Trees, Binary Search Tree

Bryn Mawr College
CS206 Intro to Data Structures

Tree

- A tree consists of a set of nodes and a set of edges that connect pairs of nodes.
- Property: there is exactly one path (no more, no less) between any two nodes of the tree.
- A path is a connected sequence of zero or more edges.
- In a rooted tree, one distinguished node is called the root. Every node c, except the root, has exactly one parent node p, which is the first node traversed on the path from c to the root. c is p's child.
- The root has no parent.
- A node can have any number of children.

Rooted Tree Terminology

- A leaf is a node with no children.
- Siblings are nodes with the same parent.
- The ancestors of a node d are the nodes on the path from d to the root. These include d's parent, d's parent's parent, d's parent's parent's parent, and so forth up to the root. Note that d's ancestors include d itself. The root is an ancestor of every node in the tree.
- If a is an ancestor of d, then d is a descendant of a.
- The length of a path is the number of edges in the path.
- The depth of a node n is the length of the path from n to the root. (The depth of the root is zero.)

Rooted Tree Terminology (cont.)

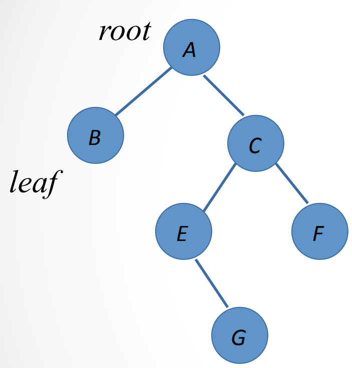
- The height of a node n is the length of the path from n to its deepest descendant. (The height of a leaf is zero.)
- The height of a tree is the depth of its deepest node = height of the root.
- The subtree rooted at node n is the tree formed by n and its descendants.
- A binary tree is a tree in which no node has more than two children, and every child is either a left child or a right child, even if it is the only child its parent has.

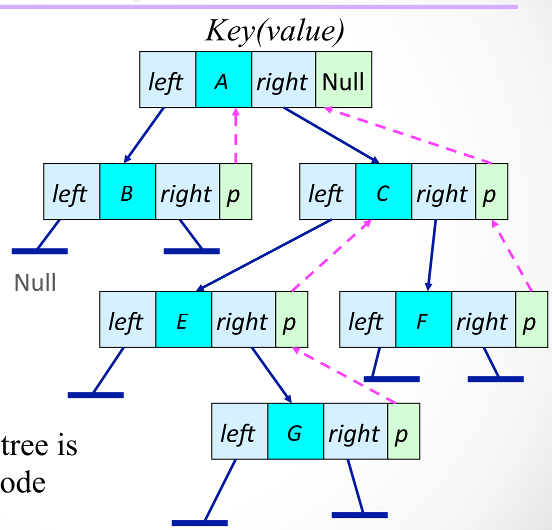
Binary Trees

Rooted trees can also be defined recursively. Here is the definition of a binary tree:

- A binary tree T is a structure defined on a finite set of nodes that either
  - Contains no nodes, or
  - Is composed of three disjoint sets of nodes:
    - a root node,
    - a binary tree called the left subtree of T, and
    - a binary tree called the right subtree of T.

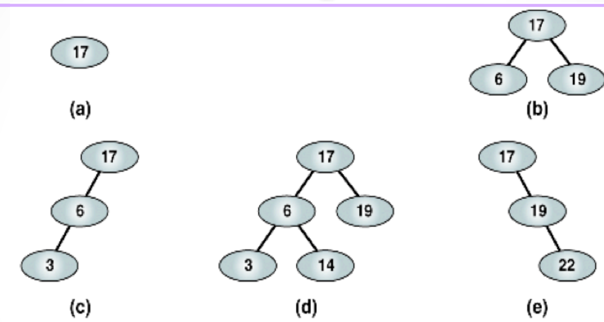
A Binary Tree





Every node in the binary tree is reachable from the root node by a unique path.

Examples



- A binary tree is
  - full if every node other than leaves has two children; (a), (b), (d)
  - complete if every level is completely filled; (a), (b)
  - nearly complete if every level except the last is completely filled, and all nodes are as far left as possible; (d)
  - balanced if the depth of left and right subtrees of every node differ at most 1. (a), (b), (d)

Representing Rooted Trees

- A direct way to represent a tree is to use a data structure where every node has three references:
  - one reference to the object stored at that node,
  - one reference to the node's parent, and
  - one reference to the node's children.
- The child-sibling (CS) representation is another popular tree representation. It spurns separately encapsulated linked lists so that siblings are directly linked.
  - It retains the item and parent references, but instead of referencing a list of children, each node references just its leftmost child.
  - Each node also references its next sibling to the right.
  - These nextSibling references are used to join the children of a node in a singly-linked list, whose head is the node's firstChild.

Basic Definition of a CSNode

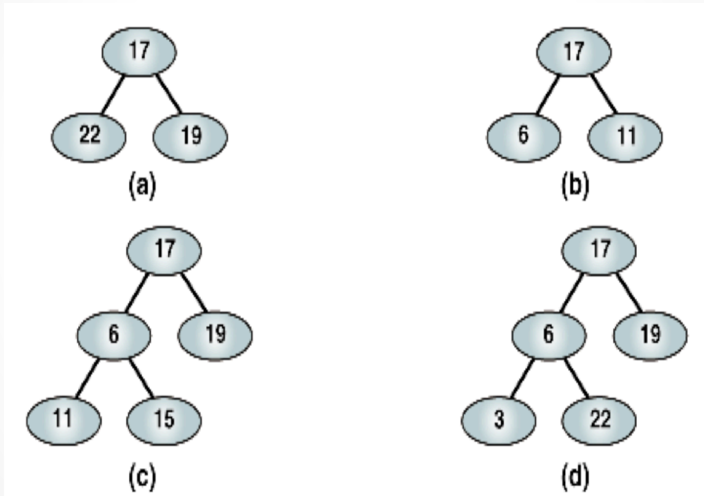
Here are the basic definitions, as well as the constructors. The rest of the code is posted separately.

```
public class CSNode<E> {
   protected CSNode<E> parent; // not really needed
   protected CSNode<E> firstChild;
   protected CSNode<E> nextSibling;
   protected E data;
   public CSNode(){}
   public CSNode(E data) { this(data, null, null); }
   public CSNode(E data, CSNode<E> child,
                          CSNode<E> sibling) {
      this.firstChild = child;
      this.nextSibling = sibling;
      this.data = data;
   }
```

Binary Search Trees

- The binary-search-tree property
  - If node y in left subtree of node x, then $key[y] \leq key[x]$.
  - If node y in right subtree of node x, then $key[y] \geq key[x]$.
- Binary search trees are an important data structure that supports dynamic set operations:
  - Search, Minimum, Maximum, Predecessor, Successor, Insert, and Delete.
  - Basic operations take time proportional to the height of the tree – $O(h)$.
- Q: Where is the minimum/maximum key?

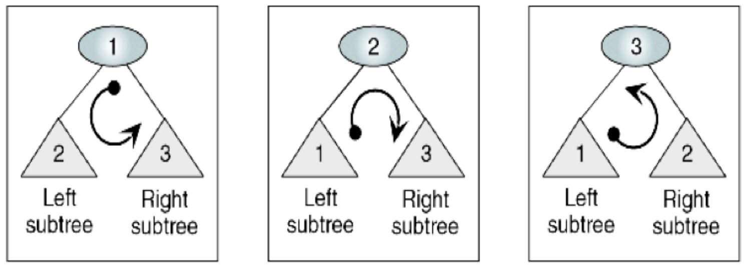
Invalid BSTs



BST Operations

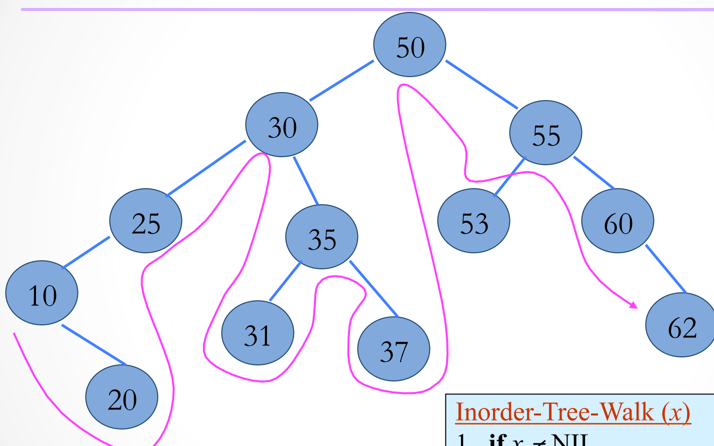
- Traversals
- Searches
- Insertion
- Deletion

Binary Tree Traversals



(a) Preorder traversal (b) Inorder traversal (c) Postorder traversal

Inorder Traversal of BST



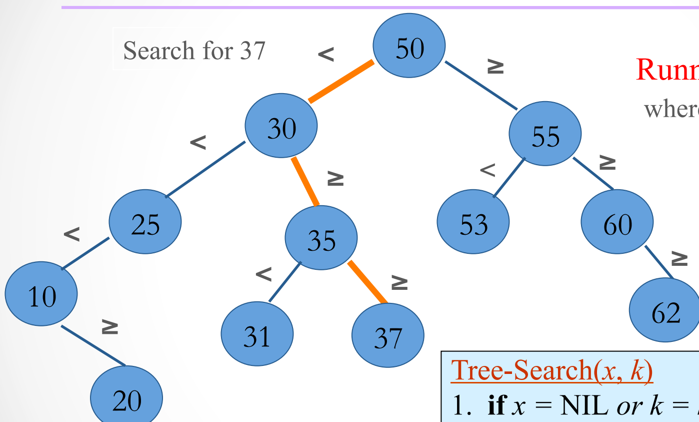
Prints out keys in sorted order:
10, 20, 25, 30, 31, 35, 37, 50, 53, 55, 60, 62

Inorder-Tree-Walk (x)
1. **if** $x \neq$ NIL
2. **then** Inorder-Tree-Walk(*left*[x])
3. print *key*[x]
4. Inorder-Tree-Walk(*right*[x])

Querying a Binary Search Tree

- All dynamic-set search operations can be supported in $O(h)$ time.
- $h = \Theta(lg\ n)$ for a balanced binary tree (and for an average tree built by adding nodes in random order.)
- $h = \Theta(n)$ for an unbalanced tree that resembles a linear chain of n nodes in the worst case.

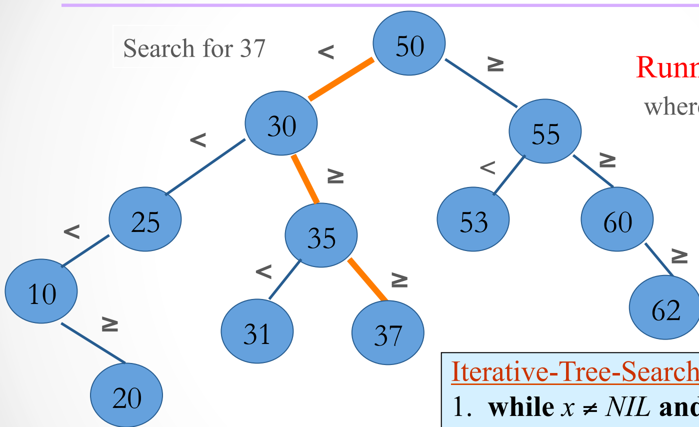
Tree Search



Running time $O(h)$
where h is tree height

```
Tree-Search(x, k)
if x = NIL 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



Running time $O(h)$

where h is tree height

<u>Iterative-Tree-Search(*x*, *k*)</u>

```
while x ≠ NIL and k ≠ key[x]
do if k < key[x]
      then x ← left[x]
     else x ← right[x]
return x
```

Finding Min & Max

- The binary-search-tree property guarantees that:
  - The minimum is located at the left-most node.
  - The maximum is located at the right-most node.

<u>Tree-Minimum(*x*)</u>

```
while left[x] ≠ NIL
   do x ← left[x]
return x
```

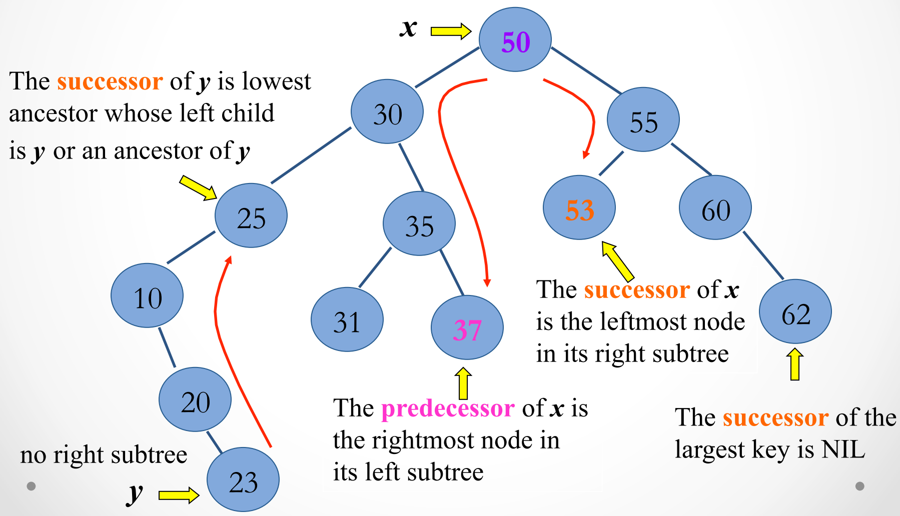
<u>Tree-Maximum(*x*)</u>

```
while right[x] ≠ NIL
      do x ← right[x]
return x
```

- Question: how long do they take?

Predecessor & Successor

10, 20, 23, 25, 30, 31, 35, 37, 50, 53, 55, 60, 62



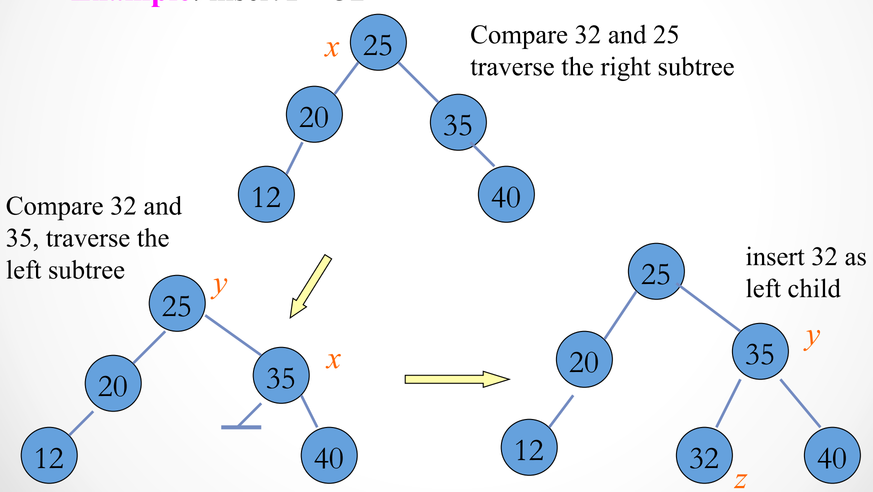
Pseudo-code for Successor

```
Tree-Successor(x)
if right[x] ≠ NIL
    then return Tree-Minimum(right[x])
y ← p[x]
while y ≠ NIL and x = right[y]
do x ← y
    y ← p[y]
return y
```

- Code for ***predecessor*** is symmetric.
- Running time: $O(h)$

Insertion Example

Example: insert $z = 32$



BST Insertion : Pseudo-code

```
Tree-Insert(T, z)
1.  y ← NIL
2.  x ← root[T]
3.  while x ≠ NIL
4.      do y ← x
5.          if key[z] < key[x]
6.              then x ← left[x]
7.          else x ← right[x]
8.  p[z] ← y
9.  if y = NIL
10.     then root[t] ← z
11. else if key[z] < key[y]
12.     then  left[y] ← z
13. else right[y] ← z
```

- Beginning at root of the tree, trace a downward path, maintaining two pointers.
  - Pointer x: traces the downward path.
  - Pointer y: "trailing pointer" to keep track of parent of x.
- Traverse the tree downward by comparing the value of node at x with *key[z]*, and move to the left or right child accordingly.
- When x is NIL, it is at the correct position for node z.
- Compare z's value with y's value, and insert z at either y's *left* or *right*, appropriately.
- Complexity: $O(h)$
  - Initialization: $O(1)$
  - While loop (3-7) : $O(h)$ time
  - Insert the value (8-13) : $O(1)$

Exercise: Sorting Using BST

Tree-Sort (A)
1. Let T be an empty BST
2. **for** $i \leftarrow 1$ *to* n
3. **do** Tree-Insert (T, $A[i]$)
4. Inorder-Tree-Walk($root[T]$)

- What are the worst case and best case running times?
- Worst case occurs when a linear chain of nodes results from the repeated insertion operation. $\Theta(n^2)$
- Best case occurs when a binary tree of height $\Theta(\lg n)$ results from repeated insertion operation. $\Theta(n \lg n)$

BST Deletion

Tree-Delete (T, x)

if x has no children ♦ case 0
    then remove x
if x has one child ♦ case 1
    then make $p[x]$ point to child
if x has two children (subtrees) ♦ case 2
    then swap x with its successor
        perform case 0 or case 1 to delete it

$\Rightarrow$ TOTAL: $O(h)$ time to delete a node