```
struct Node{
    int data degree;
    Node * child, *sibling, *parent;
};

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) += 1;
    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.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 == (*it2)->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 *> 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 *> insert (list <Node *> head, int key) {
    Node * temp = newNode (key);
    return insertATreeInHeap (head, 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.pushback(*it);
        }
        it ++;
    }
    lo = removeMinFromTreeReturnBHeap(temp);
    new_heap = unionBinomialHeap (new_heap, lo);
    new_heap = adjust (new_heap);
    return new_heap;
}
```