Lecture 10: Trees and Binary Search Trees

Read: Chapter 10, Carrano

Have seen *linear data structures* such as:
Array, vector, list, stack, queue.

We now consider *nonlinear data structure* such as:
Trees, tables, graphs, …

Recall that a tree T is a set of $n \geq 0$ elements defined as follows:

(1) If $n = 0$, T is an **empty tree**.

(2) If $n > 0$, then there exists a distinct element $r \in T$, called the root of T, such that $T - \{r\}$ can be partitioned into 0 or more disjoint subsets T_1, T_2, ..., where each of these subsets also forms a tree.

Trees are important structure in modeling

- Hierarchical relationship in computations.
- Class relationship
- Recursion
- Backtracking
- Comparison-based algorithms
- Data structure designs

Two Important Classes of Trees:

1. **k-ary trees, $k \geq 2$:**
   A tree with each node having at most k children.

2. **Binary tree:** An ordered 2-ary tree.
   T is a binary tree iff
   (i) $T = \varnothing$, or
   (ii) If $T \neq \varnothing$, T has a root r such that $T - \{r\}$ can be partitioned into two disjoint binary trees T_L and T_R, called the left subtree and right subtree of T.

Remark: Binary tree is an ordered tree.

Q: When a set of records is being maintained using a tree-based data structure, how do we systematically traverse a given binary tree so as to retrieve the data info stored in its nodes?

Some Standard Binary Traversals Algorithms:

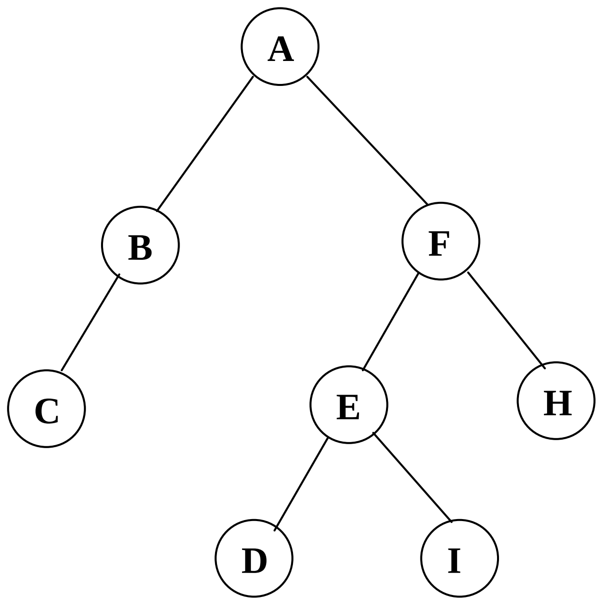
1. **Preorder traversal:**
   traverse/retrieve root,
   traverse left subtree in preorder recursively,
   traverse right subtree in preorder recursively.

2. **Postorder traversal:**
   traverse left subtree in postorder recursively,
   traverse right subtree in postorder recursively,
   traverse/retrieve root.

3. **Inorder traversal:**
   traverse left subtree in inorder recursively,
   traverse/retrieve root,
   traverse right subtree in inorder recursively.

4. **Level-order traversal:**
   Starting at level 1, traverse all the nodes at each level from left to right level by level.

Example: Binary tree traversals.



Preorder:	A B C F E D I H
Postorder:	C B D I E H F A
Inorder:	C B A D E I F H
Level order:	A B F C E H D I

Extension: General tree traversals.

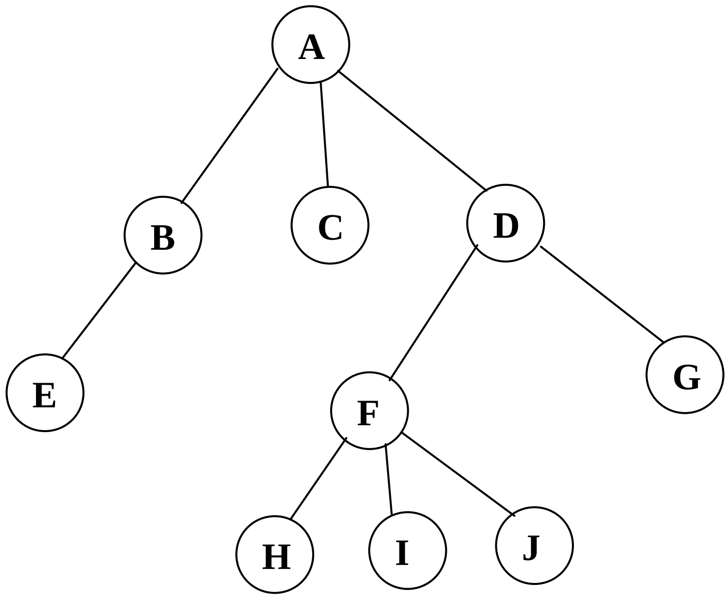
1. **Preorder traversal:**
   traverse/retrieve root,
   traverse subtree T_1 in preorder recursively,
   traverse subtree T_2 in preorder recursively,
   • • •
   traverse subtree T_k in preorder recursively.

2. **Postorder traversal:**
   traverse subtree T_1 in postorder recursively,
   traverse subtree T_2 in postorder recursively,
   • • •
   traverse subtree T_k in postorder recursively,
   traverse/retrieve root.

3. **Level order traversal:**
   Starting at level 1, traverse all the nodes at each level from left to right level by level.

Remark: Since the concept of ***in-between*** is not well-defined for a general tree, we don't usually use inorder traversal for general tree!

Example: General tree traversals.



Preorder:	A B E C D F H I J G
Postorder:	E B C H I J F G D A
Level order:	A B C D E F G H I J

Design an ADT: Binary tree.

UML diagram for the class *BinaryTree*:

Binary tree
root *left subtree* *right subtree*
createTree() *destroyBinaryTree()* *isEmpty()* *getRootData()* *setRootData()* *attachRight()* *attachLeftSubtree()* *attachRightSubtree()* *detachLeftSubtree()* *detachRightSubtree()* *getLeftSubtree()* *getRightSubtree()* *preorderTraverse()* *inorderTraverse()* *postorderTraverse()*

ADT Binary Tree Operations (See Page 508):
+createBinaryTree()
+createBinaryTree(in rootItem: TreeItemType)
+createBinaryTree(in rootItem: TreeItemType,
    inout leftTree: BinaryTree,inout rightTree: BinaryTree)
+destroyBinaryTree()
+isEmpty(): boolean {query}
+getRootData(): TreeItemType throw TreeException
+setRootData(in newItem: TreeItemType)
    throw TreeException
+attachLeft(in newItem: TreeItemType)
    throw TreeException
+attachRight(in newItem: TreeItemType)
    throw TreeException
+attachLeftSubtree(inout leftTree: BinaryTree)
    throw TreeException
+attachRightSubtree(inout rightTree: BinaryTree)
    throw TreeException
+detachLeftSubtree(out leftTree: BinaryTree) throw
TreeException
+detachRightSubtree(out rightTree: BinaryTree)
    throw TreeException
+getLeftSubtree(): BinaryTree
+getRightSubtree(): BinaryTree
+preorderTraverse(in visit:FunctionType)
+inorderTraverse(in visit:FunctionType)
+postorderTraverse(in visit:FunctionType)

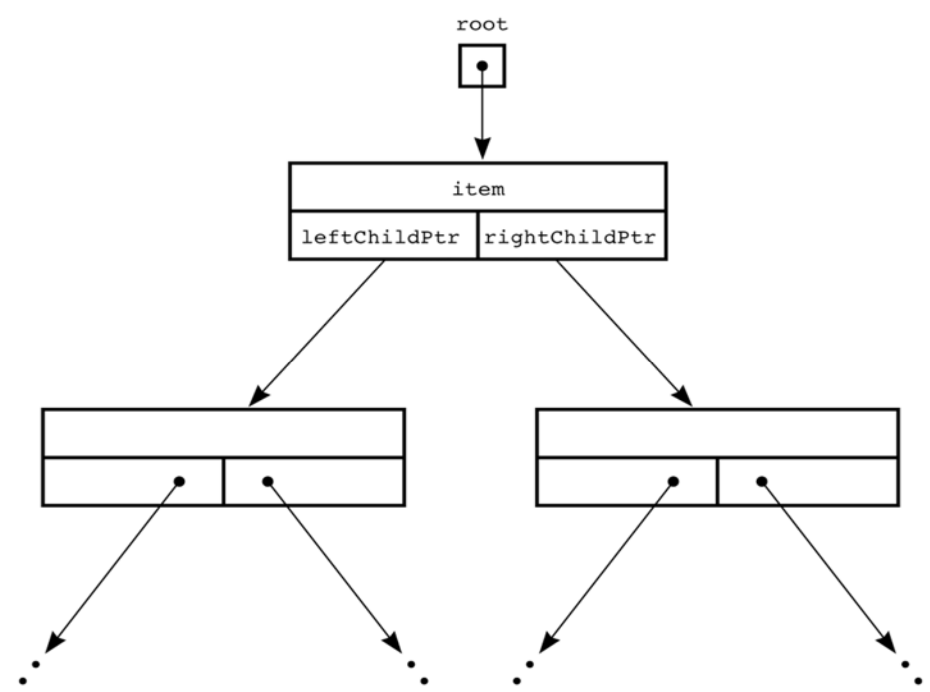
Binary tree implementations:

Depend on movement(s) in tree.

$\uparrow$ (Needs parent information)
movement $\leftarrow$ $\rightarrow$ (Needs siblings information)
$\downarrow$ (Needs children information)

Remark: Depending on application, one or more of the above information may be needed.

Pointer Implementation of Binary Tree:



The external pointer root points at the root r of the tree. If the tree is empty, root is NULL; otherwise, root$\rightarrow$leftChildPtr (root$\rightarrow$rightChildPtr) points to the root of the left (right) subtree of r.

TreeNode Class (See Page 518):

```
typedef string TreeItemType;

class TreeNode                          // node in the tree
{
private:
   TreeNode() {};
   TreeNode(const TreeItemType& nodeItem,
     TreeNode *left = NULL, TreeNode *right = NULL):
     item(nodeItem), leftChildPtr(left),rightChildPtr(right) {}

   TreeItemType item;                   // data portion
   TreeNode *leftChildPtr;              // pointer to left child
   TreeNode *rightChildPtr;             // pointer to right child

   friend class BinaryTree;             // friend class

}; // end TreeNode class
```

Remark: See Page 519 for BinaryTree.h for the ADT binary tree.

```
#include "TreeException.h"
#include "TreeNode.h"
```

```
typedef void (*FunctionType)(TreeItemType& anItem);

class BinaryTree
{
public:
// constructors and destructor:
   BinaryTree();
   BinaryTree(const TreeItemType& rootItem);
   BinaryTree(const TreeItemType& rootItem,
            BinaryTree& leftTree,
            BinaryTree& rightTree);
   BinaryTree(const BinaryTree& tree);
   virtual ~BinaryTree();

// binary tree operations:
   virtual bool isEmpty() const;
   virtual TreeItemType getRootData() const
     throw(TreeException);
   virtual void setRootData(const TreeItemType& newItem)
     throw (TreeException);
   virtual void attachLeft(const TreeItemType& newItem)
     throw(TreeException);
   virtual void attachRight(const TreeItemType& newItem)
     throw(TreeException);
   virtual void attachLeftSubtree(BinaryTree& leftTree)
     throw(TreeException);
   virtual void attachRightSubtree(BinaryTree& rightTree)
     throw(TreeException);
   virtual void detachLeftSubtree(BinaryTree& leftTree)
```

```
    throw(TreeException);
  virtual void detachRightSubtree(BinaryTree& rightTree)
    throw(TreeException);
  virtual BinaryTree getLeftSubtree() const;
  virtual BinaryTree getRightSubtree() const;
  virtual void preorderTraverse(FunctionType visit);
  virtual void inorderTraverse(FunctionType visit);
  virtual void postorderTraverse(FunctionType visit);

// overloaded operator:
  virtual BinaryTree& operator=(const BinaryTree& rhs);

protected:
  BinaryTree(TreeNode *nodePtr); // constructor
  void copyTree(TreeNode *treePtr,
                TreeNode* & newTreePtr) const;
  // Copies the tree rooted at treePtr into a tree rooted
  // at newTreePtr. Throws TreeException if a copy of the
  // tree cannot be allocated.
```

```
  void destroyTree(TreeNode * &treePtr);
  // Deallocate memory for a tree.

  // The next two functions retrieve and set the value
  // of the private data member root.
  TreeNode *rootPtr( ) const;
  void setRootPtr(TreeNode *newRoot);

  // The next two functions retrieve and set the values
  // of the left and right child pointers of a tree node.
  void getChildPtrs(TreeNode *nodePtr,
                    TreeNode * &leftChildPtr,
                    TreeNode * &rightChildPtr) const;
  void setChildPtrs(TreeNode *nodePtr,
                   TreeNode *leftChildPtr,
                   TreeNode *rightChildPtr);

  void preorder(TreeNode *treePtr, FunctionType visit);
  void inorder(TreeNode *treePtr, FunctionType visit);
  void postorder(TreeNode *treePtr, FunctionType visit);

private:
  TreeNode *root; // pointer to root of tree
}; // end class
// End of header file.
```

```
// Implementation file BinaryTree.cpp for the ADT binary
// tree.

#include "BinaryTree.h" // header file
#include <cstddef> // definition of NULL
#include <cassert> // for assert()

BinaryTree::BinaryTree() : root(NULL)
{
} // end default constructor

BinaryTree::BinaryTree(const TreeItemType& rootItem)
{
  root = new TreeNode(rootItem, NULL, NULL);
  assert(root != NULL);
} // end constructor

BinaryTree::BinaryTree(const TreeItemType& rootItem,
            BinaryTree& leftTree, BinaryTree& rightTree)
{
  root = new TreeNode(rootItem, NULL, NULL);
  assert(root != NULL);

  attachLeftSubtree(leftTree);
  attachRightSubtree(rightTree);
} // end constructor
```

```
BinaryTree::BinaryTree(const BinaryTree& tree)
{
  copyTree(tree.root, root);
} // end copy constructor

BinaryTree::BinaryTree(TreeNode *nodePtr): root(nodePtr)
{
} // end protected constructor

BinaryTree::~BinaryTree()
{
  destroyTree(root);
} // end destructor

bool BinaryTree::isEmpty() const
{
  return (root == NULL);
} // end isEmpty

TreeItemType BinaryTree::getRootData() const
{
  if (isEmpty())
    throw TreeException("TreeException: Empty tree");
  return root->item;
} // end getRootData
```

```
void BinaryTree::setRootData(const TreeItemType& newItem)
{
  if (!isEmpty())
    root->item = newItem;
  else
  {  root = new TreeNode(newItem, NULL, NULL);
    if (root == NULL)
      throw TreeException(
        "TreeException: Cannot allocate memory");
  } // end if
} // end setRootData

void BinaryTree::attachLeft(const TreeItemType& newItem)
{
  if (isEmpty())
    throw TreeException("TreeException: Empty tree");
  else if (root->leftChildPtr != NULL)
    throw TreeException(
      "TreeException: Cannot overwrite left subtree");
  else       // Assertion: nonempty tree; no left child
  {  root->leftChildPtr = new TreeNode(newItem,
                                       NULL, NULL);
    if (root->leftChildPtr == NULL)
      throw TreeException(
        "TreeException: Cannot allocate memory");
  } // end if
} // end attachLeft
```

```
void BinaryTree::attachRight(const TreeItemType& newItem)
{
  if (isEmpty())
    throw TreeException("TreeException: Empty tree");
  else if (root->rightChildPtr != NULL)
    throw TreeException(
      "TreeException: Cannot overwrite right subtree");
  else        // Assertion: nonempty tree; no right child
  {  root->rightChildPtr = new TreeNode(newItem,
                                          NULL, NULL);
    if (root->rightChildPtr == NULL)
      throw TreeException(
        "TreeException: Cannot allocate memory");
  } // end if
} // end attachRight

void BinaryTree::attachLeftSubtree(BinaryTree& leftTree)
{
  if (isEmpty())
    throw TreeException("TreeException: Empty tree");
  else if (root->leftChildPtr != NULL)
    throw TreeException(
      "TreeException: Cannot overwrite left subtree");
  else        // Assertion: nonempty tree; no left child
  {  root->leftChildPtr = leftTree.root;
    leftTree.root = NULL;
  }
} // end attachLeftSubtree
```

```
void BinaryTree::attachRightSubtree(BinaryTree& rightTree)
{
  if (isEmpty())
    throw TreeException("TreeException: Empty tree");
  else if (root->rightChildPtr != NULL)
    throw TreeException(
      "TreeException: Cannot overwrite right subtree");
  else        // Assertion: nonempty tree; no right child
  {  root->rightChildPtr = rightTree.root;
     rightTree.root = NULL;
  } // end if
} // end attachRightSubtree

void BinaryTree::detachLeftSubtree(BinaryTree& leftTree)
{
  if (isEmpty())
    throw TreeException("TreeException: Empty tree");
  else
  {  leftTree = BinaryTree(root->leftChildPtr);
    root->leftChildPtr = NULL;
  } // end if
} // end detachLeftSubtree
```

```
void BinaryTree::detachRightSubtree(BinaryTree& rightTree)
{
  if (isEmpty())
    throw TreeException("TreeException: Empty tree");
  else
  {  rightTree = BinaryTree(root->rightChildPtr);
    root->rightChildPtr = NULL;
  } // end if
} // end detachRightSubtree

BinaryTree BinaryTree::getLeftSubtree() const
{
  TreeNode *subTreePtr;
  if (isEmpty())
    return BinaryTree();
  else
  { copyTree(root->leftChildPtr, subTreePtr);
    return BinaryTree(subTreePtr);
  } // end if
} // end leftSubtree
```

```
BinaryTree BinaryTree::rightSubtree() const
{
  TreeNode *subTreePtr;
  if (isEmpty())
    return BinaryTree();
  else
  {  copyTree(root->rightChildPtr, subTreePtr);
    return BinaryTree(subTreePtr);
  } // end if
} // end rightSubtree

void BinaryTree::preorderTraverse(FunctionType visit)
{
  preorder(root, visit);
} // end preorderTraverse

void BinaryTree::inorderTraverse(FunctionType visit)
{
  inorder(root, visit);
} // end inorderTraverse

void BinaryTree::postorderTraverse(FunctionType visit)
{
  postorder(root, visit);
} // end postorderTraverse
```

```
BinaryTree& BinaryTree::operator=(const BinaryTree& rhs)
{
  if (this != &rhs)
  {  destroyTree(root);          // deallocate left-hand side
     copyTree(rhs.root, root); // copy right-hand side
  } // end if
  return *this;
} // end operator=

void BinaryTree::copyTree(TreeNode *treePtr,
                          TreeNode *& newTreePtr) const
{
  // preorder traversal
  if (treePtr != NULL)
  {  // copy node
      newTreePtr = new TreeNode(treePtr->item,
                                NULL, NULL);
     if (newTreePtr == NULL)
       throw TreeException(
         "TreeException: Cannot allocate memory");

     copyTree(treePtr->leftChildPtr,
                          newTreePtr->leftChildPtr);
     copyTree(treePtr->rightChildPtr,
                          newTreePtr->rightChildPtr);
  }
  else   newTreePtr = NULL; // copy empty tree
} // end copyTree
```

```
void BinaryTree::destroyTree(TreeNode *& treePtr)
{
  // postorder traversal
  if (treePtr != NULL)
  {  destroyTree(treePtr->leftChildPtr);
     destroyTree(treePtr->rightChildPtr);
     delete treePtr;
     treePtr = NULL;
  } // end if
} // end destroyTree

TreeNode *BinaryTree::rootPtr() const
{
 return root;
} // end rootPtr

void BinaryTree::setRootPtr(TreeNode *newRoot)
{
  root = newRoot;
} // end setRoot

void BinaryTree::getChildPtrs(TreeNode *nodePtr,
          TreeNode *& leftPtr, TreeNode *& rightPtr) const
{
  leftPtr = nodePtr->leftChildPtr;
  rightPtr = nodePtr->rightChildPtr;
} // end getChildPtrs
```

```
void BinaryTree::setChildPtrs(TreeNode *nodePtr,
                    TreeNode *leftPtr, TreeNode *rightPtr)
{
  nodePtr->leftChildPtr = leftPtr;
  nodePtr->rightChildPtr = rightPtr;
} // end setChildPtrs

void BinaryTree::preorder(TreeNode *treePtr,
                              FunctionType visit)
{
  if (treePtr != NULL)
  {  visit(treePtr->item);
     preorder(treePtr->leftChildPtr, visit);
     preorder(treePtr->rightChildPtr, visit);
  } // end if
} // end preorder

void BinaryTree::inorder(TreeNode *treePtr,
                              FunctionType visit)
{
  if (treePtr != NULL)
  {  inorder(treePtr->leftChildPtr, visit);
     visit(treePtr->item);
     inorder(treePtr->rightChildPtr, visit);
  } // end if
} // end inorder
```

```
void BinaryTree::postorder(TreeNode *treePtr,
                                   FunctionType visit)
{
  if (treePtr != NULL)
  {  postorder(treePtr->leftChildPtr, visit);
     postorder(treePtr->rightChildPtr, visit);
     visit(treePtr->item);
  } // end if
} // end postorder
// End of implementation file.
```

Example:

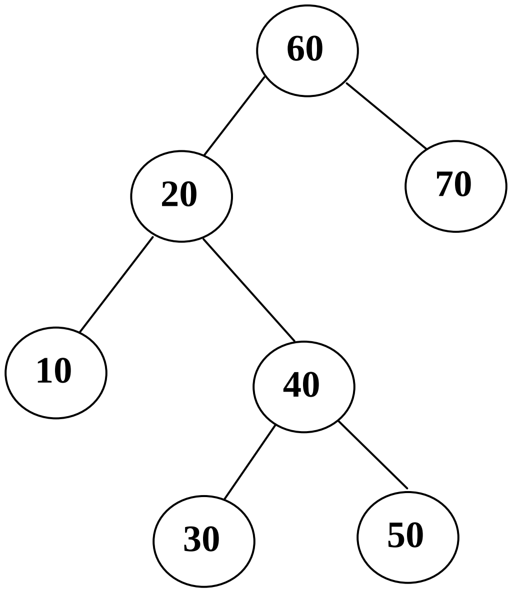
```
#include "BinaryTree.h" // binary tree operations
#include <iostream>
using namespace std;
void display(TreeItemType& anItem);

int main()
{
   BinaryTree tree1, tree2, left;      // empty trees
   BinaryTree tree3(70);               // tree with only a root 70

// build the tree in Figure 10-10
   tree1.setRootData(40);
   tree1.attachLeft(30);
   tree1.attachRight(50);
   tree2.setRootData(20);
   tree2.attachLeft(10);
   tree2.attachRightSubtree(tree1);
   BinaryTree binTree(60, tree2, tree3);

   // sample tree traversals
    binTree.inorderTraverse(display);
   binTree.leftSubtree().inorderTraverse(display);
   binTree.detachLeftSubtree(left);
   left.inorderTraverse(display);
   binTree.inorderTraverse(display);
   return 0;
} // end main
```

Binary tree from above function:



Output:

10 20 30 40 50 60 70
10 20 30 40 50
10 20 30 40 50
60 70

Designing Binary Tree Based Advance ADT:
Let S be a set of records to be maintained. To facilitate searching, let's assume that each record has a key associated with it and all the keys can be linearly ordered.

Record	↔	Instance of a class
Field	↔	Member variable
Key	↔	Form of identification

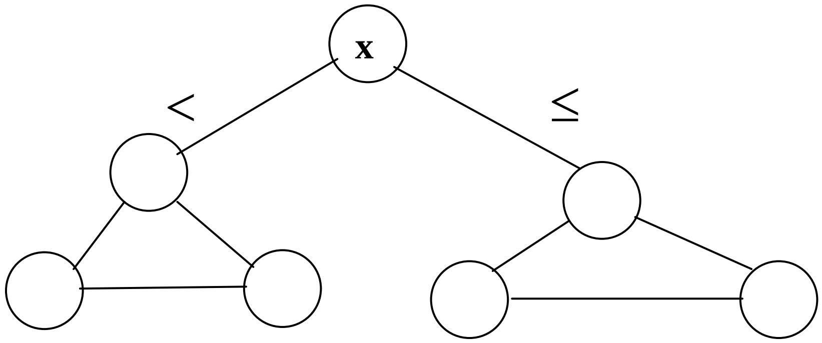
Typical Operations Required:
insertItemKey, searchMinKey, searchMaxKey, searchKey, deleteMinKey, deleteMaxKey, deleteItemKey.

Possible Approach: Sorted list.

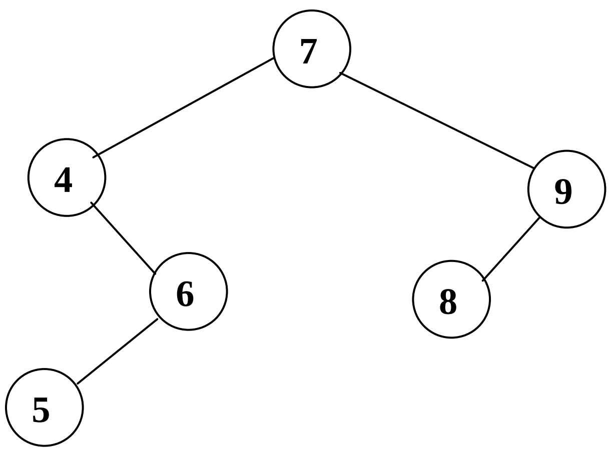
Better Approach:
Binary search tree (more efficient on the average).

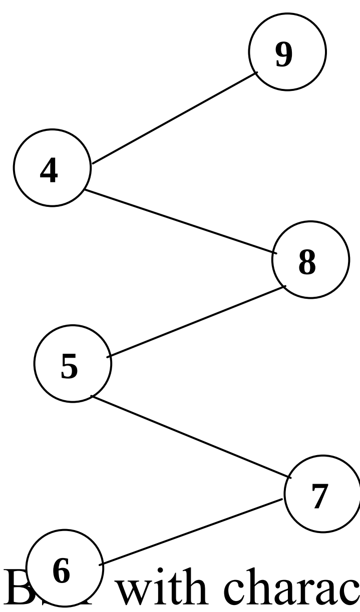
Defn: A ***binary search tree*** is a binary tree H such that the key (priority) of any node x is greater than the priority of all its left descendants and is smaller than or equal to the priority of all its right descendants. This property is often called ***binary search tree property***.

Binary Search Tree Property:

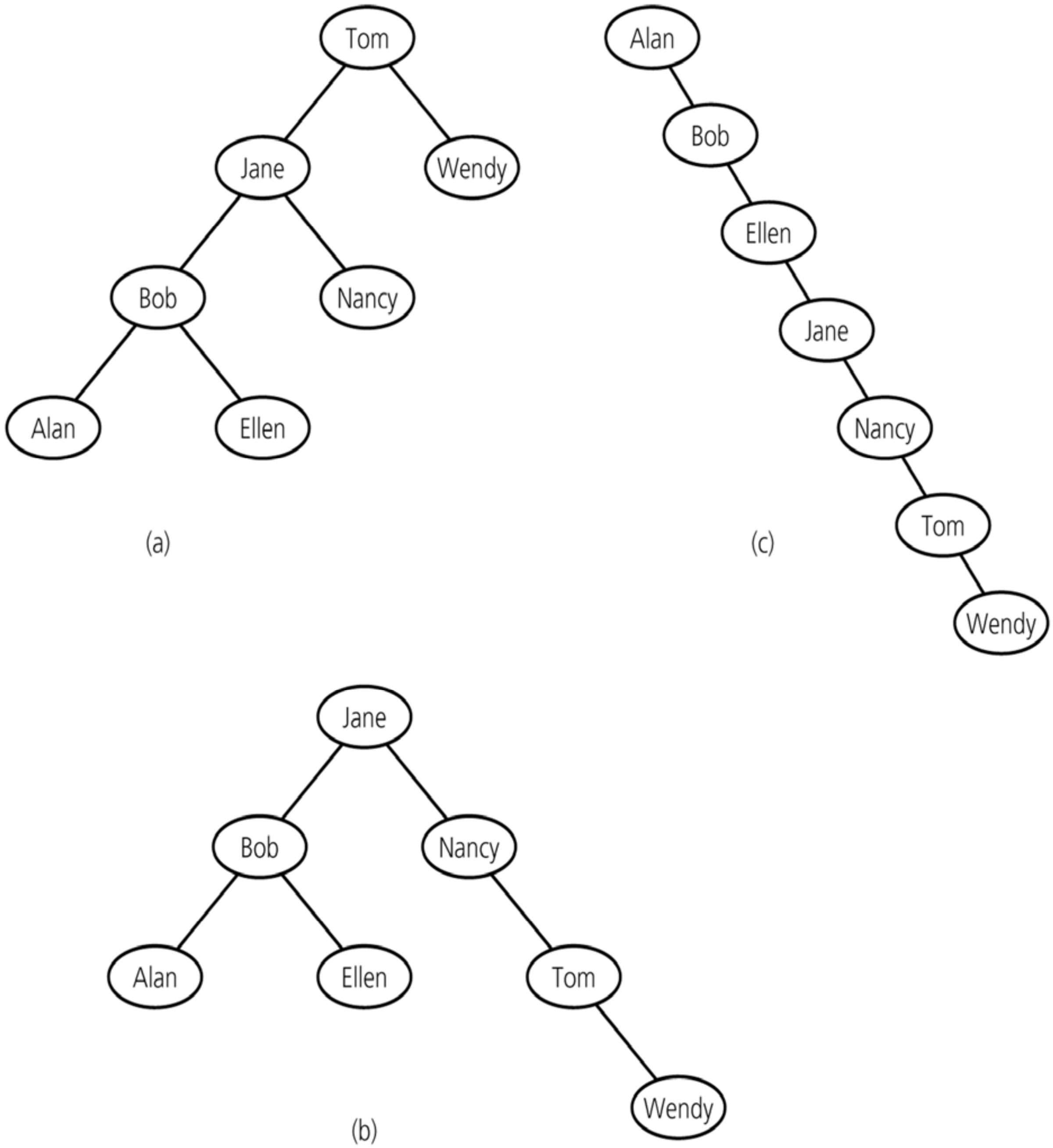


Examples: BST with integer keys.





Examples: BST with characters keys.


Tom
Jane
Wendy
Bob
Nancy
Alan
Ellen
(a)
Alan
Bob
Ellen
Jane
Nancy
Tom
Wendy
(c)
Jane
Bob
Nancy
Alan
Ellen
Tom
Wendy
(b)

Observations:

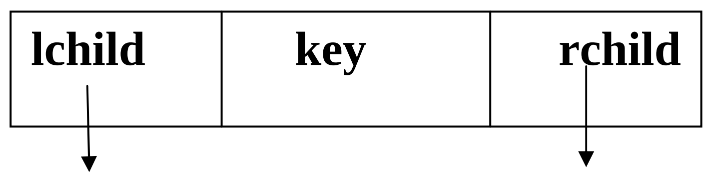
1. BST may NOT be a balanced binary tree.
2. Leftmost descendant of root = Min item.
3. Rightmost descendant of root = Max item.
4. Inorder traversal = Sorted order.
5. BST structure models binary search.

UML for ADT BST:

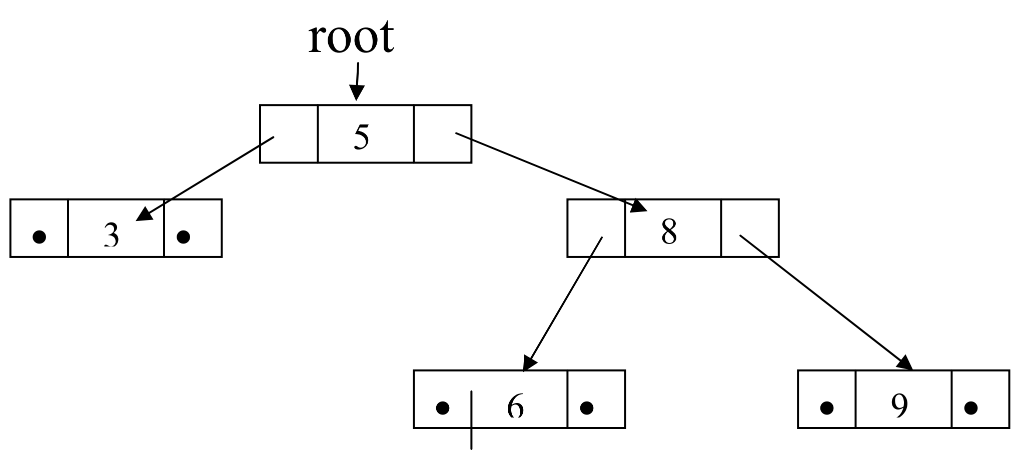
BinarySearchTree
root *left subtree* *right subtree*
createBinarySearchTree() *destroyBinarySearchTree()* *isEmpty()* *searchTreeInsert()* *searchTreeDelete()* *searchTreeRetrieve()* *preorderTraverse()* *inorderTraverse()* *postorderTraverse()*

BST Implementations:
Using Pointer-based Implementation.

TreeNode:



Example:



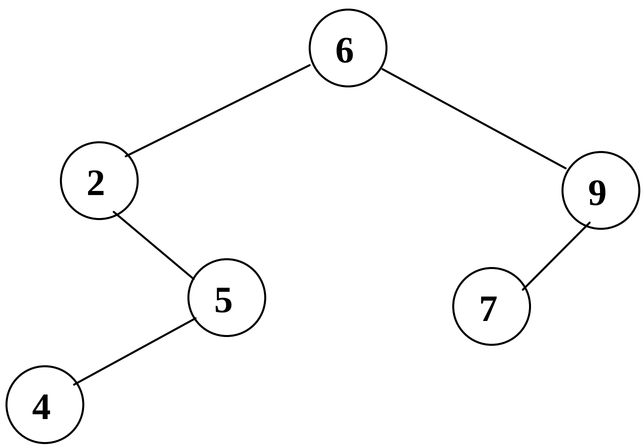
BST Operations:

1. **Search Operation:**

   Think of binary search!

Consider the following BST, how do we search for a record with a given key?

T:



Consider:

search(T,5);
search(T,8);

Algorithm:

```
search(in binTree: BST, in searchKey: KeyType)
{
  if (binTree = NULL)             // empty BST
    return not found;
  else if (binTree→key = searchKey)    // key found
        return found;
      else if (binTree→key > searchKey)   // search left tree
            return search(binTree→lchild,searchKey)
          else                                // search right tree
            return search(binTree→rchild,searchKey);
} // end search
```

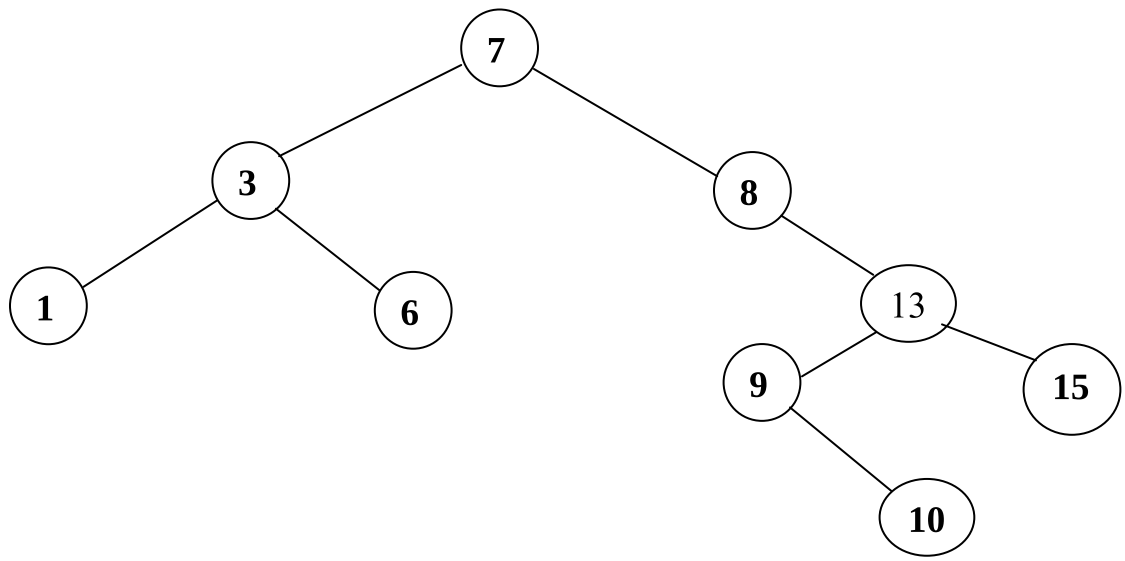
Q: How do we insert a new node containing key k into a BST?

Find position using search, create new TreeNode, and then insert.

2. **Insert Operation:**

```
insertItem(inout treePtr: TreeNodePtr, in newItem:
                                        TreeItemType)
{
  if (treePtr = NULL)  //          empty BST
    create new TreeNode and insert;
  else if (newItem.getKey() < treePtr→item.getKey())
        insertItem(treePtr→lchild, newItem);
      else insertItem(treePtr→rchild, newItem);
} // end search
```

Example: Insert items with keys 7, 3, 1, 8, 13, 15, 6, 9, 10, in the given order, into an initially empty BST to obtain the following BST.



3. **Delete Operation:**

Q: How do we delete the item with key k from a BST?

Let's first consider deleteMin operation first.

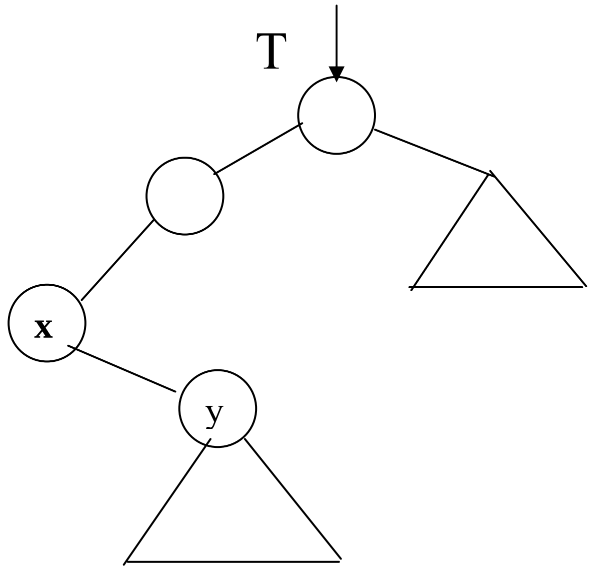
Q: Where is the item with minimum key?

It must be the leftmost descendant of the root!

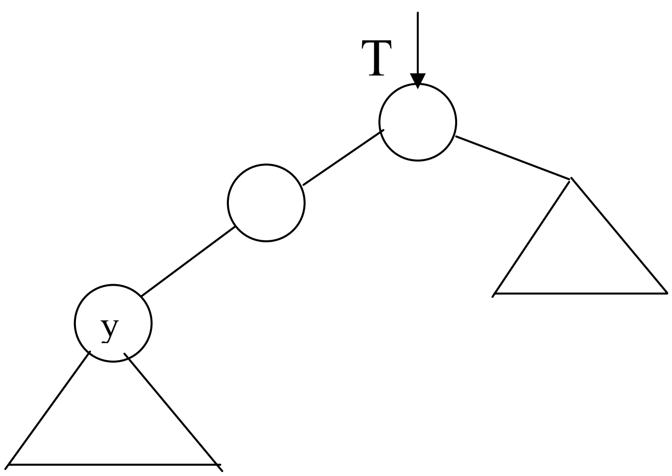
Let x be the item with min key

Observe that x must have 0 or 1 child. Either case, we can simply replace x with its right child (may be empty) in the BST.

Before:



After:



Consider general delete(T,k) operation.
Let N be the node having key k.
Three cases:

1. N has no child: Remove N.
2. N has exactly one child: Replace N with its only child.
3. N has two children: Replace N with the min priority item of its right subtree (using deleteMin operation).

Remark: We can also use deleteMax operation to support the implementation of the general remove operation.

Complexity Analysis:
For BST operations, observe that

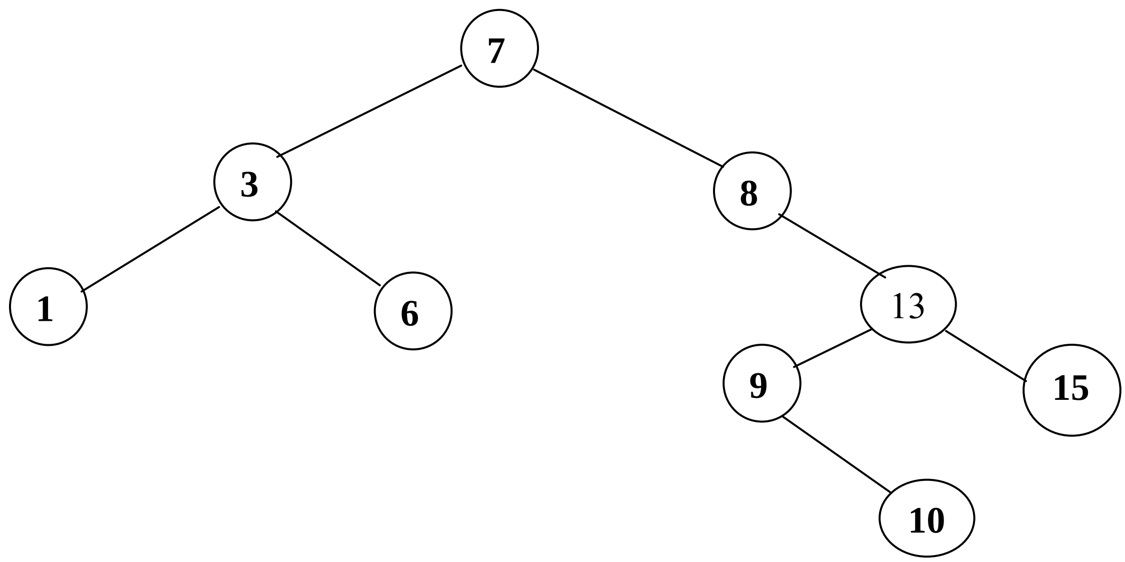
$$T_w(n) = O(\text{height of BST}) = O(n).$$

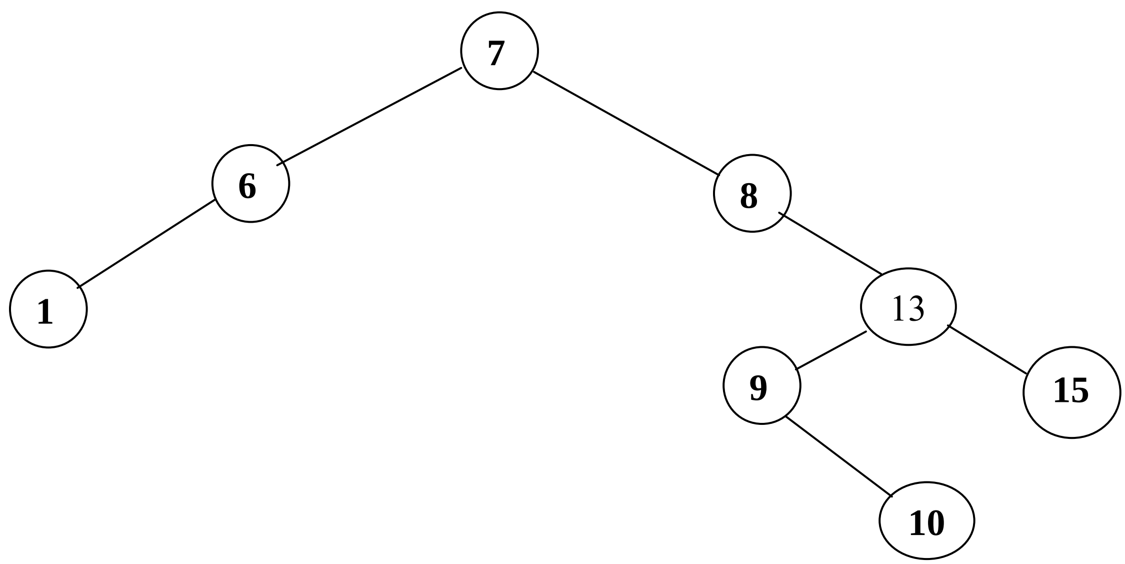
However,

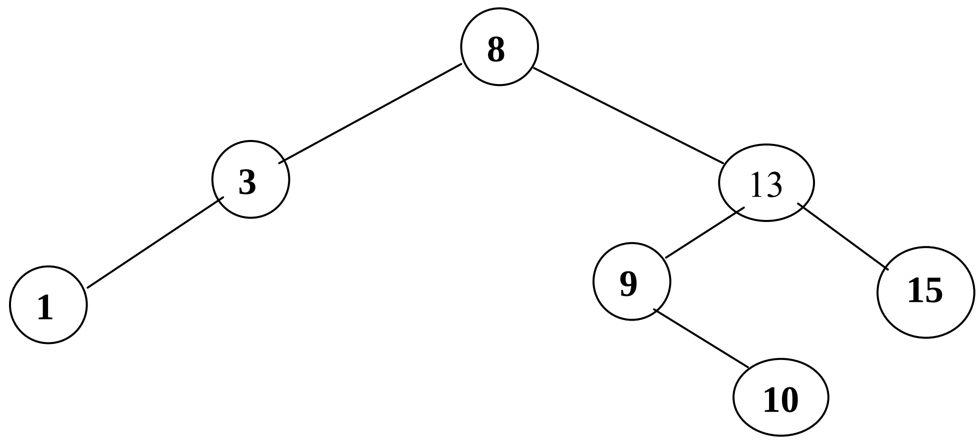
$$T_a(n) = O(\lg n).$$

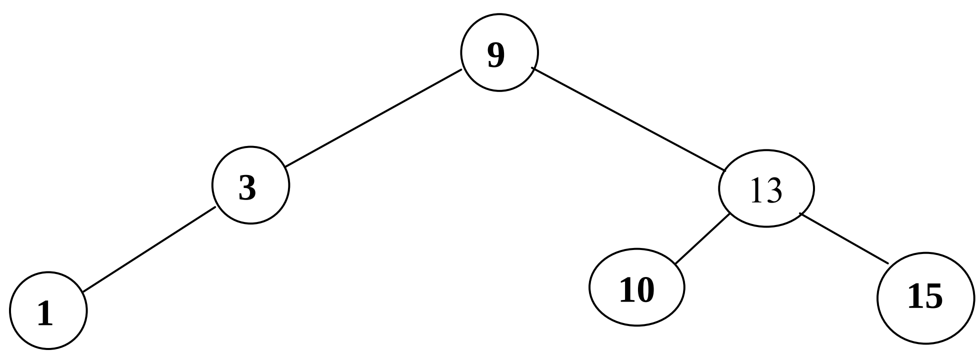
Q: How do we build a BST?
Insert the items one by one into an initially empty BST. Hence, $T_w(n) = O(n^2)$.

Example: Delete 3, 7, 8 from the following BST.



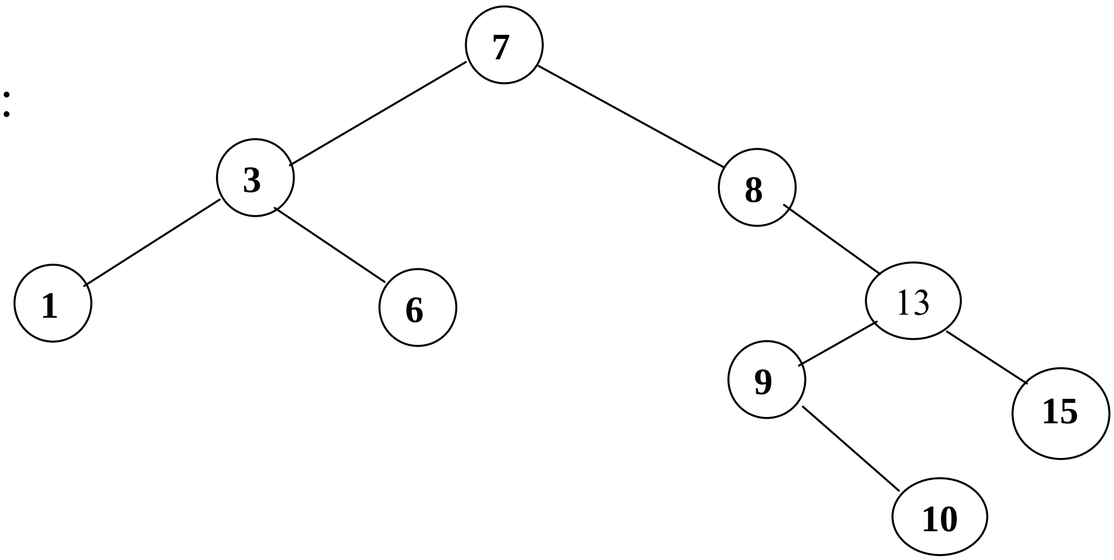



8
3
13
1
9
15
10


9
3
13
1
10
15

Saving and Restoring a BST:
Example:

T:



Consider the preorder traversal of T:
7, 3, 1, 6, 8, 13, 9, 10, 15.

Observe that the original BST T can be restored by inserting these records, in the above preorder sequence, into an initially empty BST.

Array-based Implementation of Complete Binary Tree:

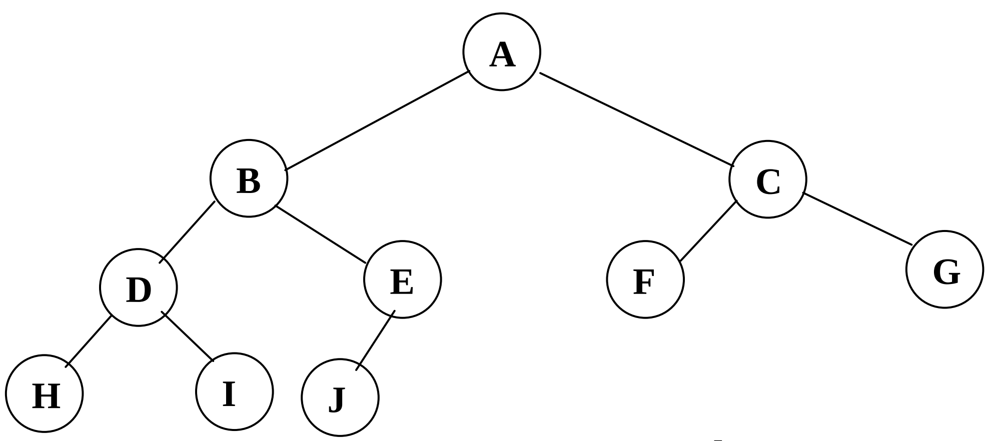
Given a complete binary tree T with n nodes, T can be represented using an array A[0:n−1] such that

(1) Root of T is at A[0],
(2) For node A[i], its left child (right child) is at A[2i+1] (A[2i+2]) if exists.

Remarks:

(1) Parent of a node A[i] is at A[(i−1)/2] if exists.
(2) For n > 1, A[i] is a leaf node iff 2i ≥ n.

Example:



last

0	1	2	3	4	5	6	7	8	9	10		
A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**		• • •	

Advantages: Fastest in accessing parent and children, $\Theta(1)$ time.

Disadvantage: Only useful when tree is complete (or "almost complete"); otherwise, memory intensive. If a tree is of height h, it requires an array with capacity $2^h - 1$. Hence, a skew tree with 10 nodes (h = 10) will need an array of capacity $2^{10} - 1 = 1023$.

Remark: Since BST can be a skew tree, it is infeasible to implement a BST using array!

More on Tree Traversals:

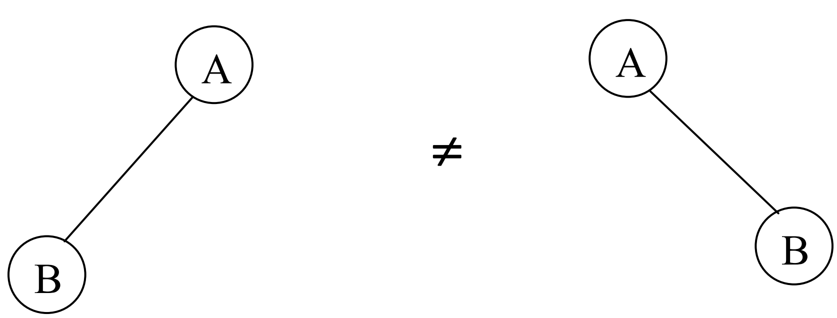
Q: How do we store a binary tree in a file so that it can be reconstructed later when needed?

Using the information of its tree traversal(s).

Q: If a single tree traversal of a binary tree T is given, can we reconstruct the corresponding binary tree T?

Not always!

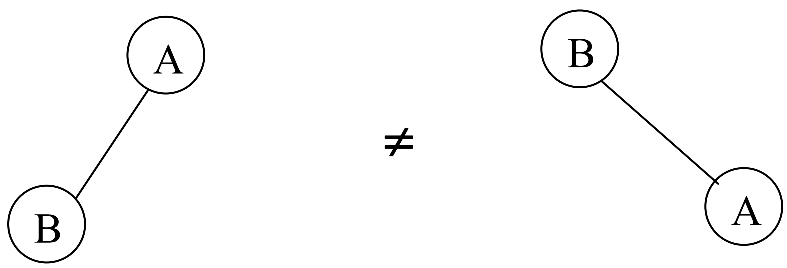
Consider the following two distinct binary trees:



Observe that both trees have preorder traversal AB, postorder traversal BA, and level-order traversal AB. Hence, these two binary trees are indistinguishable if only one of the above tree traversals is given!

Q: What if the inorder traversal of T is given?

Consider the following distinct binary trees:



Again, both trees have inorder traversal BA!

Q: How can we reconstruct the binary tree T from its traversal(s)?

Must know the root, the left and the right subtrees.

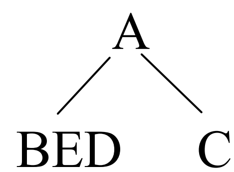
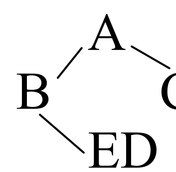
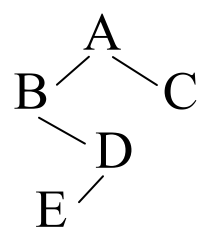
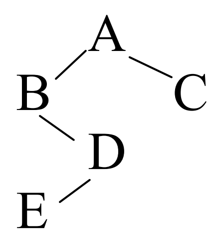
If we are given any one of the following pairs of traversals of T, T can be reconstructed if exists.

1. Preorder and inorder traversals.
2. Postorder and inorder traversals.
3. Level-order and inorder traversals.

Consider given a pair of preorder and inorder traversals of a binary tree T.

Q: How do we reconstruct T if exists?
Scan preorder traversal from left to right to determine root followed by scanning inorder traversal to determine left and right subtree.

Example: Let preorder = ABDEC and inorder = BEDAC.

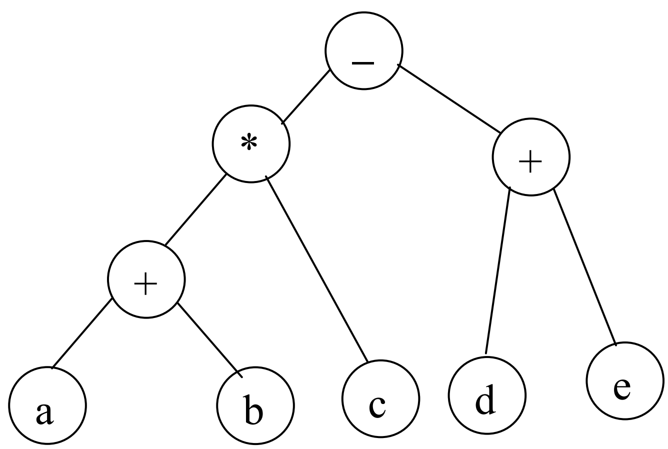
ABDEC	***BED***A***C***	
A***B***DEC	B***ED***AC	
AB***D***EC	B***E***DA***C***	
ABD***E***C	BEDA***C***	
ABDE***C***	BEDAC	

Example: A simple application of binary tree traversal.

Recall that we used infix, prefix, and postfix notations for algebraic expressions. We can also represent an algebraic expressions (with binary operations) using an ***expression tree***, which is a binary tree such that

(1) Each leaf node represents an operand, and
(2) Each non-leaf node represents an operator.

An expression tree for $((a + b) * c) - (d + e)$:



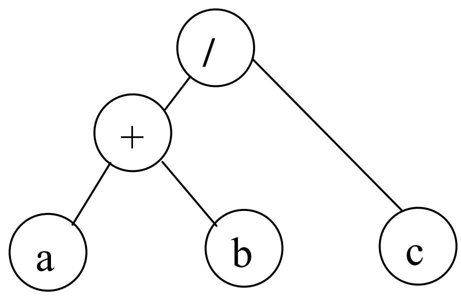
Preorder traversal:	$- * + a\ b\ c + d\ e$	(prefix)
Postorder traversal:	$a\ b + c * d\ e\ + -$	(postfix)
Inorder traversal:	$a + b * c - d + e$	(infix)

Example: Consider the algebraic expressions (a+b)/c and a+(b/c).

Let's construct expression trees and traverse them in preorder, postorder, and inorder fashions.

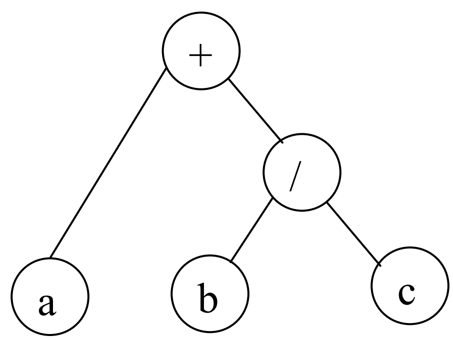
Q: What do you see?

T_1 representing (a+b)/c:



Infix: a + b / c
Prefix: / + a b c
Postfix: a b + c /

T_2 representing a+(b/c):



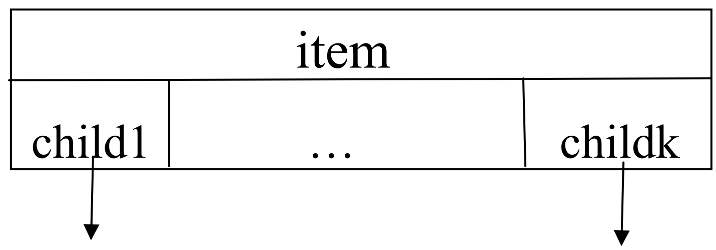
Infix: a + b / c
Prefix: + a / b c
Postfix: a b c / +

Remark: Observe that both trees have the same infix expression but different prefix and postfix expressions!

General Tree Implementations:

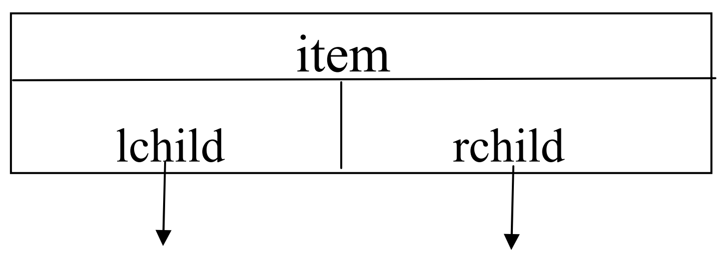
1. **k-tree:**

   NodeType:



2. **Binary tree representation of general tree:**

   NodeType:



In using a left-child right-sibling representation to represent a general tree N, at any node, lchild is used to pointed at the first child of N and all the siblings of N are linked together using the rchild pointers.

Example: