

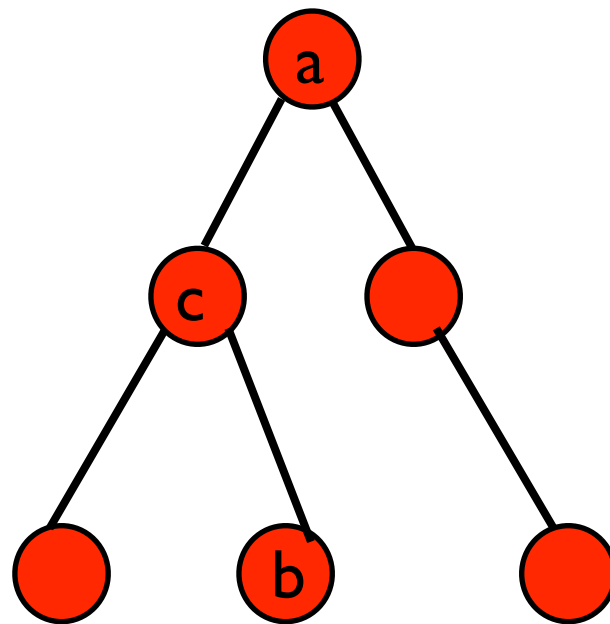
## Tree data types:

- Binary trees
- Binary search trees

Trees can be used to represent parent/child relationships between data.

Trees consist of nodes connected by edges or arcs. The connections are directional, and there are no “loops”.

The ancestor of all the nodes in a tree is the root, and has no parents. A node without children is called a leaf.



Trees are hierarchical so that if c is between a and b, then c is the child of a and the parent of b. Also b is a descendent of a, and a is an ancestor of b.

Nodes may have several children. Trees such that all nodes have at most two children are called binary trees, and we'll be studying them for a while.

A tree  $T$  is a binary tree if.

EITHER  $T$  is empty,

OR  $T$  is not empty and is a subset of nodes such that

- (a) Exactly one node is the root;
- (b) All the other nodes are partitioned into two disjoint subsets of descendants called the left subtree, and the right subtree; each subtree is a binary tree.

## Some definitions

Let  $H(T)$  be the height of a binary tree, which we define as follows.

$H(T) = 0$ , if  $T$  is empty;

$H(T) = 1 + \text{maximum}(H(T_1), H(T_2))$ , where  
 $T_1$  and  $T_2$  are the subtrees of  $T$ .

Roughly speaking, the height of the tree is the number of “levels” when a tree is drawn neatly, with all nodes of the same generation on the same level.

More definitions.

A binary tree is said to be full, if all nodes on level less than  $H(T)$  have two children each.

Roughly speaking a full binary tree has no “missing nodes”.

A binary tree is balanced, if the height of any node's right subtree differs from the height of its left subtree by no more than 1.

## Tree traversals.

Given a binary tree, we will be processing the items inside of it. To do it we will need to be able to “visit” each item. There are three ways to traverse a binary tree (ie visit each item), and we call them preorder, postorder or inorder traversals.

preorder traversal: each node is “processed” before the nodes in its subtrees;

postorder traversal: each node is “processed” after the nodes in its subtrees;

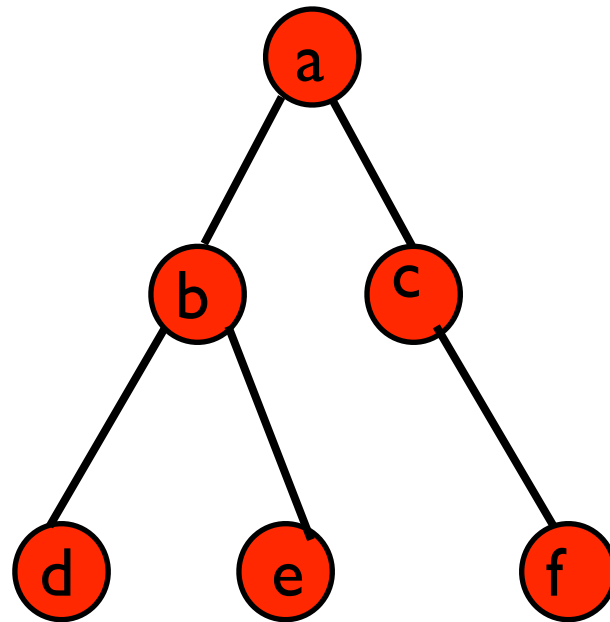
inorder traversal: each node is “processed” in between the nodes of its left and right subtrees.

Suppose that “processing” means “printing out”.

preorder output: a b d e c f

postorder output: d e b f c a

inorder output: d b e a c f



## Programming trees.

We may represent a tree in a C++ program as an array, or in a pointer-based representation. For the time being we'll use a pointer-based representation.

```
struct treeNode {  
    int item;  
    treeNode* LChildPtr;  
    treeNode* RChildPtr;  
};
```

For a pointer-based implementation in C++, we use the same idea as for linked lists, to create a struct containing the data (char, string, int etc.) together with (this time) two “links”, one for the left child and one for the right child.

Suppose that we already have a tree constructed. We can implement preorder, postorder and inorder traversals using a simple recursion.

```
void preorder (treeNode* T) {  
    if (T != NULL) {  
        cout << T-> item;  
        preorder(T -> LChildPtr);  
        preorder(T -> RChildPtr);}  
}
```

Current node's data  
printed first, before  
the left/right  
subtrees' nodes.

```
void postorder (treeNode* T) {  
    if (T != NULL) {  
        postorder(T -> LChildPtr);  
        postorder(T -> RChildPtr);  
        cout << T-> item;}  
}
```

Current node's data  
printed after the left/  
right subtrees'  
nodes.

```
void inorder (treeNode* T) {  
    if (T != NULL) {  
        inorder(T -> LChildPtr);  
        cout << T-> item;  
        inorder(T -> RChildPtr);}  
}
```

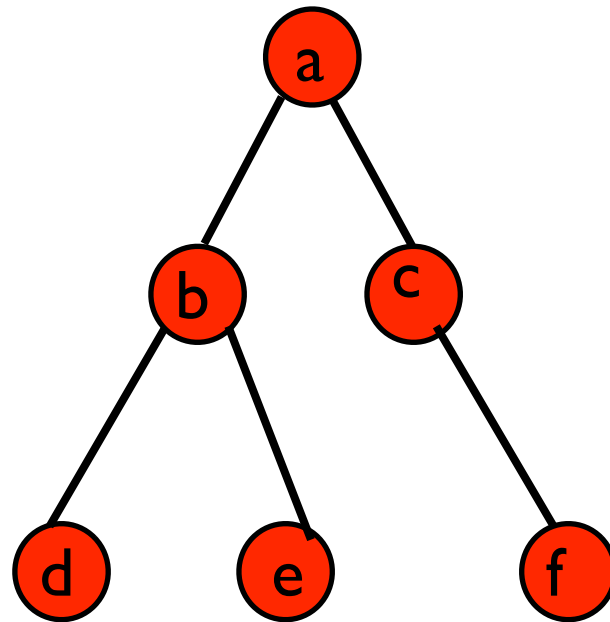
Current node's data  
printed after the left  
subtree's nodes, and  
before the right  
subtree's nodes.

Suppose that “processing” means “printing out”.

preorder output: a b d e c f

postorder output: d e b f c a

inorder output: d b e a c f



Binary search trees, are a special type of binary tree in which searching is easy, because the nodes are all ordered relative to each other. (Carrano, page 536--574)

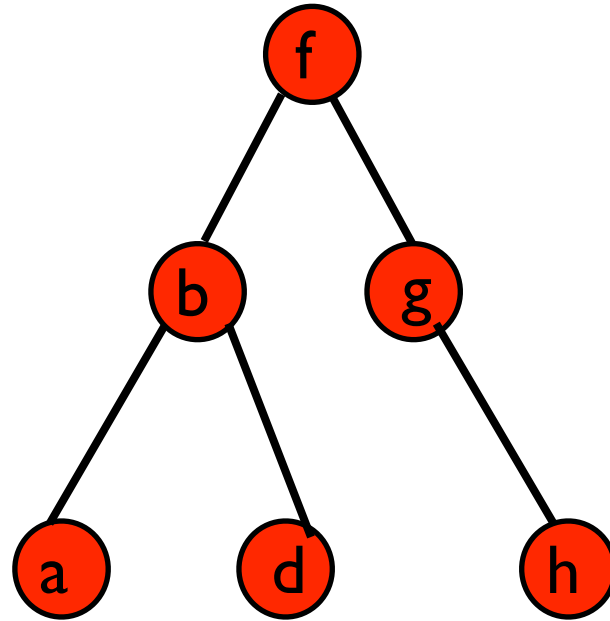
A binary search tree (BST) is defined as follows.

T is a binary search tree, if it is a binary tree, and the following “BST conditions” apply.

(a) T’s root item is greater than all the node items of its left subtree, and

(b) T’s root item is less than all the node items of its right subtree, and

(c) both of T’s left and right subtrees are binary search trees (so that (a) and (b) apply to their roots.)



Here's an example of a binary search tree.

Where is the smallest item found?

Where is the greatest item found?

What does an inorder traversal produce?

Next we'll consider how to search a binary search tree.

The basic idea is given a search key  $K$  and a tree  $T$ , we can recursively search the tree, just as in binary search, exploring either the left subtree or the right subtree according to whether  $K$  is less than depending on whether  $K$  is less than, or greater than the value at the current node.

```
void search (treeNode* TreePtr, int K, bool& success) {  
    // POST: sets success to true if K is in the tree, and false otherwise  
  
    if (TreePtr == NULL ) { success= false;}  
  
    else if (K == TreePtr-> item ) {success= true;}  
    else if (K < TreePtr-> item){search(TreePtr -> LChildPtr, K, success );}  
    else { search (TreePtr -> RChildPtr, K, success);}  
}
```

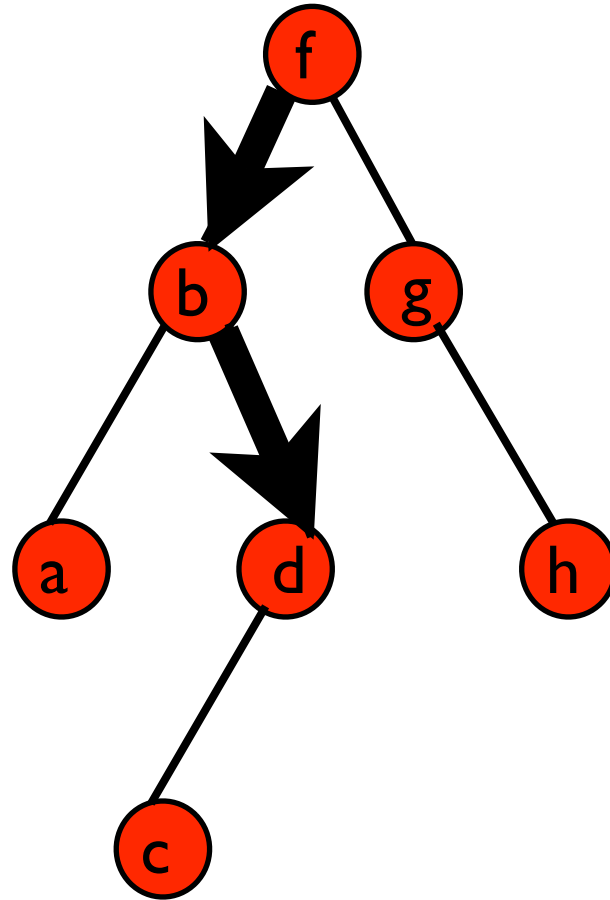
The complexity of this algorithm depends on the shape of the tree.  
(Why?)

Next we'll consider how to build a binary search tree.

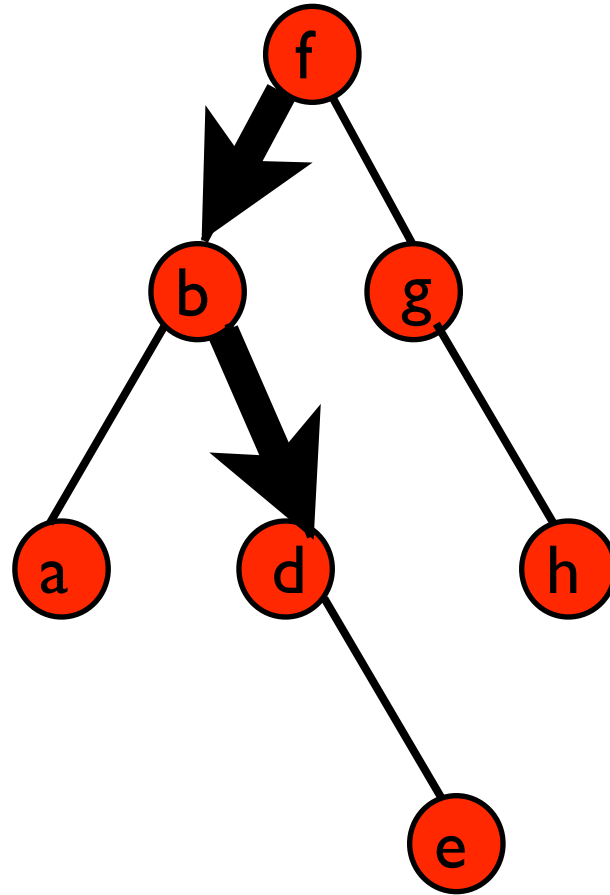
The basic idea is that we'll start from an empty tree and then just add items, making sure that the BST conditions are maintained.

To do that suppose we already have a binary search tree  $T$ , and we want to add an item,  $x$ . We have to decide where in the tree to put it. (We'll consider how to put it after we've figured out where to put it.)

An easy way to add a new item to a binary search tree is just to add it as a leaf node. To maintain the BST condition, we work our way down the tree --- effectively doing a binary search with the new item as the key --- ie taking the left branch whenever  $x$  is less than the current node, or the right branch when  $x$  is greater than the current node, until we reach a leaf. We then add it as a left/right child depending on whether  $x$  is less than or greater than the item at the leaf node.



Suppose we want to add “c” to this tree. We discover the c is less than f, so we move to f’s left child, and we discover that c is greater than b, so we move to b’s right child. At the node d we add c as a left child.



Suppose now we want to add “e” to this tree. We similarly move to node d (for the same reasons), but then add e as a right child.

```
struct treeNode {
    int item;
    treeNode* LChildPtr;
    treeNode* RChildPtr;
};
```

```
typedef treeNode* ptrType;
```

```
class BST
{
public:
    BST(); // Default constructor.

    void InsertNewItem ( int newItem , bool& success);
    ....
    void itInsert(ptrType &TreePtr, int newItem);
    ...
private :

    ptrType Root; // The top of the tree.
};
```

```

void BST::itInsert(ptrType &TreePtr, int NewItem) { // This assumes that the
    ptrType temp= TreePtr;                          // NewItem is not in the tree.

    if (TreePtr== NULL) {
        // Case 1: The tree may be empty
        //           Just create a node and add in the data
    }
    else { // Case 2: Otherwise we have to search the tree looking for the right
        // insertion position.
        bool success= false; // Use a boolean "flag" to warn when to stop....
        while(!success) {
            if ((temp -> item) < NewItem ) { // If we need to "go right"....
                if ((temp -> RChildPtr) == NULL ) { //... we may have to insert as a right node
                    // Create the node, and then stop...
                    success= true;
                }
                else temp= temp -> RChildPtr; // otherwise go right
            } ...

            else { // As for "go right", except "go left" }

```

```

void BST::itlInsert(ptrType &TreePtr, int NewItem) {
    ptrType temp= TreePtr;
    if (TreePtr== NULL) { // An empty tree, just
        TreePtr= new treeNode; // make a node
        TreePtr-> item = NewItem;
        TreePtr-> LChildPtr = NULL;
        TreePtr-> RChildPtr= NULL;
    }
    else {
        bool success= false;
        while(!success) {
            if ((temp -> item) < NewItem ) { // If we need to “go right”, and there is no right child,
                if ((temp -> RChildPtr) == NULL ) { //then insert here
                    temp->RChildPtr = new treeNode;
                    temp -> RChildPtr -> item = NewItem;
                    temp-> RChildPtr -> LChildPtr = NULL;
                    temp-> RChildPtr -> RChildPtr= NULL;
                    success= true;
                }
                else temp= temp -> RChildPtr; // otherwise go right
            } ...
        }
    }
}

```

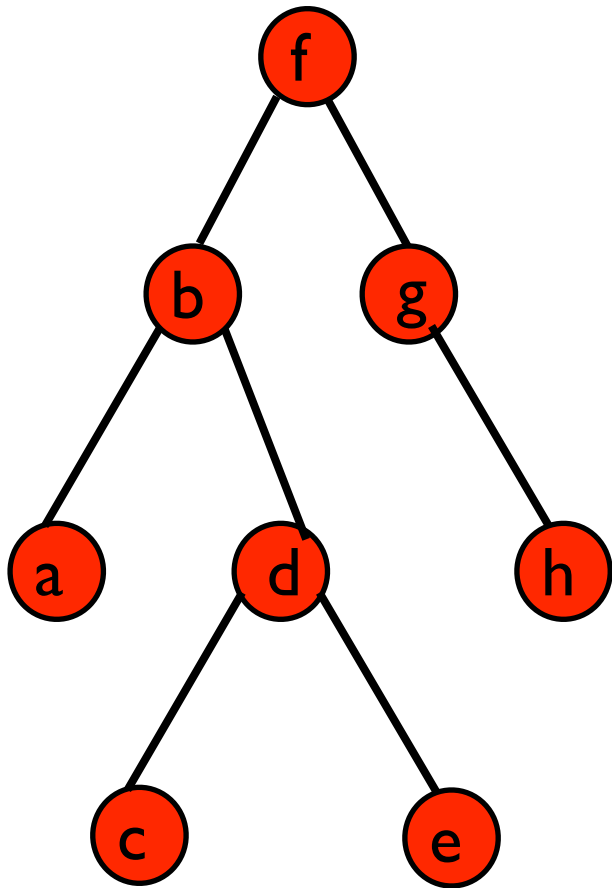
```

.....
else {
    if ((temp -> LChildPtr) == NULL ) { // if need to “go left” and there is no left
        temp->LChildPtr = new treeNode; // child then insert here.
        temp -> LChildPtr -> item = NewItem;
        temp-> LChildPtr -> LChildPtr = NULL;
        temp-> LChildPtr -> RChildPtr= NULL;
        success= true;
    }
    else temp= temp -> LChildPtr; // otherwise go left
}
}
}
}

```

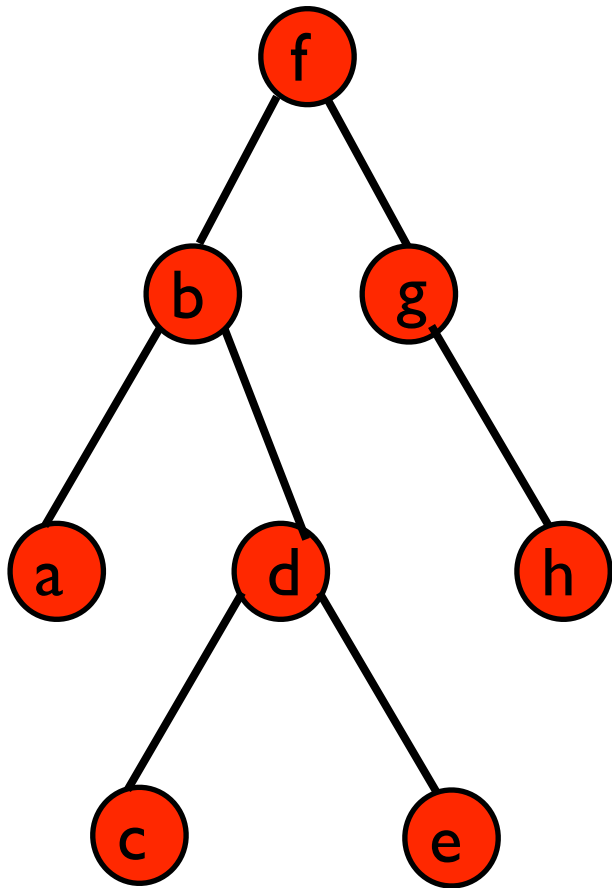
Note: we would have to do something special to deal in the case that the node is already in the tree.

To delete a node from a binary search tree is harder.  
Here's why.

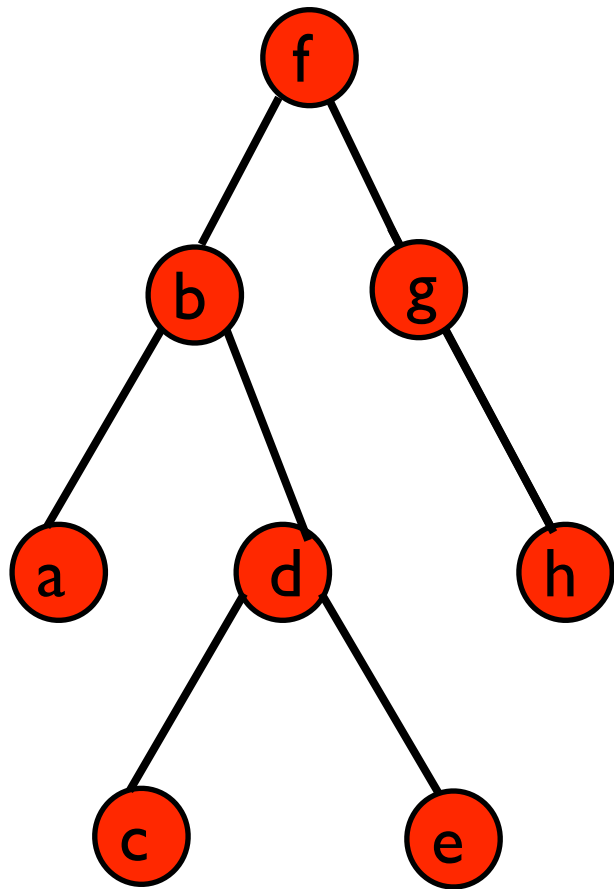


Suppose we want to delete node “b” from this tree. Obviously it will leave a gap, which must be filled by some other element already in the tree, but which one? “d” would be a possibility, but then if we moved that node wholesale, we’d have to move “a” somewhere else because, it can’t become a “third” child of “b”. We need to think again.

There are a number of cases we need to consider when we want to remove a node in general. We'll deal with each in turn, and come back to the "hard" case we've just seen.



Case 1: Suppose we want to remove a leaf node, say "h". All we do is remove it --- there's no more tidying up to do since the BST conditions in the remaining tree are not disturbed.

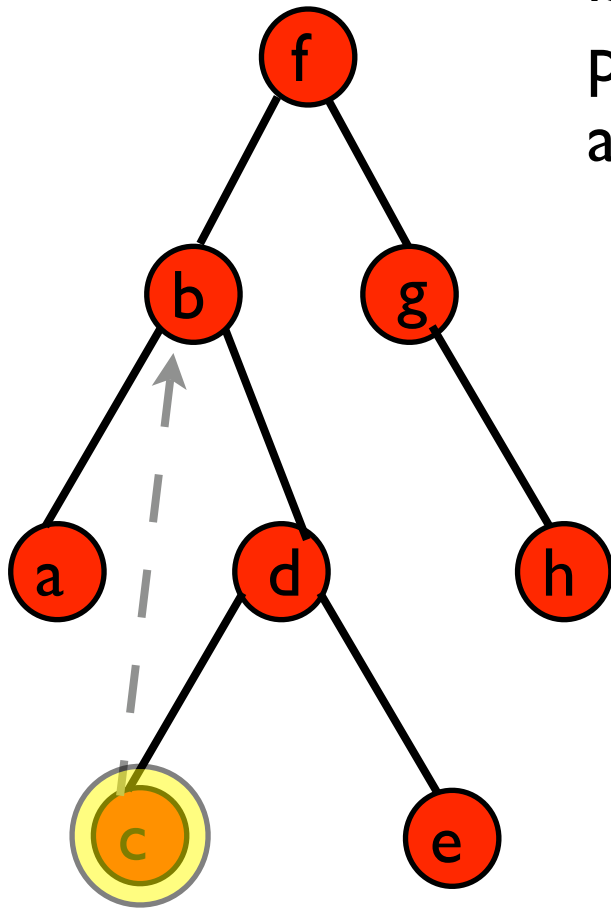


Case 2: Suppose we want to remove a node which does not have a left child, say “g”. Now in this case it is fine to move the remaining subtree rooted at its right child to “g”’s old position, because the BST conditions are not disturbed and no other node needs to be moved elsewhere.

The same idea works if there is no right child.

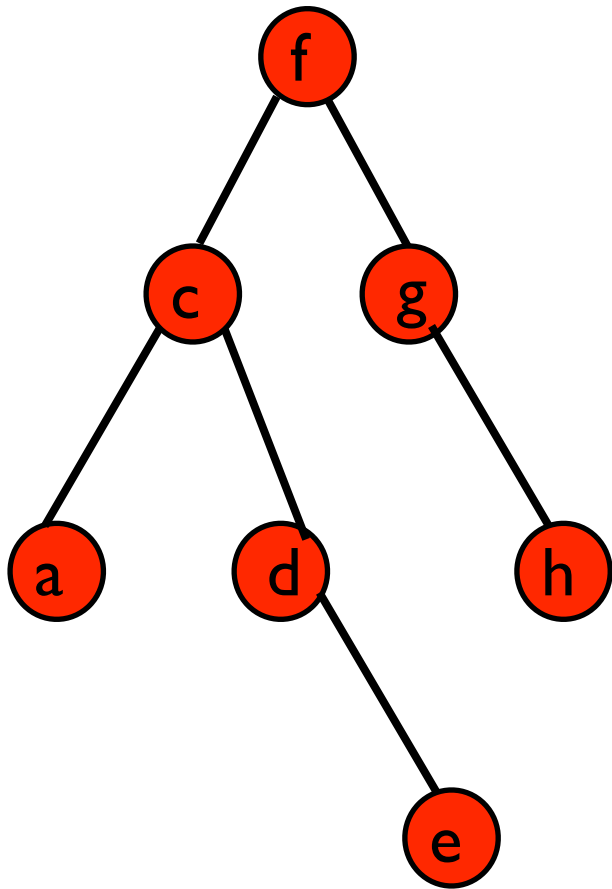
Case 3: That leaves the hard case, when a node has both a right and left child.

Suppose now we want to remove “b”. The idea is to take one of the leaves and put it in b’s place --- that way we won’t have to move anything else, only the leaf. But which one?



We need a value which is greater than all the values in b’s left subtree (in this case “a”), but less than all the values in its right subtree. (Why?) Such a value is found by taking the smallest value in (the former) “b”’s right subtree. Recall that the smallest item in a BST is at the left-most node, in this case “c”.

The tree with “b” removed now looks like this, and we can verify that the BST conditions still apply.



Overall the deletion algorithm works by doing a binary search to locate the item to be removed, working out which of the three cases it belongs to, and then either deleting the node (in the case it is a leaf); OR replacing the node with its single subtree (in the case it only has one subtree); Or locating the smallest item in its right subtree as replacement item.

In all three cases the resulting tree is still a binary search tree.

```

void Deleteltem(ptrType& TreePtr, int Key, bool &success) {
    if (TreePtr == NULL) success= false;

    else if (Key == TreePtr-> item) {
        DeleteNodeltem(TreePtr);
        success= true; // successful delete...
    }
    else if (Key < TreePtr -> item)
        Deleteltem(TreePtr ->LChildPtr, Key, success);
        // search the left....
    else Deleteltem(TreePtr ->RChildPtr, Key, success);
    //... or search the right.
}

```

This is a recursive program which searches for the node to delete (value Key). When it finds it it calls `DeleteNodeltem` to remove the item found. If the delete is successful it sets “success” to true.

```

void DeleteNodeItem(ptrType& Ptr) {
    ptrType DelPtr;
    int ReplacementItem;

    // test for leaf.. // Case: empty tree
    if ( (Ptr -> LChildPtr == NULL) && (Ptr -> RChildPtr == NULL)) {
        delete Ptr; // .. just delete
        Ptr= NULL;
    }

    // Case: no left child..
    else if (Ptr -> LChildPtr == NULL) { // .. move the right child up and
        DelPtr= Ptr;
        Ptr= Ptr -> RChildPtr;
        DelPtr->RChildPtr= NULL;        // ... delete the node.
        delete DelPtr;
    }

    // Case: no right child..
    else if (Ptr -> RChildPtr == NULL) { // .. move the left child up and
        DelPtr= Ptr;
        Ptr= Ptr -> LChildPtr;
        DelPtr->LChildPtr= NULL;        // ... delete the node.
        delete DelPtr;
    }
}

```

```

void DeleteNodeItem(ptrType& Ptr) {
    ptrType DelPtr;
    int ReplacementItem;
    .....

    //... if none of the above, the node must have two children..
    else { //.. take the first right, and then go left all the way..
        ProcessLeftmost(Ptr-> RChildPtr, ReplacementItem); //.. get the replacement item
        Ptr->item= ReplacementItem;
    }
}

```

This is the hard case: we have to find the smallest node in the right subtree, get its value (ie assign it to variable “ ReplacementItem”), delete it, and then replace the current node’s data with that value. To do the find and delete we call a function called **ProcessLeftMost**.

```

void ProcessLeftmost (ptrType& Ptr, int& rep) {
    if (Ptr-> LChildPtr == NULL)
    {
        rep= Ptr-> item; // .. this is the one..
        ptrType DelPtr= Ptr;
        Ptr= Ptr->RChildPtr; // .. move the right subtree up
        delete DelPtr; // Get rid of old node.
    }
    else ProcessLeftmost(Ptr-> LChildPtr, rep); // ... otherwise, keep going left.
}

```

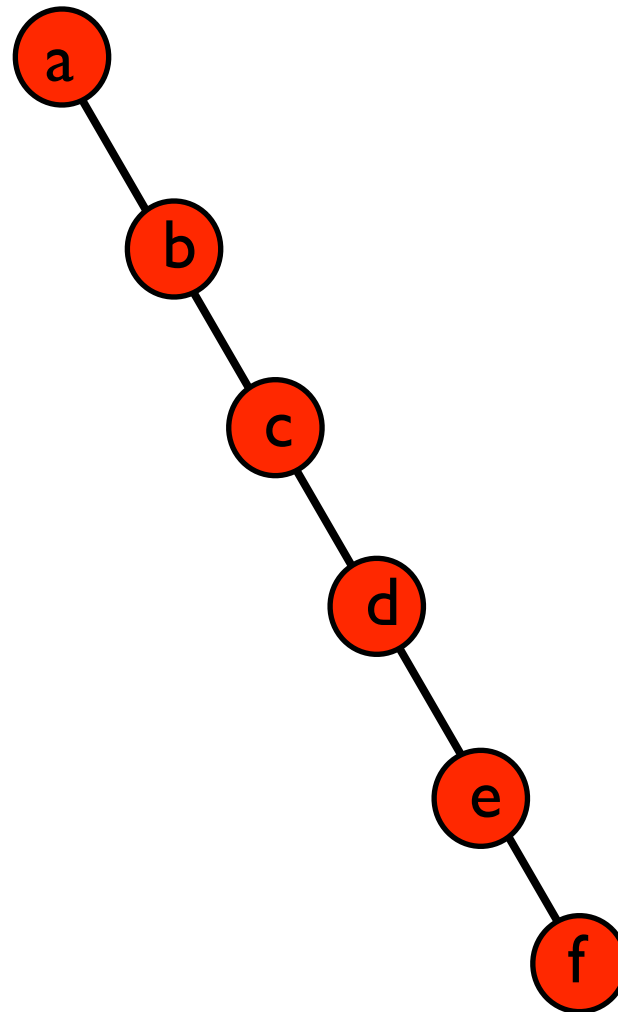
This is a recursive program, which searches the tree by taking the left branch as far as it can. When it can't go any further (the base case), it puts the value found in variable rep, and then deletes the node.

This finishes off the deletion of a node in a binary search tree!

The order of insertion of nodes in a binary search tree affects the final shape of the tree.

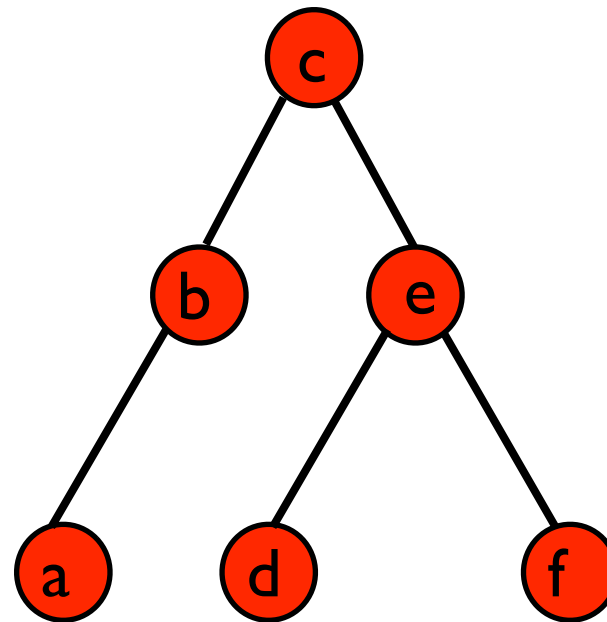
If we insert the nodes in this order a, b, c, d, e, f, then the shape of the tree will be linear.

The efficiency of searches in linear trees containing  $N$  nodes is  $O(N)$ .



The order of insertion of nodes in a binary search tree affects the final shape of the tree.

If we insert the nodes in this order c,b,e,a,d,f then the shape of the tree will be balanced.



The efficiency of searches in balanced trees containing  $N$  nodes is  $O(\log N)$ .

There are two other common ways to search a general binary tree: a breadth-first search and depth-first search.

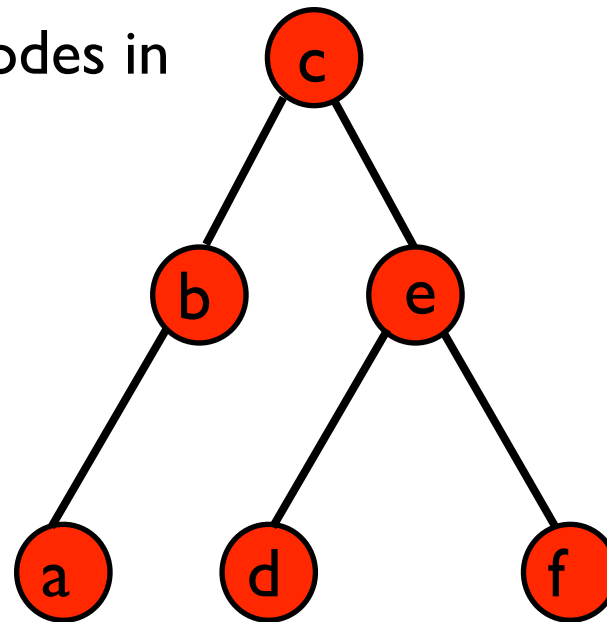
A depth-first search searches the tree by systematically looking down each path, as far as it can go;

A breadth-first search searches (effectively) by searching systematically “across” each level or generation, starting from the top.

The implementations of these algorithms are very similar however!

A depth-first search visits the nodes in order c b a e d f

A breadth-first search visits the nodes in order c b e a d f



To implement a depth first search we need a way of storing the subtrees yet to be explored, so that once we have gone all the way down a path, we may “backtrack” to the place where we branched off.

Backtracking is a common technique in searching and is implemented with a stack.

```
void BST::depthFirst(ptrType& t) {
    stack<ptrType> st; // Here's the stack for storing subtrees
    st.push(t);
    ptrType temp;
    while(!st.empty()) {
        temp= st.top(); // Look at the top item
        if (temp!= NULL) {
            cout << temp-> item << " ";
            st.pop();
            st.push(temp->RChildPtr); // push the right tree first
            st.push(temp->LChildPtr);
        }
        else st.pop(); // remove null subtrees
    }
    cout << endl;
}
```

To implement a breadth first search we need a way of storing the subtrees yet to be explored, so that once we have gone all the way through a generated, we may continue to the next level.

To make sure we search earlier generations before later ones, we need a data structure to store the subtrees so that the later we add subtrees, the later they will be processed. That's a queue data structure.

```
void BST::breadthFirst(ptrType& t) {
    queue<ptrType> st; // Here's the queue for storing subtrees
    st.push(t);
    ptrType temp;
    while(!st.empty()) {
        temp= st.front(); // Look at the front item
        if (temp!= NULL) {
            cout << temp-> item << " ";
            st.pop();
            st.push(temp->LChildPtr); // Add the left child first
            st.push(temp->RChildPtr);
        }
        else st.pop(); // remove null subtrees
    }
    cout << endl;
}
```

## Tree sort

We can use binary search trees for sorting. Here's the strategy:

- Read data from an array, inserting the items into a binary search tree;
- Traverse the constructed tree using an inorder traversal, and put each item back into the array.

### Analysis:

Building a binary search tree has worst case complexity  $O(n^2)$ , where  $n$  is the number of nodes (why?)

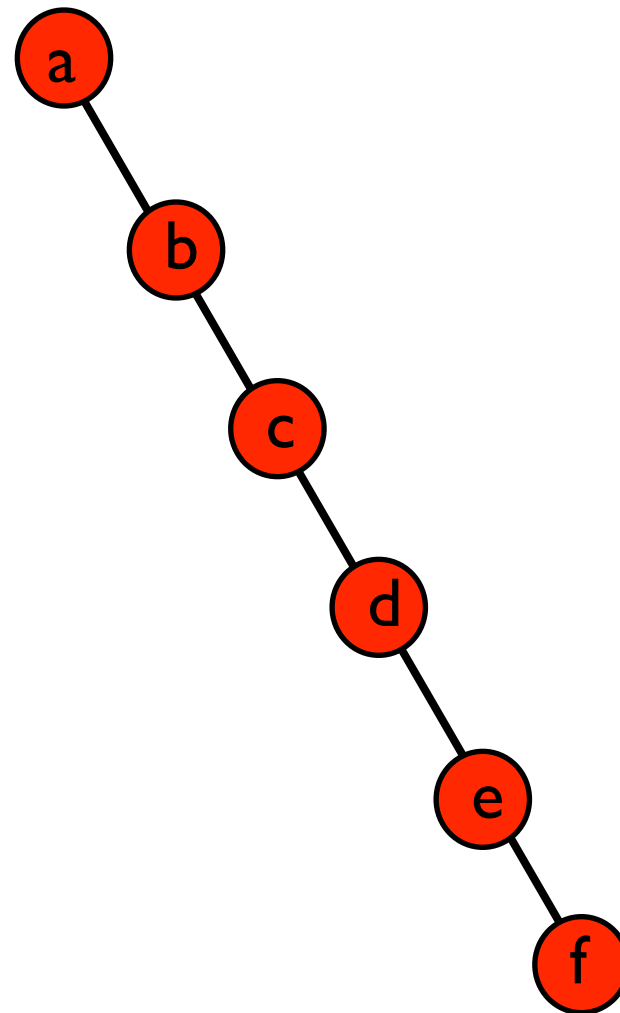
Traversing the binary search tree has worst case complexity  $O(n)$ .

How to make a balanced tree.

Suppose we have a sequence of nodes: a b c d e f. If we add them to the tree in this order then we will get this.

We would however rather have a balanced tree --- to get that we should add the nodes in order

c b a e d f



```

void restore( BST &T, int A[], int first, int last){
// PRE:A is sorted
  if (first <= last) {
    bool s;
    int mid= (first+last)/2;
    T.InsertNewItem(A[mid], s);
    restore(T,A , first, mid- 1);
    restore(T,A, mid+ 1, last);
  }
}

```

To construct balanced tree from a sorted array, we select the middle item from the array, insert that, and then recursively insert the left half, and then the right.

We can now turn any binary search tree into a balanced tree:

- Read the nodes of the tree into an array using an inorder traversal (why?)
- Use the restore function to create a balanced binary search tree from the array of items.