Week 9-LAB B

1.

```
#include <iostream> #include <vector> #include <algorithm> using namespace std;

class BTreeNode { public:

vector<int> keys; vector<BTreeNode*> children; bool leaf;
int minDegree;

BTreeNode(int degree, bool leaf);

void traverseInOrder();
int findKey(int key);
void insertNonFull(int key);
void splitChild(int i, BTreeNode *y); void deleteKey(int key);

void removeFromLeaf(int idx); void removeFromNonLeaf(int idx); int getPredecessor(int idx);
int getSuccessor(int idx);
void fill(int idx);
void borrowFromPrev(int idx); void borrowFromNext(int idx); void merge(int idx);

friend class BTree; };

class BTree { public:

BTreeNode *root; int minDegree; int splitCount;
int mergeCount;

BTree(int degree) { root = nullptr; minDegree = degree; splitCount = 0; mergeCount = 0;

}

void insert(int key); void deleteKey(int key); void traverseInOrder();

};

BTreeNode::BTreeNode(int degree, bool isLeaf) { minDegree = degree;

leaf = isLeaf;
keys.resize(2 * minDegree - 1); children.resize(2 * minDegree);

}
```

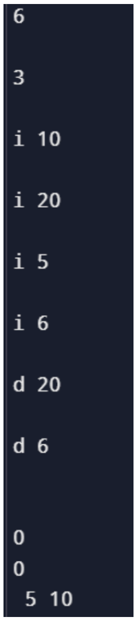
```
int BTreeNode::findKey(int key) {
intidx=0;
while (idx < keys.size() && keys[idx] < key)

++idx; return idx;

}

void BTreeNode::traverseInOrder() { int i;

for (i = 0; i < keys.size(); i++) { if (!leaf)

children[i]->traverseInOrder(); cout << " " << keys[i];

}
if (!leaf)

children[i]->traverseInOrder(); }

void BTree::traverseInOrder() { if (root != nullptr)

root->traverseInOrder(); }

void BTree::insert(int key) { if (root == nullptr) {

root = new BTreeNode(minDegree, true); root->keys[0] = key;
root->keys.resize(1);

}else{
if (root->keys.size() == 2 * minDegree - 1) {

BTreeNode *s = new BTreeNode(minDegree, false); s->children[0] = root;
s->splitChild(0, root);
root = s;

splitCount++;
int i = (root->keys[0] < key) ? 1 : 0; root->children[i]->insertNonFull(key);

}else{ root->insertNonFull(key);

} }

}

void BTreeNode::insertNonFull(int key) { int i = keys.size() - 1;
if (leaf) {

keys.resize(keys.size() + 1); while (i >= 0 && keys[i] > key) {
```

```
keys[i + 1] = keys[i];

i--; }

keys[i + 1] = key; }else{

while (i >= 0 && keys[i] > key) i--;

if (children[i + 1]->keys.size() == 2 * minDegree - 1) { splitChild(i + 1, children[i + 1]);
if (keys[i + 1] < key)

i++; }

children[i + 1]->insertNonFull(key); }

}

void BTreeNode::splitChild(int i, BTreeNode *y) { BTreeNode *z = new
BTreeNode(y->minDegree, y->leaf); z->keys.resize(minDegree - 1);

for(intj=0;j<minDegree-1;j++) z->keys[j] = y->keys[j + minDegree];

if (!y->leaf) { z->children.resize(minDegree); for (int j = 0; j < minDegree; j++)

z->children[j] = y->children[j + minDegree]; }

y->keys.resize(minDegree - 1); children.insert(children.begin() + i + 1, z);
keys.insert(keys.begin() + i, y->keys[minDegree - 1]);

}

void BTree::deleteKey(int key) { if (!root) {

return; }

root->deleteKey(key);
if (root->keys.size() == 0) { BTreeNode *tmp = root;

if (root->leaf) root = nullptr;

else
root = root->children[0];

delete tmp; }

}

void BTreeNode::deleteKey(int key) { int idx = findKey(key);
```

```
if (idx < keys.size() && keys[idx] == key) { if (leaf)

removeFromLeaf(idx); else

removeFromNonLeaf(idx); }else{

if (leaf) { return;

}

bool flag = (idx == keys.size());
if (children[idx]->keys.size() < minDegree)

fill(idx);

if (flag && idx > keys.size()) children[idx - 1]->deleteKey(key);

else children[idx]->deleteKey(key);

} }

void BTreeNode::removeFromLeaf(int idx) { keys.erase(keys.begin() + idx);

}

void BTreeNode::removeFromNonLeaf(int idx) { int key = keys[idx];

if (children[idx]->keys.size() >= minDegree) { int pred = getPredecessor(idx);
keys[idx] = pred; children[idx]->deleteKey(pred);

} else if (children[idx + 1]->keys.size() >= minDegree) { int succ = getSuccessor(idx);
keys[idx] = succ;
children[idx + 1]->deleteKey(succ);

}else{
merge(idx); children[idx]->deleteKey(key);

} }

int BTreeNode::getPredecessor(int idx) { BTreeNode *cur = children[idx];
while (!cur->leaf)

cur = cur->children[cur->keys.size()]; return cur->keys[cur->keys.size() - 1];

}

int BTreeNode::getSuccessor(int idx) { BTreeNode *cur = children[idx + 1]; while (!cur->leaf)
```

```
cur = cur->children[0]; return cur->keys[0];

}

void BTreeNode::fill(int idx) {
if (idx != 0 && children[idx - 1]->keys.size() >= minDegree)

borrowFromPrev(idx);
else if (idx != keys.size() && children[idx + 1]->keys.size() >= minDegree)

borrowFromNext(idx); else {

if (idx != keys.size()) merge(idx);

else
merge(idx - 1);

} }

void BTreeNode::borrowFromPrev(int idx) { BTreeNode *child = children[idx]; BTreeNode
*sibling = children[idx - 1];

child->keys.insert(child->keys.begin(), keys[idx - 1]); if (!child->leaf)

child->children.insert(child->children.begin(), sibling->children[sibling->keys.size()]);

keys[idx - 1] = sibling->keys[sibling->keys.size() - 1];

sibling->keys.pop_back(); }

void BTreeNode::borrowFromNext(int idx) { BTreeNode *child = children[idx]; BTreeNode
*sibling = children[idx + 1];

child->keys.push_back(keys[idx]);

if (!child->leaf) child->children.push_back(sibling->children[0]);

keys[idx] = sibling->keys[0];

sibling->keys.erase(sibling->keys.begin()); }

void BTreeNode::merge(int idx) { BTreeNode *child = children[idx]; BTreeNode *sibling =
children[idx + 1];

child->keys.push_back(keys[idx]);
for (int i = 0; i < sibling->keys.size(); i++)

child->keys.push_back(sibling->keys[i]);
```

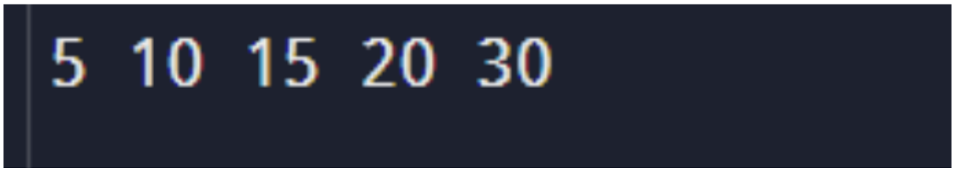
```
if (!child->leaf) {
for (int i = 0; i <= sibling->keys.size(); i++)

child->children.push_back(sibling->children[i]); }

keys.erase(keys.begin() + idx); children.erase(children.begin() + idx + 1);

delete sibling; }

int main() {
int NQ, degree;
cin >> NQ >> degree;

BTree btree(degree); char op;
int val;

while (NQ--) {
cin >> op >> val; if(op=='i'){

btree.insert(val); }elseif(op=='d'){

btree.deleteKey(val); }

}

cout << btree.splitCount << endl; cout << btree.mergeCount << endl; btree.traverseInOrder();
cout << endl;

return 0; }
```



```
6

3

i 10

i 20

i 5

i 6

d 20

d 6


0
0
 5 10
```

2.

```
 #include <iostream> using namespace std;

struct ThreadedNode { int data;

ThreadedNode* left;
ThreadedNode* right;
bool rightThread; // true if right pointer is a thread

ThreadedNode(int data) { this->data = data;
left = right = nullptr; rightThread = false;

} };

class ThreadedBinaryTree { private:

ThreadedNode* root;

void createThreadedUtil(ThreadedNode* current, ThreadedNode*& prev) { if (current == nullptr)

return;

createThreadedUtil(current->right, prev);

if (current->right == nullptr && prev != nullptr) { current->right = prev;
current->rightThread = true;

}

prev = current;

createThreadedUtil(current->left, prev); }

void inOrderTraversalUtil(ThreadedNode* node) { ThreadedNode* current = leftmost(node);
while (current != nullptr) {

cout << current->data << " "; if (current->rightThread)

current = current->right; else

current = leftmost(current->right); }

}

ThreadedNode* leftmost(ThreadedNode* node) { if (node == nullptr)

return nullptr;
while (node->left != nullptr) node = node->left;

return node; }

public: ThreadedBinaryTree() {

root = nullptr; }

void setRoot(ThreadedNode* node) { root = node;

}

void createThreaded() { ThreadedNode* prev = nullptr; createThreadedUtil(root, prev);

}

void inOrderTraversal() { inOrderTraversalUtil(root);

} };

int main() {
ThreadedBinaryTree tree;
ThreadedNode* root = new ThreadedNode(20); root->left = new ThreadedNode(10);
root->right = new ThreadedNode(30); root->left->left = new ThreadedNode(5); root->left->right =
new ThreadedNode(15);

tree.setRoot(root); tree.createThreaded(); tree.inOrderTraversal();

return 0;

}
```



```
5 10 15 20 30
```