Binary Search Trees

A binary search tree is a data structure that is used to represent sets whose elements are ordered by some linear order. As usual the order will be represented by $<$. In particular a binary search tree is either

1. the empty set

2. a node (called the root) together with two binary search trees, one called the left subtree and the other called the right subtree. Each node in the left subtree is less than the root and each node in the right subtree is larger than the root.

All binary trees that we shall consider will be finite. Binary search trees support the operations insert, delete, member, minimum, maximum.

We will assume that the nodes of the tree are given by the deceleration:

```
struct tnode
{
     int Item;
     tnode *Left, *Right;
};
```

We will also assume that there is a pointer

```
     tnode *Root
```

that points to the root of the tree.

First to find the minimum value in a binary search tree we start with a procedure to find the minimum value in a child tree.

Procedure subTreeMin (tnode *Node)
begin

```
    if (*Node has no left child)
      return *Node.Item
    else
      return subTreeMin (*Node.Left)
end
```

Next the procedure to find the minimum in a binary search tree.

```
Algorithm Find_Min (tnode *Root)
begin
    subTreeMin (Root)
end
```

Note that Root points to the root of the binary search tree in which we are looking for a minimum. Also we are assuming that the tree is not empty. The maximum can be found in a similar manner.

The algorithm to test for membership is similar. To test for membership in a binary search

tree we start with a procedure to test for membership in a child tree.

```
Procedure subTreeMember
          (tnode *Node, int searchValue)
begin
   if (Node is NULL)
     return false
   else if (searchValue < *Node.Item)
     subTreeMember (*Node.Left, searchValue)
   else if (searchValue > *Node.Item)
     subTreeMember (*Node.Right,
          searchValue)
   else
     return true
end
```

Next the procedure to test for membership in a binary search tree.

Algorithm TreeMember (tnode *Root, int searchValue)
begin
    subTreeMember (Root, searchValue)
end
Note that Root points to the root of the binary search tree which we are searching.

Next to insert into a binary search tree. Again we start with a procedure to insert into a child tree.

When reading the following procedure be sure to recall that all values in a binary search tree must be distinct.

```
Procedure childInsert (tnode *Node,
          int insertValue)
begin
   if (insertValue < *Node.Item)
     if (*Node.Left is not NULL)
       subInsert (*Node.left, insertValue)
     else
       Insert insertValue as
       the left child of *Node.
   else
     if (*Node.right is not NULL)
       subInsert (*Node.Right, insertValue)
     else
       Insert insertValue as
       the right child of *Node.
end
```

Now the procedure to insert in a binary search tree.

Procedure Insert (tnode *Root,
int insertValue)
begin
subInsert (Root, insertValue)
end
As usual, Root points to the root of the binary search tree in which we are inserting.

For simplicity, and to illustrate the recursive nature of binary trees, our algorithms have been recursive. In pratice, however, it is often more efficient to implement these algorithms using a loop.

The next program reads in integer values and stores them in a binary search tree and then prints out the values in such a way that they are displayed in order.

```
#include <iostream>
using namespace std;

struct tnode
{
     int item;
     tnode *left, *right;
     tnode(int Item, tnode *Left = 0,
                     tnode *Right = 0)
          {item = Item; left = Left;
                     right = Right;}
};

void tprint(tnode *tptr)
{
     if ( tptr )
     {
          tprint( tptr->left );
          cout << tptr->item << '\n';
          tprint( tptr->right );
     }
```

```
}

int main()
{
     tnode *tree = 0, *prev, *curr;
     int ival;
     cout << "Input values, negative "
          << "to terminate" << endl;
     cout << "Input value : ";
     cin >> ival;
     while ( ival >= 0 )
     {
          tnode *tptr = new tnode(ival, 0, 0);
          if ( !tree )
          {
               tree = tptr;
          }
          else
          {
               curr = tree;
               while ( curr )
```

```
                {
                    prev = curr;
                    if (ival < curr->item)
                        curr = curr->left;
                    else
                        curr = curr->right;
                }
                if (ival < prev->item)
                    prev->left = tptr;
                else
                    prev->right = tptr;
            }
        cout <<"Input value : ";
        cin >> ival;
        }
        tprint( tree );
}
```

Finally we see how to delete a node from a binary search tree. We notice that

- If a node has at most one child then we can treat removal just as we do for a linked list.

- For any node with a left child, the largest node that is less than the given node has no right child.

The procecure to remove a node is as follows:

- If the node to be removed has no left child, remove it as we would in a linked list.

- To remove node with a left child, find the largest node that is less than the given node, call this node X. Swap values with

Node X and the node to be removed. Finally delete node X.

```
#include <iostream>
using namespace std;
struct tnode
{
     int item;
     tnode *left, *right;
     tnode(int Item, tnode *Left, tnode *Right)
          {item = Item; left = Left; right = Right;
};

void tprint(tnode *tptr)
{
     if ( tptr )
     {
          tprint( tptr->left );
          cout << tptr->item << ' ';
          tprint( tptr->right );
     }
```

```
}

int main()
{
     tnode *tree = 0, *prev, *curr, *trptr;
     int ival;
     cin >> ival;
     while ( ival > 0 )
     {
          tnode *tptr = new tnode(ival, 0, 0);
          if ( !tree )
          {
               tree = tptr;
          }
          else
          {
               curr = tree;
               while ( curr )
               {
                    prev = curr;
                    if (ival < curr->item)
```

```
                    curr = curr->left;
                else
                    curr = curr->right;
            }
            if (ival < prev->item)
                prev->left = tptr;
            else
                prev->right = tptr;
    }
cin >> ival;
}
tprint( tree );
cout << '\n';
cout << "Input value to be removed ";
cin >> ival;
prev = 0;
curr = tree;
while ( (curr) && ( curr->item != ival) )
{
    if( curr->item > ival )
    {
```

```
                prev = curr;
                curr = curr->left;
           }
           else
           {
                prev = curr;
                curr = curr->right;
           }
      }

     if ( curr )
     {
/* if node to be removed has no left child */
           if (curr->left == 0)
           {
                if ( !prev )
                     tree = curr->right;
                else if ( prev ->left == curr )
                     prev->left = curr->right;
                else
                     prev->right = curr->right;
```

```
            }
            else
            {
/* else find largest node less than curr->item */
                trptr = curr;
                prev = curr;
                curr = curr->left;
                while ( curr->right )
                {
                     prev = curr;
                     curr = curr->right;
                }
                trptr->item = curr->item;
                /* delete curr */
                if ( prev->left == curr)
                     prev->left = curr->left;
                else
                     prev->right = curr->left;
            }
            delete curr;
      }
```

```
        tprint( tree );
        cout << '\n';
}
```