WAP to Binomial heap

```
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*  mergeBinomialTree (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! = l1end() && 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*> insertATreeInHeap(list<Node*>_
                                        heap,
                               Node *tree)
{
   list <Node*> temp;
      temp.push_back(tree);
      temp = unionBinomialHeap(_heap, temp);
     return adjust(temp);
    }
list <Node *> removeMinFromTreeReturnBHeap(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;
    }
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
         new_heap = adjust(new_heap);
        return new_heap;
      }
```