# Data Structures – LECTURE 8

## Binary search trees

- Motivation
- Operations on binary search trees:
  - Search
  - Minimum, Maximum
  - Predecessor, Successor
  - Insert, Delete
- Randomly built binary search trees



## Motivation: binary search trees

- A dynamic ADT that efficiently supports the following common operations on *S*:
  - Search for an element
  - Minimum, Maximum
  - Predecessor, Successor
  - Insert, Delete
- Use a binary tree! All operations take $\Theta(\lg n)$
- The tree must always be balanced, for otherwise the operations will not take time proportional to the height of the tree!



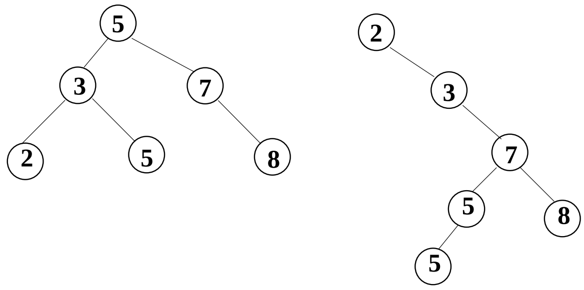
## Binary search tree

- A binary search tree has a root, internal nodes with at most two children each, and leaf nodes
- Each node *x* has *left*(*x*), *right*(*x*), *parent*(*x*), and *key*(*x*) fields.
- **Binary-search-tree property**:

  Let *x* be the root of a sub-tree, and *y* a node below it.
  - left sub-tree: $key(y) \leq key(x)$
  - right sub-tree: $key(y) > key(x)$



## Examples of binary trees




*In-order, pre-order, and post-order traversal*



## Tree-Search routine

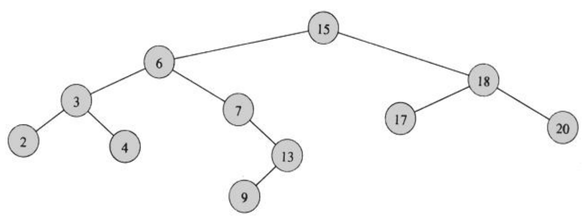
**<u>Tree-Search</u>**(*x*,*k*)
 **if** $x = null$ **or** $k = key[x]$
  **then return** *x*
 **if** $k < key[x]$
  **then return** Tree-Search(*left*[*x*],*k*)
  **else return** Tree-Search(*right*[*x*],*k*)

**<u>Iterative-Tree-Search</u>**(*x*,*k*)
 **while** $x \neq null$ **and** $k \neq key[x]$
  **do if** $k < key[x]$
   **then** $x \leftarrow left[x]$
   **else** $x \leftarrow right[x]$
**return** *x*

***Complexity: O(h)***



## Example: search in a binary tree




Search for 13 in the tree

## Tree traversal

**Inorder-Tree-Walk**($x$)
  **if** $x \neq null$
    **then** Inorder-Tree-Walk(*left*[$x$])
      **print** *key*[$x$]
      Inorder-Tree-Walk(*right*[$x$])

Recurrence equation:

$T(0) = \Theta(1)$

$T(n)=T(k) + T(n - k -1) + \Theta(1)$

***Complexity:* $\Theta(n)$**



## Max and Min-Search routines

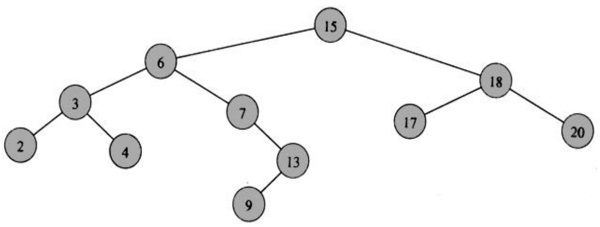
**Tree-Minimum**($x$)
  **while** *left*[$x$] $\neq$ *null*
    **do** $x \leftarrow$ *left*[$x$]
  **return** $x$

**Tree-Maximum**($x$)
  **while** *right*[$x$] $\neq$ *null*
    **do** $x \leftarrow$ *right*[$x$]
  **return** $x$

***Complexity:* $O(h)$**



## Example: Min and Max search



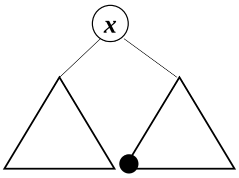



## Tree-Successor routine (1)

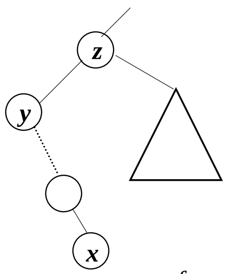
- The successor of $x$ is the smallest element $y$ with a key greater than that of $x$
- The successor of $x$ can be found without comparing the keys. It is either:
  1. *null* if $x$ is the *maximum node.*
  2. the *minimum of the right child of t* when $t$ has a right child.
  3. or else, *the lowest ancestor of x whose left child is also an ancestor of x.*



## Tree-Successor: cases




*Minimum of right child of t*



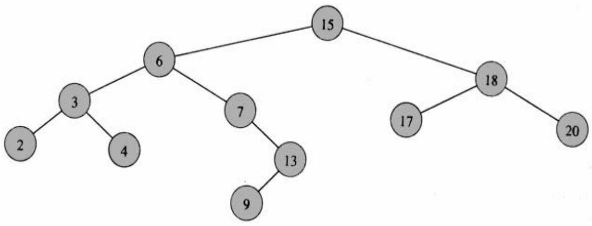

*Lowest ancestor z of t whose left child y is also an ancestor of t*



## Tree-Successor routine (2)

**Tree-Successor**($x$)
  **if** *right*[$x$] $\neq$ *null*    /* *Case 2*
    **then return** Tree-Minimum(*right*[$x$])
    $y \leftarrow$ *parent*[$x$]
  **while** $y \neq null$ and $x$ = *right*[$y$]   /* *Case 3*
    **do** $x \leftarrow y$
      $y \leftarrow$ *parent*[$y$]
    **return** $y$

## Example: finding a successor




*Find the successors of 15, 13*



## Proof (1)

- Case 3: If *x* doesn't have a right child, then its successor is *x*'s first ancestor such that its left child is also an ancestor of *x*. (This includes the case that there is no such ancestor, and then *x* is the maximum and the successor is *null*.)
- Proof: To prove that a node *z* is the successor of *x*, we need to show that *key*[*z*] > *key*[*x*] and that *x* is the maximum of all elements smaller than *z*.
- Start from *x* and climb up the tree as long as you move from a right child up. Let the node you stopped at be *y*, and denote *z* = *parent*[*y*].



## Proof (2)

- Sub-claim: *x* is the max of the sub-tree rooted at *y*.
- Proof of sub-claim: *x* is the node you reach if you go right all the time from *y*.
- Now we claim *z* = *parent*(*y*) is the successor of *x*. First, *key*[*z*] > *key*[*x*] because *y* is the left child of *z* by the definition of *y*, so *x* is in *z*'s left sub-tree.
- Now, *x* is the maximum of all items that are smaller than *z*, because by the sub-claim *x* is the maximum of the sub-tree rooted at *y*, and all elements smaller than *z* are in this sub-tree by the property of binary search trees.

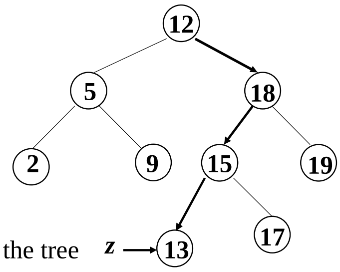


## Insert

- Insert is very similar to search: we essentially find the place in the tree where we want to insert the new node *z*.
- The new node *z* will always be a leaf.
- We assume that initially *left*(*z*) and *right*(*z*) are both *null*.



## Example: insertion




Insert 13 in the tree



## Tree-insert routine

```
Tree-Insert(T,z)
y ← null
x ← root[T]                      y is the parent of x
while x ≠ null
  do y ← x
     if key[z] < key[x]
       then x ← left[x]
       else x ← right[x]
parent[z] ← y
```

```
/* When the tree is empty
if y = null then root[T] ← z
else if key[z] < key[y]
     then left[y] ← z
     else right[y] ← z
```

# Delete (1)

Delete is more complicated than insert. There are three cases to delete node *z*:

1. *z* has no children
2. *z* has one child
3. *z* has two children

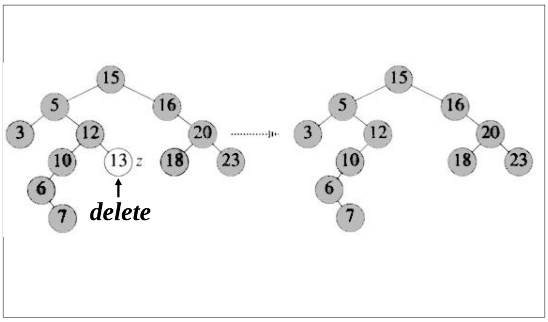
Case 1: delete *z* and update the child's parent child to *null*.

Case 2: delete *z* and connect its parent to its child.

Case 3: more complex; we can't just take the node out and reconnect its parent with its children, because the tree will no longer be a binary tree!

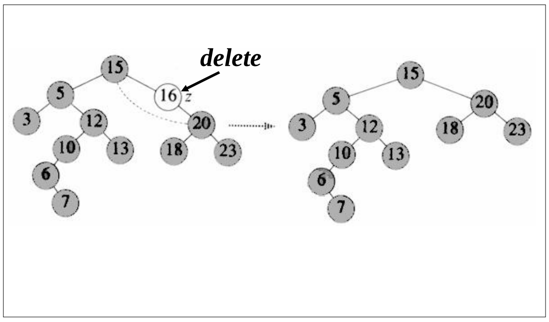


# Delete case 1: no children!






# Delete case 2: one child






# Delete (2)

*For case 3, the solution is to replace the node by its successor, and "pull" the successor, which necessarily has one child at most.*

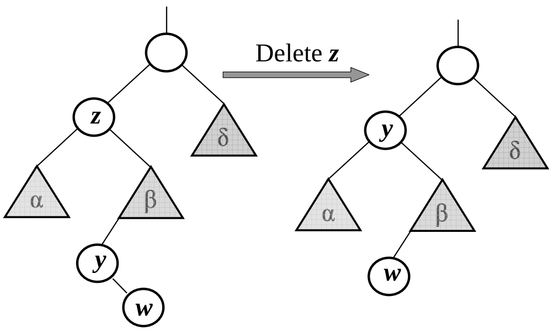
Claim: if a node has two children, its successor has at most one child.

Proof: This is because if the node has two children, its successor is the minimum of its right sub-tree. This minimum cannot have a left child because then the child would be the minimum…

Invariant: in all cases the binary search tree property is preserved after the deletion.

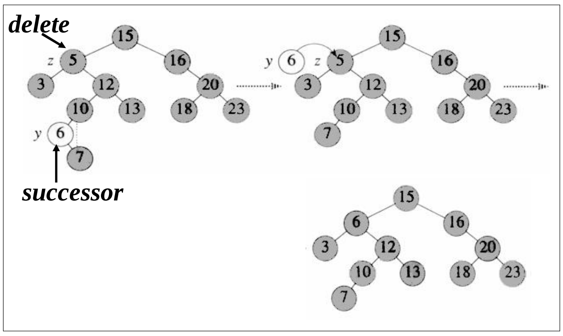


# Delete: case 3 proof






# Delete: case 3



## Tree-Delete routine

```
Tree-Delete(T,z)
if left[z] = null or right[z] = null    /* Cases 1 or 2
   then  y ← z                          /* find a node y to splice
   else  y ← Tree-Successor(z)          /* to splice out
if left[y] ≠ null                       /* set the child x
   then x ← left[y]
   else  x ← right[y]
if x ≠ null                             /* splicing operation
   then parent[x] ← parent[y]
if parent[y] = null
   then root[T] ← x
   else if y = left[parent[y]]
           then left[parent[y]] ← x
           else  right[parent[y]] ← x
                                        /* copy y's satellite
                                           data into z
                                        if y ≠ z
                                           then key[z] ← key[y]
                                        return y
```



## Complexity analysis

- Delete: The two first cases take $O(1)$ operations: they involve switching the pointers of the parent and the child (if it exists) of the node that is deleted.
- The third case requires a call to Tree-Successor, and thus can take $O(h)$ time.
- In conclusion: all dynamic operations on a binary search tree take $O(h)$, where $h$ is the height of the tree.
- In the worst case, the height of the tree can be $O(n)$



## Randomly-built Binary Search Trees

- Definition: A **randomly-built binary search tree** over $n$ distinct keys is a binary search tree that results from inserting the $n$ keys in random order (each permutation of the keys is equally likely) into an initially empty tree.
- Theorem: The average height of a randomly-built binary search tree of $n$ distinct keys is $O(\lg n)$
- Corollary: The dynamic operations Successor, Predecessor, Search, Min, Max, Insert, and Delete all have $O(\lg n)$ average complexity on randomly-built binary search trees.