Binomial Heap

```
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;
    b2->sibling = b1->child;
    b1->child = b2;
    b1->degree++;
    return b1;
}

list<Node *> unionBionomialHeap(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(it1 != l1.end())
{
    _new.push_back(*ot);
    ot++;
  }
  return _new;
}

list<Node*> insertATreeInHeap(list<Node*>
                _heap, Node *tree)
{
    list<Node*> temp;
    temp.push_back(tree);
    temp = unionBinomialHeap(_heap, temp);
    return adjust(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++~~ it++;
    }
    lo = removeMinFromTreeReturnBHeap(temp);
  newheap= unionBinomialHeap(new_heap, lo);
  new_heap = adjust(newheap);
  return new_heap;
}
```