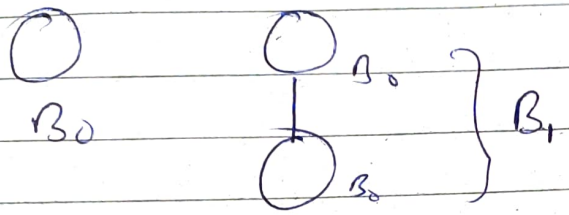
Implementation of Binomial Heap

Binomial heap is a collection of Binomial tree



Three operations to be performed

1 insert(H,k): Inserts a key 'k' to Binomial Heap 'H'. This operation first creates a binomial heap with single key 'k', then calls union on H and the new binomial heap

2 getMin(H): A simple way to get Min() is to traverse the list of root of Binomial trees and return the minimum key. This implementation requires O(log n) time. It can be optimized to O(1) by maintaining a pointer to minimum key root.

3 extractMin(): This operation also uses union(). We first call getMin() to find the minimum key Binomial tree, then we remove the node and create a new binomial heap by connecting all subtrees of the removed minimum node. Finally we call union() on H and the newly created binomial heap. This operation requires O(log n) time

```
#include <bits/stdc++.h>
using namespace std;

// A binomial tree node
struct Node
{
    int data, degree;
    Node *child, *sibling, *parent;
};

Node* newNode(int key)
{
    // initializing Node fields
}
// Merging two binomial trees
Node* mergeBinomialTree(Node *b1, Node* b2)
{
    if(b1->data > b2->data)  // Make sure b1 is
        swap(b1, b2);        smaller

    //Make larger valued tree a child of smaller valued tree
    b2->parent = b1;
    b2->sibling = b1->child;
    b1->child = b2;
    b1->degree++;
    return b1;
}

// Function performs union operation on two binomial heap
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 D(l1) <= D(l2)
        if((*it)->degree <= (*ot)->degree)
        {
            _new.push_back(*it);
            it++;
        }
        else
        {
            _new.push_back(*ot);
            ot++;
        }
    }

    // If there remains some elements in l1
    while(it != l1.end())
    {
        // Push(it) and increment it
    }
    while(ot != l2.end())
    {
        // Push(ot) and increment it
    }
return _new;
}
```

// adjust function rearranges the heap so that heap is in increasing order of degree and no two binomial trees have same degree in this heap.

```
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((*it1)->degree == (*it2)->degree)
    {
        Node* temp;
        *it1= mergeBinomialTrees(*it1,*it2);
        it2=_heap.erase(it2);
        if(it3!=_heap.end())
            it3++;
    }
}
return _heap;
}

//Function to insert binomial tree into binomial heap
list<Node*> insertATreeHeap(list<Node*> _heap, Node *tree)
{
    list<Node*> temp;
    temp.push_back(tree);
    temp= unionBinomialHeap(_heap,temp);
    return adjust(temp);
}
    toreturn
// function pointer of minimum value node
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 = removeMinFromTreeReturnHeap(temp);
    new_heap = unionBinomialHeap(new_heap, lo);
    new_heap = adjust(new_heap);
    return new_heap;
}
```