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

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* 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*> unionBionomialHeap(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 == (*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 = unionBionomialHeap(_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;
}

list<Node*> insert(list<Node*> _head, int key) {
    Node *temp = newNode(key);
    return insertATreeInHeap(_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,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 = unionBionomialHeap(new_heap,lo);
    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++;
    }
}


int main() {
    int ch,key;
    list<Node*> _heap;
    int n;
    cin >> n;
    while(n--) {
      int k;
      cin >> k;
      _heap = insert(_heap, k);
    }

    cout << "Heap elements after insertion:\n";
    printHeap(_heap);

    Node *temp = getMin(_heap);
    cout << "\nMinimum element of heap " << temp -> data << "\n";

    _heap = extractMin(_heap);
    cout << "Heap after deletion of minimum element\n";
    printHeap(_heap);
    return 0;
}
```

OUTPUT:

```
5
1
2
3
4
5
Heap elements after insertion:
5 1 3 4 2
Minimum element of heap 1
Heap after deletion of minimum element
2 3 4 5
```