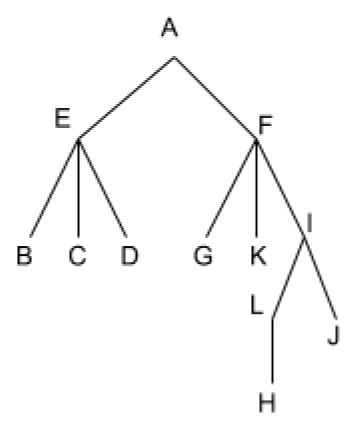
Trees: acyclic, connected graph (directed or undirected).

1. Undirected, unrooted tree: represented as a graph (e.g., Q2 of SP1). Output of algorithms of Kruskal and Boruvka are list of edges forming MSTs. This can be converted into a rooted tree, by running DFS/BFS on this graph (induced by the edges of this tree).
2. Undirected, rooted tree: represented as binary tree or arbitrary tree. A special node is designated as root. Edges connect parent node to its children, and it takes O(1) time to iterate per child. Nodes with no children are called “leaves” or “leaf nodes”. Other nodes are “internal” nodes. It is optional to store information about a node’s parent.
3. Up tree: Each node knows its parent. A node does not store its children.
4. Directed tree (also known as branching or arborescence). Outgoing or incoming tree.

Rooted trees: Defined recursively as root node attached to zero or more subtrees.



Root = A. Leaves = { B, C, D, G, K, H, J }.
Children(A) = { E, F }. E and F are siblings. Parent(C) = E.
Descendants(F) = { F, G, K, I, L, J, H }. Ancestors(I) = { I, F, A }.
Proper descendants(I) = { L, J, H }. Proper ancestors(I) = { F, A }.

```
class Tree<T> {
   class Entry<T> {
      T element;
      List<Entry<T>> children;
   }
   Entry<T> root;
   int size;
}
```

Concepts: **Path**: Sequence of edges to go from one node to another: F → I → L → H is a path of length 3 from F to H. The **length** of a path is the number of its edges. **Depth** of a node is the length of the path from the root of the tree to that node. So, depth(A) = 0, depth(F) = 1, depth(L) = 3, etc. **Height** of a node is a maximum length of a path from the node to any of its descendants, i.e., length of path from the node to a farthest leaf node that is its descendant. So, height(G) = 0, height(F) = 3. **Subtree** rooted at a node X is the part of the tree induced by X and its descendants. So, subtree of E is the tree rooted at E, connecting it to its descendants B, C, and, D.

$$\text{depth}(u) = \begin{cases} 0, & \text{if } u \text{ is root,} \\ 1 + \text{depth}(u.\text{parent}), & \text{otherwise.} \end{cases}$$

从root向下数

$$\text{height}(u) = \begin{cases} 0, & \text{if } u \text{ is leaf,} \\ 1 + \max\limits_{c \in u.children} \text{height}(c), & \text{otherwise.} \end{cases}$$

从叶子向上数

Binary trees: an important subclass of rooted trees, in which each node has at most 2 children. Binary trees have many applications, such as in expression trees (programming languages), binary search trees (TreeMap), Huffman coding.

Tree traversals: Algorithms for going through the nodes of a tree in different orders.

```
class BinaryTree<T> {
    class Entry<T> {
        T element;
        Entry<T> left, right;
        // Optional parent
        Entry<T> parent;
    }
    Entry<T> root;
    int size; }
```

```
preOrder( ) {
    preOrder( root );
}

preOrder(Entry<T> r) {
    if (r != null) {
        visit(r);
        preOrder(r.left);
        preOrder(r.right);
    }
}
```

```
postOrder( ) {
    postOrder( root );
}

postOrder(Entry<T> r) {
    if (r != null) {
        postOrder(r.left);
        postOrder(r.right);
        visit(r);
    }
}
```

```
inOrder( ) {
    inOrder( root );
}

inOrder(Entry<T> r) {
    if (r != null) {
        inOrder(r.left);
        visit(r);
        inOrder(r.right);
    }
}
```

preOrder and postOrder can be generalized easily to arbitrary trees. All that preorder usually requires, is to visit a node, before visiting any of its proper descendants. Therefore, it is possible to use a level-order traversal (BFS) of the tree for preorder, if visit(u) is called when u is removed from the queue. Similarly, DFS can be used for postorder, if a node is visited at the end of dfsVisit. Reverse of a preorder traversal can also be used as postorder.

Care should be taken when using depth( ) and height( ) functions. Depth of a node u can be calculated in time proportional to depth(u) if each node stores a link to its parent. There is no efficient implementation of depth(u) if the tree does not store parent link. Time to calculate height(u) is proportional to the number of descendants of u (which is the number of nodes in the subtree rooted at u).

```
int depth( Entry<T> u ) {
    return u == null ? -1 : 1 + depth( u.parent );
}

int height( Entry<T> u ) {
    if ( u == null ) { return -1; }
    lh = height( u.left );
    rh = height( u.right );
    return 1 + max( lh, rh );
}

// Bad code.  RT = O(n^2)
void traversal( Entry<T> u ) {
    if ( u != null ) {
        traversal( u.left );
        traversal( u.right );
        print u, depth(u), height(u)
    }
}
```

```
// Better code:  RT = O(n)
traversal( ) {
    traversal( root, 0 );
}

// Return height of tree. Depth is passed as param
int traversal( Entry<T> u, int d ) {
    if ( u != null ) {
        lh = traversal( u.left, d+1 );
        rh = traversal( u.right, d+1 );
        h = 1+max(lh, rh)
        print u, d, h
        return h
    } else {
        return -1;
    }
}
```

Dictionary ADT: An abstract data type on elements that are totally ordered (i.e., elements of T are Comparable or a Comparator is available for T), supporting the following operations: contains, add [insert], remove [delete], min, max, get. Iterating a dictionary goes through elements in sorted order of its keys.

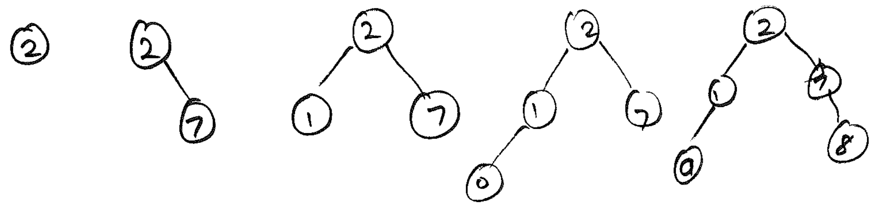
Design Goals: Binary Search Trees (BST)

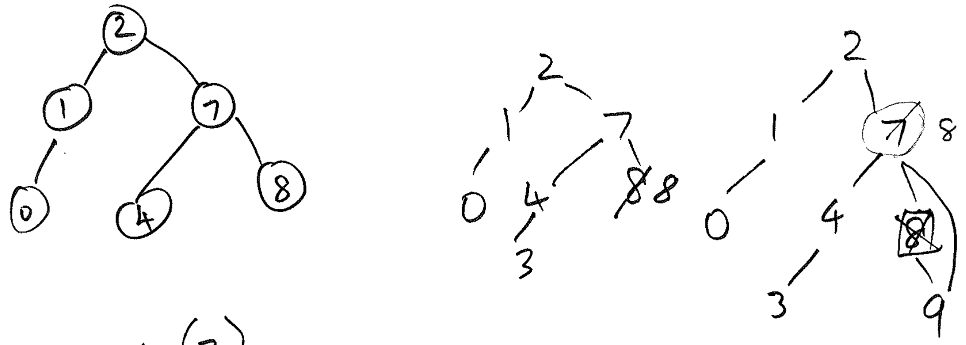
1. BST code should extend to balanced BST data structures such as AVL, RedBlack, splay trees.
2. Single pass algorithms, where possible. – avoid recursion, if possible.

   $r \longrightarrow \cdots\cdots \longrightarrow x$ (return)
3. Avoid duplication of work.
4. No parent link in Entry class. – saves space

```
// Helper function
Entry<T> find(x):
  stack ← Create a stack of Entry<T>
  return find(root, x)
Entry<T> find(t, x):   // LI: stack has path from root to t.parent
    if t = null or t.element = x then return t
    while true do
       if x < t.element then
if t.left ≠ null ~~then~~ { stack.push(t) ; t ← t.left } else break
       else if x = t.element { break }
       else   // x > t.element
if t.right ≠ null then { stack.push(t) ; t ← t.right } else break
    return t
```

~~Example~~ Add to initially empty tree: 2 7 1 0 8 4 3 8 9





remove (7)

return 7

```
contains(x):
    t ← find(x)
    return t ≠ null and t.element = x
```