ESO207 Programming Assignment-2.1

By: S Pradeep(200826) and Yash Gupta(201144)

Pseudo Code for Merge

```
typedef struct Node{                          //structure of nodes of tree
    int kind                                  //represent leaf,two or three node
    int x,y                                   //to store minimum values of middle and right
    struct Node*lchild,*mchild,*rchild        //children
}node

node* create3node()                           //Creates a three-node with all children to set to null
    Declare and allot memory to node n              //and min values initialize to 0
    n.lchild=n.mchild=n.rchild=NULL
    n.x=n.y=0
    n.kind=3                                  //To represent 3node
    return n

node* create2node()                           //Creates a two node
    Declare and allot memory to node n
    n.lchild=n.mchild=n.rchild=NULL
    n.x=n.y=0
    n.kind=2                                  //To represent 2node
    return n

node* createleafnode()                        //Creates a leaf node
    Declare and allot memory to node n
    n.lchild=n.mchild=n.rchild=NULL
    n.x=n.y=0
    n.kind=1                                  //To represent leaf node
    return n

int height(node*root)                         //function which returns height of a tree
    if(root==NULL) :
        return 0
    return 1+height(root.lchild)

int min(node*root)                            //function which returns minimum element of a tree
    Declare and allot memory to node n
    while(n.kind!=1) :
        n=n.lchild
    endwhile
    return n.x

typedef struct Tree {    //structure which stores 2 nodes and min. value of 2nd node if present
```

```
   node*n1,*n2          //this struct. helps in returning final nodes and value after insertion
   int m                //n1, n2 act as left and right child of their parent and m is min of n2
}tree

tree*maketree()                        //Creates a tree structure
   Declare and allot memory to tree t
   t.n1=NULL
   t.n2=NULL
   return t
node* Merge(node*s1,node*s2)           //Function which returns final node of merged tree
   node*s=create2node()                //creates node to store final tree to be returned
   h1=height(s1)                       //heights of trees s1 and s2 respectively
   h2=height(s2)
   if(h1==h2) :                        //If heights are equal, then add s1 and s2 are children to s
      s.lchild=s1                      //s-parent node whose left child is s1 and middle child is s2
      s.mchild=s2
      s.x=min(s2)                      //stores min. of s2 in s
      return s
   endif
   h=h1-h2
   tree*t=maketree()                   //to store tree struct which insert1/2 will return
   node*ptr=s1                         //pointer to find node where second tree has to be merged
   if(h1>h2) :                         //at a height of h2+1, in case h(s1)>h(s2)
      while(--h) :                              //moves ptr to rightmost node at height h2+1
         if(ptr.rchild!=NULL) :
            ptr=ptr.rchild
         else :
            ptr=ptr.mchild
      endwhile
      t=insert1(s1,s2,ptr)        //Function call to insert s2 at node referred by ptr to the right
   endif
   h=-h
   ptr=s2                         //in case h(s2)>h(s1) ptr points to left most node at height h1+1
   if(h2>h1) :
      while(h) :
         ptr=ptr.lchild
         h--
      endwhile
      t=insert2(s1,s2,ptr)
   endif
   if(t.n2==NULL) :
      return t.n1            //if n2, i.e 2nd node returned by insert is NULL then tree is simply
                             //rooted at n1 so return it only
   else :                         //if n2 is not NULL then return their parent s
      s.lchild=t.n1               //whose left child is n1 and right child is n2
      s.mchild=t.n2
      s.x=t.m                     //min of middle child of s is min of n2 which is returned in t.m
   endifelse
   return s
//end of Merge function definition

// insert1 funct. which return tree struct after insertion in the case when h(s1) > h(s2).  In this
//function, s2 is merged in the tight side of s1 at appropriate height
tree*insert1(node*s1,node*s2,node*ptr) {      //recursively insert1
   tree*t=maketree()                      //here s1 is pointer which will traverse downwards from root
                                          //to ptr, on the way recursively calling insert1
   if(s1==ptr) :                          //Base case
      if(s1.kind==2) :                    //if s1 is 2node then insert s2 by converting s1 to 3node
         s1.kind=3                        //and s2 be its rightmost child
         s1.rchild=s2
         s1.y=min(s2)                     //stores min. of s2 in s1 in y
         t.n1=s1
         t.n2=NULL
         return t                         //return tree struct as explained earlier
      else :                              //if s1 is 3node so split s1 to two 2nodes
         node*new=create2node()   //new is created, here value of elements of s1<new
         new.x=min(s2)                       //stores min of s2 as required
         new.lchild=s1.rchild        //left child of new node is former right child of s1
         new.mchild=s2                    //middle child of new node is s2
         s1.kind=2                        //s1 is converted into 2node after rchild copied to new node
         s1.rchild=NULL
         t.n1=s1                         //now n1 points to s1
         t.n2=new                        //n2 points to new
         t.m=s1.y                        //m stores min. of n2 which is - what was min of rchild of s1
         return t
      endifelse
   endif
   if(s1.kind==2) :                      //if not base case and it is 2node
      t=insert1(s1.mchild,s2,ptr)      //recursive call to obtain tree after inserting inside mchild
      if(t.n2==NULL) :                    //if after insertion n2 is still null
         s1.mchild=t.n1                   //then add only n1 to its parent this case is when merging
         tree*t1=maketree()          //doesn’t necessitate the creation of new node at this height
         t1.n1=s1
         t1.n2=NULL
         return t1                      //return tree
      else :                             //if t.n2 is not null and parent(s1) is 2node
         s1.kind=3                       //so convert s1 to 3node and add n1 and n2
         s1.mchild=t.n1
         s1.rchild=t.n2
         s1.y=t.m                        //and now also store min. of n2(rightmost child)
         tree*t1=maketree()
         t1.n1=s1
         t1.n2=NULL
         return t1
      endifelse
   endif
   if(s1.kind==3) :                    //if it is not base case and is 3node
      t=insert1(s1.rchild,s2,ptr)    //recursive call returns tree after insertion of s2 in rchild
      if(t.n2!=NULL) :                 //if n2 is not null and it is 3node
         node*n=create2node()           //so split it into two 2nodes one is s1 and other n
         n.lchild=t.n1                //s1 with original children except rchild and n with n1 and n2
         n.mchild=t.n2
         n.x=t.m                     //min. of mchild of n is as returned by t in m
         s1.rchild=NULL          //as splitting done so rchild of s1 now present with n
         s1.kind=2                //Change into 2node
         tree*t1=maketree()
         t1.n1=s1
         t1.n2=n
         t1.m=s1.y               //min of whole n remains same as min of former rchild of s1 as
         return t1                //as min of rchild doesn't change because merge done to right.
      else :                         //if n2 is null then simply add n1 to s1 only
         s1.rchild=t.n1          //min of n1 doesn't change as all elements that are inserted are
         tree*t1=maketree()  //bigger than those that were in rchild
         t1.n1=s1
         t1.n2=NULL
         return t1
      endifelse
   endif
//End of insert1 function. No return statement as all cases are already taken into account

//insert2 funct. which return tree struct after insertion in case when h(s2) > h(s1). In this
//function, s1 is merged in the left side of s2 at appropriate height
tree*insert2(node*s1,node*s2,node*ptr) {     //recursively insert2
   tree*t=maketree()           //here s2 is pointer which will traverse downwards from root
                               //to ptr, on the way recursively calling insert2
   if(s2==ptr) :               //Base case when we arrive at done where insertion to be done
      if(s2.kind==2) :           //if s2 is 2node then insert s1 by making s2 to be 3node
         s2.kind=3
         s2.rchild=s2.mchild  //as s1 should be added as left child, shift mchild to rchild
         s2.mchild=s2.lchild   // and lchild to mchild
         s2.lchild=s1            //adding s1 as lchild
         s2.y=s2.x               //min of current rchild is min of previous mchild
         s2.x=min(s2.mchild)         //storing min. of mchild as s1.x
         t.n1=s2
         t.n2=NULL
         return t
      else :                         //if s2 is 3node so splitting s2 to two 2nodes
         node*new=create2node()        //one is s2 and other new
         new.x=min(s2.lchild)      //with value of elements of new < that of s2
         new.lchild=s1               //hence making lchild of s2 as mchild of new
         new.mchild=s2.lchild
         s2.kind=2                   //change s2 into 2node
         s2.lchild=s2.mchild        //swapping as a consequence of above
         s2.mchild=s2.rchild
```

```
        int a=s2.x                   //a stores min. of initial mchild (current lchild)
        s2.rchild=NULL               //as it is min. of modified s2
        s2.x=s2.y
        t.n1=new
        t.n2=s2
        t.m=a                        //which should be returned as minimum of t.n2
        return t
      endifelse
    endif
    if(s2.kind==2) :                 //if not base case and 2node
      t=insert2(s1,s2.lchild,ptr)      //recursive call
      if(t.n2==NULL) :                 //if n2 is null only add n1 as lchild
        s2.lchild=t.n1
        tree*t1=maketree()
        t1.n1=s2
        t1.n2=NULL
        return t1
      else :                         //if n2 not null
        s2.kind=3                    //convert s2 to 3node
        s2.rchild=s2.mchild          //swapping so as to add n1 and n2
        s2.mchild=t.n2               //as l and m child
        s2.lchild=t.n1
        s2.y=s2.x                    //as a consequence of swapping
        s2.x=t.m
        tree*t1=maketree()
        t1.n1=s2
        t1.n2=NULL
        return t1
      endifelse
    endif
    if(s2.kind==3) :                 //if not base case and 3node
      t=insert2(s1,s2.lchild,ptr)      //recursive call
      if(t.n2!=NULL) :                 //if n2 not null split s2 to two 2node s2 and n
        node*n=create2node()
        n.lchild=t.n1                  //withvalue of elements of n < that of s2
        n.mchild=t.n2
        n.x=t.m
        s2.lchild=s2.mchild            //swapping so as to convert to 2node as n added to left of s2
        s2.mchild=s2.rchild
        s2.rchild=NULL
        s2.kind=2
        int a=s2.x                   //storing so as to return min of modified s2 as m in t struct
        s2.x=s2.y
        tree*t1=maketree()
        t1.n1=n
        t1.n2=s2
        t1.m=a                     //min value
        return t1
    else :                          //if n2 is null so only adding n1 as lchild
        s2.lchild=t.n1
        tree*t1=maketree()
        t1.n1=s2
        t1.n2=NULL
        return t1
    endifelse
  endif
//End of insert2 function. No return statement as all cases are already taken into account
```

Complexity analysis of Merge

Function height(node *s) and min(node *s) have a time complexity of O(h) where h is the height of the 2-3 tree rooted at s. This is because they traverse from root to one of the leaf nodes in a sequential and straight manner. In Merge, we first call height(s1) and height(s2). These two steps take a complexity of O(h(s1)+h(s2)). If h1==h2, then Merge ends in finitely more steps and is thus O(h(T1)+h(T2)) complexity. If h1 and h2 are not equal, the pointer ptr is moved to the node where the merge has to happen. This step takes O(|h1-h2|) steps. This is followed by a function call of insert1 or insert2 depending on the case. Both s1 and s2 are recursive functions. In insert1, pointer s1 recursively moves down with each call to pointer ptr at height h2+1. In each recursive call, a fixed number of if and assignment conditions are executed. In the base case, when s1 is same as ptr, a call to min(s2) is made. This call takes O(h2) time to execute. This means the overall function insert1 takes C_1(h1-h2)+C_2(h2) < C_3(h1+h2) for any C_3>C_2. This means insert1 has time complexity O(h1+h2). Function insert2 also is a similar function which adds s1 to left side of s2 at appropriate height, while insert1 adds s2 to right of s1 at appropriate height. So in insert2, pointer s2 recursively traverses down till ptr and insertion is cascaded up the recursion stack at that node. So, insert2 also has time complexity of O(h1+h2). Thus, worst case time complexity of Merge is O(h1+h2)+O(|h1-h2|)+O(h1+h2)+c < O(h1+h2). Thus time complexity of Merge operation is **O(h(T1)+h(T2))**.