

Program-09

```
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 = t;
    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*> 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
```

```
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 *temp;

```

```

    *it1 = mergeBinomialTrees(*it1, *it2);

```

```

    it2 = -heap.erase(it2);

```

```

    if (it3 != -heap.end())

```

```

        it3++;

```

```

}

```

```

}

```

```

return -heap;

```

```

}

```

```

list<Node*> insertATreeInHeap(list<Node*> -heap,
                               Node *tree)
{

```

```

    list<Node*> temp;

```

```

    temp.push_back(tree);

```

```

    temp = UnionBinomialHeap(-heap, temp);

```

```

    return adjust(temp);

```

```

}

```



```
list<Node*> removeMinFromTree ReturnBHeap  
(Node* tree)
```

```
{  
    list<Node*> heap;  
    Node* temp = tree->child;  
    Node* do;
```

```
    While(temp)
```

```
{  
    do = temp;  
    temp = temp->sibling;  
    do->sibling = NULL;  
    heap.push-front(do);
```

```
}
```

```
    return heap;
```

```
}
```

```
list<Node*> insert(list<Node*> head, int key)
```

```
{
```

```
    Node* temp = newNode(key);
```

```
    return insertA tree in heap(-head, 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, do;
    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++;
    }
    do = removeMinFromTreeReturnBHeap(temp);
    new-heap = UnionBionomialHeap(new-heap, do);
    new-heap = adjust(new-heap);
    return new-heap;
}

```

```

void printTree (Node *h)

```

```

{
    while (h)
    {
        cout << h->data << " ";
        printTree(h->child);
        h = h->Sibling;
    }
}

```



```
void printHeap (list<Node*> _heap)
{
    list<Node*> :: iterator it;
    it = _heap.begin();
    while (it != _heap.end())
    {
        printTree(*it);
        it++;
    }
}
```