1.2 : Implementing Binary Trees

Tree Structure

- A binary tree *T* consists of a reference (the *root*) and (zero or more) nodes
  - it is empty when there are no nodes i.e. $n = 0$
  - in an empty tree, root has the value *null.*
  - the node referenced by root (in a non-empty tree) is called the *root node*

Tree structure

- A binary tree is a recursive structure
- Each node consists of
  - data
  - two members, each of which acts as a root reference of a subtree
- Each member may refer to another node or have the value 0
- If the reference is 0, the subtree is empty

Tree structure

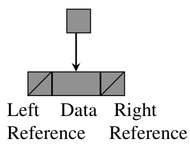
- The members are called *left* and *right* references
- Therefore, each node has a left and right subtree
- A node with no children is called a *leaf* or *external* node
  - i.e. left reference = 0 and right reference = 0

Empty Binary Tree

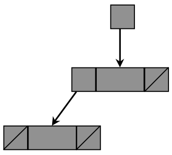
Root = 0

Binary Tree of zero nodes

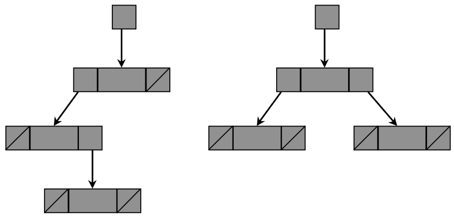
Binary tree with one node



Binary tree with two nodes



Binary trees with three nodes



Comparing Trees and linked lists

- Both composed of nodes connected by references
- LL node references one successor. Tree node references more than one
- LL linear, Trees are non-linear structures
- Representation and fundamental operations of Trees are similar to those of LL but complicated by existence of multiple successors

Binary Search Tree

- A Binary search tree is a Binary tree with the following properties
  - one of the data members is regarded as the *key* of the node (e.g., an ID *key* for searching)
  - all key data values to the left of the root are less than the root key
  - all key data values to the right of the root are greater than the root key
- This allows us to use the binary search method

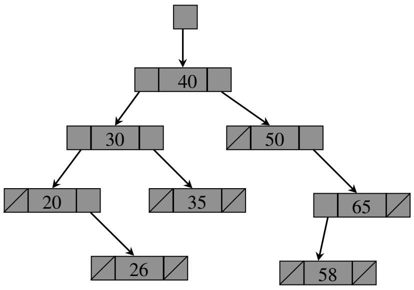
Search key

```
public abstract class KeyedItem{
   private Comparable searchKey;

   public KeyedItem(Comparable Key) {
       searchKey=key;
      } //end constructor
   public Comparable getKey() {
     return searchKey;
   }
```


Binary search - revision

- "Divide and conquer " strategy
  - in an array, repeatedly divide it into two pieces and focus on the piece that could contain the target value
  - in a binary search tree, the division is done by considering if the value to be searched for is less than or greater than the current value
    - » if less than, search the left subtree
    - » if greater than, search the right subtree

Binary search tree example



Functions of a binary search tree

- Initialise the tree
- Delete the tree
- Add a new value in its proper place in the tree
- Print the contents of the tree
- Search the tree for a particular data value
- Delete a value from the tree
  - need to reorganise tree to maintain the order

1.3 : Binary tree code

Some Preliminaries

- Define three classes
  - Class TreeNode containing the node's data and left and right references
  - Abstract Class BinaryTreeBasis defines the root of the tree
  - Class BinaryTree containing the methods to build a binary tree. This class extends the BinaryTreeBasis class.

Class TreeNode

```
public class TreeNode {
  private Object item;
  private TreeNode leftChild;
  private TreeNode rightChild;

  public TreeNode(Object newItem) {
  // Initializes tree node with item and no children.
    item = newItem;
    leftChild  = null;
    rightChild = null;
  }  // end constructor
```


Class TreeNode

```
public TreeNode(Object newItem,
                TreeNode left, TreeNode right) {
// Initializes tree node with item and
// the left and right children references.
  item = newItem;
  leftChild  = left;
  rightChild = right;
}  // end constructor
```

Class TreeNode

```
public Object getItem() {
// Returns the item field.
  return item;
}  // end getItem

public void setItem(Object newItem) {
// Sets the item field to the new value newItem.
item  = newItem;
}  // end setItem
```


Class TreeNode

```
public TreeNode getLeft() {
// Returns the reference to the left child.
  return leftChild;
}  // end getLeft

public void setLeft(TreeNode left) {
// Sets the left child reference to left.
  leftChild  = left;
}  // end setLeft
```


Class TreeNode

```
public TreeNode getRight() {
// Returns the reference to the right child.
  return rightChild;
}  // end getRight

public void setRight(TreeNode right) {
// Sets the right child reference to right.
  rightChild  = right;
}  // end setRight
}  // end TreeNode
```


Abstract Class BinaryTreeBasis

```
public abstract class BinaryTreeBasis {
  protected TreeNode root;

  public BinaryTreeBasis() {
    root = null;
  }  // end default constructor
```


Abstract Class BinaryTreeBasis

```
public BinaryTreeBasis(Object rootItem) {
    root = new TreeNode(rootItem, null, null);
  }  // end constructor

  public boolean isEmpty() {
// Returns true if the tree is empty, else returns false.
    return root == null;
  }  // end isEmpty
```


Abstract Class BinaryTreeBasis

```
public void makeEmpty() {
// Removes all nodes from the tree.
    root = null;
  }  // end makeEmpty
```

Abstract Class BinaryTreeBasis

```
public Object getRootItem() throws TreeException {
// Returns the item in the tree's root.
  if (root == null) {
    throw new TreeException("TreeException:
  Empty tree");
  }
  else {
    return root.getItem();
  }  // end if
 }  // end getRootItem
```



```
public class TreeException extends RuntimeException
   {
  public TreeException(String s) {
    super(s);
  }  // end constructor
} // end TreeException
```


Class BinaryTree

```
public class BinaryTree extends BinaryTreeBasis {
  public BinaryTree() {
  }  // end default constructor

  public BinaryTree(Object rootItem) {
    super(rootItem);
  }  // end constructor
```



```
public BinaryTree(Object rootItem,
                  BinaryTree leftTree,
                  BinaryTree rightTree) {
  root = new TreeNode(rootItem, null, null);
  attachLeftSubtree(leftTree);
  attachRightSubtree(rightTree);
 }  // end constructor
```



```
public void setRootItem(Object newItem) {
  if (root != null) {
    root.setItem(newItem);
  }
  else {
    root = new TreeNode(newItem, null, null);
  }  // end if
 }  // end setRootItem
```



```
public void attachLeft(Object newItem) {
  if (!isEmpty() && root.getLeft() == null) {
    // assertion: nonempty tree; no left child
    root.setLeft(new TreeNode(newItem, null, null));
  }  // end if
 }  // end attachLeft

 public void attachRight(Object newItem) {
  if (!isEmpty() && root.getRight() == null) {
    // assertion: nonempty tree; no right child
    root.setRight(new TreeNode(newItem, null, null));
  }  // end if
 }  // end attachRight
```

```
public void attachLeftSubtree(BinaryTree leftTree)
                        throws TreeException {
  if (isEmpty()) {
    throw new TreeException("TreeException:
  Empty tree");
  }
  else if (root.getLeft() != null) {
    // a left subtree already exists; it should have
  been
    // deleted first
    throw new TreeException("TreeException: " +
                  "Cannot overwrite left subtree");
  }
```



```
  else {
      // assertion: nonempty tree; no left child
      root.setLeft(leftTree.root);
      // don't want to leave multiple entry points into
      // our tree
      leftTree.makeEmpty();
    }  // end if
  }  // end attachLeftSubtree
```



```
public void attachRightSubtree(BinaryTree
   rightTree)
                        throws TreeException {
  if (isEmpty()) {
    throw new TreeException("TreeException:
  Empty tree");
  }
  else if (root.getRight() != null) {
    // a right subtree already exists; it should have
  been
    // deleted first
    throw new TreeException("TreeException: " +
                  "Cannot overwrite right subtree");
  }
```



```
  else {
      // assertion: nonempty tree; no right child
      root.setRight(rightTree.root);
      // don't want to leave multiple entry points into
      // our tree
      rightTree.makeEmpty();
    }  // end if
  }  // end attachRightSubtree
```



```
protected BinaryTree(TreeNode rootNode) {
   root = rootNode;
 }  // end protected constructor
```



```
public BinaryTree detachLeftSubtree()
                 throws TreeException {
  if (isEmpty()) {
    throw new TreeException("TreeException:  Empty
  tree");
  }
  else {
    // create a new binary tree that has root's left
    // node as its root
    BinaryTree leftTree;
    leftTree = new BinaryTree(root.getLeft());
    root.setLeft(null);
    return leftTree;
  }  // end if
}  // end detachLeftSubtree
```

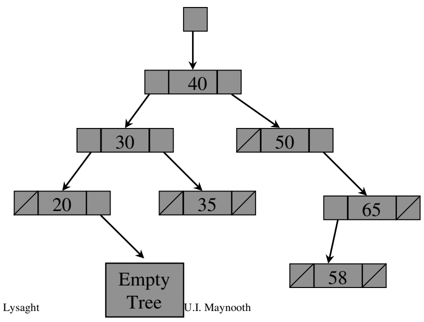

6

```
public BinaryTree detachRightSubtree()
                throws TreeException {
  if (isEmpty()) {
    throw new TreeException("TreeException:  Empty tree");
  }
  else {
    BinaryTree rightTree;
    rightTree = new BinaryTree(root.getRight());
    root.setRight(null);
    return rightTree;
  }  // end if
}  // end detachRightSubtree

} // end BinaryTree
```


Binary Search Tree

- Operations
  - Insert a new item into the binary search tree in its proper place
  - delete the item with a given search key from the binary search tree
  - retrieve the item with a given search key from the binary tree
  - Traverse the items in a binary search tree
  - Print the binary search tree.

Add to a Binary Search Tree



Add to a binary search tree

```
insertItem(treeNode, newItem)
//Inserts newItem into the binary search tree of which
  treeNode is the root. Let parentNode be the parent of the
  empty subtree at which search terminates when it seeks
  newItem's search key

if(Search terminated at parentNode's left
  subtree) { Set leftChild of parentNode
  to reference newItem}
else {Set rightChild of parentNode to
  reference newItem}
```


Add to a binary search tree

// A recursive function to add a node to a binary
// tree. First call passes over the root of the
// tree. Subsequent (recursive) calls pass the
// left or right subtree as appropriate
// From main() this is called by means of the insert
// method declared in the Binary Search Tree class
definition.

Add to a binary search tree

```
insertItem(treeNode, newItem)
//Inserts newItem into the binary search tree of which
  treeNode is the root.

if(treeNode is null){
//Create a new node and let treeNode reference it
// Create a new node with newItem as the data portion
  and
// set the references in the new node to null
}
```


7

```
else if (newItem < treeNode.getItem() ) {
treeNode.setLeft(insertItem(treeNode.getLeft
  (), newItem))
else{
  treeNode.setRight(insertItem(treeNode.getR
  ight(), newItem))
}
return treeNode;
```


Add to a binary search tree

```
protected TreeNode insertItem(TreeNode tNode,
                    KeyedItem newItem) {
  TreeNode newSubtree;
  if (tNode = = null) {
   // position of insertion found; insert after leaf
   // create a new node
    tNode = new TreeNode(newItem, null, null);
   return tNode;
  }  // end if
```



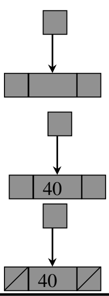
```
KeyedItem nodeItem = (KeyedItem)tNode.getItem();

  // search for the insertion position
  if (newItem.getKey().compareTo(nodeItem.getKey()) < 0) {
   // search the left subtree
   newSubtree = insertItem(tNode.getLeft(), newItem);
   tNode.setLeft(newSubtree);
   return tNode;
  }
```



```
else { // search the right subtree
   newSubtree = insertItem(tNode.getRight(),
  newItem);
   tNode.setRight(newSubtree);
   return tNode;
  }  // end if
 }  // end insertItem
```

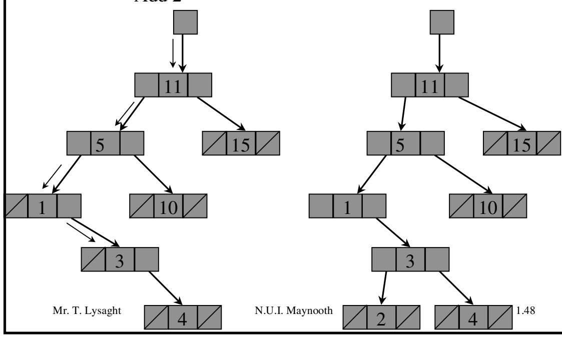

Add to an empty tree

```
if (p == NULL)
// root is NULL

{
tNode = new TreeNode(data, null, null);


    item = data;



     leftChile = null;
     rightChild = null;
}
```



Add to a non-empty tree

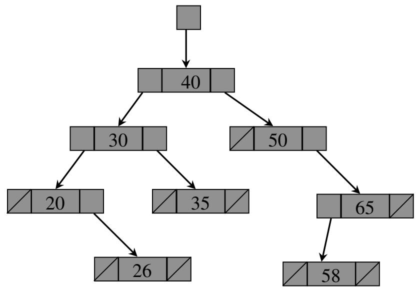
Add 2



Print contents of binary tree

```
// recursive function to print the contents of
// the tree as they appear.
// the tree is printed sideways i.e. root at
// extreme left
// the print function is called from main ()
// print function passes over the root and
// tspace=0
// tspace indicates how many positions have been
// used for the parent (if any)I.e. to judge
// where to print the values
```


Binary tree



Binary tree printed as :

```
                65
                          58
       50

40
                35

       30
                          26
                20
```


Binary search

```
// Binary search algorithm for a binary
  tree
// In a binary search tree values less
// than the root are in the left subtree,
// values greater than/equal to the root
// are in the right subtree

// Returns NULL if not found
// Else returns a pointer to the node
```



```
protected KeyedItem retrieveItem(TreeNode
  tNode, Comparable searchKey) {

    KeyedItem treeItem;
    if (tNode == null) {
      treeItem = null;
    }
```



```
 else {
 KeyedItem nodeItem =
  (KeyedItem)tNode.getItem();

if (searchKey.compareTo(nodeItem.getKey())==0)
  {
   // item is in the root of some subtree
        treeItem = nodeItem;
}
```

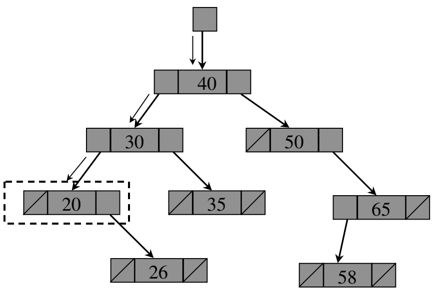
```
else if
  (searchKey.compareTo(nodeItem.getKey())<0)
  {

   // search the left subtree
    treeItem = retrieveItem(tNode.getLeft(),
  searchKey);
}
```



```
else  {
   // search the right subtree
 treeItem = retrieveItem(tNode.getRight(),
  searchKey);
     }  // end if
 }  // end if
    return treeItem;
}  // end retrieveItem
```

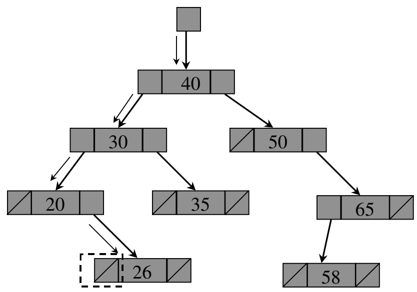

Binary search

Search for 20



Binary search

Search for 25



Returns NULL

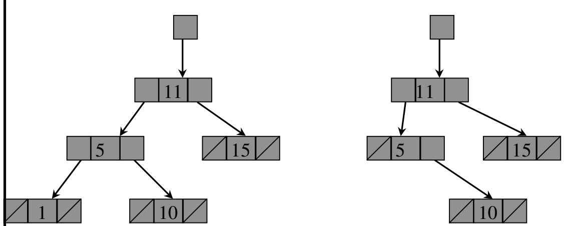
Deleting from a binary tree

- Leaf node
- Node with a left child only
- Node with a right child only
- Node with 2 children and no grandchildren
- Node with 2 children and grandchildren

NOTE : deletion is a non-trivial function - need to maintain order in the tree

Let p = pointer to node being deleted

Binary tree delete - leaf node

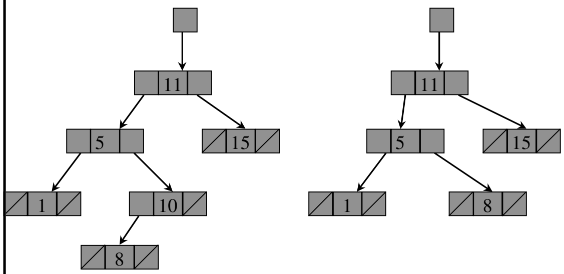


Delete 1

Before

After

Binary tree delete - left node only



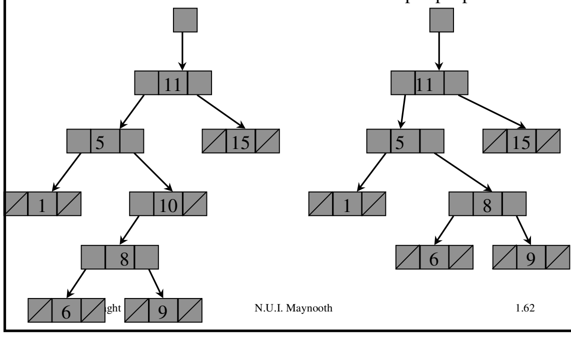
Delete 10

p = p->pleft
i.e. promote left subtree

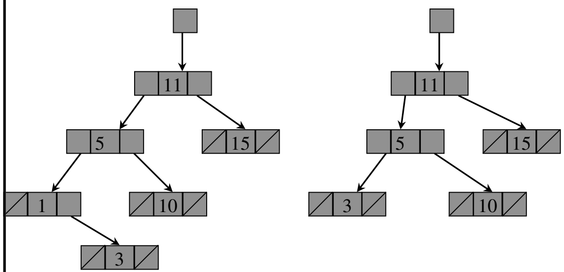
Binary tree delete - left node only

Delete 10

p = p->pleft



Binary tree delete - right node only



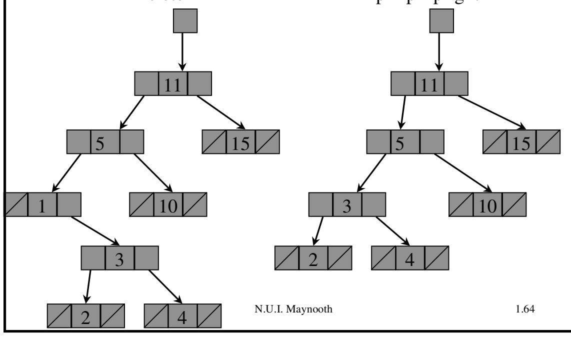
Delete 1

p = p->pright
i.e. promote right subtree

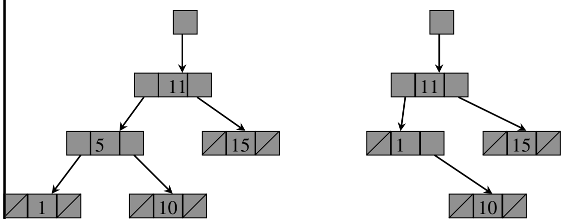
Binary tree delete - right node only

Delete 1

p = p->pright



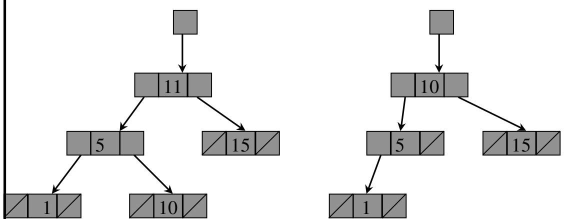
Binary tree delete - 2 children, no grandchildren



Delete 5

p = p->pleft
i.e. promote left child

Binary tree delete - 2 children and grandchildren



Delete 11

promote largest value in left subtree of node to be deleted

Binary tree delete - 2 children and grandchildren

Delete 11