1 Node Insertion Using Recursion

Information can be inserted into dynamic data structures using recursion.

1.1 Lists

1.1.1 Node Declaration

```
struct node
{
   int info;
   struct node *next;
};

typedef struct node * NodePtr;
```

1.1.2 Adding a Node Recursively — Description

Q: Where is the node added?

Q: Where is the node added?
A: End of the list. Elsewhere?

Q: What is the base case?

Q: Where is the node added?
A: End of the list.
A: Elsewhere? Ordered list.

Q: What is the base case?
A: Empty list.

Q: What is the general case?

Q: Where is the node added?
A: End of the list.

Q: What is the base case?
A: Empty list.

Q: What is the general case?

Q: Where is the node added?
A: End of the list.

Q: What is the base case?
A: Empty list.

Q: What is the general case?
A: Non-empty list.

Q: How do we insert a value?

Q: Where is the node added?
A: End of the list.

Q: What is the base case?
A: Empty list.

Q: What is the general case?
A: Non-empty list.

Q: How do we insert a value?
A: Traverse to end of list (location) recursively.

Q: Where is the node added?
A: End of the list.

Q: What is the base case?
A: Empty list.

Q: What is the general case?
A: Non-empty list.

Q: How do we insert a value?
A: Traverse to end of list (location) recursively.

Q: How?

1.1.3 Adding a Node Recursively — Pass By Reference

```
void AddNodeRecursive( NodePtr& h, int x )
{
    if( h != NULL )
    {
        AddNodeRecursive( h->next, x );
    }
    else
    {
        NodePtr n = new node;

        n->info = x;
        n->next = NULL;

        h = n;
    }
}
```

A couple of questions:

Q: How does this work?

Q: Will it work without the `&`?

1.1.4 Adding a Node Recursively — Pass By Pointer

```
void AddNodeRecursive2( NodePtr* h, int x )
{
    if( *h != NULL )
    {
        AddNodeRecursive2( &(*h)->next, x );
    }
    else
    {
        NodePtr n = new node;

        n->info = x;
        n->next = NULL;

        *h = n;
    }
}
```

Q: What is a `NodePtr*`?
A: `struct node **`

Note the extra `*` needed to access `h`!

1.2 Trees

1.2.1 BST Node Declaration

```
typedef struct  BSTreeNode
{
    int        data;
    BSTreeNode *left;
    BSTreeNode *right;
} *BSTreePtr;
```

Q: Where is the node added?
A: Appropriate location in tree.

Q: What is the base case?
A: Empty tree.

Q: What is the general case?
A: Non-empty tree.

Q: How do we insert a value?
A: Traverse to location recursively.

Q: How?

1.2.2 Adding a Node Recursively — Pass By Reference

```
void Add_BST_Recursive( BSTreePtr & t, int val )
{
   if( t == NULL)
   {
       BSTreePtr newPtr = new BSTreeNode;
                                // Initialize
       newPtr->data  = val;
       newPtr->left  = NULL;
       newPtr->right = NULL;

       t = newPtr;
   }
   else if( val <= t->data )  // Add to left subtree
   {
       Add_BST_Recursive( t->left, val );
   }
   else   // Add right to subtree
   {
       Add_BST_Recursive( t->right, val );
   }
}
```

1.2.3 Adding a Node Recursively — Pass By Pointer

An exercise for the interested student.

Compare Deletion

Compare the recursive and non-recursive versions of the code and the *inorder predecessor* and *inorder successor* versions.

Table 1: Recursive

```
void Add_BST_Recursive(
       BSTreePtr & t, int val )
{
   if( t == NULL)
   {
      BSTreePtr newPtr;

      newPtr = new BSTreeNode;
             // Initialize
      newPtr->data  = val;
      newPtr->left  = NULL;
      newPtr->right = NULL;

      t = newPtr;
   }
   else if( val <= t->data )  // Add left subtree
   {
      Add_BST_Recursive( t->left, val );
   }
   else   // Add right subtree
   {
      Add_BST_Recursive( t->right, val );
   }
}
```

Table 2: Non-recursive

```
void AddNode(
       DATA_TYPE newData )
{
   TreePtr newPtr;

   newPtr = new BSTreeNode;
      // Add new data in the new node's data field
   newPtr->data     = newData;
   newPtr->leftPtr  = NULL;
   newPtr->rightPtr = NULL;

   //  If the BST is empty, insert the new data in r
   if( rootPtr == NULL )
   {
      rootPtr = newPtr;
   }
   else   //  Look for the insertion location
   {
      TreePtr   treePtr = rootPtr;
      TreePtr   targetNodePtr;

      while( treePtr != NULL )
      {
        targetNodePtr = treePtr;
        if( newData == treePtr->data )
           // Found same data; ignore it.
           return;
        else if( newData < treePtr->data )
           // Search left subtree for insertion loca
           treePtr = treePtr->leftPtr;
        else   //  newData > treePtr->data
           // Search right subtree for insertion loc
           treePtr = treePtr->rightPtr;
      }

      //  "targetNodePtr" is the pointer to the
      //  parent of the new node.  Decide where
      //  it will be inserted.
      if( newData < targetNodePtr->data )
         targetNodePtr->leftPtr = newPtr;
      else  // insert it as its right child
         targetNodePtr->rightPtr = newPtr;
   }
}
```

Table 3: Wirth's Inorder Predecessor

```
void Delete( TreePtr &p, int x )
{
   TreePtr  q;

   if( p == NULL )
      return;  /* item not in tree */
   else if( x < p->data )
      Delete( p->leftPtr, x  );
   else if( x > p->data )
      Delete( p->rightPtr, x );
   else    /* delete p   */
   {
      q = p;
      if(     q->rightPtr == NULL)
         p = q->leftPtr;
      else if( q->leftPtr == NULL)
         p = q->rightPtr;
      else Del( q, q->leftPtr );
   }
}


void Del( TreePtr& q, TreePtr& r )
{
   if( r->rightPtr != NULL )
      Del( q, r->rightPtr );
   else
   {
      q->data = r->data;
      q = r;
      r = r->leftPtr;
   }
}
```

Table 4: Standard Inorder Successor

```
void DeleteNode(
       TreePtr& treePtr, DATA_TYPE val )
{
   if( treePtr == NULL )
      return;
*/else if( val == treePtr->data )
      DeleteNodeItem( treePtr );
   else if( val < treePtr->data )
      DeleteNode( treePtr->leftPtr, val );
   else
      DeleteNode( treePtr->rightPtr, val );
}

void DeleteNodeItem( TreePtr& treePtr )
{
   TreePtr delPtr;

   if( IsLeaf(treePtr) ) {
      delete treePtr;
      treePtr = NULL;
   }
   else if( treePtr->leftPtr == NULL ) {
      delPtr = treePtr;
      treePtr = treePtr->rightPtr;
      delPtr->rightPtr = NULL;
      delete delPtr;
   }
   else if( treePtr->rightPtr == NULL ) {
      delPtr = treePtr;
      treePtr = treePtr->leftPtr;
      delPtr->leftPtr = NULL;
      delete delPtr;
   }
   else {
      DATA_TYPE  replacementItem;

      ProcessLeftMost( treePtr->rightPtr, replacement
      treePtr->data = replacementItem;
   }
}

void ProcessLeftMost( TreePtr& treePtr, DATA_TYPE& the
{
   if( treePtr->leftPtr != NULL )
      ProcessLeftMost( treePtr->leftPtr, theItem );
   else
   {
```