# Data Structures and Algorithms: Lecture 13

Barbara Morawska

October 25, 2018

# Binary Search Trees

Implementation of a dynamic set as a tree with the following operations supported:

- SEARCH
- MINIMUM
- MAXIMUM
- PREDECESSOR
- SUCCESSOR
- INSERT

Running time for these operations is proportionate to the height of a tree. Hence notice:

- If the tree is a complete binary tree with $n$-nodes: $O(\lg n)$
- In the worst case, the tree is a chain of nodes: $O(n)$

# What is a binary-search tree?

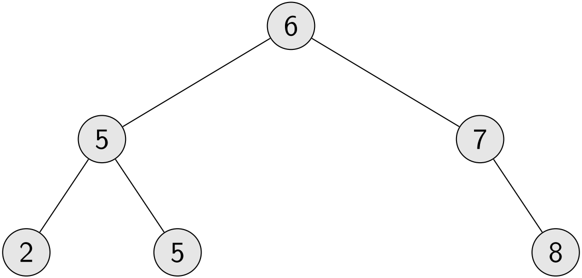
It is a binary tree which:

- can be implemented as a linked data
- each node contains key and satellite data
- each node has attributes: left, right, p
- if a node does not have a left child, a right child or a parent, it has the attribute pointing to NIL
- the root is the only node that has the parent attribute set to NIL.

# Binary-search-tree property

## Let $T$ be a binary tree

- Let $x, y$ be nodes in $T$.
- 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 $y.key \geq x.key$.



# How to print out all keys in a sorted order?

- **inorder tree walk**, prints values of:
  the root, the left subtree, the right subtree
- **preorder tree walk**, prints values of:
  the root, left subtree, the right subtree
- **postorder tree walk**, prints the values of:
  the left subtree, the right subtree, the root.

# Inorder tree walk

**Procedure** INORDER-TREE-WALK($x$)

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

Running time: $\Theta(n)$.

## Theorem

*If $x$ is the root of an $n$-node subtree, then the call* INORDER-TREE-WALK($x$) *takes $\Theta(n)$ time.*

## Proof

We have to show:

- $T(n) = \Omega(n)$
- $T(n) = O(n)$

# INORDER-TREE-WALK($x$) takes $\Theta(n)$ time

- Since all nodes in the tree are visited: $T(n) = \Omega(n)$
- We have to show $T(n) = O(n)$
- For empty tree INORDER-TREE-WALK takes constant time, $T(0) = c$
- Hence assume $n > 0$
- Let INORDER-TREE-WALK be called on a node $x$:
  - left subtree has $k$ nodes
  - right subtree has $n - k - 1$ nodes (1 is for the root).
- Hence the recurrence is:

$$T(n) = T(k) + T(n - k - 1) + d$$

  where $d$ is a constant time required for printing of $x.key$, etc.
- We show that $T(n) = O(n)$ by substitution method.

# Proof that $T(n) = O(n)$

Notice: we have to prove that $T(n) \leq c \cdot n$, for a constant $c$ and sufficiently large $n$.
Instead we will prove that $T(n) \leq (c+d)n + c$ for a constant $c$ and sufficiently large $n$.
Then $T(n) \leq (c+d)n + c = O(n)$

## Proof by induction

- **Base case** if $n = 0$, $T(0) = c \leq (c+d) \cdot 0 + c$.
- **Induction hypothesis:** $\forall 0 \leq m < n.(T(n) \leq (c+d)n + c)$
- **Induction step:**

$$
\begin{aligned}
T(n) &= T(k) + T(n-k-1) + d \\
&\leq_{IH} ((c+d)k + c) + ((c+d)(n-k-1) + c) + d \\
&= (c+d)k + c + (c+d)n - (c+d)k - (c+d) + c + d \\
&= (c+d)n + d
\end{aligned}
$$

# Querying a binary search tree

## Querying operations:

SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR

## Searching

Input: pointer to the root of a tree and a key
Output: pointer to a node with the key or NIL.

# Searching

**Procedure** TREE-SEARCH$(x, k)$

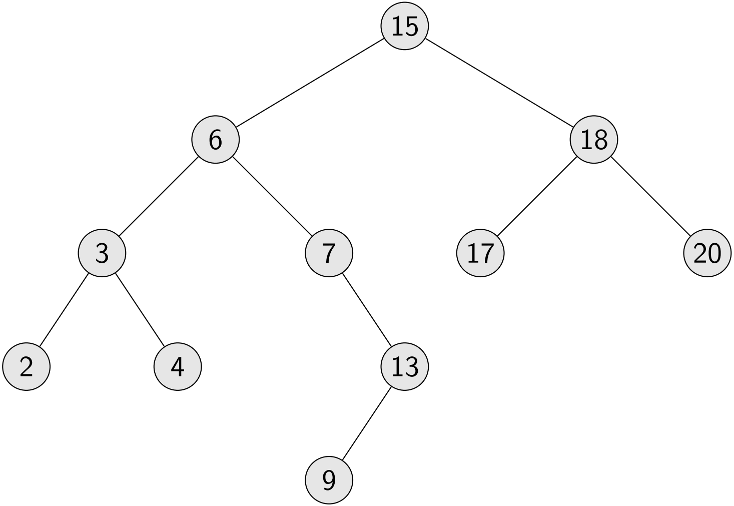
```
if x == NIL or k == x.key then
    return x
if k < x.key then
    return TREE-SEARCH(x.left, k)
else
    return TREE-SEARCH(x.right, k)
```

## Running time:

In TREE-SEARCH we follow the path from the root to a leaf, hence there are at most $O(h)$ steps, where $h$ is the height of the tree.

# Example

Search the tree for $k = 13$.



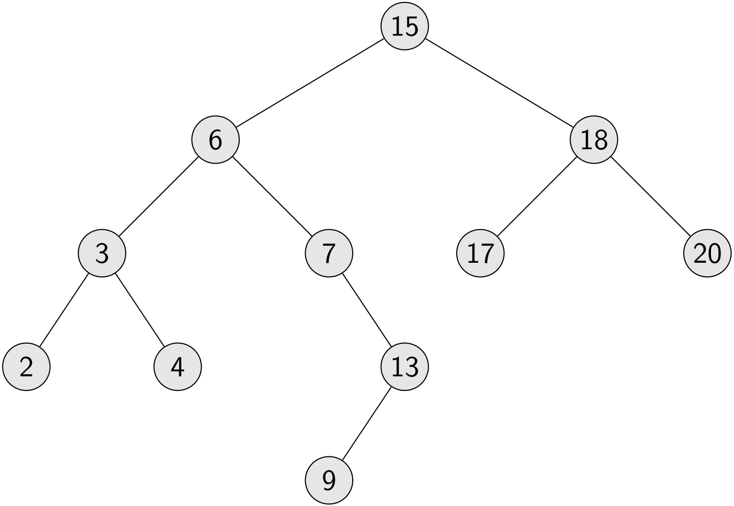
# Iterative version of TREE-SEARCH

**Procedure** ITERATIVE-TREE-SEARCH($x, k$)

```
while x ≠ NIL or k ≠ x.key do
    if k < x.key then
        x = x.left
    else
        x = x.right
return x
```

# Minimum and Maximum

To find minimum, follow the left pointers from the root down to NIL.



# Minimum and Maximum

Input: pointer to the root.

**Procedure** TREE-MINIMUM($x$)

**while** $x.left \neq NIL$ **do**
    $x = x.left$
**return** $x$

**Procedure** TREE-MAXIMUM($x$)

**while** $x.right \neq NIL$ **do**
    $x = x.right$
**return** $x$

Running time: $O(h)$, where $h$ is the height of the tree.

# Successor and Predecessor

The successor of $x$ is the node with the smallest key greater than $x.key$. We can find the successor of $x$ without comparing the keys.

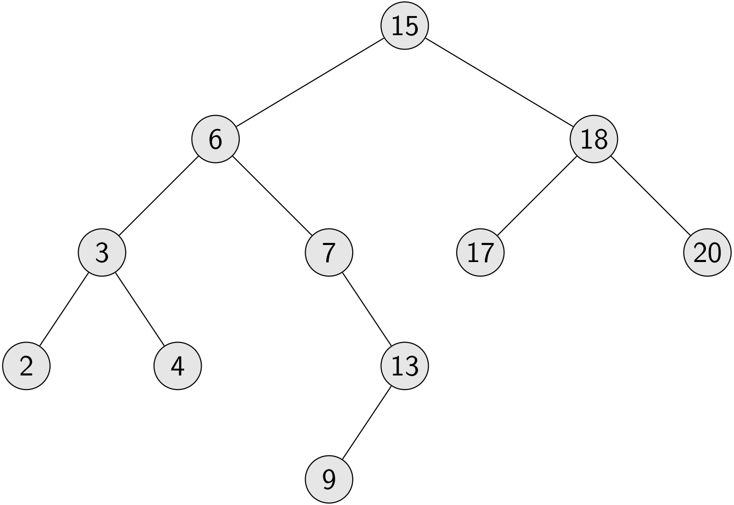
**Procedure** TREE-SUCCESSOR($x$)

```
if x.right ≠ NIL then
    return TREE-MINIMUM(x.right)
y = x.p
while y ≠ NIL and x == y.right do
    x = y
    y = y.p
return y
```

- If $x.right$ exists, then the successor of $x$ is the leftmost node in the right subtree.
- If $x.right$ is empty, and $x$ has a successor, then it is its lowest parent, whose left child is an ancestor of $x$.

# Example of finding a successor

- Find successor of a node with the key 17.
- Find successor of a node with the key 13.



# TREE-SUCCESSOR and TREE-PREDECESSOR

- Running time for TREE-SUCCESSOR is $O(h)$, where $h$ is the height of the tree.
  (We follow either a path down the tree, or up the tree, and the paths have length at most $h$.)
- The procedure TREE-PREDECESSOR is symmetric to TREE-SUCCESSOR. It runs also in $O(h)$ time.
- Even if the keys are not all distinct, we can use the procedures.
  (We just define a successor/predecessor node as the one returned by the procedure.)

Thus we have shown:

**Theorem**
*The dynamic query operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR take $O(h)$ time on a binary search tree of height $h$.*

# Insertion and Deletion

Notice:

- These operations change the tree
- We have to ensure that the binary-search-property holds
- Insertion is easy
- Deletion is more complicated

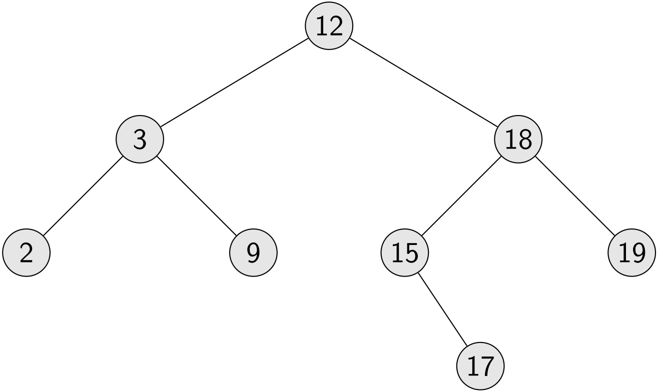
# Insertion

We want to insert a node with a value $v$. First create a node $z$:
$z.key = v$, $z.left = NIL$, $z.right = NIL$

**Procedure** TREE-INSERT($T, z$)

```
y = NIL
x = T.root
while x ≠ NIL do
    y = x
    if z.key < x.key then
        x = x.left
    else
        x = x.right
z.p = y
if y == NIL then
    T.root = z
else if z.key < y.key then
    y.left = z
else
    y.right = z
```

# Example: inserting a node with value 13



# Deletion

## We want to delete node $z$

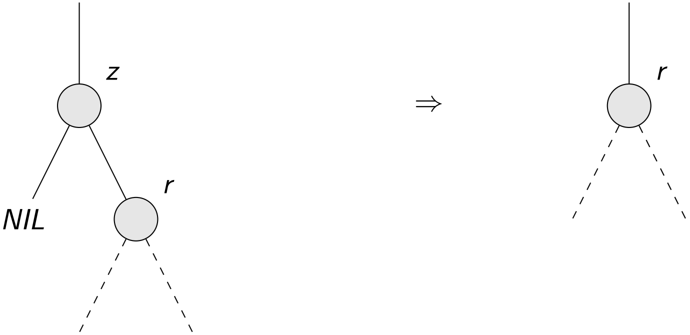
3 cases:

- $z$ has no children: simply remove $z$, parent of $z$ has now *NIL* in the place of the pointer to $z$.
- $z$ has one child: then this child will replace $z$ in the tree.
- $z$ has 2 children, then:
  - find the successor of $z$, $y$,
  - $y$ replaces $z$
  - $y.left$ points now to $z.left$
  - $y.right$ points now to $z.right$
  - have to be careful if $y$ was the right child of $z$... (cannot point to itself)

In the delete procedure we organize these cases in a little different way.

# Cases for DELETION

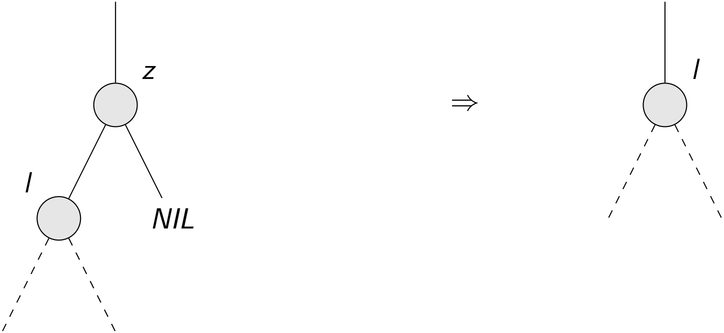
If $z$ has no left child:




We replace $z$ by $z.right$, even if $z.right$ is $NIL$.

# Cases for DELETION

If $z$ has left child only:



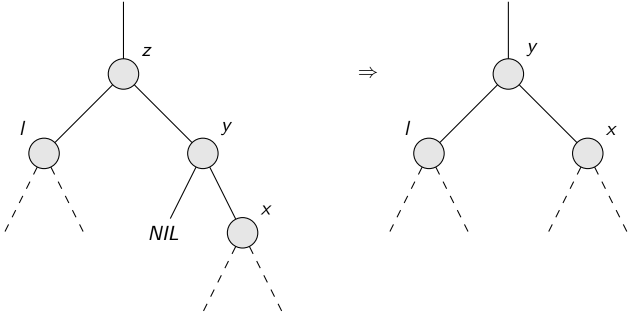

We replace $z$ by its left child $l$

Otherwise, $z$ has both children: left and right.

# Cases for DELETION

$z$ has left and right children:

- $y$ is the successor of $z$
- Slice $y$ from its place and put it in place of $z$
- If $y$ is the right child of $z$, replace $z$ by $y$:

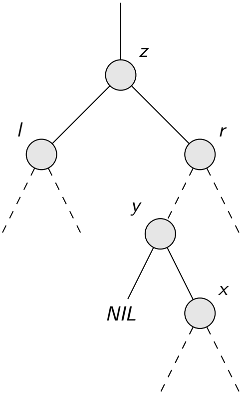


# Cases for Deletion

## $z$ has left and right children:

If $y$ is not the right child of $z$:

- $y$ (successor of $z$) is in $z$'s right subtree:



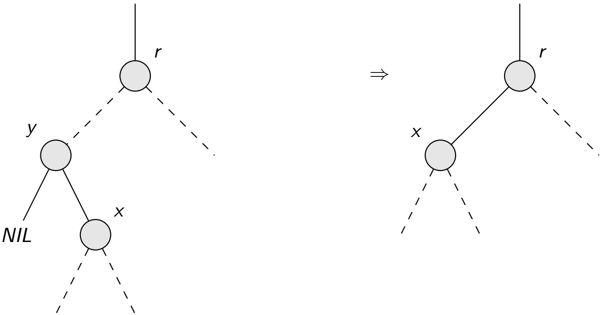

Notice: $y$ cannot have a left child.

# Cases for DELETION

$z$ has left and right children:

$y$ (successor of $z$) is in $z$'s right subtree:

- First replace $y$ by its own right child:



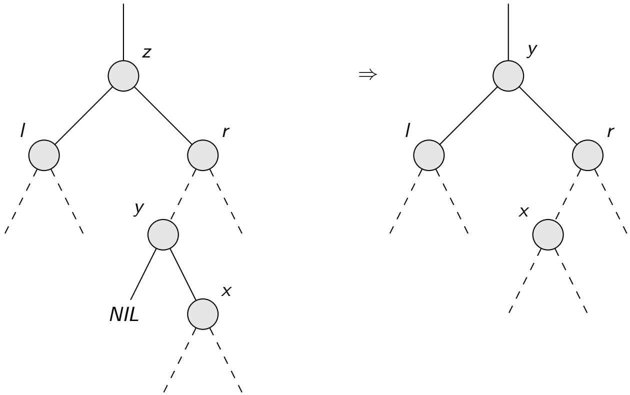

- Second, replace $z$ by $y$

# Cases for Deletion

## $z$ has left and right children:

$y$ (successor of $z$) is in $z$'s right subtree:

- Replacing $z$ by $y$:



# Moving subtrees

To move subtrees in a binary search tree, we use a procedure TRANSPLANT.
It replaces subtree rooted at $u$ with the subtree rooted at $v$.

**Procedure** TRANSPLANT($T, u, v$)

**if** $u.p == NIL$ **then**
    $T.root = v$
**else if** $u == u.p.left$ **then**
    $u.p.left = v$
**else**
    $u.p.right = v$
**if** $v \neq NIL$ **then**
    $v.p = u.p$

TRANSPLANT does not update the binary search tree. The property of binary search tree should be secured by the deleting algorithm.

## Tree-Delete

**Procedure** TREE-DELETE($T, z$)

```
if z.left == NIL then
    Transplant(T, z, z.right)
else if z.right == NIL then
    Transplant(T, z, z.left)
else
    y = Tree-Minimum(z.right)
    if y.p ≠ z then
        Transplant(T, y, y.right)
        y.right = z.right
        y.right.p = y
    Transplant(T, z, y)
    y.left = z.left
    y.left.p = y
```

Running time: $O(h)$ because of Tree-Minimum.

# Conclusion

## Theorem

*We can implement* INSERT *and* DELETION *for binary search trees in such a way, that each one runs in $O(h)$ time on a binary search tree of height $h$.*