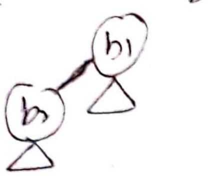
Binomial Heaps

```
#include <bits/stdc++.h>

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* merge_BT(Node* b1, Node *b2)
{
    if (b1 -> data > b2 -> data)
        swap (b1, b2);
    // considering b1 < b2

    b2 -> parent = b1;
    b2 -> sibling = b1 -> child;
    b1 -> child = b2;
    ~~b2~~ b1 -> degree ++;

    return b1;

}
```



```
list <Node*>   unionBT ( list<Node*> l1,
                         list <Node*> l2)
{
    list <Node*> merged_l;
    list <Node*>:: iterator it1= l1.begin();
    list <Node*>:: iterator it2= l2.begin();

  while (it1!= l1.~~begin~~end() &&
         it2!= l2.end())
  {
      if ((*it1)->degree <= (*it2)->degree)
      {   merged_l.push_back(*it1);
          it1++;
      }
      else
      {
          merged_l.push_back(*it2);
          it2++;
      }
  }
   while( it1 != l1.end())
   {   merged_l.push_back(*it1);
       it1++;
   }
   while( it2 != l2.end())
   {   merged_l.push_back(*it2);
       it2++;
   }
  return merged_l;
}
```

```
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++;
            }
```

```
else if ((*it1)->degree == (*it2)->degree)
{
        Node * tmp;
        *it1 = mergeBT(*it1, *it2);
        it2 = _heap.erase(it2);
        if (it3 != _heap.end())
                it3++;

    }

return _heap;

}


list <Node*>        insert_BT (list<Node*> _head,
                                        int key)

{       Node *    tmp = newNode (key);
        ~~return~~    ~~insert A~~
        list<Node *> tmpH;
        tmpH . push_back (tmp);

        tmp = union_BT (tempH, _heap);

        return Adjust (tmp);

}

list <Node*>        extract_Min( list<Node*> _heap)

{
            list <Node*> new_heap, lo;
        Node  *tmp;
            tmp = getMin (_heap);
            list<Node *> ::iterator it;
            it = _heap-begin();
```

```
while (it != heap.end())
{
        if (*it != tmp)
        {
              new heap.pushback(*it);
        }
        it ++;
}
lo = removeMinFromTreeReturnBH(tmp);
new.heap = union_BT(new.heap, lo);
new.heap = adjust(new heap);
  return new.heap;
}
```