Tirgul 7

Binary Search Trees

Motivation

- We would like to have a dynamic ADT that efficiently supports the following common operations:
  - Insert & Delete
  - Search for an element
  - Minimum & Maximum
  - Predecessor & Successor
- Use a binary tree! All operations take $\Theta(h)$
- When the tree is balanced, $h=\log(n)$

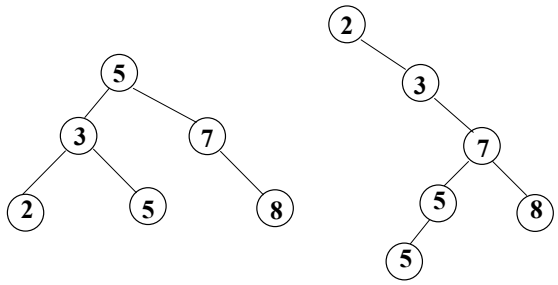
Binary search tree

- 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 (and possibly other fields as well).

Binary-search-tree properties:

- Let x be the root of a sub-tree, and y a node below it. y resides in the:
  - left sub-tree if $key(y) < key(x)$
  - right sub-tree if $key(y) > key(x)$

Examples of binary trees

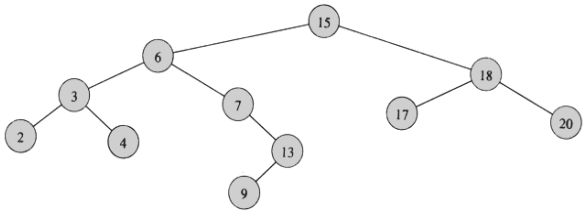


Tree-Search routine

```
Tree-Search(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)

Iterative-Tree-Search(x,k)
    while x ≠ null and k ≠ key[x]
          do if k < key[x]
                   then  x ← left[x]
                   else   x ← 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 ≠ 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

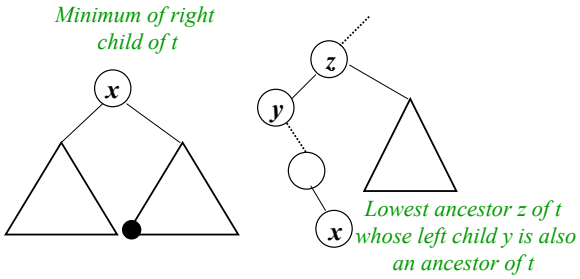
```
Tree-Minimum(x)
     while left[x] ≠ null
          do  x ← left[x]
     return x

Tree-Maximum(x)
     while right[x] ≠ null
          do  x ← right[x]
     return x
```

➲ ***Complexity:*** $\Theta(h)$

Tree-Successor routine

- 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 2,3



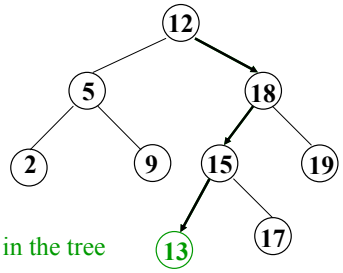
Tree-Successor routine

```
Tree-Successor(x)
  if right[x] ≠ null     // Case 2
     then return Tree-Minimum(right[x])
     y ← parent[x]       // Case 3
     while y ≠ null  and x = right[y] do
          x ← y
          y ← parent[y]
     return y
```


Insert

- Insert is very similar to search:
- We search for the value, if we do not find it, we continue searching along either the left or right branch.
- Eventually we will reach a null leaf, and simply add the value at that position.
- The complexity is proportional to the height of the tree

Example: insertion



Insert 13 in the tree

Tree-insert routine

```
Tree-Insert(T,z)
y ← null
x ← root[T]
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
elseif key[z]< key[y]
  then left[y] ←z
else right[y] ←z
```

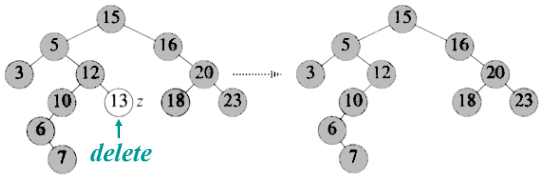
y is the parent of x

Delete

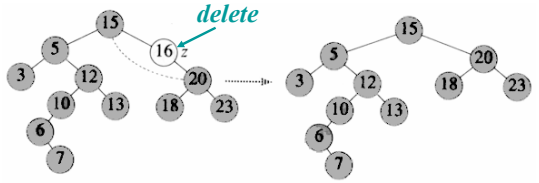
- Delete is more complicated than insert. There are three cases to delete node *z*:
  1. *z* has no children
  2. *z* has one child
  3. *z* has two children

- **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!



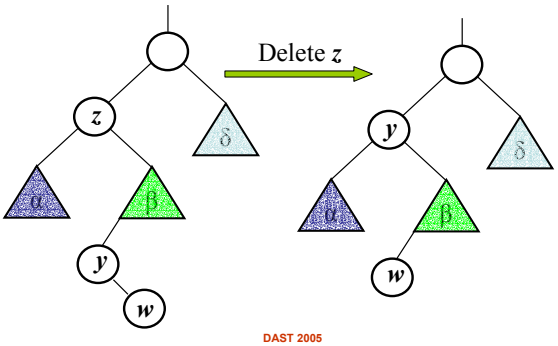
Delete case 2: one child



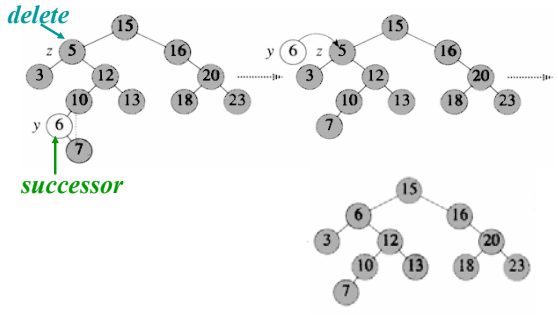
Delete

- *For case 3, the solution is to replace the node by its successor (or predecessor), 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



Delete: case 3



Tree-Delete routine

```
TREE-DELETE(T, z)
 1  if left[z] = NIL or right[z] = NIL
 2       then y ← z
 3       else y ← TREE-SUCCESSOR(z)
 4  if left[y] ≠ NIL
 5       then x ← left[y]
 6       else x ← right[y]
 7  if x ≠ NIL
 8       then p[x] ← p[y]
 9  if p[y] = NIL
10       then root[T] ← x
11       else if y = left[p[y]]
12                 then left[p[y]] ← x
13                 else right[p[y]] ← x
14  if y ≠ z
15       then key[z] ← key[y]
16            copy y's satellite data into z
17  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 $O(n)$**