Set Operations

- Maximum
- Extract-Max
- Insert
- Increase-key
- Search
- Delete
- Successor
- Predecessor

We can use priority queue (implemented by heap)

Not efficient with heap!

Linked list? Array?
Search Tree

- Support all of previous operations
  - Time proportional to tree height

- Can be used to implement priority queues

- Different types of search trees
  - Binary search tree, red-black trees, etc
Binary Search Tree

- First, it is a binary tree
- Represented by a linked data structure
- Each node contains at least fields:
  - Key
  - Left
  - Right
  - Parent
  Nil if does not exists
Secondly, the binary-search-tree property

For any node $x$,

- $\text{Key}[y] \leq \text{Key}[x]$ if $y$ in left subtree of $x$
- $\text{Key}[x] \leq \text{Key}[y]$ if $y$ in right subtree of $x$
Example

```
    13
   /   \
  6     18
 /   \   /   \
3     7 17     20
|     |   |     |
2     4   9   13
```

Properties

- Not unique for the same set of elements
- Minimum
  - Does it have to be a leaf?
- Maximum
Query A Tree

- **Tree-search** \((x, k)\)
  - Given a node \(x\) and a key \(k\), returns a node \(y\) in the subtree rooted at \(x\) if \(y\).key = \(k\) if exists, Nil otherwise

```plaintext
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)
```

Time complexity:
\(O(h)\), where \(h\) is height of tree.
Iterative Tree Search

procedure IterativeTreeSearch(x,K)

1 while (x = NIL) and (K ≠ x.key) do
2     if (K ≤ x.key) then
3         x ← x.left;
4     else
5         x ← x.right;
6     end
7 end
8 return (x);
Binary search tree on the same set of numbers.

Which tree is better?
Assign the following numbers so that it becomes a binary search tree: 8, 12, 16, 17, 20, 24, 31, 35, 42
procedure InorderTreeWalk(x)
1 if \( x \neq \text{NIL} \) then
2 InorderTreeWalk(x.left);
3 print \( x\.\text{key} \);
4 InorderTreeWalk(x.right);
5 end
Inorder Tree Walk

- Inorder tree walk / transversal
- Provide a way to visit all nodes in the tree
- Visit tree nodes in sorted order!
  - During binary-search tree property.
- Running time:
  \[ \Theta(n). \]
  Why?

- Preorder, postorder
More Properties

- For any subtree rooted at $x$
  - Minimum
  - Maximum

- For any subtree rooted at $x$
  - Inorder-tree-walk ($x$) produces a subsequence of
    Inorder-tree-walk ($root$)

- Let $y$ be an ancestor of $x$
  - Relations between $\text{Key}[y]$ and $\text{Key}[x]$
Minimum / Maximum

- Tree-minimum($x$)
  while ($x$.left ≠ Nil)
    do $x = x$.left;
  return $x$;

- Tree-maximum($x$)
  while ($x$.right ≠ Nil)
    do $x = x$.right;
  return $x$;

Time: $O(h)$
Successor

- **Successor**(*x*)
  - The node *y* with the smallest key greater than or equal to *x*.key
  - Ambiguous when multiple nodes have same key
  - Successor in the inorder tree walk
  - Return Nil if none exists -- has largest key
Example
Two Cases

- If right child of $x$ exists
  - Leftmost node in right subtree
- Otherwise
  - Lowest ancestor $y$ where $x$ is in the left-subtree of $y$

Tree-successor($x$)

```c
if x.right != Nil
then return Tree-minimum(x.right)
y = x.parent
while y != Nil and x = y.right
    do x = y
        y = y.parent
return y
```

Time: $O(h)$

why?
Predecessor (x)

- Two cases
  - The left child of $x$ exists
    - Rightmost child in the left subtree of $x$
  - Otherwise
    - Lowest ancestor $y$ where $x$ is in the right subtree of $y$.

- Completely symmetric
Summary

- Using binary search tree
  - Traversal (inorder tree walk)
  - Min / max
  - Search
  - Successor / predecessor

- All (other than traversal) have running time $O(h)$
  - $h$ is height of the tree.

We could use a sorted array to do all these in efficiently!

However, array is not efficient in dynamic updates.
Tree-Insert(T, z)

- Insert z into tree T, and resulting tree still binary search tree

Use Tree-search!
Pseudocode

Tree-insert($T$, $z$)

$y = \text{Nil}$, $x = \text{root}[T]$
while ($x \neq \text{Nil}$)
do 
  $y = x$
  if ($z$.key $<$ $x$.key )
    then $x = x$.left
  else $x = x$.right

$z$.parent = $y$
if ($y = \text{Nil}$) then root[$T$] = $z$
else if ($z$.key $<$ $y$.key) 
  then $y$.left = $z$
  else $y$.right = $z$

Time: O(h)

Locate potential parent $y$ of $z$. 
Build Binary Search Tree

- Use Tree-insert
  - i.e., insert the elements one by one.
  - Worst case: quadratic
  - Best case: $\Theta(n \lg n)$

- Sorting, and then construct a tree
  - $\Theta(n \lg n)$
Deletion

```
delete 4 ?
delete 17 ?
delete 7 ?
delete 3 ?
delete 6 ?
delete 15 ?
```
Three Cases for Deleting $z$

- **Case 1:** Leaf node
- **Case 2:** Has only one child
  - Replace $z$ with its child
- **Case 3:** Has both children
  - Find its successor $y$
    - $y$ cannot have left child!
    - replacing $y$ with its right child!
  - Replace $z$ with $y$

The “replacement” is implemented by the transplant operation
Transplant Operation

```
function Transplant(T, u, v)
    /* Replace subtree rooted at u with subtree rooted at v. */
    1 p ← u.parent;
    2 if (p = NIL) then T.root ← v;
    3 else if (u = p.left) then p.left ← v;
    4 else p.right ← v;
    5 if (v ≠ NIL) then v.parent ← p;
```

Does not touch parent info. for u.
Case 3: More Detailed View

- Goal: delete node $z$
- Let $y$ be the successor of $z$
- Case-(3.1)
  - $y$ is the right child of $z$
  - Then replace $z$ by $y$
Case 3: cont.

- Case (3.b):
  - $y$ is not the right child of $z$
  - First replace $y$ by its own right child, and then replace $z$ by $y$. 

![Tree Diagram]
Delete

$$\text{TREE-DELETE}(T, z)$$

if $z.left == \text{NIL}$

$$\text{TRANSPLANT}(T, z, z.right) \quad \text{// } z \text{ has no left child}$$

elseif $z.right == \text{NIL}$

$$\text{TRANSPLANT}(T, z, z.left) \quad \text{// } z \text{ has just a left child}$$

else // $z$ has two children.

$$y = \text{TREE-MINIMUM}(z.right) \quad \text{// } y \text{ is } z\text{'s successor}$$

if $y.p \neq z$

$$\text{// } y \text{ lies within } z\text{'s right subtree but is not the root of this subtree.}$$

$$\text{TRANSPLANT}(T, y, y.right)$$

$$y.right = z.right$$

$$y.right.p = y$$

$$\text{// Replace } z \text{ by } y.$$ 

$$\text{TRANSPLANT}(T, z, y)$$

$$y.left = z.left$$

$$y.left.p = y$$

Time complexity: $O(h)$

together cover Cases 1 & 2.
An Exercise

Delete I?  
Delete G?  
Delete K?  
Delete B?
Remarks

- All complexity depends on height $h$
- $h = \Omega(lg n), \ h = \Theta(n)$
- To guarantee performance:
  - Balanced tree!

- Randomly build a tree
  - Expected height is $O(lg n)$
  - Still problem: future insertions …