2

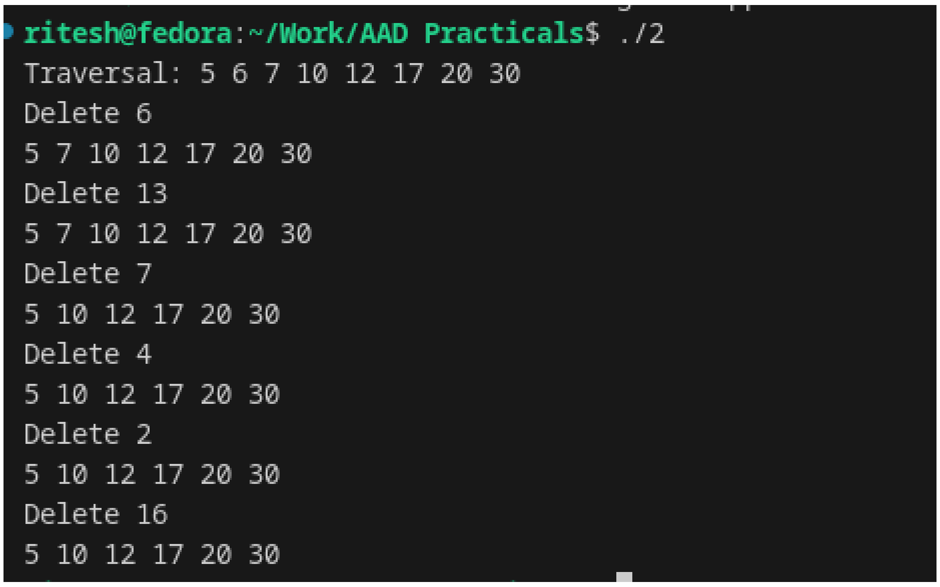
B-Tree Implementation

Code:

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

class BTreeNode {
public:
    vector<int> keys;
    vector<BTreeNode*> C;
    bool leaf;
    int m, t; // m = order, t = ceil(m/2)

    BTreeNode(int _m, bool _leaf): m(_m), leaf(_leaf) {
        t = (m+1)/2;
    }

    BTreeNode* search(int k) {
        int i = 0;
        while (i < (int)keys.size() && k > keys[i]) i++;
        if (i < (int)keys.size() && keys[i] == k) return this;
        if (leaf) return nullptr;
        return C[i]->search(k);
    }

    void traverse() {
        int i;
        for (i = 0; i < (int)keys.size(); i++) {
            if (!leaf) C[i]->traverse();
            cout << keys[i] << " ";
        }
        if (!leaf) C[i]->traverse();
    }

    void splitChild(int i) {
        BTreeNode* y = C[i];
        BTreeNode* z = new BTreeNode(m, y->leaf);
        int mid = t-1;

        z->keys.assign(y->keys.begin()+mid+1, y->keys.end());
        if (!y->leaf) z->C.assign(y->C.begin()+mid+1, y->C.end());

        int upKey = y->keys[mid];
```

```cpp
        y->keys.resize(mid);
        if (!y->leaf) y->C.resize(mid+1);

        C.insert(C.begin()+i+1, z);
        keys.insert(keys.begin()+i, upKey);
    }

    void insertNonFull(int k) {
        int i = keys.size()-1;
        if (leaf) {
            keys.push_back(0);
            while (i >= 0 && keys[i] > k) { keys[i+1] = keys[i]; i--; }
            keys[i+1] = k;
        } else {
            while (i >= 0 && keys[i] > k) i--;
            i++;
            if ((int)C[i]->keys.size() == m-1) {
                splitChild(i);
                if (keys[i] < k) i++;
            }
            C[i]->insertNonFull(k);
        }
    }

    int findKey(int k) { int idx=0; while (idx<(int)keys.size() &&
keys[idx]<k) idx++; return idx; }

    void remove(int k);
    void removeFromLeaf(int idx) { keys.erase(keys.begin()+idx); }
    void removeFromNonLeaf(int idx);
    int getPred(int idx){ BTreeNode* cur=C[idx]; while(!cur->leaf)
cur=cur->C.back(); return cur->keys.back(); }
    int getSucc(int idx){ BTreeNode* cur=C[idx+1]; while(!cur->leaf)
cur=cur->C.front(); return cur->keys.front(); }
    void fill(int idx);
    void borrowFromPrev(int idx);
    void borrowFromNext(int idx);
    void merge(int idx);
};

class BTree {
public:
    BTreeNode* root; int m,t;
    BTree(int _m): root(nullptr), m(_m), t((_m+1)/2) {}

    void traverse() { if(root) root->traverse(); cout<<"\n"; }
    BTreeNode* search(int k) { return root? root->search(k):nullptr; }

    void insert(int k) {
        if(!root){ root=new BTreeNode(m,true); root->keys.push_back(k);
return; }
        if((int)root->keys.size()==m-1){
            BTreeNode* s=new BTreeNode(m,false);
            s->C.push_back(root);
            s->splitChild(0);
            int i= (s->keys[0]<k); s->C[i]->insertNonFull(k);
            root=s;
        } else root->insertNonFull(k);
    }

    void remove(int k) {
        if(!root) return;
        root->remove(k);
        if(root->keys.empty()){
            BTreeNode* tmp=root;
            root= root->leaf? nullptr: root->C[0];
            delete tmp;
        }
    }
};

// ---- Delete helpers ----
void BTreeNode::remove(int k) {
    int idx=findKey(k);
    if(idx<(int)keys.size() && keys[idx]==k){
        if(leaf) removeFromLeaf(idx);
        else removeFromNonLeaf(idx);
    } else {
        if(leaf) return;
        bool flag=(idx==(int)keys.size());
        if((int)C[idx]->keys.size()<t) fill(idx);
        if(flag && idx>(int)keys.size()) C[idx-1]->remove(k);
        else C[idx]->remove(k);
    }
}
void BTreeNode::removeFromNonLeaf(int idx) {
    int k=keys[idx];
    if((int)C[idx]->keys.size()>=t){
        int pred=getPred(idx); keys[idx]=pred; C[idx]->remove(pred);
    } else if((int)C[idx+1]->keys.size()>=t){
        int succ=getSucc(idx); keys[idx]=succ; C[idx+1]->remove(succ);
    } else { merge(idx); C[idx]->remove(k); }
}
void BTreeNode::fill(int idx) {
    if(idx!=0 && (int)C[idx-1]->keys.size()>=t) borrowFromPrev(idx);
    else if(idx!=(int)keys.size() && (int)C[idx+1]->keys.size()>=t)
borrowFromNext(idx);
    else { if(idx!=(int)keys.size()) merge(idx); else merge(idx-1); }
}
void BTreeNode::borrowFromPrev(int idx) {
    BTreeNode* child=C[idx]; BTreeNode* sib=C[idx-1];
    child->keys.insert(child->keys.begin(), keys[idx-1]);
    if(!child->leaf){ child->C.insert(child->C.begin(), sib->C.back());
sib->C.pop_back();}
    keys[idx-1]=sib->keys.back(); sib->keys.pop_back();
}
void BTreeNode::borrowFromNext(int idx) {
    BTreeNode* child=C[idx]; BTreeNode* sib=C[idx+1];
    child->keys.push_back(keys[idx]);
    if(!child->leaf){ child->C.push_back(sib->C.front()); sib-
>C.erase(sib->C.begin()); }
    keys[idx]=sib->keys.front(); sib->keys.erase(sib->keys.begin());
}
void BTreeNode::merge(int idx) {
    BTreeNode* c=C[idx]; BTreeNode* s=C[idx+1];
    c->keys.push_back(keys[idx]);
    c->keys.insert(c->keys.end(), s->keys.begin(), s->keys.end());
    if(!c->leaf) c->C.insert(c->C.end(), s->C.begin(), s->C.end());
    keys.erase(keys.begin()+idx); C.erase(C.begin()+idx+1);
    delete s;
}

// ---- Demo ----
int main() {
    BTree t(3); // order = 3
    t.insert(10); t.insert(20); t.insert(5); t.insert(6); t.insert(12);
    t.insert(30); t.insert(7); t.insert(17);

    cout<<"Traversal: "; t.traverse();
    cout<<"Delete 6\n"; t.remove(6); t.traverse();
    cout<<"Delete 13\n"; t.remove(13); t.traverse();
    cout<<"Delete 7\n"; t.remove(7); t.traverse();
    cout<<"Delete 4\n"; t.remove(4); t.traverse();
    cout<<"Delete 2\n"; t.remove(2); t.traverse();
    cout<<"Delete 16\n"; t.remove(16); t.traverse();
}
```

Output:



```
ritesh@fedora:~/Work/AAD Practicals$ ./2
Traversal: 5 6 7 10 12 17 20 30
Delete 6
5 7 10 12 17 20 30
Delete 13
5 7 10 12 17 20 30
Delete 7
5 10 12 17 20 30
Delete 4
5 10 12 17 20 30
Delete 2
5 10 12 17 20 30
Delete 16
5 10 12 17 20 30
```