is not ensured that the resulting tree is a Red-Black tree and so, $h > \log(n)$ might be possible.

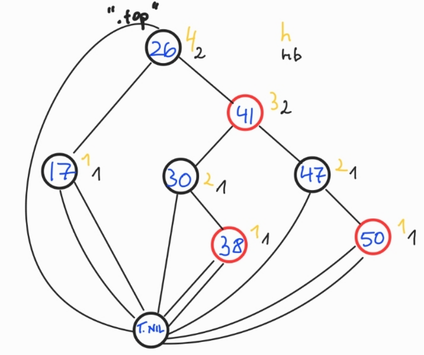
In this case, it is not ensured anymore that the latter operations still run in $O(\log n)$ time

Part 3: Red-Black Trees

In Red-Black trees we have one extra bit of storage per node: its color, either RED or BLACK.

A Red-Black tree is a binary search tree that satisfies the following Red-Black properties:

I. (a) Every node is either RED or BLACK. (b) The root is BLACK. (c) Every leaf ($T.nil$) is BLACK.
II. If a node is RED, then each of its children must be BLACK
III. For each node x, all simple paths from x to descendant leaves contain the same number of BLACK nodes.



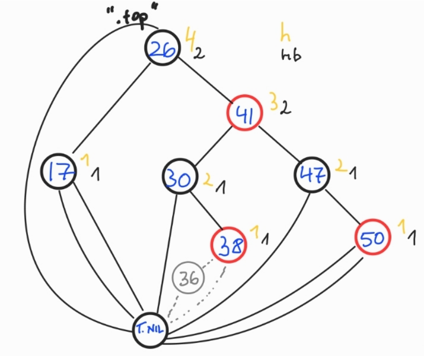
insert key=36: where and which color?

Part 3: Red-Black Trees

In Red-Black trees we have one extra bit of storage per node: its color, either RED or BLACK.

A Red-Black tree is a binary search tree that satisfies the following Red-Black properties:

I. (a) Every node is either RED or BLACK. (b) The root is BLACK. (c) Every leaf ($T.nil$) is BLACK.
II. If a node is RED, then each of its children must be BLACK
III. For each node x, all simple paths from x to descendant leaves contain the same number of BLACK nodes.



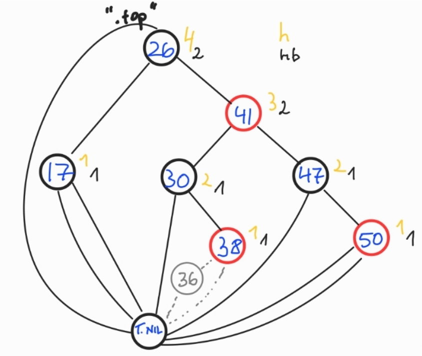
insert key=36: where and which color?

Part 3: Red-Black Trees

In Red-Black trees we have one extra bit of storage per node: its color, either RED or BLACK.

A Red-Black tree is a binary search tree that satisfies the following Red-Black properties:

I. (a) Every node is either RED or BLACK. (b) The root is BLACK. (c) Every leaf ($T.nil$) is BLACK.
II. If a node is RED, then each of its children must be BLACK
III. For each node x, all simple paths from x to descendant leaves contain the same number of BLACK nodes.



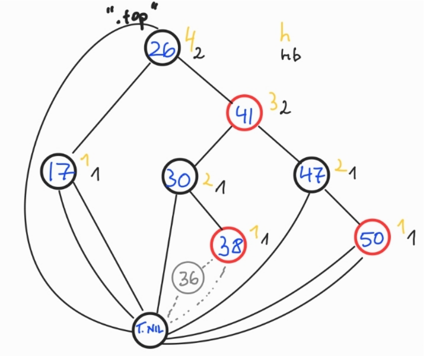
insert key=36: where and which color? Due to Cond. II it must be BLACK

Part 3: Red-Black Trees

In Red-Black trees we have one extra bit of storage per node: its color, either RED or BLACK.

A Red-Black tree is a binary search tree that satisfies the following Red-Black properties:

I. (a) Every node is either RED or BLACK. (b) The root is BLACK. (c) Every leaf ($T.nil$) is BLACK.
II. If a node is RED, then each of its children must be BLACK
III. For each node x, all simple paths from x to descendant leaves contain the same number of BLACK nodes.



insert key=36: where and which color? Due to Cond. II it must be BLACK BUT then Cond. III is violated

Part 3: Red-Black Trees

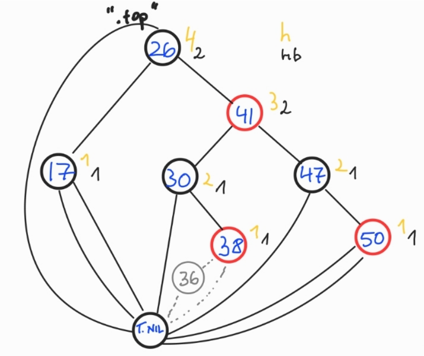
In Red-Black trees we have one extra bit of storage per node: its color, either RED or BLACK.

A Red-Black tree is a binary search tree that satisfies the following Red-Black properties:

I. (a) Every node is either RED or BLACK. (b) The root is BLACK. (c) Every leaf ($T.nil$) is BLACK.

II. If a node is RED, then each of its children must be BLACK

III. For each node x, all simple paths from x to descendant leaves contain the same number of BLACK nodes.



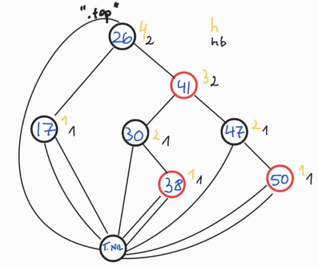
insert key=36: where and which color? Due to Cond. II it must be BLACK BUT then Cond. III is violated

We show now how to insert into and delete from a Red-Black tree in $O(\log n)$ time while preserving the Red-Black properties.

Part 3: Red-Black Trees (insert)

```
RB-Insert(T, z)
1  Tree-Insert(T, z)
2  z.right := T.NIL and z.left := T.NIL
   //both of z's children are the sentinel
3  z.color := RED
4  RB-Insert-Fixup(T, z)
```

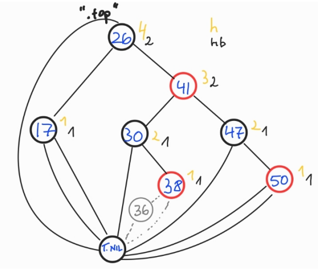
Insert as in usual binary search tree + Line 2,3,4



Part 3: Red-Black Trees (insert)

RB-Insert(T, z)
1 Tree-Insert(T, z)
2 $z.right$:= $T.NIL$ and $z.left$:= $T.NIL$
//both of z's children are the sentinel
3 $z.color$:= Red
4 RB-Insert-Fixup(T, z)

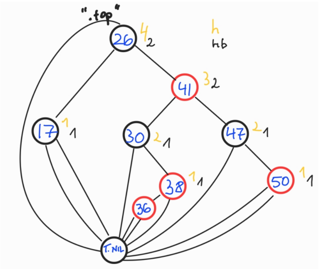
Insert as in usual binary search tree + Line 2,3,4



Part 3: Red-Black Trees (insert)

```
RB-Insert(T, z)
1  Tree-Insert(T, z)
2  z.right := T.NIL and z.left := T.NIL
   //both of z's children are the sentinel
3  z.color := RED
4  RB-Insert-Fixup(T, z)
```

Insert as in usual binary search tree + Line 2,3,4



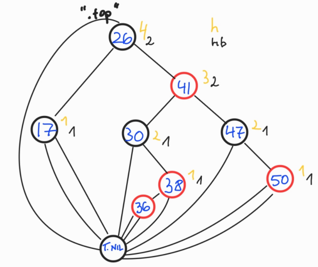
Which Red-Black properties could be violated?

- I.a Every node is either RED or BLACK.
- I.b The root is BLACK
- I.c Every leaf ($T.nil$) is BLACK.
- II If a node is RED, then each of its children must be BLACK
- III For each node x, all simple paths from x to descendant leaves contain the same number of BLACK nodes.

Part 3: Red-Black Trees (insert)

RB-Insert(T, z)
1 Tree-Insert(T, z)
2 $z.right := T.NIL$ and $z.left := T.NIL$
//both of z's children are the sentinel
3 $z.color :=$ RED
4 RB-Insert-Fixup(T, z)

Insert as in usual binary search tree + Line 2,3,4



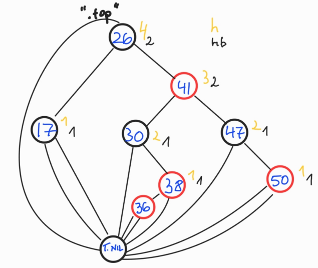
Which Red-Black properties could be violated?

- I.a Every node is either RED or BLACK.
- I.b The root is BLACK
- I.c Every leaf ($T.nil$) is BLACK.
- II If a node is RED, then each of its children must be BLACK
- III For each node x, all simple paths from x to descendant leaves contain the same number of BLACK nodes.

Part 3: Red-Black Trees (insert)

```
RB-Insert(T, z)
1  Tree-Insert(T, z)
2  z.right := T.NIL and z.left := T.NIL
   //both of z's children are the sentinel
3  z.color := RED
4  RB-Insert-Fixup(T, z)
```

Insert as in usual binary search tree + Line 2,3,4



Which Red-Black properties could be violated?

I.a Every node is either RED or BLACK. **OK!**

I.b The root is BLACK

I.c Every leaf ($T.nil$) is BLACK.

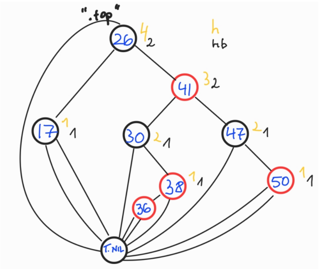
II If a node is RED, then each of its children must be BLACK

III For each node x, all simple paths from x to descendant leaves contain the same number of BLACK nodes.

Part 3: Red-Black Trees (insert)

```
RB-Insert(T, z)
1  Tree-Insert(T, z)
2  z.right := T.NIL and z.left := T.NIL
   //both of z's children are the sentinel
3  z.color := RED
4  RB-Insert-Fixup(T, z)
```

Insert as in usual binary search tree + Line 2,3,4



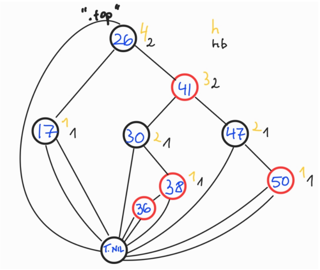
Which Red-Black properties could be violated?

- I.a Every node is either RED or BLACK. **OK!**
- I.b The root is BLACK
- I.c Every leaf ($T.nil$) is BLACK.
- II If a node is RED, then each of its children must be BLACK
- III For each node x, all simple paths from x to descendant leaves contain the same number of BLACK nodes.

Part 3: Red-Black Trees (insert)

```
RB-Insert(T, z)
1  Tree-Insert(T, z)
2  z.right := T.NIL and z.left := T.NIL
   //both of z's children are the sentinel
3  z.color := RED
4  RB-Insert-Fixup(T, z)
```

Insert as in usual binary search tree + Line 2,3,4



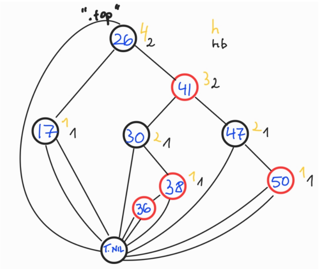
Which Red-Black properties could be violated?

- I.a Every node is either RED or BLACK. **OK!**
- I.b The root is BLACK OK, unless z is now root.
- I.c Every leaf ($T.nil$) is BLACK.
- II If a node is RED, then each of its children must be BLACK
- III For each node x, all simple paths from x to descendant leaves contain the same number of BLACK nodes.

Part 3: Red-Black Trees (insert)

RB-Insert(T, z)
1 Tree-Insert(T, z)
2 $z.right := T.NIL$ and $z.left := T.NIL$
//both of z's children are the sentinel
3 $z.color :=$ RED
4 RB-Insert-Fixup(T, z)

Insert as in usual binary search tree + Line 2,3,4



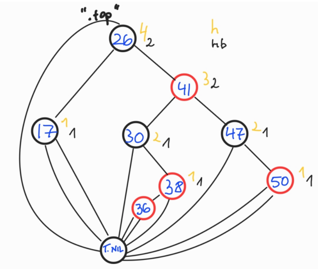
Which Red-Black properties could be violated?

- I.a Every node is either RED or BLACK. **OK!**
- I.b The root is BLACK OK, unless z is now root.
- I.c Every leaf ($T.nil$) is BLACK.
- II If a node is RED, then each of its children must be BLACK
- III For each node x, all simple paths from x to descendant leaves contain the same number of BLACK nodes.

Part 3: Red-Black Trees (insert)

`RB-Insert`(T, z)
1 `Tree-Insert`(T, z)
2 $z.right := T.NIL$ and $z.left := T.NIL$
//both of z's children are the sentinel
3 $z.color :=$ RED
4 `RB-Insert-Fixup`(T, z)

Insert as in usual binary search tree + Line 2,3,4



Which Red-Black properties could be violated?

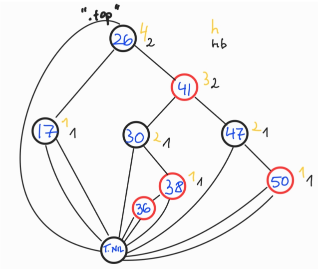
- I.a Every node is either RED or BLACK. **OK!**
- I.b The root is BLACK OK, unless z is now root.
- I.c Every leaf ($T.nil$) is BLACK. **OK!**
- II If a node is RED, then each of its children must be BLACK
- III For each node x, all simple paths from x to descendant leaves contain the same number of BLACK nodes.

Part 3: Red-Black Trees (insert)

RB-Insert(T, z)

1 Tree-Insert(T, z)
2 $z.right := T.NIL$ and $z.left := T.NIL$
  //both of z's children are the sentinel
3 $z.color :=$ RED
4 RB-Insert-Fixup(T, z)

Insert as in usual binary search tree + Line 2,3,4



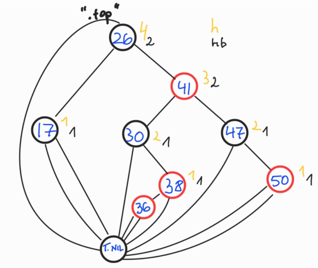
Which Red-Black properties could be violated?

- I.a Every node is either RED or BLACK. **OK!**
- I.b The root is BLACK OK, unless z is now root.
- I.c Every leaf ($T.nil$) is BLACK. **OK!**
- II If a node is RED, then each of its children must be BLACK
- III For each node x, all simple paths from x to descendant leaves contain the same number of BLACK nodes.

Part 3: Red-Black Trees (insert)

RB-Insert(T, z)
1 Tree-Insert(T, z)
2 $z.right := T.NIL$ and $z.left := T.NIL$
//both of z's children are the sentinel
3 $z.color :=$ RED
4 RB-Insert-Fixup(T, z)

Insert as in usual binary search tree + Line 2,3,4



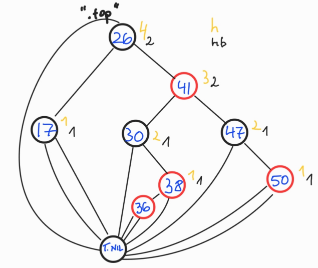
Which Red-Black properties could be violated?

- I.a Every node is either RED or BLACK. **OK!**
- I.b The root is BLACK OK, unless z is now root.
- I.c Every leaf ($T.nil$) is BLACK. **OK!**
- II If a node is RED, then each of its children must be BLACK
  violated if $(z.top).color$ is RED
- III For each node x, all simple paths from x to descendant leaves contain the same number of BLACK nodes.

Part 3: Red-Black Trees (insert)

RB-Insert(T, z)
1 Tree-Insert(T, z)
2 $z.right := T.NIL$ and $z.left := T.NIL$
//both of z's children are the sentinel
3 $z.color :=$ RED
4 RB-Insert-Fixup(T, z)

Insert as in usual binary search tree + Line 2,3,4



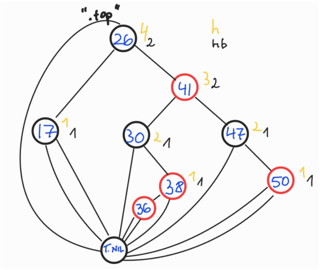
Which Red-Black properties could be violated?

- I.a Every node is either RED or BLACK. **OK!**
- I.b The root is BLACK OK, unless z is now root.
- I.c Every leaf ($T.nil$) is BLACK. **OK!**
- II If a node is RED, then each of its children must be BLACK violated if $(z.top).color$ is RED
- III For each node x, all simple paths from x to descendant leaves contain the same number of BLACK nodes.

Part 3: Red-Black Trees (insert)

```
RB-Insert(T, z)
1  Tree-Insert(T, z)
2  z.right := T.NIL and z.left := T.NIL
   //both of z's children are the sentinel
3  z.color := RED
4  RB-Insert-Fixup(T, z)
```

Insert as in usual binary search tree + Line 2,3,4



Which Red-Black properties could be violated?

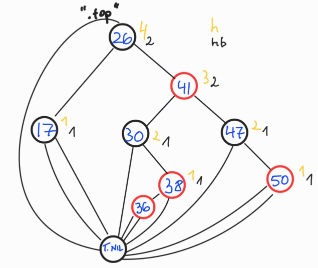
- I.a Every node is either RED or BLACK. **OK!**
- I.b The root is BLACK OK, unless z is now root.
- I.c Every leaf ($T.nil$) is BLACK. **OK!**
- II If a node is RED, then each of its children must be BLACK violated if $(z.top).color$ is RED
- III For each node x, all simple paths from x to descendant leaves contain the same number of BLACK nodes. **OK! hence, $bh(x)$ remains unchanged for all x**

Part 3: Red-Black Trees (insert)

```
RB-Insert(T, z)
1 Tree-Insert(T, z)
2 z.right := T.NIL and z.left := T.NIL
  //both of z's children are the sentinel
3 z.color := RED
4 RB-Insert-Fixup(T, z)
```

Insert as in usual binary search tree + Line 2,3,4

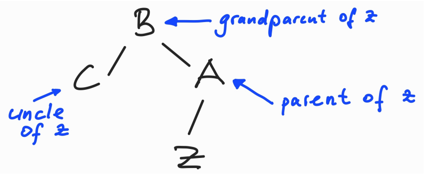
We use `RB-Insert-Fixup`(T, z) which is based on re-coloring and rotations to fix issues that may occur in I.b and II.
Coloring z BLACK would yield other issues that are harder to fix!



Which Red-Black properties could be violated?

- I.a Every node is either RED or BLACK. **OK!**
- I.b The root is BLACK OK, unless z is now root.
- I.c Every leaf ($T.nil$) is BLACK. **OK!**
- II If a node is RED, then each of its children must be BLACK
  violated if $(z.top).color$ is RED
- III For each node x, all simple paths from x to descendant leaves contain the same number of BLACK nodes.
  OK! hence, $bh(x)$ remains unchanged for all x

Part 3: Red-Black Trees (insert)



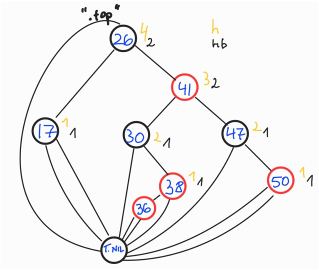
Notation in binary tree (left&right not important in this fig!)
uncle always exist (uncle = T.NIL possible)

Scenarios where z needs some fix up:

I.b z is the root (T was empty at start)

II. parent of z is RED
Then we distinguish:

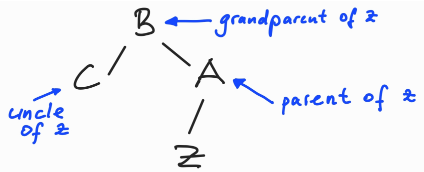
i. uncle of z is RED
ii. uncle of z is BLACK (triangle)
iii. uncle of z is BLACK (line)



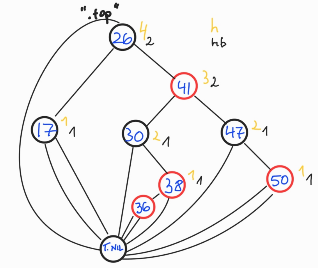
Which Red-Black properties could be violated?

I.a Every node is either RED or BLACK. **OK!**
I.b The root is BLACK OK, unless z is now root.
I.c Every leaf ($T.nil$) is BLACK. **OK!**
II If a node is RED, then each of its children must be BLACK
violated if $(z.top).color$ is RED
III For each node x, all simple paths from x to descendant leaves contain the same number of BLACK nodes.
OK! hence, $bh(x)$ **remains unchanged for all** x

Part 3: Red-Black Trees (insert)



Notation in binary tree (left&right not important in this fig!)
uncle always exist (uncle = T.NIL possible)



Scenarios where z needs some fix up:

I.b z is the root (T was empty at start)

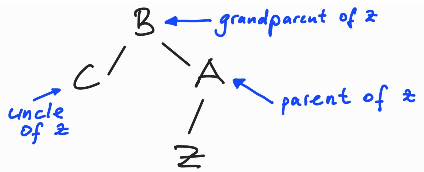
II. parent of z is RED
Then we distinguish:

i. uncle of z is RED
ii. uncle of z is BLACK (triangle)
iii. uncle of z is BLACK (line)

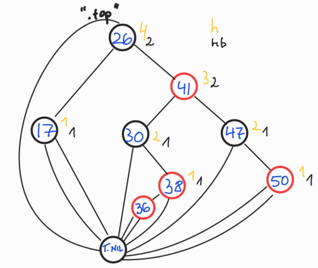
Which Red-Black properties could be violated?

I.a Every node is either RED or BLACK. **OK!**
I.b The root is BLACK OK, unless z is now root.
I.c Every leaf ($T.nil$) is BLACK. **OK!**
II If a node is RED, then each of its children must be BLACK
violated if $(z.top).color$ is RED
III For each node x, all simple paths from x to descendant leaves contain the same number of BLACK nodes.
OK! hence, $bh(x)$ **remains unchanged for all** x

Part 3: Red-Black Trees (insert)



Notation in binary tree (left&right not important in this fig!)
uncle always exist (uncle = T.NIL possible)



Scenarios where z needs some fix up:

I.b z is the root (T was empty at start)

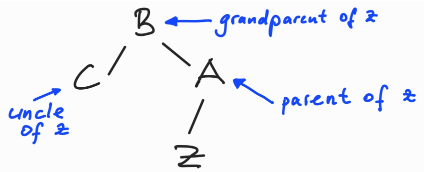
II. parent of z is RED
Then we distinguish:

i. uncle of z is RED
ii. uncle of z is BLACK (triangle)
iii. uncle of z is BLACK (line)

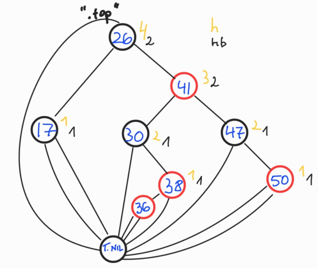
Which Red-Black properties could be violated?

I.a Every node is either RED or BLACK. **OK!**
I.b The root is BLACK OK, unless z is now root.
I.c Every leaf ($T.nil$) is BLACK. **OK!**
II If a node is RED, then each of its children must be BLACK
violated if $(z.top).color$ is RED
III For each node x, all simple paths from x to descendant leaves contain the same number of BLACK nodes.
OK! hence, $bh(x)$ remains unchanged for all x

Part 3: Red-Black Trees (insert)



Notation in binary tree (left&right not important in this fig!)
uncle always exist (uncle = T.NIL possible)



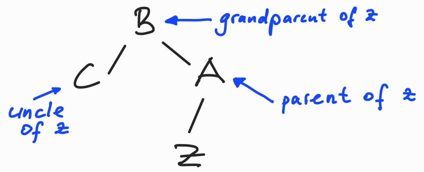
Scenarios where z needs some fix up:

- I.b z is the root (T was empty at start)
- II. parent of z is RED
  Then we distinguish:
  - i. uncle of z is RED
  - ii. uncle of z is BLACK (triangle)
  - iii. uncle of z is BLACK (line)

Which Red-Black properties could be violated?

- I.a Every node is either RED or BLACK. **OK!**
- I.b The root is BLACK OK, unless z is now root.
- I.c Every leaf ($T.nil$) is BLACK. **OK!**
- II If a node is RED, then each of its children must be BLACK
  violated if $(z.top).color$ is RED
- III For each node x, all simple paths from x to descendant leaves contain the same number of BLACK nodes.
  OK! hence, $bh(x)$ remains unchanged for all x

Part 3: Red-Black Trees (insert)



Notation in binary tree (left&right not important in this fig!)

Scenarios where z needs some fix up:

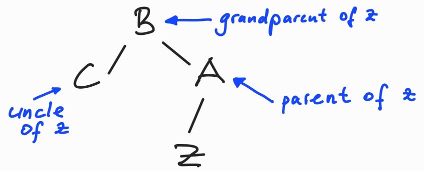
I.b z is the root (T was empty at start)

II parent of z is RED

Then we distinguish:

i. uncle of z is RED
ii. uncle of z is BLACK (triangle)
iii. uncle of z is BLACK (line)

Part 3: Red-Black Trees (insert)



Notation in binary tree (left&right not important in this fig!)

Scenarios where z needs some fix up:

I.b z is the root (T was empty at start)

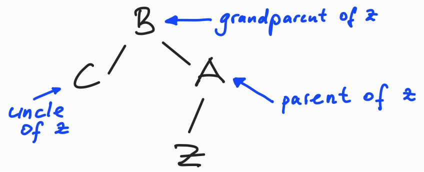
II parent of z is RED

Then we distinguish:

i. uncle of z is RED
ii. uncle of z is BLACK (triangle)
iii. uncle of z is BLACK (line)

Case (I.b): recolor z to BLACK $\Longrightarrow$ we get a valid Red-Black-tree with single root z

Part 3: Red-Black Trees (insert)



Notation in binary tree (left&right not important in this fig!)

Scenarios where z needs some fix up:

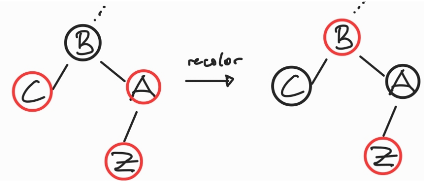
I.b z is the root (T was empty at start)

II parent of z is RED

Then we distinguish:

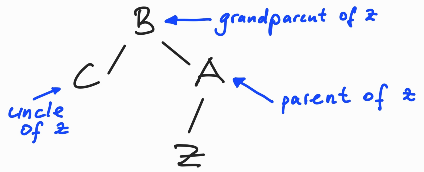
i. uncle of z is RED
ii. uncle of z is BLACK (triangle)
iii. uncle of z is BLACK (line)

Case (II.i):



Of course this may cause further violations if parent of B is red, but this will be corrected afterwards.

Part 3: Red-Black Trees (insert)



Notation in binary tree (left&right not important in this fig!)

Scenarios where z needs some fix up:

I.b z is the root (T was empty at start)

II parent of z is RED

Then we distinguish:

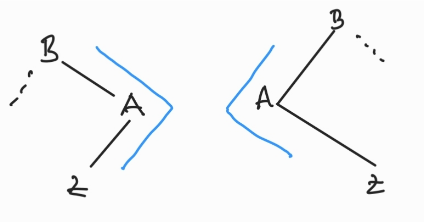
i. uncle of z is RED

ii. uncle of z is BLACK (triangle)

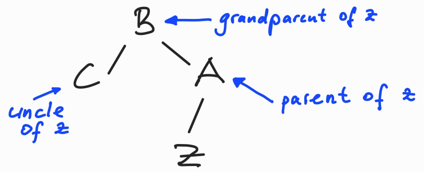
iii. uncle of z is BLACK (line)

Case (II.ii):

triangle: either A left of B and z right of A or A right of B and z left of A



Part 3: Red-Black Trees (insert)



Notation in binary tree (left&right not important in this fig!)

Scenarios where z needs some fix up:

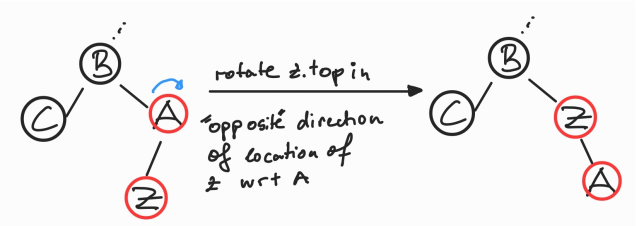
I.b z is the root (T was empty at start)

II parent of z is RED

Then we distinguish:

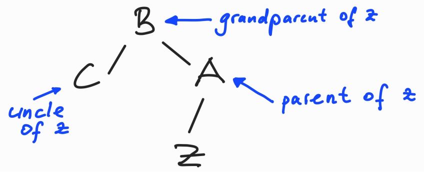
i. uncle of z is RED
ii. uncle of z is BLACK (triangle)
iii. uncle of z is BLACK (line)

Case (II.ii):



Still II (node is RED, children BLACK) is violated, but now we are in Case II.iii with A playing the role of z

Part 3: Red-Black Trees (insert)



Notation in binary tree (left&right not important in this fig!)

Scenarios where z needs some fix up:

I.b z is the root (T was empty at start)

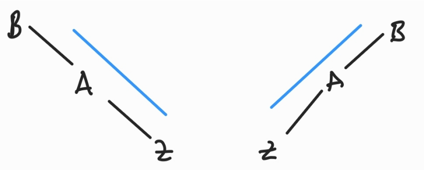
II parent of z is RED

Then we distinguish:

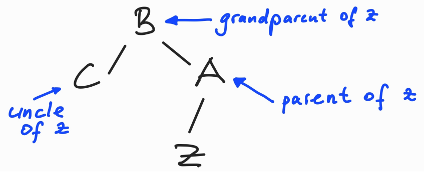
i. uncle of z is RED
ii. uncle of z is BLACK (triangle)
iii. uncle of z is BLACK (line)

Case (II.iii):

line: either A left of B and z left of A or A right of B and z left of A



Part 3: Red-Black Trees (insert)



Notation in binary tree (left&right not important in this fig!)

Scenarios where z needs some fix up:

I.b z is the root (T was empty at start)

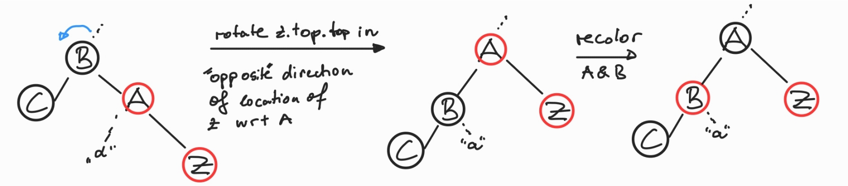
II parent of z is RED

Then we distinguish:

i. uncle of z is RED
ii. uncle of z is BLACK (triangle)
iii. uncle of z is BLACK (line)

Case (II.iii):

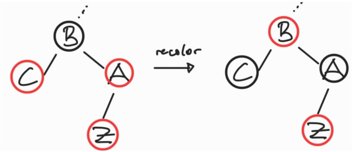
line: either *A* left of *B* and *z* left of *A* or *A* right of *B* and *z* left of *A*



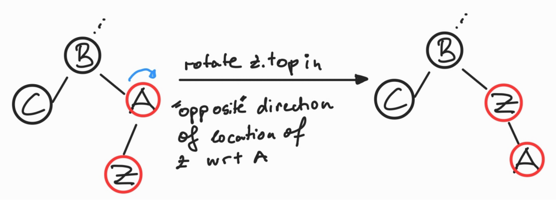
Part 3: Red-Black Trees (insert)

Case (I.b): recolor z to BLACK $\Longrightarrow$ we get a valid Red-Black-tree with single root z

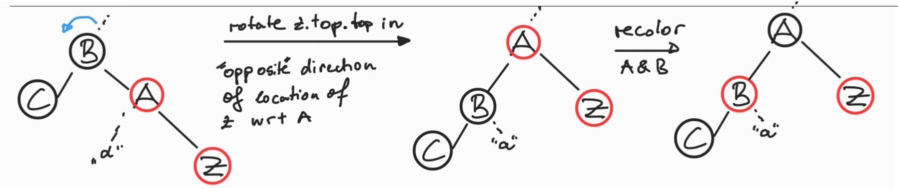
Case (II.i):



Case (II.ii):



Case (II.iii):



Working Example Board.

Part 3: Red-Black Trees (insert)

I.b z is the root $\Longrightarrow$ Re-color z

II parent of z is RED

i. uncle of z is RED
$\Longrightarrow$ Recolor + Repeat with "new z" (L.8)

ii. uncle of z is BLACK (triangle)
$\Longrightarrow$ Rotate parent of z +
Repeat with "new z" (L.10)

iii. uncle of z is BLACK (line)
$\Longrightarrow$ Rotate grandparent of z and recolor

To summarize in a nutshell:

Insert z as in usual BST and then `RB-Insert-Fixup`(T, t) which is based on 4 scenarios:

- I.b z is the root $\Longrightarrow$ Re-color z
- II parent of z is RED
  - i. uncle of z is RED
    $\Longrightarrow$ Recolor + Repeat with "new z" (L.8)
  - ii. uncle of z is BLACK (triangle)
    $\Longrightarrow$ Rotate parent of z +
    Repeat with "new z" (L.10)
  - iii. uncle of z is BLACK (line)
    $\Longrightarrow$ Rotate grandparent of z and recolor

Part 3: Red-Black Trees (insert)

To summarize in a nutshell:

Insert z as in usual BST and then `RB-Insert-Fixup`(T, t) which is based on 4 scenarios:

- I.b z is the root $\Longrightarrow$ Re-color z
- II parent of z is RED
  - i. uncle of z is RED
    $\Longrightarrow$ Recolor + Repeat with "new z" (L.8)
  - ii. uncle of z is BLACK (triangle)
    $\Longrightarrow$ Rotate parent of z +
    Repeat with "new z" (L.10)
  - iii. uncle of z is BLACK (line)
    $\Longrightarrow$ Rotate grandparent of z and recolor

To summarize in a nutshell:

Insert z as in usual BST and then `RB-Insert-Fixup`(T, t) which is based on 4 scenarios:

I.b z is the root $\Longrightarrow$ Re-color z

II parent of z is RED

i. uncle of z is RED
$\Longrightarrow$ Recolor + Repeat with "new z" (L.8)

ii. uncle of z is BLACK (triangle)
$\Longrightarrow$ Rotate parent of z +
Repeat with "new z" (L.10)

iii. uncle of z is BLACK (line)
$\Longrightarrow$ Rotate grandparent of z and recolor

Part 3: Red-Black Trees (insert)

To summarize in a nutshell:

Insert z as in usual BST and then `RB-Insert-Fixup`(T, t) which is based on 4 scenarios:

- I.b z is the root $\Longrightarrow$ Re-color z
- II parent of z is RED
  - i. uncle of z is RED
    $\Longrightarrow$ Recolor + Repeat with "new z" (L.8)
  - ii. uncle of z is BLACK (triangle)
    $\Longrightarrow$ Rotate parent of z +
    Repeat with "new z" (L.10)
  - iii. uncle of z is BLACK (line)
    $\Longrightarrow$ Rotate grandparent of z and recolor

Part 3: Red-Black Trees (insert)

To summarize in a nutshell:

Insert z as in usual BST and then `RB-Insert-Fixup`(T, t) which is based on 4 scenarios:

- I.b z is the root $\implies$ Re-color z
- II parent of z is RED
  - i. uncle of z is RED
    $\implies$ Recolor + Repeat with "new z" (L.8)
  - ii. uncle of z is BLACK (triangle)
    $\implies$ Rotate parent of z +
    Repeat with "new z" (L.10)
  - iii. uncle of z is BLACK (line)
    $\implies$ Rotate grandparent of z and recolor

Part 3: Red-Black Trees (insert)

To summarize in a nutshell:

Insert z as in usual BST and then `RB-Insert-Fixup`(T, t) which is based on 4 scenarios:

I.b z is the root $\Longrightarrow$ Re-color z

II parent of z is RED

i. uncle of z is RED
$\Longrightarrow$ Recolor + Repeat with "new z" (L.8)

ii. uncle of z is BLACK (triangle)
$\Longrightarrow$ Rotate parent of z +
Repeat with "new z" (L.10)

iii. uncle of z is BLACK (line)
$\Longrightarrow$ Rotate grandparent of z and recolor

Part 3: Red-Black Trees (insert)

To summarize in a nutshell:

Insert z as in usual BST and then `RB-Insert-Fixup`(T, t) which is based on 4 scenarios:

- I.b z is the root $\implies$ Re-color z
- II parent of z is RED
  - i. uncle of z is RED
    $\implies$ Recolor + Repeat with "new z" (L.8)
  - ii. uncle of z is BLACK (triangle)
    $\implies$ Rotate parent of z +
    Repeat with "new z" (L.10)
  - iii. uncle of z is BLACK (line)
    $\implies$ Rotate grandparent of z and recolor

RB-INSERT-FIXUP(T, z)

```
1  while z.p.color == RED
2      if z.p == z.p.p.left
3          y = z.p.p.right
4          if y.color == RED
5  //case II.i       z.p.color = BLACK
6  //case II.i       y.color = BLACK
7  //case II.i       z.p.p.color = RED
8  //case II.i       z = z.p.p
9          else if z == z.p.right
10 //case II.ii          z = z.p
11 //case II.ii          LEFT-ROTATE(T, z)
12 //case II.iii      z.p.color = BLACK
13 //case II.iii      z.p.p.color = RED
14 //case II.iii      RIGHT-ROTATE(T, z.p.p)
15     else (same as then clause
                   with "right" and "left" exchanged)
16 T.root.color = BLACK //case I.b
```

$v.p$ means parent of v (=$v.top$)

Souce: Introduction to Algorithms (3rd edition), Cormen

Part 3: Red-Black Trees (insert)

To summarize in a nutshell:

Insert z as in usual BST and then `RB-Insert-Fixup`(T, t) which is based on 4 scenarios:

- I.b z is the root $\Longrightarrow$ Re-color z
- II parent of z is RED
  - i. uncle of z is RED
    $\Longrightarrow$ Recolor + Repeat with "new z" (L.8)
  - ii. uncle of z is BLACK (triangle)
    $\Longrightarrow$ Rotate parent of z +
    Repeat with "new z" (L.10)
  - iii. uncle of z is BLACK (line)
    $\Longrightarrow$ Rotate grandparent of z and recolor

Theorem. *Insertion of elements into Red-Black tree while maintaining Red-Black properties can be done $O(\log(n))$ time*

```
RB-INSERT-FIXUP(T, z)
1  while z.p.color == RED
2      if z.p == z.p.p.left
3          y = z.p.p.right
4          if y.color == RED
5  //case II.i      z.p.color = BLACK
6  //case II.i      y.color = BLACK
7  //case II.i      z.p.p.color = RED
8  //case II.i      z = z.p.p
9          else if z == z.p.right
10 //case II.ii         z = z.p
11 //case II.ii         LEFT-ROTATE(T, z)
12 //case II.iii    z.p.color = BLACK
13 //case II.iii    z.p.p.color = RED
14 //case II.iii    RIGHT-ROTATE(T, z.p.p)
15     else (same as then clause
                  with "right" and "left" exchanged)
16 T.root.color = BLACK //case I.b
```

$v.p$ means parent of v (=$v.top$)

Souce: Introduction to Algorithms (3rd edition), Cormen

Part 3: Red-Black Trees (insert)

To summarize in a nutshell:

Insert z as in usual BST and then `RB-Insert-Fixup`(T, t) which is based on 4 scenarios:

- I.b z is the root $\Longrightarrow$ Re-color z
- II parent of z is RED
  - i. uncle of z is RED
    $\Longrightarrow$ Recolor + Repeat with "new z" (L.8)
  - ii. uncle of z is BLACK (triangle)
    $\Longrightarrow$ Rotate parent of z + Repeat with "new z" (L.10)
  - iii. uncle of z is BLACK (line)
    $\Longrightarrow$ Rotate grandparent of z and recolor

Theorem. *Insertion of elements into Red-Black tree while maintaining Red-Black properties can be done $O(\log(n))$ time*

Proof. Correctness of `RB-Insert`(T, z) and `RB-Insert-Fixup`(T, z), see Sec 13.3 in Cormen-course-book for more details.

```
RB-INSERT-FIXUP(T, z)
1  while z.p.color == RED
2      if z.p == z.p.p.left
3          y = z.p.p.right
4          if y.color == RED
5 //case II.i      z.p.color = BLACK
6 //case II.i      y.color = BLACK
7 //case II.i      z.p.p.color = RED
8 //case II.i      z = z.p.p
9          else if z == z.p.right
10 //case II.ii          z = z.p
11 //case II.ii          LEFT-ROTATE(T, z)
12 //case II.iii     z.p.color = BLACK
13 //case II.iii     z.p.p.color = RED
14 //case II.iii     RIGHT-ROTATE(T, z.p.p)
15     else (same as then clause
                with "right" and "left" exchanged)
16 T.root.color = BLACK //case I.b
```

$v.p$ means parent of v (=$v.top$)

Souce: Introduction to Algorithms (3rd edition), Cormen

Part 3: Red-Black Trees (insert)

To summarize in a nutshell:

Insert z as in usual BST and then `RB-Insert-Fixup`(T, t) which is based on 4 scenarios:

- I.b z is the root $\Longrightarrow$ Re-color z
- II parent of z is RED
  - i. uncle of z is RED
    $\Longrightarrow$ Recolor + Repeat with "new z" (L.8)
  - ii. uncle of z is BLACK (triangle)
    $\Longrightarrow$ Rotate parent of z +
    Repeat with "new z" (L.10)
  - iii. uncle of z is BLACK (line)
    $\Longrightarrow$ Rotate grandparent of z and recolor

Theorem. *Insertion of elements into Red-Black tree while maintaining Red-Black properties can be done $O(\log(n))$ time*

Proof. Correctness of `RB-Insert`(T, z) and `RB-Insert-Fixup`(T, z), see Sec 13.3 in Cormen-course-book for more details.

`RB-Insert`(T, z) runs in $O(h(T)) = O(\log(n))$ time.

```
RB-INSERT-FIXUP(T, z)
 1   while z.p.color == RED
 2       if z.p == z.p.p.left
 3           y = z.p.p.right
 4           if y.color == RED
 5 //case II.i       z.p.color = BLACK
 6 //case II.i       y.color = BLACK
 7 //case II.i       z.p.p.color = RED
 8 //case II.i       z = z.p.p
 9           else if z == z.p.right
10 //case II.ii          z = z.p
11 //case II.ii          LEFT-ROTATE(T, z)
12 //case II.iii       z.p.color = BLACK
13 //case II.iii       z.p.p.color = RED
14 //case II.iii       RIGHT-ROTATE(T, z.p.p)
15       else (same as then clause
                 with "right" and "left" exchanged)
16   T.root.color = BLACK //case I.b
```

$v.p$ means parent of v (=$v.top$)

Souce: Introduction to Algorithms (3rd edition), Cormen

Part 3: Red-Black Trees (insert)

To summarize in a nutshell:

Insert z as in usual BST and then `RB-Insert-Fixup`(T, t) which is based on 4 scenarios:

- I.b z is the root $\Longrightarrow$ Re-color z
- II parent of z is RED
  - i. uncle of z is RED
    $\Longrightarrow$ Recolor + Repeat with "new z" (L.8)
  - ii. uncle of z is BLACK (triangle)
    $\Longrightarrow$ Rotate parent of z + Repeat with "new z" (L.10)
  - iii. uncle of z is BLACK (line)
    $\Longrightarrow$ Rotate grandparent of z and recolor

Theorem. *Insertion of elements into Red-Black tree while maintaining Red-Black properties can be done $O(\log(n))$ time*

Proof. Correctness of `RB-Insert`(T, z) and `RB-Insert-Fixup`(T, z), see Sec 13.3 in Cormen-course-book for more details.

`RB-Insert`(T, z) runs in $O(h(T)) = O(\log(n))$ time.
`RB-Insert-Fixup`(T, z): for each z constant "reassignments" of pointers.
All "new.z" that might cause conflicts and need to be fixed up are ancestors of the "original z" $\Longrightarrow$ While-loop executions: $O(\log(n))$.

RB-INSERT-FIXUP(T, z)

```
1  while z.p.color == RED
2      if z.p == z.p.p.left
3          y = z.p.p.right
4          if y.color == RED
5  //case II.i      z.p.color = BLACK
6  //case II.i      y.color = BLACK
7  //case II.i      z.p.p.color = RED
8  //case II.i      z = z.p.p
9          else if z == z.p.right
10 //case II.ii          z = z.p
11 //case II.ii          LEFT-ROTATE(T, z)
12 //case II.iii     z.p.color = BLACK
13 //case II.iii     z.p.p.color = RED
14 //case II.iii     RIGHT-ROTATE(T, z.p.p)
15     else (same as then clause
               with "right" and "left" exchanged)
16 T.root.color = BLACK //case I.b
```

$v.p$ means parent of v (=$v.top$)

Souce: Introduction to Algorithms (3rd edition), Cormen

Part 3: Red-Black Trees (insert)

To summarize in a nutshell:

Insert z as in usual BST and then `RB-Insert-Fixup`(T, t) which is based on 4 scenarios:

- I.b z is the root $\Longrightarrow$ Re-color z
- II parent of z is RED
  - i. uncle of z is RED
    $\Longrightarrow$ Recolor + Repeat with "new z" (L.8)
  - ii. uncle of z is BLACK (triangle)
    $\Longrightarrow$ Rotate parent of z + Repeat with "new z" (L.10)
  - iii. uncle of z is BLACK (line)
    $\Longrightarrow$ Rotate grandparent of z and recolor

Theorem. *Insertion of elements into Red-Black tree while maintaining Red-Black properties can be done $O(\log(n))$ time*

Proof. Correctness of `RB-Insert`(T, z) and `RB-Insert-Fixup`(T, z), see Sec 13.3 in Cormen-course-book for more details.

`RB-Insert`(T, z) runs in $O(h(T)) = O(\log(n))$ time.
`RB-Insert-Fixup`(T, z): for each z constant "reassignments" of pointers.
All "new.z" that might cause conflicts and need to be fixed up are ancestors of the "original z" $\Longrightarrow$ While-loop executions: $O(\log(n))$.

RB-INSERT-FIXUP(T, z)

```
1  while z.p.color == RED
2      if z.p == z.p.p.left
3          y = z.p.p.right
4          if y.color == RED
5  //case II.i     z.p.color = BLACK
6  //case II.i     y.color = BLACK
7  //case II.i     z.p.p.color = RED
8  //case II.i     z = z.p.p
9          else if z == z.p.right
10 //case II.ii        z = z.p
11 //case II.ii        LEFT-ROTATE(T, z)
12 //case II.iii    z.p.color = BLACK
13 //case II.iii    z.p.p.color = RED
14 //case II.iii    RIGHT-ROTATE(T, z.p.p)
15     else (same as then clause
              with "right" and "left" exchanged)
16 T.root.color = BLACK  //case I.b
```

$v.p$ means parent of v (=$v.top$)

Souce: Introduction to Algorithms (3rd edition), Cormen

Deletion is much more involved and omitted here, see Sec 13.4 in Cormen-course-book for more details.

Search-Trees: Summary

We considered the class of binary search trees (BST).
In particular, we had a closer look to the subclasses:

- AVL trees
- Red-Black trees

Question: when using AVL tree, when Red-Black trees?

Since the invention of AVL trees in 1962 and Red-black trees in 1978, researchers were divided in two separated communities, AVL supporters and Red-Black ones.

Often, AVL trees are used for retrieval applications (Search Engines, Database queries) whereas Red Black trees are used in updates operation (insertion, replace information)

Worst case	**AVL**	**Red-Black**
Height	1.44 Log (n)	2 log (n+1)
Updates complexity	O(Log (n))	O(Log (n))
Retrieval Complexity	O(Log (n))	O(Log (n))
Rotations for insert	2	2
Rotations for delete	Log (n)	3

* AVL and Red-Black tree as a single balanced tree, Bounif and Zegour, Proc. of the Fourth Intl. Conf. Advances in Computing, Communication and Information Technology, 2016