Chapter 10-3, Lecture notes

The ADT Binary Search Tree

1. Searching through a binary search tree means the tree must be set up so that at a particular node of n:
   - n's value is greater than all values in its left subtree T_L,
   - n's value is less than all values in its right subtree T_R.
   - Both T_L and T_R are binary search trees

   1.1 The search goes by the value of a parameter and NOT the position and can be very efficient.

   1.1.1 The search is also more useful if each node is either a structure or a class that can contain multiple values like name, ID, address, phone, etc…

   1.1.2 That would be considered a "record" with "fields" (i.e. the name, ID, etc…).

   1.1.3 So that the request of:

   Find the record for the person whose ID number is 123456789

   1.1.4 Can be done.

   1.1.5 The field that is searched is called the "search key" or just "key" and must be unique (name is not acceptable because of duplicate names like the proverbial "John Smith" and "Jane Brown").

   1.1.6 However, you could make a request to find ALL records for "John Smith" and "Jane Brown".

   1.1.7 For instance:

   Find the records for all people named Jane Brown

   1.1.8 Or limit it by doing:

   Find the records for the Jane Brown whose phone number is 401-555-1212

   1.2 The following is a class KeyedItem:

```
class KeyedItem
{
public:
        KeyedItem() {}
        KeyedItem(const KeyType& keyValue) : searchKey(keyValue) {}
        KeyType getKey() const // returns search key
        {
                return searchKey;
        } / end getKey
private:
        KeyType searchKey;
}; // end KeyedItem
```

1.3 So using a search key, you modify the binary search tree definition such that:

- o N's search key is greater than all values in its left subtree T_L,
- o N's search key is less than all values in its right subtree T_R.
- o Both T_L and T_R are binary search trees

2. An operational contract for the ADT Binary Search Tree would be:

2.1 TreeItemType is the type of the items stored in the binary search tree. It should be based upon KeyedItem, which has a search key field of type KeyType.

searchTreeInsert(in newItem:TreeItemType) throw TreeException

2.2 Inserts newItem into this binary search tree whose items have distinct search keys that differ from newItem's search key. Throws TreeException if the insert is not successful.

searchTreeDelete(in searchKey:KeyType) throw TreeException

2.3 Deletes from this binary search tree the item whose search key equals searchKey. If no such item exists, the operations fails and throws TreeException.

searchTreeRetrieve(in searchKey:KeyType, out treeItem:TreeItemType) {query}
throw TreeException

2.4 Retrieves into treeItem the item in this binary search tree whose search key equals searchKey. If no such item exists, the operation fails and throws TreeException.

preoderTraverse(in visit:FunctdionType)

2.5 Traverses this binary search tree in preorder and calls the function void one for each item.

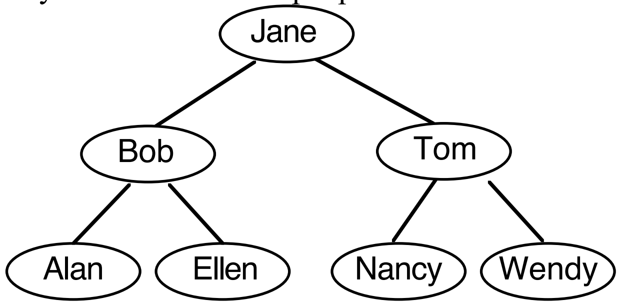
inoderTraverse(in visit:FunctdionType)

2.6 Traverses this binary search tree in inorder and calls the function void one for each item.

postoderTraverse(in visit:FunctdionType)

2.7 Traverses this binary search tree in postorder and calls the function void one for each item.

3. So as in figure 10-19, you had the familiar binary search tree, called nameTree, that shows the search key for the records of people such that:



nameTree.searchTreeRetrieve("Nancy", nameRecord)

3.1 Would retrieve the record of Nancy. In you insert a record into nameTree that describes Hal by doing:

nameTree.searchTreeInsert(HalRecord)

3.2 You could get the record back later. If you delete Jane's record by doing:

nameTree.searchTreeDelete("Jane")

3.3 You can still get the records for Nancy and Hal. And if you want to display the records in alphabetic order, you can do:

nameTree.inorderTraverse(displayName)

3.4 Algorithms for the ADT Binary Search Tree algorithms would be based on the following classes:

```
#include <string>

using namespace std;

typedef string KeyType;

class KeyedItem
{
public:
        KeyedItem() {}
        KeyedItem(const KeyType& keyValue) : searchKey(keyValue) {}
        KeyType getKey() const // returns search key
        {
                return searchKey;
        } / end getKey
private:
        KeyType searchKey;
}; // end KeyedItem

class Person : public KeyedItem
{
public:
        Person() : KeyedItem() {}
        Person(const string& name, const string& id, const string& phone) :
                        KeyedItem(name), idNum(id), phoneNumber(phone) {}
private:
        // Key item is the person's name
        string idNum;
        string phoneNumber;
        // .. and other data about the person
}; // end Person
```

4. The recursive search would be:

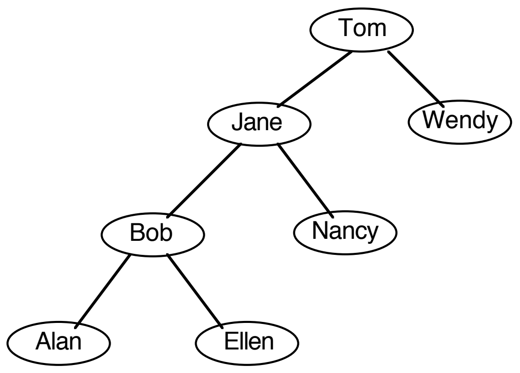
```
Search(in binTree:BinarySearchTree, in searchKey:KeyType)
// Searches the binary search tree binTree for the item whose search key is searchKey

        if (binTree is empty)
                The desired record is not found

        else if (searchKey == search key of root's item)
                The desired record is found

        else if (searchKay < search key of root's item)
                search(Left subtree of binTree, searchKey)

        else
                search(Right subtree of binTree, searchKey)
```

4.1 When the algorithm hits an empty tree, it means that the node is not to be found.

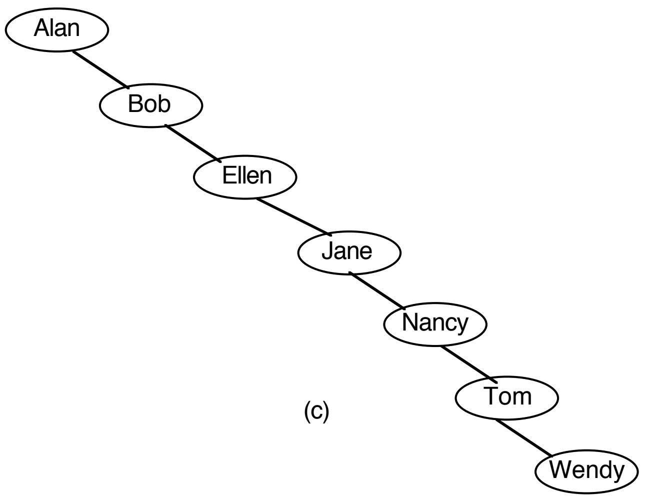
4.2 We will assume that we are using the pointer-based implementation for binary trees though this would apply to the array based implementations also.

4.3 However, not all Binary Search Trees are equal.

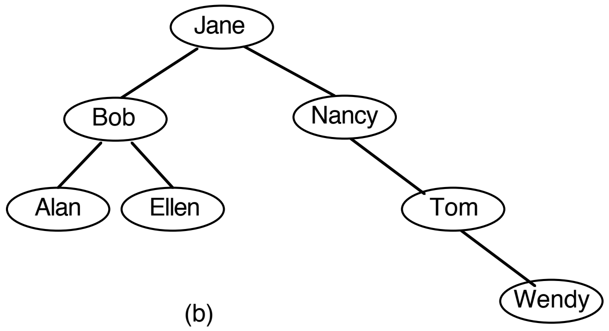
4.4 If the tree is not a complete binary tree, you can get some interesting shapes like figure 10-20 would show:



(a)



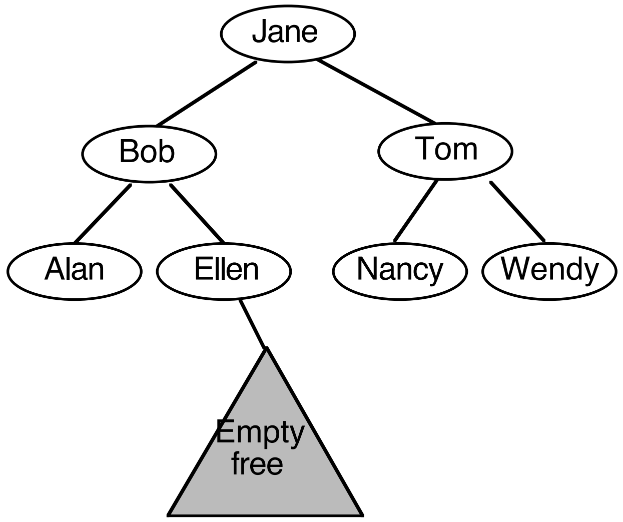
(c)



(b)

4.5 To insert, we can use the search routine to point to the proper place in the binary tree to place the new node.

4.5.1 For instance, if we wanted to insert Frank, the search of Frank would have the search terminate at the place where Frank should go, like so in Figure 10-22:



4.5.2 This makes for an easy insert routine.

4.5.3 However, you need to pass the treePtr as a reference to allow changing the pointers like we have done before and so the insert algorithm would look like:

```
insertItem(inout treePtr:TreeNodePtr, in NewItem:TreeItemType)
// inserts newItem into the binary search tree to which treePtr points.

        if (treePtr is NULL)
        {       Create a new node and let treePtr point to it
                Copy newItem into new node's data portion
                Set the pointers in the new node to NULL
        }

        else if (newItem.getKey() < treePtr->item.getKey())
                insertItem(treePtr->leftChildPtr, newItem)

        else
                insertItem(treePtr->rightChildPtr, newItem)
```

4.6 This is similar to the recursive-linked list where the "treePtr" keeps "moving down" and when you change it you are actually changing a left or right pointer or even the root.

5. Deletion is a bit more complicated if you want to maintain the tree as a binary search tree.
   5.1 A first draft of the deletion routine would be:

```
deleteItem(inout treePtr:TreeNodePtr, in searchKey:KeyType) throw TreeException
// Deletes from the binary search tree to which treePtr points the item whose search key
// equals searchKey. If no such item exists, the operation fails and throws TreeException.

        Locate (by using the search algorithm) the item i whose search key
                equals searchKey

        if (item I is found)
                remove item i from the tree
        else
                throw a tree exception
```

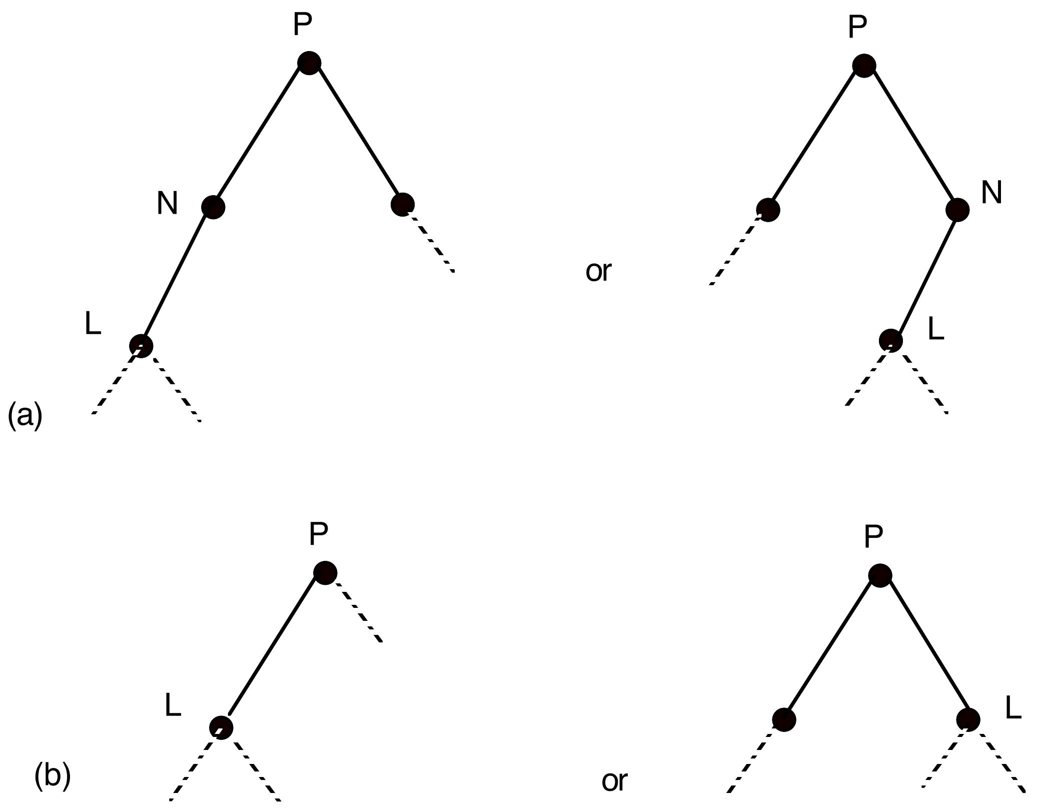
   5.2 The important task is the "remove item i from the tree".
   5.3 Depending on if the deleteItem finds the item i, there are three cases to ponder for a node N:

      5.3.1 N is a leaf
      5.3.2 N has only one child
      5.3.3 N has two children

5.4 The first case is the easiest: all you need to do is to set the pointer to it to Null and delete the node.

5.5 The second case you need to consider if:
- N has only a left child
- N has only a right child

5.6 The two possibilities are symmetrical so we only need examine the left child one.
   5.6.1 Figure 10-24 has an example of deleting N and two possible binary search trees:



   5.6.2 L is the left child of N and P is the parent of N.
   5.6.3 N can be either the right or left child of P.
   5.6.4 If you delete N then L would be without a parent and P would be without one of its children.
   5.6.5 Suppose we let L become “adopted” by P?
   5.6.6 L is a binary search tree and moving it up would still maintain a binary search tree – especially with N having only one child.
   5.6.7 So adoption can be done.

5.7 But the most difficult case is the third case where the item to be deleted has two children.

5.7.1 Which child do you use to move up?

5.7.2 Actually, you do not delete N at all – you find an easier node to delete and delete it instead of N.

5.7.3 It may seem like cheating to delete the node – after all the user expects the specified node to be gone.

5.7.4 However, since there is a wall, one could argue that the user would and should not know which specific node is deleted – only that the value specified is gone from the tree.

5.8 So, let's look at an alternate strategy for deleting node N that has two children:

5.8.1 Locate another node M that is easier to remove from the tree than the node N

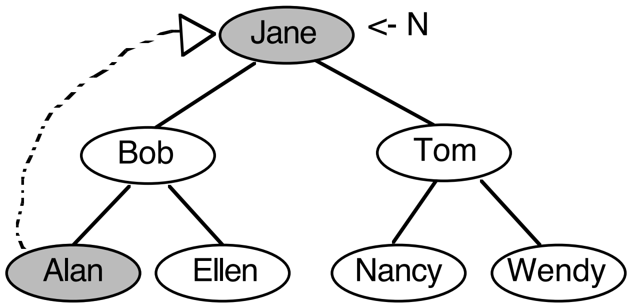
5.8.2 Copy the item that is in M to N, thus effectively deleting from the tree the item originally in N

5.8.3 Remove the node M from the tree

5.8.4 Well, a node that is easier to delete would be a leaf node with no children or a node with only one child.

5.8.4.1 However, you need to be careful in selecting a node since you must preserve the tree as a binary search tree.

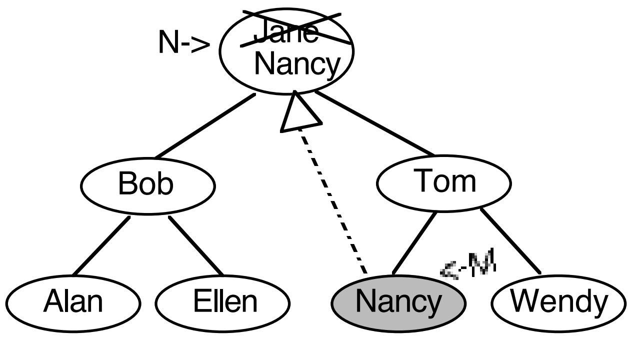
5.8.4.2 For example, in figure 10-26, selecting the wrong node to switch with would mean the tree is no longer a binary search tree:



5.8.5 We want to find a replacement that is close to the node being deleted.

5.8.5.1 We know that all values to the right of N would be greater than the value in N and that all values to the left of N would be less than the value in N.

5.8.5.2 If we choose the right subtree then choose the leftmost node, we will have a successor M (it would be a "inorder successor), like figure 10-28 shows:



5.8.6 Note, if we go right from N then try to go left, we stop at the first node with no left child.

5.8.6.1 Therefore if Nancy was deleted, then Tom would replace it.

5.8.6.2 A more detailed deletion algorithm would be:

```
deleteItem(inout treePtr:TreeNodePtr, in searchKey:KeyType) throw TreeException
// Deletes from the binary search tree to which treePtr points the item whose search key
// equals searchKey. If no such item exists, the operation fails and throws TreeException.

    Locate (by using the search algorithm) the item whose search key equals
        searchKey; it occurs in node N

    if (item is found in node N)
        deleteNodeItem(N) // defined next
    else
        Throw a tree exception

deleteNodeItem(inout N:TreeNode)
// Deletes the item in a node N of a binary search tree

    if (N is a leaf)
        Remove N from the tree

    else if (N has only one child C)
    {   if (N was a left child of its parent P)
            Make C the left child of P
        else
            Make C the right child of P
    }
    else // node has two children
    {   Find M, the node that contains N's inorder successor
        Copy the item from node M into node N
        Remove M from the tree by using the previous technique
            for a leaf or a node with one child
    } // end if
```

5.9 Now we need to get a new more details.

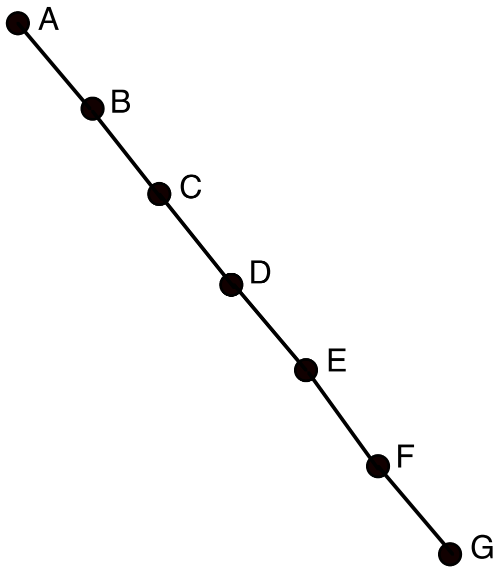
5.9.1 The deleteNodeItem uses the method processLeftmore to find the node M (that has the inorder successor of node N).

```
deleteItem(inout treePtr:TreeNodePtr, in searchKey:KeyType) throw TreeException
// Deletes from the binary search tree to which treePtr points the item whose search key
// equals searchKey. If no such item exists, the operation fails and throws TreeException.

    if (treePtr == NULL)
        Throw a tree exception indicating item not found

    else if (searchKey == treePtr->item.getKey())
        // item is in the root of some subtree
        deleteNodeItem(treePtr)         // delete the item

    else if (searchKey < treePtr->item.getKey())
        // search the left subtree
        deleteItem(treePtr->leftChildPtr, searchKey)

    else // search the right subtree
        deleteItem(treePtr->rightChildPtr, searchKey)

deleteNodeItem(inout nodePtr:TreeNodePtr)
// Deletes the item in node N to which nodePtr points

    if (N is a leaf)
    {   // remove leaf from the tree
        delete nodePtr
        nodePtr = NULL
    }

    else if (N has only one child C)
    {   // C replaces N as the child of N's parent
        delPtr = nodePtr
        if (C is the left child of N)
            nodePtr = nodePtr->leftChildPtr
        else
            nodePtr = nodePtr->rightChildPtr
        delete delPtr
    }
    else // N has two children
    {   // find the inorder successor of the search key in N: it is in the leftmost
        // node of the subtree rooted at N's right child
        processLeftmost(nodePtr0>rightChildPtr, replacementItem)
        Put replacementItem in node N
    } // end if
```

```
processLeftmost(inout nodePtr:TreeNodePtr, out treeItem:TreeItemType)
// Retrieves into treeItem the item in the leftmost descendant of the node to which
// nodePtr points. Deletes the node that contains this item.

        if (nodePtr->leftChildPtr == NULL)
        {       // this is the node you want; it has no left child, but it might
                // have a right subtree
                treeItem = nodePtr->item
                delPtr = nodePtr

                // actual argument corresponding to nodePtr is a child pointer of nodePtr's
                // parent; thus, the following "moves up" nodePtr's right substree
                nodePtr = nodePtr->rightChildPtr

                delete delPtr
        }
        else
                processLeftmost(nodePtr->leftChildPtr, treeItem)
```

5.9.2 On pages 555 to 563 is the implementation of Binary Search Tree.

6. A "path" of a binary search tree starts at the root and it would go right and left.
   - 6.1 The maximum number of comparisons to find a node would equal to the height of the tree.
   - 6.2 The following is the height of a tree that has no left nodes (Figure 10-30) and it is the maximum height:



   - 6.3 The minimum height of a tree is a bit more difficult to get.
   - 6.4 You need to determine the maximum number of nodes at a particular height.

6.5 Theorem 10-2 would have the following:

A full binary tree of height $h \geq 0$ has $2^h - 1$ nodes

6.6 And theorem 10-3 states:

The maximum number of nodes that a binary tree of height h can have is $2^h - 1$.

6.7 So a tree of height 3 can have, at maximum, 7 nodes.

6.7.1 On page 566 has Theorem 10-4 states:

The minimum height of a binary tree with n nodes is $[\log_2(n + 1)]$

6.7.2 The binary search tree would have a height of $[\log_2(n + 1)]$ to be efficient.

6.8 If you insert the names in order like Alan, Bob, Ellen, Jane, Nancy, Tom, and Wendy, you would get a maximum height tree.

6.8.1 If you inserted the names in order of Jane, Bob, Tom, Alan, Ellen, Nancy, and Wendy you would get a tree of minimum height.

6.8.2 If the insertion and deletion operations in real life are random and are sufficiently large number of items then you would get a height close to $[\log_2(n + 1)]$.

6.8.3 But real life operations would probably not be random.

6.8.4 For instance, what if someone "arranged" a list in alphabetic order.

6.8.5 That would give the tree of maximum height. However, there are ways to balance a binary search tree.

7. Summarizing the order of operations in figure 10-34 is:

Operation	Average case	Worst case
Retrieval	O(log n)	O(n)
Insertion	O(log n)	O(n)
Deletion	O(log n)	O(n)
Traversal	O(n)	O(n)

7.1 Treesort uses the binary search tree to sort an array.

7.2 The basic idea of the algorithm is:

```
Treesort(inout anArray:ArrayType, in n:integer)
// Sorts the n integers in an array anArray into ascending order

Insert anArray's elements into a binary search tree bTree

Traverse bTree in inorder. As you visit bTree's nodes, copy their data items into
successive locations of anArray
```

7.3 You can save a binary search tree in a file two different ways: one to allow restoring the tree in the original shape and the second to restore it in the balanced shape.

7.4 If you write out the binary search tree in preorder, you will be able to read the tree back in using the searchTreeInsert to insert these values in the same form as they were written out.

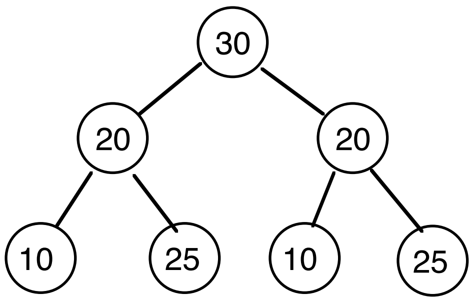
7.5 To make the tree balanced, you need to write the values inorder then attempt to recreate the tree.

7.5.1 For instance the algorithm would create a balanced tree from a sorted file:

```
readFull(out treePtr:TreeNodePtr, in n:integer)
// Builds a full binary search tree from n sorted values in a file.
// treePtr will point to the tree's root.

        if (n > 0)
        {       // construct the left subtree
                treePtr = pointer to new node with NULL child pointers
                readFull(treePtr->leftChildPtr, n/2)

                // get the root
                Read item from file into treePtr->item

                // construct the right subtree
                readFull(treePtr->rightChildPtr, n/2)
        } // end if
```

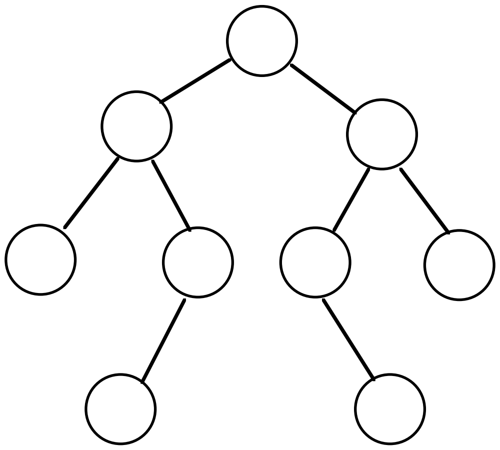
7.5.2 If you know what n is, you can read from the file.

7.5.3 For instance, figure 10-36:



10	20	25	**30**	40	50	60

7.5.4 However there are problems with the readFull tree is if the tree is not full, you could have the following, figure 10-37:



7.5.5 That is not a complete tree.

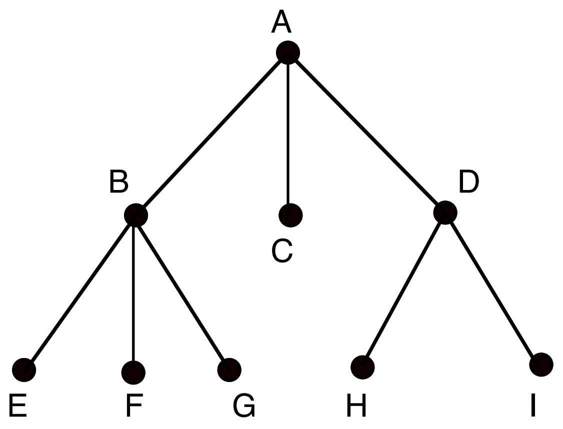
7.5.6 The following algorithm would compensate for that:

```
readTree(out treePtr:TreeNodePtr, is n:integer)
// Builds a minimum-height binary search tree from n sorted values in a file.
// treePtr will point to the tree's root.

        if (n > 0)
        {       // construct the left subtree
                treePtr = pointer to new node with NULL child pointers
                readTree(treePtr->leftChildPtr, n/2)

                // get the root
                Read item from file into treePtr->item

                // construct the right subtree
                readTree(treePtr0>rightChildPtr, (n-1)/2)
        } // end if
```

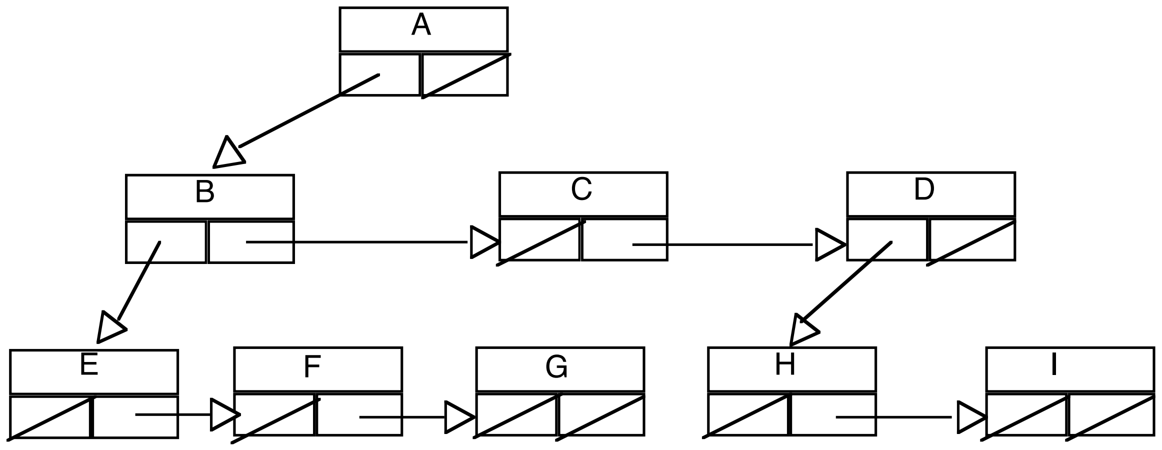
7.6 There are also STL Search Algorithms, of course.

Chapter 10-4, General Trees

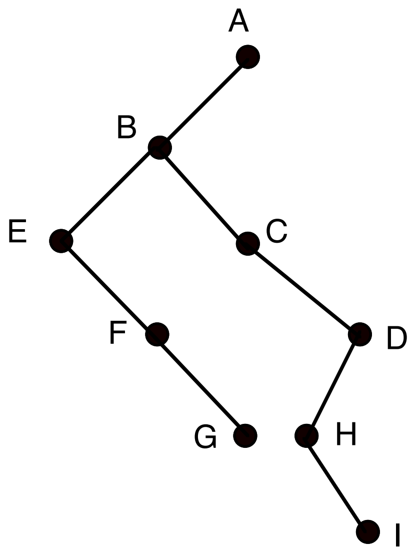
1 General trees can have more than two children.
   1.1 For instance, figure 10-38:



   1.2 You can implement the above using a binary tree structure.
      1.2.1 For instance figure 10-39 shows the same general tree using binary tree:

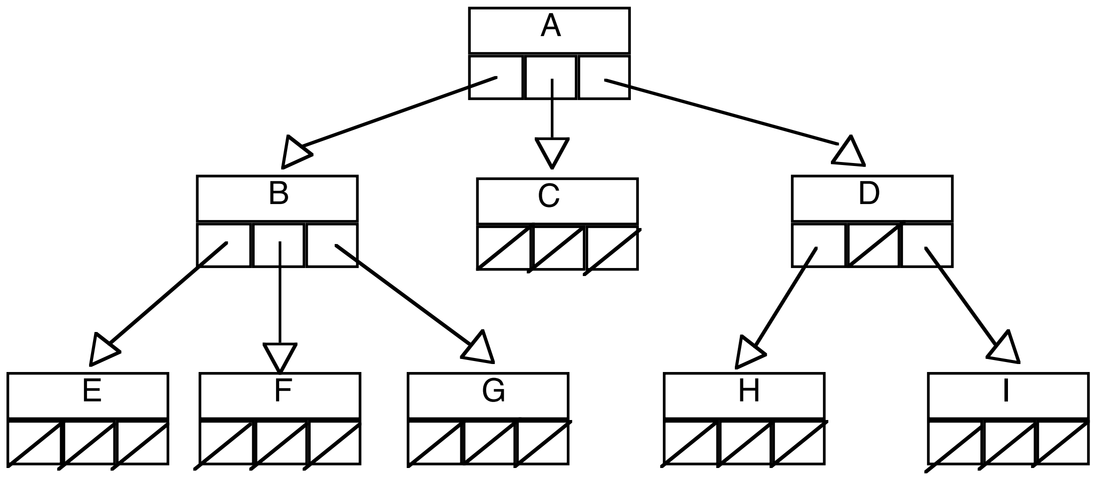


      1.2.2 The shape of the above would be figure 10-40:



1.3 Though you can have more than one pointer defined.

1.3.1 For instance, a three pointer general tree would look like:



1.3.2 The children would be left, middle and right.