A report outlining pseudo code and complexity analysis of MERGE and CREATE_INTERVAL_TREE routines

Shashwat Gupta (14IE10028)

February 12, 2016

1 CREATE_INTERVAL_TREE

The Create Subroutine takes in parameters as l,u,n and generates a Tree within [l,u] and with n nodes. The nodes are created recursively and all the nodes are dynamically generated and linked with each other.
There are 2 base conditions: when n = 0 or 1, for which the code is provided. This speciality indicates that there is a NULL point (at n=0) and there is a terminal node (n=1). For all other values of n, the median is found and the limits are calculated and passed recursively. rem is also taken into consideration. rem is the remainder. If we want the range to be from l to u and say n is not divisible by u-l+1, then some nodes would have more elements. This balance is done by rem. The following is the Code

```
node* CREATE_INTERVAL_TREE(int l, int u, int n) //working correctly
{
    if(n==0)
        return NULL;
    if(n==1)
    {
        node *temp = (node*)malloc(sizeof(node));
        temp->lower=l;
        temp->upper=u;
        list *templist = (list*)malloc(sizeof(list));
        templist->next=NULL;
        templist->data=0;
        temp->listptr=templist;
        temp->lChild=NULL;
        temp->rChild=NULL;
        return temp;
    }
    int nn=n/2+1; //median interval number
    int rem=(u-l+1)%n;
    node *temp = (node*)malloc(sizeof(node));
    temp->lower=(nn-1)*interval+l+rem/2;
    temp->upper=nn*interval+l-1+rem-rem/2;
    list *templist = (list*)malloc(sizeof(list));
    templist->next=NULL;
    templist->data=0;
    temp->listptr=templist;
    temp->lChild=CREATE_INTERVAL_TREE(l,temp->lower-1,nn-1);
    temp->rChild=CREATE_INTERVAL_TREE(temp->upper+1,u,n-nn);
    return temp;
}
```

Listing 1: elimin_solve

2 Time Complexity Calculation of CREATE_INTERVAL_TREE

Time complexity has the recursive relation:

$$T(n) = T(n/2) + T(n/2 - 1) + \theta(1)$$

where n is the number of nodes to be created.
Solving this equation by recursive method yields T(n) $\in O(logn)$.

3 MERGE

The Merge Subroutine takes in parameters as T,l,u and traverses the Tree until it fins the mother root from where the changes will take effect. The nodes are traversed recursively. Once we reach the node from where the effects will take place, we dynamically create a node N which will save all the changes within it self and N will be returned finally to complete the merged tree. The following is the Code

```
node* MERGE(node *T,int l,int u)
{
    if(T==NULL)
        return NULL;
    if(T->lower>u)
    {
        T->lChild=MERGE(T->lChild,l,u);
        return T;
    }
    if(T->upper<l)
    {
        T->rChild=MERGE(T->rChild,l,u);
        return T;
    }
    node *N = (node*)malloc(sizeof(node));
    N->lower=l;
    N->upper=u;
    list *templist = (list*)malloc(sizeof(list));
    templist->next=NULL;
    templist->data=0;
    N->listptr=templist;
    N->lChild=NULL;
    N->rChild=NULL;
    eat(T,N,2,0);
    return N;
}
```

Listing 2: elimin_solve

The eat subroutine is the recursive function created to traverse the tree and make the changes once the root has been located from where the changes begin to occur. It takes into consideration several cases and carefully traverses the tree to make the necessary changes. It starts with a base case for T=NULL and then checks for several possibilities like if the node is overlapping from left side or right side or it is completely inside. It then makes calls to its right and left child to get their data and make them the children of the main node N.

The following is the Code of eat

```
void eat(node* T, node* N, int check, int flag)
{
    if (T==NULL)
        return;
    if(T->lower<N->lower&&T->upper<=N->upper) //overlapping left
    {
        list *listdata = (list*)malloc(sizeof(list));
        listdata=T->listptr;
        while(listdata->data<N->lower&&listdata->data!=0)
            listdata=listdata->next;
        while(listdata->next!=NULL)
        {
            INSERT(N,listdata->data);
            listdata=listdata->next;
        }
        listdata=T->listptr;
        if (listdata->data>=N->lower)
        {
            listdata->data=0;
            listdata->next=NULL;
        }
        while(listdata->next!=NULL&&listdata->next->data<N->lower)
            listdata=listdata->next;
        listdata->next=NULL;
        T->upper=N->lower-1;
        if(flag==1)
            N->lChild=T;
        return;
    }
    if(T->lower>=N->lower&&T->upper>N->upper) //overlapping right
    {
        list *listdata = (list*)malloc(sizeof(list));
        listdata=T->listptr;
        while(listdata->next!=NULL&&listdata->data<=N->upper)
        {
            INSERT(N,listdata->data);
            listdata=listdata->next;
        }
        T->listptr=listdata;
        eat(T->lChild,N,0,1);
        T->lower=N->upper+1;
        if (T->rChild==NULL)
        {
            N->rChild=T;
            T->lChild=NULL;
        }
        else
        {
            T->rChild->lower=N->upper+1;
            while(listdata->next!=NULL)
            {
                INSERT(T->rChild,listdata->data);
                listdata=listdata->next;
            }
            N->rChild=T->rChild;
        }
        return;
    }
    if(T->lower>=N->lower&&T->upper<=N->upper) //completely within
    {
        list *listdata = (list*)malloc(sizeof(list));
        listdata=T->listptr;
        while(listdata->next!=NULL&&listdata->data!=0)
        {
            INSERT(N,listdata->data);
            listdata=listdata->next;
        }
        eat(T->lChild,N,0,0);
        eat(T->rChild,N,1,0);
        return;
    }
    else if(check==0&&N->lower>T->upper)
    {
        eat(T->rChild,N,1,1);
        if(T->rChild!=NULL&&T->rChild->lower>=N->lower&&T->rChild->upper<=N->upper)
            T->rChild=NULL;
        if(flag==0)
            N->lChild=T;
    }
    else if(check==1&&T->lower>N->upper)
    {
        eat(T->lChild,N,0,1);
        if(T->lChild!=NULL&&T->lChild->lower>=N->lower&&T->lChild->upper<=N->upper)
            T->lChild=NULL;
        if(flag==0)
            N->rChild=T;
    }
    else if(check==0&&flag==0)
        N->lChild=T;
    else if(check==1&&flag==0)
        N->rChild=T;
}
```

Listing 3: elimin_solve

4 Time Complexity Calculation of MERGE

Time complexity has the recursive relation:

$$T(n) = T(n/2) + \theta(1)$$

where n is the number of nodes.

Solving this equation by recursive method yields T(n) $\in O(logn)$.