ITT DUBLIN
Institute of Technology Tallaght
Institiúid Teicneolaíochta Tamhlacht

Binary Trees

And the Binary Search Tree

Binary Trees

- A Binary Tree is a special tree in which every node has exactly two children.
  - none, one, or both of the nodes may be NULL
- Since each node has at most 2 children, we can refer to them as *right child* and the *left child*, and to the sub-trees rooted at these children as the *right sub-tree* and the *left sub-tree* of the tree rooted at the parent.

Implementing a binary tree

- A binary tree can be implemented as a linked structure, where each node has 2 pointers, one to its right child and the other to its left child.

<u>Binary Tree Node implemented as a class</u>

```
class TreeNode
{
  public:
    TreeNode(DATATYPE theData);
  private:
     TreeNode *leftPtr;
     DATATYPE data;
     TreeNode *rightPtr;
   };
```

<u>The Binary Tree class itself</u> will contain

- a pointer to the root node (much like a list maintains a pointer to the head of the list)
- methods to insert and delete data from the tree

```
class BinaryTree
{
   public:
     BinaryTree();
     // methods to maintain the tree
   private:
     TreeNode *root;
};
```

ITT DUBLIN
Institute of Technology Tallaght
Institiúid Teicneolaíochta Tamhlacht

Recursion in the Binary Tree : coding example

See Savitch chapter 17

Variants on the pre-order traversal may be written to supply whatever functionality is required (e.g. to print each value on the tree, build up a coding map in a Huffman tree, or to find the depth of a tree)

```
class BTreeNode
{
  friend class BTree;
   public:
     TreeNode(DATATYPE theData);
     bool isLeaf();
    private:
      TreeNode *leftChild;
      DATATYPE data;
      TreeNode *rightChild;
};
```

```
class BTree
{
  public:
       BTree();
       void preOrderTraversal();

  protected:
   //this will be used internally by the traversal above
       void preOrderTraversal(BTreeNode *subTreeRoot)
       // specialised traversals may have other args,
       // depending what has to happen when a node is visited
       BTreeNode *root;
};
```

Some Binary Tree Methods

```
BTree::BTree() : root(NULL)
{}
```

```
void BTree:: preOrderTraversal()
{
        preOrderTraversal(root);
}
```

‘Visit the subtree root’ may mean, for example, `’print its value’, or ‘if it’s a leaf, then add something to the coding map’, or ‘add one to depth before going on to sub-trees’

```
void BTree:: preOrderTraversal(BTreeNode *subtreeRoot)
{
    if (subTreeRoot != NULL)
    {
            //’visit’ the subtree root, then …
            preOrderTraversal(root->leftChild);
            preOrderTraversal(root->rightChild);
    }
}
```

Example of recursion in the Binary Tree

- In a binary tree each node has maximum of 2 children (a right child and a left child)
  - Exercise: Use recursion to find the height of a binary tree

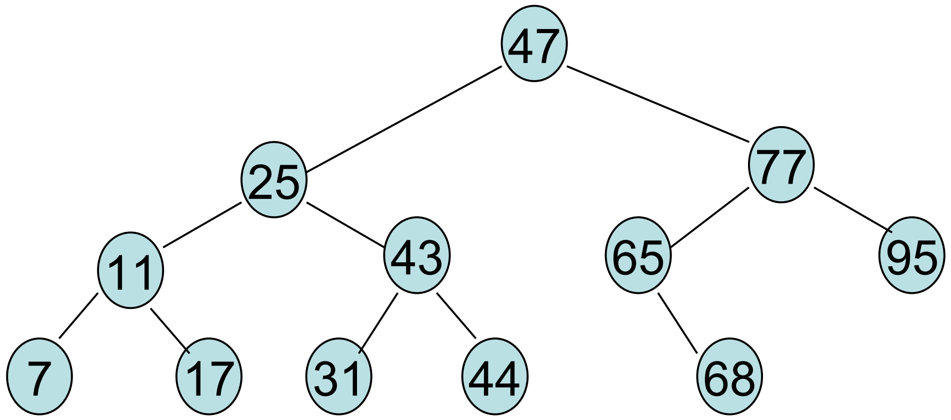
uses a pre-order traversal of the tree

If a node has neither a left or a right child, then height of the tree rooted at this node is 0 (this is the simple base case, which will always be reached)

Otherwise, the height of the tree rooted at this node is 1 + (the maximum of the heights of its right subtree and its left subtree)

A Binary Search Tree

- A binary search tree is a special binary tree which has the property that the values in any left subtree are less than the values in its root node, and the values in any right sub-tree are greater than the value in its root node.
- Binary search trees provide a fast way of searching for a value that matches a key value. We compare the value with the root. If it is smaller, we know we must search the left subtree of the root. If the value is larger than the root value, we know we must search the right subtree of the root.



A (well-balanced) binary search tree enables us to search for a data item in O(logn) time as in a binary search of a sorted array, *but also to insert and delete items efficiently (not so easy with a sorted array!)*

<u>Pseudo-code to search for val in a Binary Search Tree</u>

```
Current node = root
found = FALSE
while not found and current node not null
  if val is equal to current node val
     found = true
  else if val is less than current node val
     current node = left child
  else if val is greater than current node val
     current node = right child
End while
```

Recursive traversals of the BST

1. Exercise: Write the code for a binary search tree of chars, and use a recursive method to search the tree for a particular character and return true if it is present, false otherwise

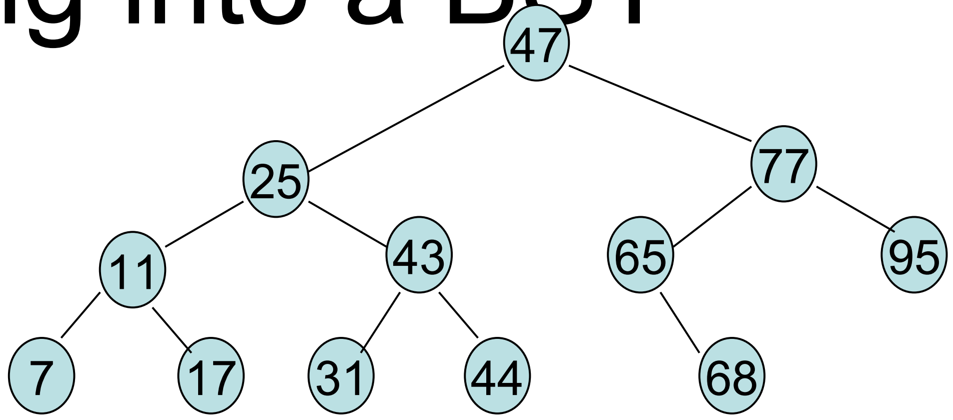
2. Exercise: Write the code to print all values in a binary search tree in ascending order
   A. Will this use pre-order traversal? Post-order traversal? Some other traversal??

   A. An in-order traversal, will visit all values stored in the left sub treè, then the root node, then all value stored in its right sub-tree

Inserting into a BST

Pseudo-code to insert a node val in a BST

```
Currentnode = root
While val not placed
   If val is less than current node val
      if left child is null
         left child = val (val is placed)
      else
         current node = left child
   Else if val greater than current node val
      if right child is null
         right child = val (val is placed)
      else
         current node = right child
  Else val is placed already (equal to current node)
End while val not placed
```

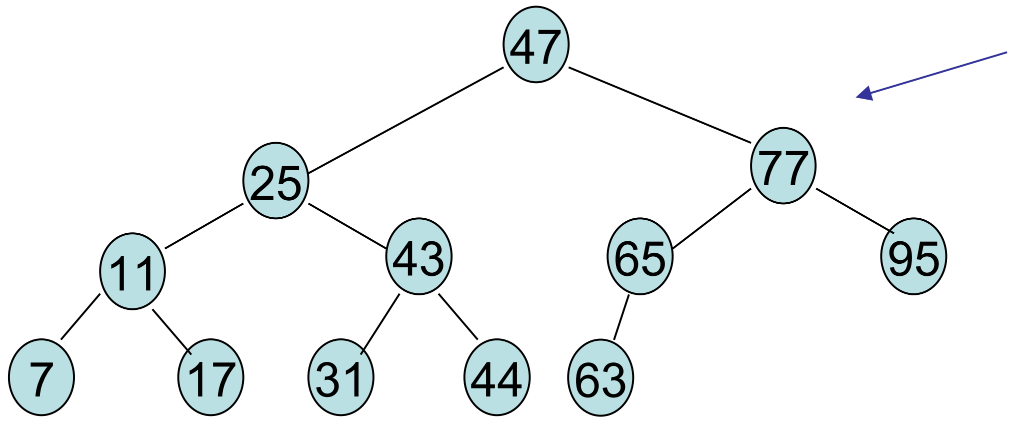


 Use the pseudo code to see where 19, 26, 78 would be inserted

… the shape of a binary search tree that corresponds to a particular set of data can vary, depending on the order in which the elements are inserted into the tree.

Suppose every value entered is smaller than the last one … ?

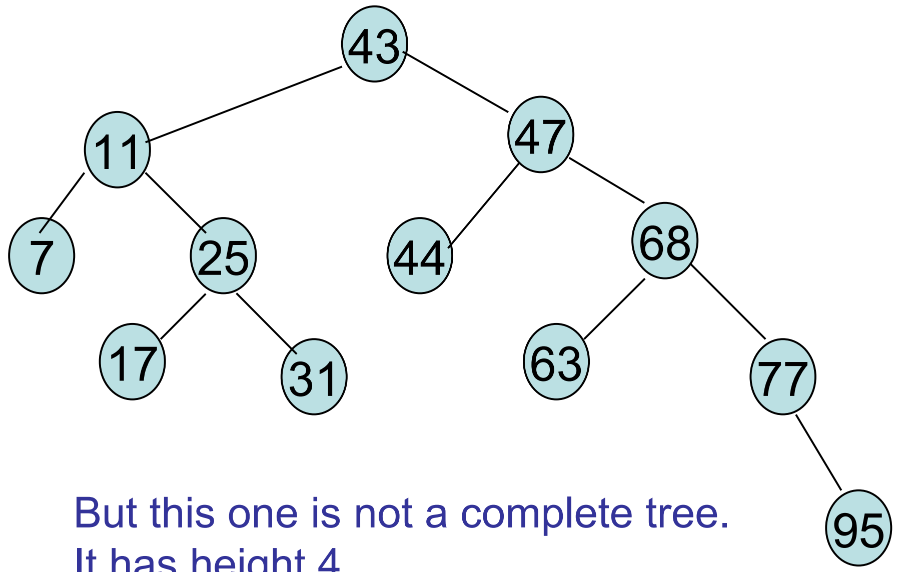
Efficient binary search trees



This is a complete binary tree, because all levels except the bottom are filled, and the bottom level has spaces only on the right. It has height 3

To be an efficient BST, it should be of the minimum possible height, and therefore be a *complete binary tree.*

The search algorithm will be O(log n) *– as it was for the binary search of an array*



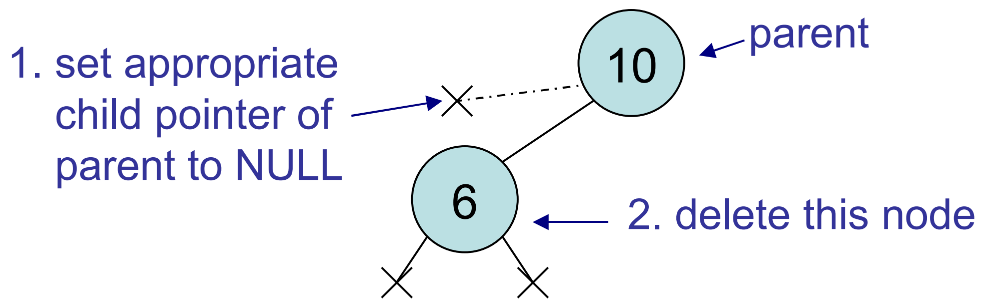
But this one is not a complete tree. It has height 4

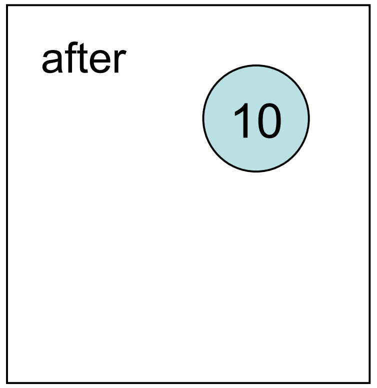
Deleting from a binary search tree

- This is a more complex problem than inserting. First we must search for the value to be deleted, and then remove it from the tree. But what happens to its right and left subtrees? There are three possible cases: (in all the diagrams below, the value to be deleted from the tree is 6)
- When we have identified which node to delete (probably by searching for it) we must make sure that we also maintain a pointer to its parent

Case 1: The node to be deleted was a leaf node

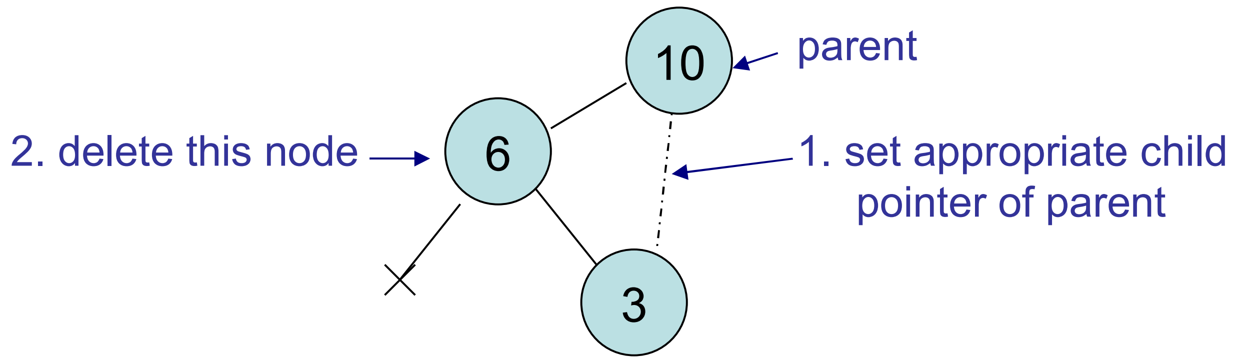
This is a simple case. The node is deleted, and then the pointer to it in the parent node is set to NULL.

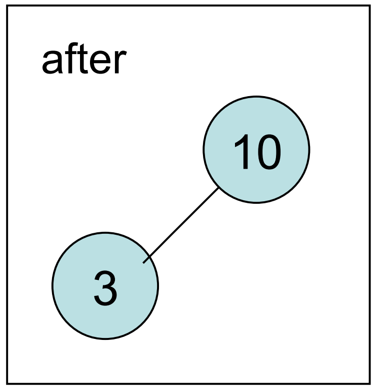




Case 2: The node has one subtree only.

The node is deleted, but first the pointer to it in the parent node is set to point to its (only) child

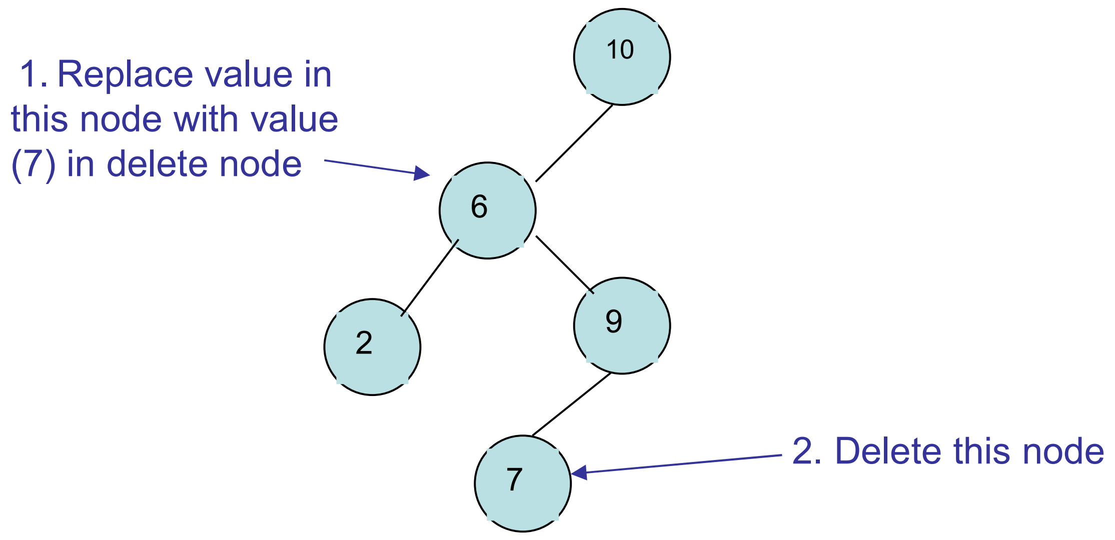


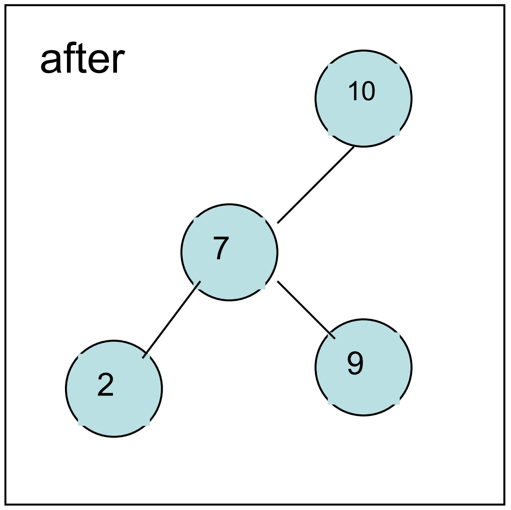


Deleting from a binary search tree - cont

Case 3: The node to be deleted has 2 sub-trees (i.e. both a right and a left child

- This is the tricky case. The solution is that we don't actually remove this node at all.
- We delete an easier (leaf) node, and replace the value to be deleted with the value in the easier node.
- To maintain the binary search tree property, choose the minimum of the right sub-tree – that is the left most node of the right sub-tree.
- *(note: the maximum of the left sub-tree would do just as well – that is the rightmost node of the left sub-tree)*


1. Replace value in this node with value (7) in delete node
10
6
2
9
7
2. Delete this node


after
10
7
2
9