Lab - 9.

19UI18CS058
Neelesh Gangopadhyay

Implementation of Binomial Heap

Program

```
struct Node
{ int data, degree;
  Node * child, *sibling, *parent;
};
Node * newNode(int key)
{
  Node * temp = new Node;
  temp -> data = key;
  temp -> degree = 0;
  temp -> child = temp -> parent = temp
       -> sibling = NULL;
  return temp;
}

Node * mergeBinomialTrees(Node* b1, Node *b2)
{ if (b1 -> data > b2 -> data)
      swap(b1, b2);
  b2 -> parent = b1;
  b1 -> child = b2;
  b1 -> degree++;
  return b1;
```

```
list<Node*> & union BinomialHeap (list<Node*> l1,
                                  list<Node*> l2)
{
    BinomialHeap *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.pushback(*it);
            it++;
        }
        else
        {
            new.pushback(*ot);
            ot++;
        }
    }
    while (it != l1.end())
    {
        new.pushback(*it);
        it++;
    }
    return new;
}

list<Node*> adjust(BinomialHeap *heap)
{
    if (heap.size() <= 1)
        return heap;
```

```
list<Node*> new_heap;
list<Node*>::iterator it1, it2, it3;
it1 = it2 = it3 = heap.begin();

if (heap.size() == 2)
{
it2 = it1;
it2++;
it3 = heap.end();

}
else
{
it2++;
it3 = it2;
it3++;
}
while (it1 != heap.end())
{ if (it2 == heap.end())
    it++;

else if ((*it1)->deg == (*it2)->deg)
    {it1++;
    it2++;
    if (it3 != heap.end())
        it3++;
```

```
else if ((it-3) != heap.end() &&
        (*(it+1))->degree == (*(it+2))->degree &&
        (*(it+1))->degree == (*(it+3))->degree)
{
    it++;
    it2++;
    it3++;
}
else if ((*it)->degree == (*it1)->degree)
{ Node* temp;
  *it1 = mergeBinomialTrees(*it1, *it2);
   it2 = heap.erase(it2);
   if ((it3) != heap.end())
       it3++;
}
return heap
}

list<Node*> insertATreeInHeap(list<Node*> heap
                              Node* tree)
{
  list<Node*> temp;
  temp.push_back(tree);
  temp = unionBinomialHeap(heap, temp);
  return adjust(temp);
}

list<Node*> removeMinFromTreeReturnBHeap
(Node* H)
```

```
{
  list<Node*> heap;
  Node* temp = root->child;

    Node* lo;
while (temp)

{ lo = temp;
  temp = temp -> sibling;
    lo -> sibling = NULL;
    heap.push_front(lo);
      ?
    return heap
      ?

list<Node*> insert(list<Node*> _heap, int val)

{ Node* temp = newNode(key);
  return insertATreeInHeap(heap, 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)
{ new_heap.push_back(*it);
}
it++;
}
lo = removeMinFromTreeReturnBHeap(temp);
new_heap = unionBinomialHeap(new_heap, lo);
new_heap = adjust(new_heap);
return new_heap;
}