Binary Trees

– Special class of trees, where each node has (up to) 2 children.

Applications: Expression trees, Binary search trees, Huffman Coding algorithm, ... (TreeMap)

Implementation: Entry class: element, Entry left, right.

element	
left	right

Sometimes Entry parent is included.

Tree traversals: going through nodes of a tree in specific order.

Preorder

```
preOrder():
    preOrder(root)

preOrder(t):
if t ≠ null then
    visit(t.element)
    preOrder(t.left)
    preOrder(t.right)
```

InOrder

```
inOrder():
    inOrder(root)

inOrder(t):
  if t ≠ null then
    inOrder(t.left)
    visit(t.element)
    inOrder(t.right)
```

PostOrder

```
postOrder():
    postOrder(root)

postOrder(t):
  if t ≠ null then
    postOrder(t.left)
    postOrder(t.right)
    visit(t.eleme
```

Remarks: (1) preOrder, postOrder can be defined for arbitrary trees. inOrder is specific to binary trees.

(2) in most applications, preorder just requires a node to be processed before its descendants — BFS (level order) can also be used. PostOrder = process node after descendants → DFS can also be used.

Dictionary ADT

Type T that implements Comparable <? super T>

Elements of T are comparable.

class method : int compareTo(T other):

this < other → return negative value (usually -1)

this = other → return 0

this > other → return positive value (usually +1)

Mathematically,

T – total order.

Ex: Integers, Strings

Operations:

add(x) – add a new word to dictionary. duplicates are not allowed, no null keys.

contains(x) – is x in dictionary?

remove (x) – remove x from dictionary.

min ( ) – smallest word in dictionary. w.r.t. the ordering.

max ( ) – last word in dict.

std: isEmpty( ), size( ), clear( )

Additional ops: succ(x) [ceiling(x)] – Element after x in sorted order. [Element in dict that is ≥ x, but smallest possible

pred(x), floor(x). – similar.

Binary Search Trees: a binary tree that implements dictionaries.

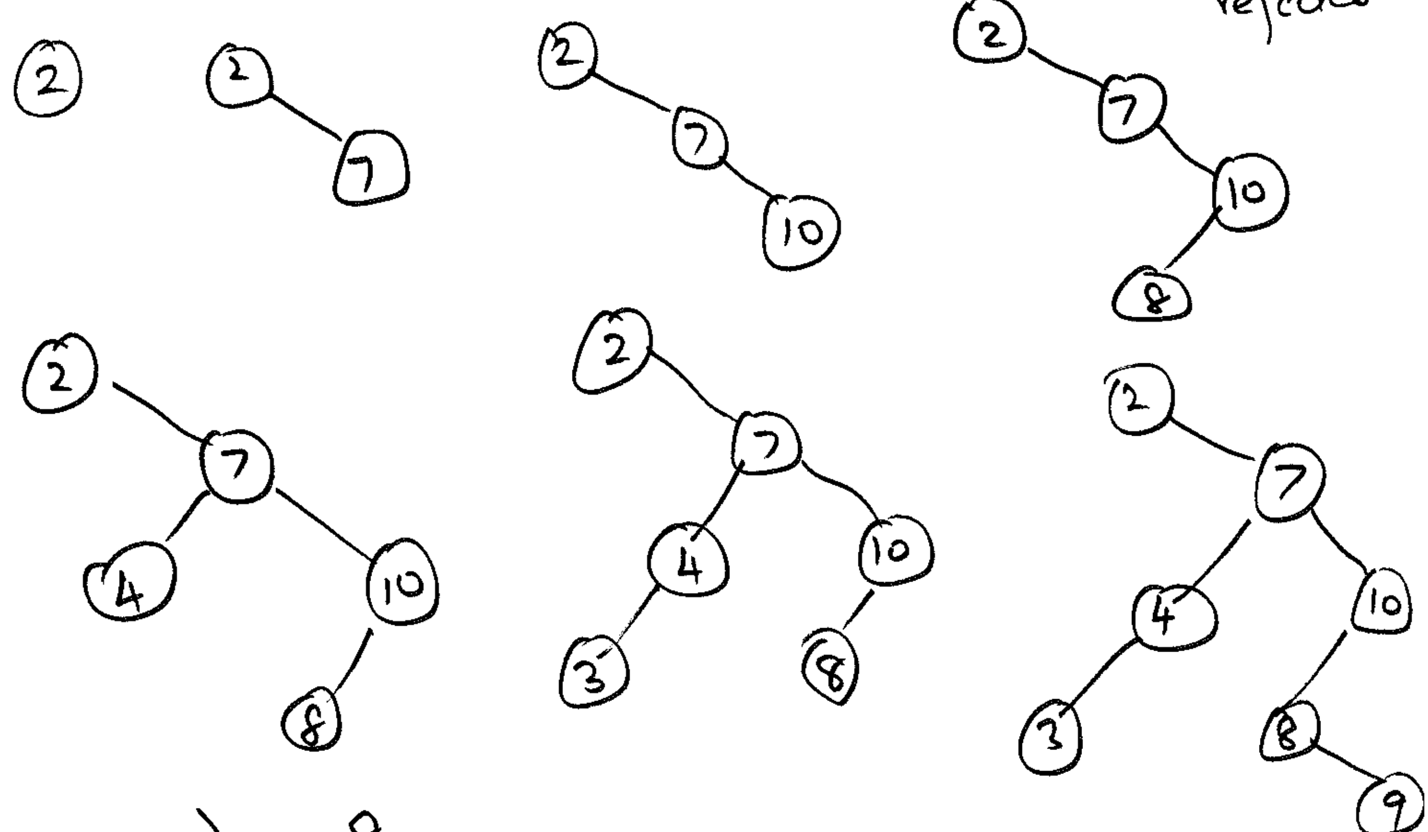
Element at node $>$ all elements in left subtree
$<$ all elements in right subtree.

Rule satisfied at every node.

Example: Add the following words in order into a dict:

2, 7, 10, 8, 4, 3, 8, 9

rejected



succ(7) = 8

floor(7) = 7

ceiling(7) = 7

pred(4) = 3

floor(6) = 4

ceiling(6) = 7

Fact: Inorder traversal of a BST visits nodes in sorted order

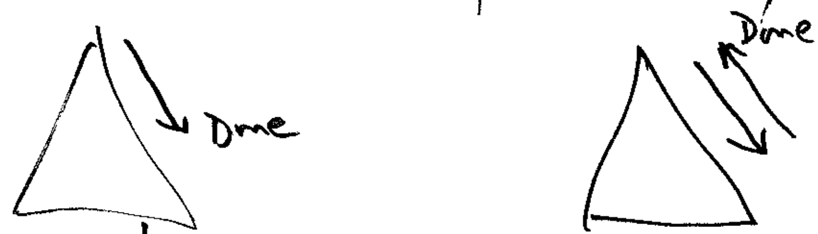
Implementation of Binary Search Trees (BST)

Design goals:

1. Write code that can be extended to other implementations of dictionaries, such as AVL Trees / Red-Black trees.

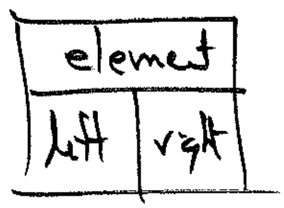
   Java's TreeMap

2. Single-pass algorithms, where possible. Avoid recursion — for efficiency.



single pass   Double pass.

3. Save space — avoid storing parent link at node.



Entry class does not have ~~storage~~ field for parent.

4. Avoid duplication of code.

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). Initial call: traversal(root)
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.