Multi-merge

Shashwat Gupta (14IE10028)

March 21, 2016

1 Data Structures

The following is the code for my data structures used in the program

```
Node:
typedef struct _node
{
    struct _node *parent,*left,*right,*child;
    int key,degree;
    bool mark;
}node;

Heap
typedef struct _heap
{
    node *root,*min;
    int n;
}heap;
```

Listing 1: Structures

2 Initialisations of the data structures

1. Heap

```
heap* initHeap()
{
    heap* h = (heap *)malloc(sizeof(heap));
    h->root=NULL;
    h->min=NULL;
    h->n=0;
    return h;
}
```

Listing 2: heap

2. Node

```
node* initNode(int n)
{
    node *N = (node *)malloc(sizeof(node));
    N->parent=NULL;
```

```
    N->left=NULL;
    N->right=NULL;
    N->child=NULL;
    N->key=n;
    N->degree=0;
    N->mark=0;
    return N;
}
```

Listing 3: node

3 Heap Functions

1. Insert

The following is the Code for inserting into the Heap.

```
heap* insertToHeap(heap *h,node *n)
{
    if(h->root==NULL)
    {
        h->root=n;
        h->root->left=NULL;
        h->root->right=NULL;
        h->root->parent=NULL;
        h->min=n;
        h->n++;
        return h;
    }
    if(h->min->key>n->key)
        h->min=n;
    node *temp;
    temp=h->root;
    while(temp->right!=NULL)
        temp=temp->right;
    temp->right=n;
    n->left=temp;
    n->right=NULL;
    n->parent=NULL;
    h->n++;
    return h;
}
```

Listing 4: insert

2. Extract min

The following is the Code for extracting the minimum.

```
node* extractMin(heap *h)
{
    node* oldMin=h->min;
    if(h->min==h->root)
    {
        if(h->root->right==NULL)
        {
            h->root=h->root->child;
            h->min=h->root;
```

```
            h->root->parent=NULL;
            oldMin->left=NULL;
            oldMin->right=NULL;
            h->n--;
            return oldMin;
        }
        else
        {
            h->root=h->root->right;
            h->root->left=NULL;
            h->min=h->root;
        }
    }
    else if(h->min->right==NULL)
    {
        h->min->left->right=NULL;
        h->min=h->min->left;
    }
    else
    {
        h->min->left->right=h->min->right;
        h->min->right->left=h->min->left;
        h->min=h->min->right;
    }
    oldMin->left=NULL;
    oldMin->right=NULL;
    h->n--;
    node *temp;
    temp=oldMin->child;
    node* A[h->n];
    int i;
    for (i = 0; i < h->n; i++)
        A[i] = NULL;
    i =0;
    while(temp!=NULL)
    {
        A[i]=temp;
        temp=temp->right;
        i++;
    }
    for(i=0;i<h->n;i++)
    {
        if(A[i]!=NULL)
        {
            h=insertToHeap(h,A[i]);
            h->n--;
        }
    }
    return oldMin;
}
```

Listing 5: extract

3. Consolidate heap

The following is the Code for consolidating .

```
heap* consolidateHeap(heap *H)
{
```

```
    if(H->root==NULL)
        return;
    int i,d;
    node* A[H->n];
    for (i = 0; i < H->n; i++)
        A[i] = NULL;
    node *z,*y,*x;
    x = H->root;
    while(x!=NULL)
    {
        d = x->degree;
        while(A[d]!=NULL)
        {
            y = A[d];
            if (x->key > y->key)
            {
                z = x;
                x = y;
                y = z;
            }
            x=linkHeap(H, y, x);
            A[d] = NULL;
            d = d + 1;
        }
        A[d] = x;
        x = x-> right;
    }
    H->min=NULL;
    H->root=NULL;
    for(i=0;i<H->n;i++)
    {
        if(A[i]!=NULL)
        {
            H=insertToHeap(H,A[i]);
            H->n--;
        }
    }
    return H;
}
```

Listing 6: consolidate

4. Delete node

The following is the Code for deletion of node from the Heap

```
heap* removeFromHeap(heap *h, node *x)
{
    h=decKey(h,x,-10000);
    extractMin(h);
    return h;
}
```

Listing 7: delete

5. Dec node key

The following is the Code for decreasing the key of node.

```
heap* cut(heap *h, node* x, node* y)
```

```
{
    if(x==y->child)
    {
        if(x->right!=NULL)
        {
            y->child=x->right;
            x->right->left=NULL;
            x->right=NULL;
        }
        else
            y->child=NULL;
    }
    else
    {
        if(x->right!=NULL)
        {
            x->left->right=x->right;
            x->right->left=x->left;
            x->left=NULL;
            x->right=NULL;
        }
        else
        {
            x->left->right=NULL;
            x->left=NULL;
        }
    }
    h=insertToHeap(h,x);
    h->n--;
    x->parent=NULL;
    x->mark=0;
    return h;
}
heap* cascadingCut(heap *h, node* y)
{
    node *z=y->parent;
    if(z!=NULL)
    {
        if(y->mark==0)
            y->mark=1;
        else
        {
            h=cut(h,y,z);
            h=cascadingCut(h,z);
        }
    }
    return h;
}
heap* decKey(heap *h, node* x, int k)
{
    if(k>x->key)
    {
        printf("Error, key is greater than original\n");
        return h;
    }
    x->key=k;
    node *y=x->parent;
```

```
59      if(y!=NULL && x->key < y->key)
60      {
61          h=cut(h,x,y);
62          h=cascadingCut(h,y);
63      }
64      if(x->key < h->min->key)
65          h->min=x;
66      return h;
67  }
```

Listing 8: decrease

4 Amortised Costs

These are the costs:

1. Create $\in O(1)$.
2. Merge $\in O(1)$.
3. Decrease Key $\in O(1)$.
4. Insert into Heap $\in O(1)$.
5. Extract min from Heap $\in O(logn)$.