1BM18CS125
VYSHNAVI.M.N

BINOMIAL HEAP

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

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

}

list<Node*> unionBinomialHeap(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;
    }

list<Node*> adjust(list<Node*> _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())
        it1++;
    else if ((*it1)->degree < (*it2)->degree)
    {
        it1++;
        it2++;
        if (it3 != _heap.end())
            it3++;
    }
    else if ( it3 != _heap.end() &&
            (*it1)->degree == (*it2)->degree &&
            (*it1)->degree == (*it3)->degree )
    {
        it1++;
        it2++;
        it3++;
    }

}
return ~~[illegible]~~ _heap;
}

list<Node*> insertATreeHeap (list<Node*> _heap,
                             Node *tree)
{   list <Node*> temp;
    temp.push_back(tree);
    temp = unionBinomialHeap(_heap, temp);
    return adjust(temp);
}
```

```
list<Node*> removeMinFromTree (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;
}

list<Node*> insert(list<Node*> _head, int key)
{
   Node *temp = new Node(key);
   return insertATreeInHeap(_head, temp);
}
```