Algorithms (COMP3600/6466)
Data Structures: Binary Search Trees

Pascal Bercher

(working in the Intelligent Systems Cluster)

School of Computing
The Australian National University

Tuesday, 29.8.2023

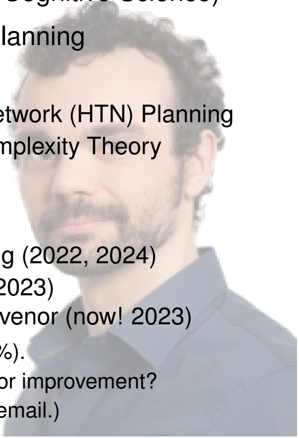
Australian
National
University

Orga

About me!

(Dr.) Pascal (Bercher) (Co-Convenor)

`https://comp.anu.edu.au/people/pascal-bercher/`



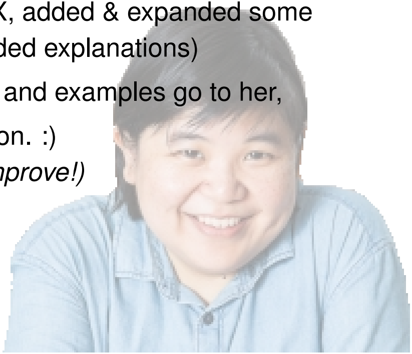
- *Studies:* Computer Science (with minor Cognitive Science)
- *PhD:* Computer Science: Hierarchical Planning
- *Research:*
  - problem classes: Hierarchical Task Network (HTN) Planning
  - research areas: Heuristic Search, Complexity Theory
- *Teaching:*
  - Convenor of Logic (2021, 2022)
  - Convenor of Foundations of Computing (2022, 2024)
  - Convenor of Theory of Computation (2023)
  - Lecturer of Algorithms (2021), Co-convenor (now! 2023)
    - I will teach weeks 6 to 8 and 12 (33%).
    - Spotted errors in the slides? Ideas for improvement?
      $\longrightarrow$ ***please, let me know!*** (Drop an email.)

About my Slides

Dr. Hanna Kurniawati (past Convenor) 2019–2022

`https://comp.anu.edu.au/people/hanna-kurniawati/`

- My slides are based significantly on material by Hanna (I just converted slides to LaTeX, added & expanded some examples, and sometimes added explanations)
- So the credit for good content and examples go to her,
- and blame me for bad execution. :)
  (But recall to tell me how to improve!)



Introduction

Motivation

What do we want to achieve? Low runtime (average and worst case) for any of the typical data management operations:

- insertion
- deletion
- access (i.e., more general search)
- min/max (i.e., specialized search)

Motivation

What do we want to achieve? Low runtime (average and worst case) for any of the typical data management operations:

- insertion
- deletion
- access (i.e., more general search)
- min/max (i.e., specialized search)

What does *data* mean? *Anything*, but represented by Integers! Why?

- They serve as keys to reference the *actual* (satellite) data.
- Keys for some data might be derived from identifiers like personal names, date of birth, production year, etc.
- One application of storing and processing data is AI planning and AI search. The key of a search node might be the goal distance (i.e., heuristic value), and its data the actual content (e.g., “state”).

Abstract Data Structures

How to achieve low runtime? Via suitable abstract data structures.

- Abstract data structures can be thought of as a mathematical model for data representation.
- An abstract data structure consists of:
  - A container that holds a key as well as the data, so-called satellite data (can be ignored for our purposes).
  - A set of operations on the data (cf. previous slide). These operations are defined based on their *behavior* (input/output relation and their runtime) rather than by any exact implementation.

Overview of Covered Data Structures

Data structures covered:

- Binary Search Trees (today, week 6)
- Heaps (also today, week 6)
- AVL Trees (tomorrow, week 6)
- Red/Black Trees (week 7)
- Hash Tables (week 8)

Which operations should be supported by *Binary Search Trees*?

- Search
  - $\rightarrow$ Does the given key exist in the tree?
- List all data
  - $\rightarrow$ List all existing keys in a specific order.
- Min, Max
  - $\rightarrow$ What's the minimum (resp. maximum) key in the tree?
- Successor, Predecessor
  - $\rightarrow$ What's the smallest value greater than the given key in the tree?
  - $\rightarrow$ What's the highest value smaller than the given key in the tree?
- Insert, Delete
  - $\rightarrow$ Insert the given key into the tree, or remove it from it.

All these operations except "list all data" should run in $O(h)$, where h is the height of the search tree.

Basics

Binary Search Tree

A *binary search tree* is a tree $\mathcal{T} = (N, E)$ consisting of a finite set of nodes N and edges $E \subseteq N \times N$, such that each node has at most two children, i.e., for all $x \in N$ holds $|\{x' \in N \mid x \in N, (x, x') \in E\}| \leq 2$.

Binary Search Tree

A *binary search tree* is a tree $\mathcal{T} = (N, E)$ consisting of a finite set of nodes N and edges $E \subseteq N \times N$, such that each node has at most two children, i.e., for all $x \in N$ holds $|\{x' \in N \mid x \in N, (x, x') \in E\}| \leq 2$.

Implementation and Notation:

- Each node $x \in N$ has four values:
  - Its key $x.key$, (note: all keys are distinct!)
  - parent $x.p$,
  - left child $x.left$, and right child $x.right$
- The root node is the only one without parent, $x.p = NIL$

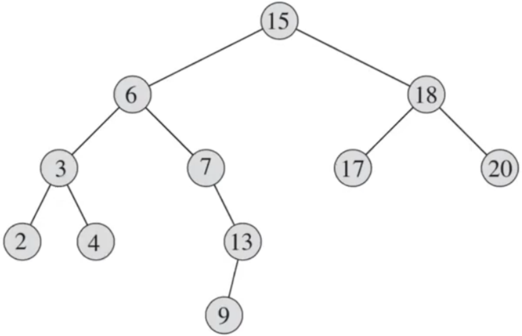
Binary Search Tree

A *binary search tree* is a tree $\mathcal{T} = (N, E)$ consisting of a finite set of nodes N and edges $E \subseteq N \times N$, such that each node has at most two children, i.e., for all $x \in N$ holds $|\{x' \in N \mid x \in N, (x, x') \in E\}| \leq 2$.

Implementation and Notation:

- Each node $x \in N$ has four values:
  - Its key $x.key$, (note: all keys are distinct!)
  - parent $x.p$,
  - left child $x.left$, and right child $x.right$
- The root node is the only one without parent, $x.p = NIL$
- With h we refer to the *height* of the tree, i.e., the length of the longest path (number of edges). (Also referred to as depth.)
- With n we refer to the number of nodes $|N|$.
- For all $x \in N$ holds:
  - For all x' in the tree rooted in $x.left$ holds $x'.key < x.key$
  - For all x' in the tree rooted in $x.right$ holds $x'.key > x.key$

Binary Search Tree, Example



Here you see:

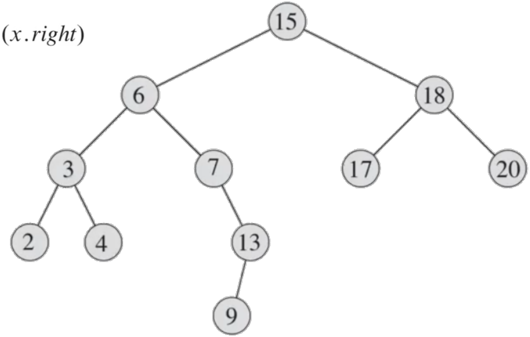
- This is a binary tree.
- All keys are sorted with respect to their parent (=left to right!).
- It has a height of 4.

In-Order Tree Walk

Procedure

Binary search trees allow us to output all keys in sorted order in $\Theta(n)$ via *In-Order Tree Walk*: (called with *T.root*)

```
INORDER-TREE-WALK(x)
if x ≠ NIL
    INORDER-TREE-WALK(x.left)
    print x.key
    INORDER-TREE-WALK(x.right)
```



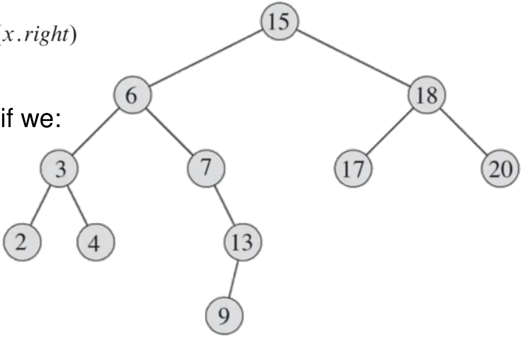
Procedure

Binary search trees allow us to output all keys in sorted order in $\Theta(n)$ via *In-Order Tree Walk*: (called with *T.root*)

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.left)
3      print x.key
4      INORDER-TREE-WALK(x.right)
```

Homework: What happens if we:

- switch lines 2 & 3?
  (pre-order tree walk)
- switch lines 3 & 4?
  (post-order tree walk)



Runtime Complexity

Let $T(n)$ be the runtime of this algorithm. Want to show $T(n) \in \Theta(n)$.

- $T(n) \in \Omega(n)$ since each node is visited at least once.
- We show $T(n) \in O(n)$ (via induction).

Can assume $T(n) = T(k) + T(n - k - 1) + 1$, where k is the number of nodes in the root's left subtree and $T(1) = 1$.

Runtime Complexity

Let $T(n)$ be the runtime of this algorithm. Want to show $T(n) \in \Theta(n)$.

- $T(n) \in \Omega(n)$ since each node is visited at least once.
- We show $T(n) \in O(n)$ (via induction).

Can assume $T(n) = T(k) + T(n - k - 1) + 1$, where k is the number of nodes in the root's left subtree and $T(1) = 1$.

Induction hypothesis:

Runtime Complexity

Let $T(n)$ be the runtime of this algorithm. Want to show $T(n) \in \Theta(n)$.

- $T(n) \in \Omega(n)$ since each node is visited at least once.
- We show $T(n) \in O(n)$ (via induction).

Can assume $T(n) = T(k) + T(n-k-1) + 1$, where k is the number of nodes in the root's left subtree and $T(1) = 1$.

Induction hypothesis:
$T(n) \leq c \cdot n$ with constant $c \geq 1$.

Runtime Complexity

Let $T(n)$ be the runtime of this algorithm. Want to show $T(n) \in \Theta(n)$.

- $T(n) \in \Omega(n)$ since each node is visited at least once.
- We show $T(n) \in O(n)$ (via induction).

Can assume $T(n) = T(k) + T(n-k-1) + 1$, where k is the number of nodes in the root's left subtree and $T(1) = 1$.

Induction hypothesis:
$T(n) \leq c \cdot n$ with constant $c \geq 1$.

Base case:
$T(1) = 1 \leq c \cdot 1$.

Runtime Complexity

Let $T(n)$ be the runtime of this algorithm. Want to show $T(n) \in \Theta(n)$.

- $T(n) \in \Omega(n)$ since each node is visited at least once.
- We show $T(n) \in O(n)$ (via induction).

Can assume $T(n) = T(k) + T(n-k-1) + 1$, where k is the number of nodes in the root's left subtree and $T(1) = 1$.

Induction hypothesis:
$T(n) \leq c \cdot n$ with constant $c \geq 1$.

Base case:
$T(1) = 1 \leq c \cdot 1$.

Induction step:
$T(n+1) = T(k) + T(n+1-k-1) + 1 = T(k) + T(n-k) + 1$
$\leq (c \cdot k) + (c \cdot (n-k)) + 1 = c \cdot n + 1 \leq c \cdot (n+1)$.

Runtime Complexity

Let $T(n)$ be the runtime of this algorithm. Want to show $T(n) \in \Theta(n)$.

- $T(n) \in \Omega(n)$ since each node is visited at least once.
- We show $T(n) \in O(n)$ (via induction).

Can assume $T(n) = T(k) + T(n-k-1) + 1$, where k is the number of nodes in the root's left subtree and $T(1) = 1$.

Induction hypothesis:
$T(n) \leq c \cdot n$ with constant $c \geq 1$.

Base case:
$T(1) = 1 \leq c \cdot 1$. (Alt.: $T(1) = 1 \in O(1) \in O(n)$)

Induction step: (Alt.: $T(n+1) = O(k) + O(n-k) + O(1) \in O(n)$)
$T(n+1) = T(k) + T(n+1-k-1) + 1 = T(k) + T(n-k) + 1$
$\leq (c \cdot k) + (c \cdot (n-k)) + 1 = c \cdot n + 1 \leq c \cdot (n+1)$.

Access

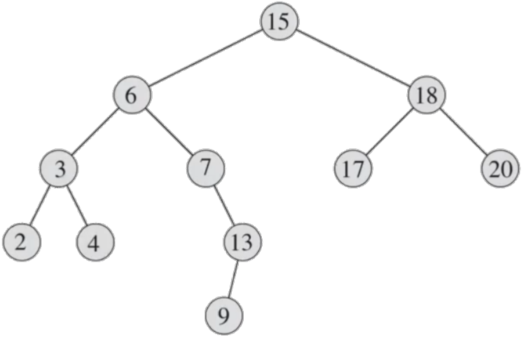
Iterative-Tree Search

To search for a key whose value is k we can use *Iterative-Tree Search*:

(Start with x being the root node *T.root*.)

```
ITERATIVE-TREE-SEARCH(x, k)
while x ≠ NIL and k ≠ x.key
    if k < x.key
        x = x.left
    else x = x.right
return x
```

(Might return NIL.)



Min & Max

Tree-Minimum and Tree-Maximum

To find the minimum, respective maximum:

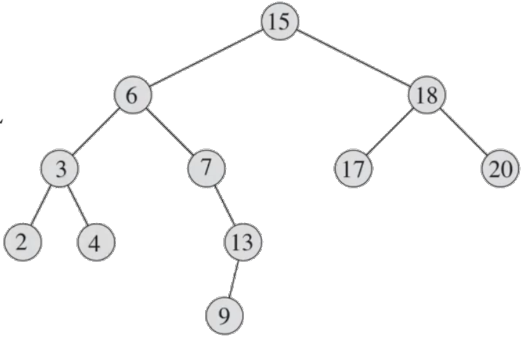
TREE-MINIMUM(x)

```
while x.left ≠ NIL
    x = x.left
return x
```

TREE-MAXIMUM(x)

```
while x.right ≠ NIL
    x = x.right
return x
```

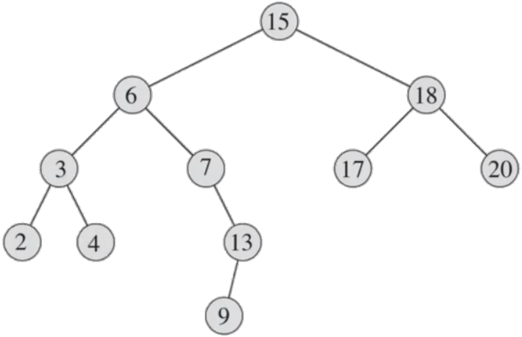
Again: Call with *T.root*



Predecessor and Successor

Predecessor

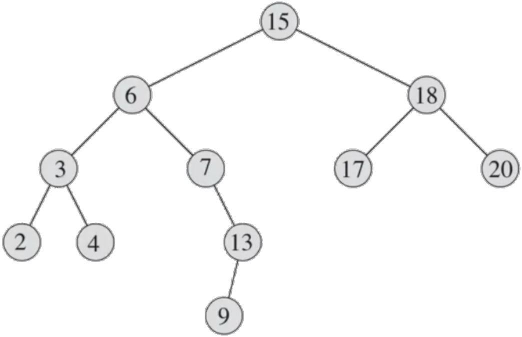
To find the predecessor of k, find node x with $x.key = k$. Then,



Predecessor

To find the predecessor of k, find node x with $x.key = k$. Then,

- predecessor of k is the maximum in the subtree rooting in $x.left$.
  $\rightarrow$ For example, predecessor of 15 is 13.

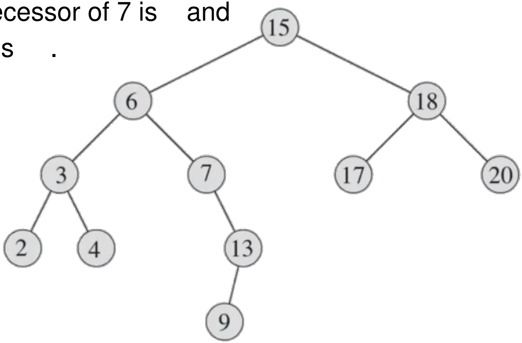


Predecessor

To find the predecessor of k, find node x with $x.key = k$. Then,

- predecessor of k is the maximum in the subtree rooting in $x.left$.
  $\rightarrow$ For example, predecessor of 15 is 13.
- if that subtree is empty, k's predecessor is the lowest ancestor of x whose right child is also an ancestor of x.
  $\rightarrow$ For example, predecessor of 7 is and
  $\rightarrow$ predecessor of 17 is .

Each node is an ancestor of itself.

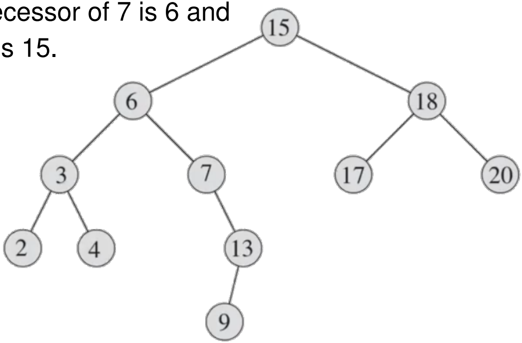


Predecessor

To find the predecessor of k, find node x with $x.key = k$. Then,

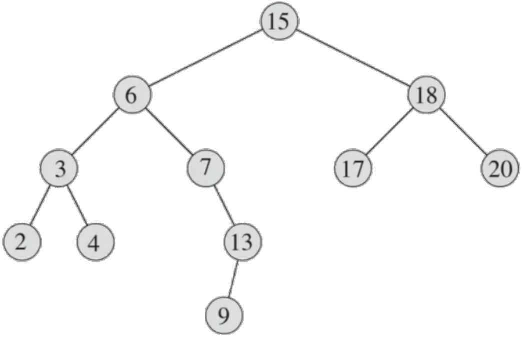
- predecessor of k is the maximum in the subtree rooting in $x.left$.
  $\rightarrow$ For example, predecessor of 15 is 13.
- if that subtree is empty, k's predecessor is the lowest ancestor of x whose right child is also an ancestor of x.
  $\rightarrow$ For example, predecessor of 7 is 6 and
  $\rightarrow$ predecessor of 17 is 15.

Each node is an ancestor of itself.



Successor

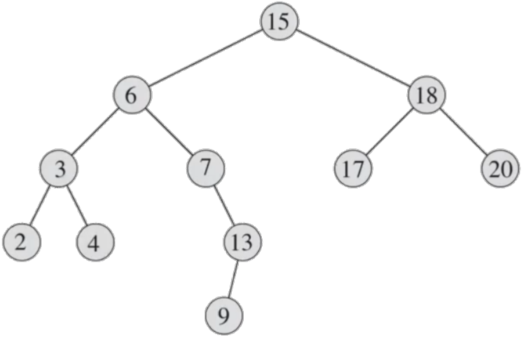
To find the successor of k, find node x with $x.key = k$. Then,



Successor

To find the successor of k, find node x with $x.key = k$. Then,

- successor of k is the minimum in the subtree rooting in $x.right$.
  $\rightarrow$ For example, successor of 15 is 17.

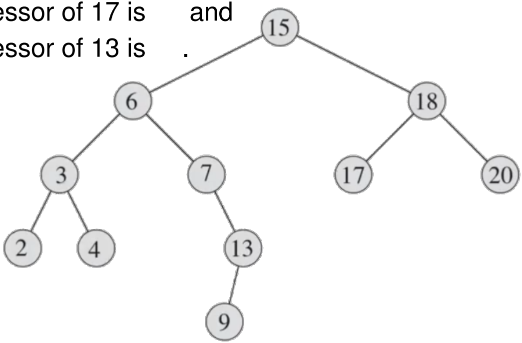


Successor

To find the successor of k, find node x with $x.key = k$. Then,

- successor of k is the minimum in the subtree rooting in $x.right$.
  $\rightarrow$ For example, successor of 15 is 17.
- if that subtree is empty, k's successor is the lowest ancestor of x whose left child is also an ancestor of x.
  $\rightarrow$ For example, successor of 17 is and
  $\rightarrow$ For example, successor of 13 is .

Each node is an ancestor of itself.

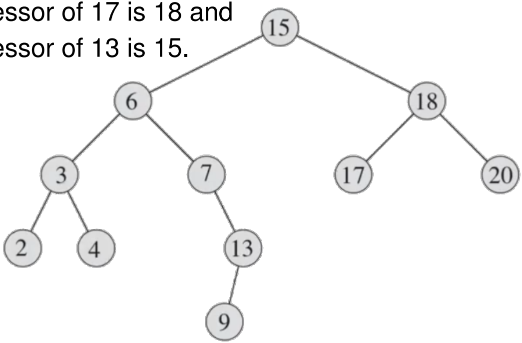


Successor

To find the successor of k, find node x with $x.key = k$. Then,

- successor of k is the minimum in the subtree rooting in $x.right$.
  $\rightarrow$ For example, successor of 15 is 17.
- if that subtree is empty, k's successor is the lowest ancestor of x whose left child is also an ancestor of x.
  $\rightarrow$ For example, successor of 17 is 18 and
  $\rightarrow$ For example, successor of 13 is 15.

Each node is an ancestor of itself.



Insertion and Deletion

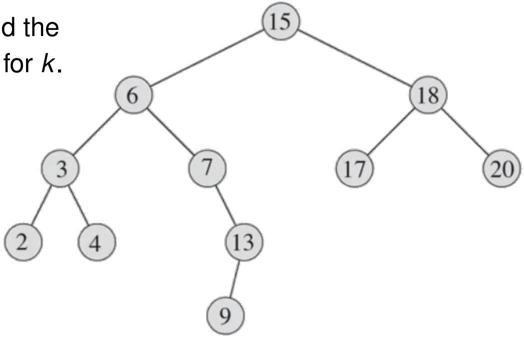
Tree-Insert

New values always get inserted as leaves.

Suppose we want to insert a key k using a node called z. Thus, $z.key = k$, $z.left = NIL$, $z.right = NIL$.

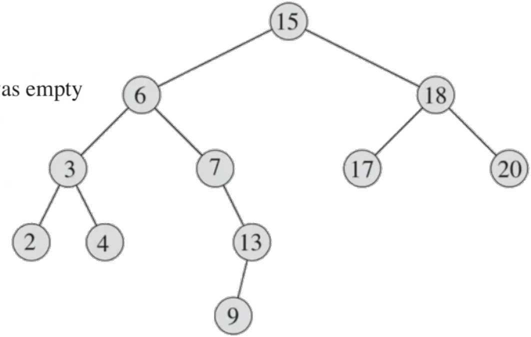
How to insert?

- Traverse the tree to find the correct (leaf!) position for k.
- Add z to the tree.



Tree-Insert, cont'd

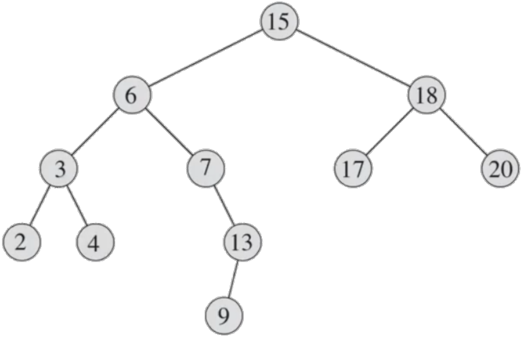
```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```



Deletion

To delete node z with key k, we have three cases:

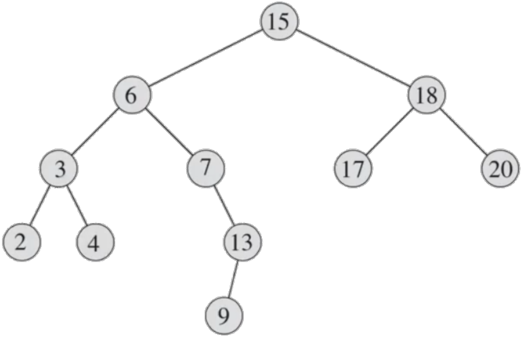
1. If z has no children: Remove z and modify its parent to replace z with NIL.



Deletion

To delete node z with key k, we have three cases:

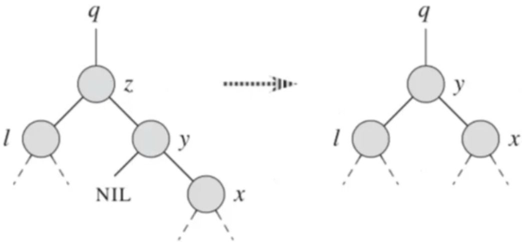
2. If z has one child: Elevate the child to take z's position in the tree by modifying its parent to replace z with z's child.



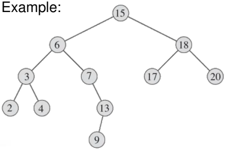
Deletion

To delete node z with key k, we have three cases:

3. If z has two children: Consider its successor y (y must be in z's right sub tree and it does not have a left child).
   - If y is z's right child, replace z with y. (z's left child becomes y's left child, and y replaced z's position.)



Example:

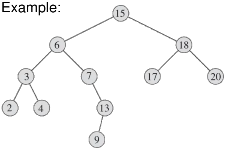


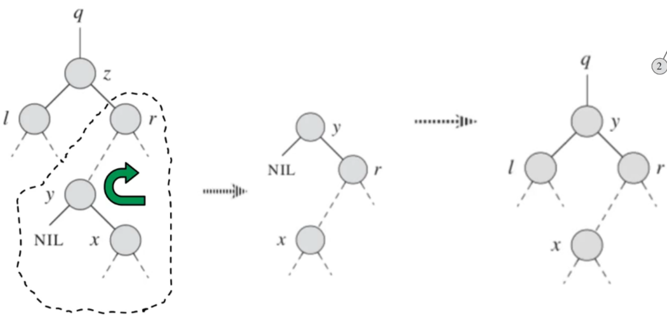
Deletion

To delete node z with key k, we have three cases:

3. If z has two children: Consider its successor y (y must be in z's right sub tree and it does not have a left child).
   - If y is not z's right child, first replace y by its right child an then replace z with y.

Example:





Deletion

To delete node z with key k, we have three cases:

1. If z has no children: Remove z and modify its parent to replace z with NIL.
2. If z has one child: Elevate the child to take z's position in the tree by modifying its parent to replace z with z's child.
3. If z has two children: Consider its successor y (y must be in z's right sub tree and it does not have a left child).
   - If y is z's right child, replace z with y. (z's left child becomes y's left child, and y replaced z's position.)
   - If y is not z's right child, first replace y by its right child an then replace z with y.

   $\rightarrow$ The requirement to find a successor causes deletion to take $O(h)$ rather than constant time.

Properties

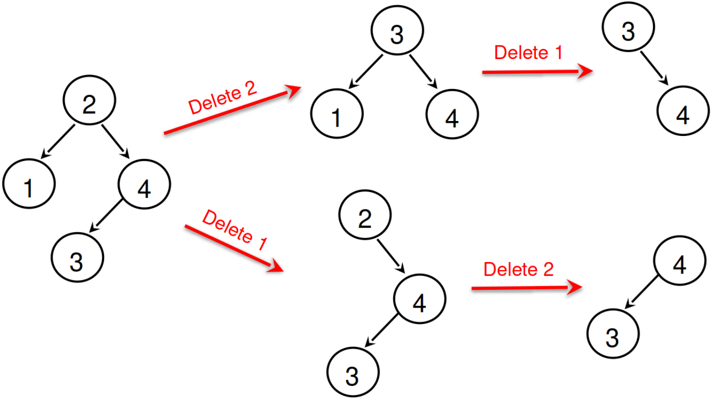
Are insertion resp. deletion commutative?

I.e., do these operations always lead to the same result, no matter of the order? For example,

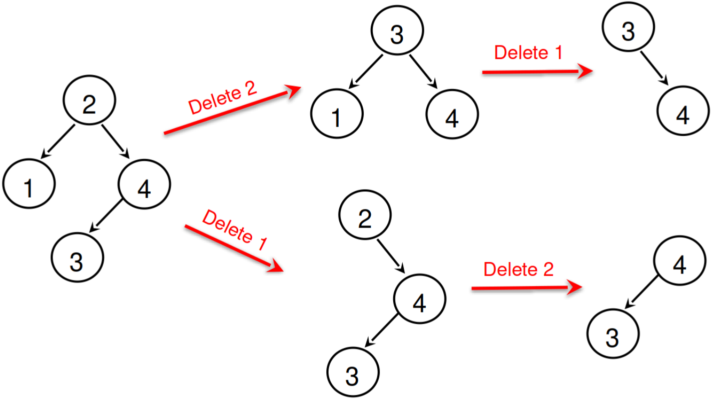
- does inserting “a, then b” always lead the same result
- as inserting “b, then a”?

(Same for deletion.)

Properties, cont'd



Properties, cont'd



So, no! Not commutative. Insertion: *homework!*

(Also make sure you can perform above's deletions yourself!)

Summary

Summary

Today we covered **Binary Search Trees**.

Operations considered:

- Inorder-Tree-Walk (plus its runtime analysis)
- Iterative-Tree-Search
- Tree-Minimum and Tree-Maximum
- Tree-Insert
- Deletion (only with textual description, no pseudo code)