09/12/2020
Wednesday

Arjun A. S
1BM18CS019
5A1

ADS

Lab - 9

Writeup

Binomial Heap

Functions:

(1) insert(H,K): Inserts a key 'k' To Binomial Heap 'H'. This creates a Heap with single key 'k', then calls union on H and the new Binomial Heap.

(2) getMin (H): It traverses the list of Binomial trees and returns the minimum key.

(3) extractMin(H): This function first calls getMin() then removes the nodes and create a new Binomial Heap by connecting all subtrees of the removed minimum node and Union) is called on H and newly created Binomial Heap.

```
struct Node {
        int data, degree;
        node * child, * sibling, * parent
};

Node * newNode ( int data)
{
        Node *t = ~~newNode~~ new Node;
        t→ data =data;
        t→ degree = 0;
        t→ chid = t→ parent = t→ sibling = NULL;
        return t;
}
```

Arjun AS

Arjun A.S
1BM18CS019

```
list <Node *> insertionof tree ( list <Node *> heap , Node * tree)
{
        list <Node *> temp;
        temp. push_back (tree)
        temp = Union of heap (heap, temp)
        return   adjust (temp);       //rearranging heap
}

list <Node *> unionof heap (list <Node *> l1 , list <Node *> l2)
{
        list <Node *> new;
        list <Node *> :: iterator  it = l1. begin();
        list <Node *> :: iterator  ot = l2. begin();

        while (it != l1.end () && ot != l2.end() )
        {
                if( (*it) -> degree <= (*ot) -> degree)
                {
                        new. push_back (*it);
                        it ++;
                }
                else
                {
                        new. push_back (*ot);
                        ot ++;
                }
        }
        while (it != l1.end())
        {
                new. push_back (*it);
                it ++)
        }
        while (ot != l2.end())
        {
                new. push_back (*ot);
                ot ++;
        }
        return  new;

}
```

Arjun A.S
1BM18CS019

```
list <Node *> insert (list <Node *> head, int data)
 {
        node *temp = newNode (data);
        return insertionof tree(head, temp);
 }

Node * getMin (list <Node *> heap)
{
      list <Node *> :: iterator it = heap.begin();
      Node *temp = *it;
      while (it != heap.end())
      {
            if((*it) ->data < temp->data)
                  temp = *it;
            it ++;
      }
      return temp;
}

list <Node *> extractMin (list <Node *> heap)
{
        list <Node *> new heap, lo;
        Node *temp;
        temp = getMin (heap);
        list <Node *>:: iterator it;
        it = heap. begin ();
        while (it != heap. end())
        {
              if (*it != temp)
              {
                newheap . push_back (*it);
              }

        it ++;
        }
        lo = remove min andrettemp (temp);
        new heap = union of heap (newheap, lo);
        new heap = adjust (newheap);
        return newheap;
}

list <Node *> remove min and rettemp (Node * tree)
{
        list <Node *> heap;
        Node * temp = tree -> child;
        Node * lo;
        while (temp)
        {
                lo = temp;
                temp = temp -> sibling;
                lo -> sibling = NULL;
                heap.push_front (lo);
        }
        return heap;
}
```