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


class BTreeNode
{
	int *keys;
	int t;
	BTreeNode **C;
	int curr_n;
	bool leaf;

public:

	BTreeNode(int _t, bool _leaf);
	void PRINT();
	BTreeNode *SEARCH(int k);

	int findKey(int k);

	void insertNonFull(int k);

	void splitChild(int i, BTreeNode *y);

	void DELETE(int k);

	void removeFromLeaf(int idx);

	void removeFromNonLeaf(int idx);

	int Predecessor(int idx);

	int Successor(int idx);

	void fill(int idx);

	void BorrowLeft(int idx);

	void BorrowRight(int idx);

	void MERGE(int idx);

	friend class BTree;
};

class BTree
{
	BTreeNode *root;
	int t;
public:

	BTree(int _t)
	{
		root = NULL;
		t = _t;
	}

	void PRINT()
	{
		if (root != NULL) root->PRINT();
	}


	BTreeNode* SEARCH(int k)
	{
		return (root == NULL)? NULL : root->SEARCH(k);
	}

	void INSERT(int k);

	void DELETE(int k);

};

BTreeNode::BTreeNode(int t1, bool leaf1)
{

	t = t1;
	leaf = leaf1;


	keys = new int[2*t-1];
	C = new BTreeNode *[2*t];

	curr_n = 0;
}
```

```
int BTreeNode::findKey(int k)
{
        int idx=0;
        while (idx<curr_n && keys[idx] < k)
                ++idx;
        return idx;
}


void BTreeNode::DELETE(int k)
{
        int idx = findKey(k);

        if (idx < curr_n && keys[idx] == k)
        {

                if (leaf)
                        removeFromLeaf(idx);
                else
                        removeFromNonLeaf(idx);
        }
        else
        {

                if (leaf)
                {
                        cout << k <<" does not exist\n";
                        return;
                }


                bool flag = ( (idx==curr_n)? true : false );


                if (C[idx]->curr_n < t)
                        fill(idx);


                if (flag && idx > curr_n)
                        C[idx-1]->DELETE(k);
                else
                        C[idx]->DELETE(k);
        }
        return;
}

void BTreeNode::removeFromLeaf (int idx)
{


        for (int i=idx+1; i<curr_n; ++i)
                keys[i-1] = keys[i];

        curr_n--;

        return;
}


void BTreeNode::removeFromNonLeaf(int idx)
{

        int k = keys[idx];

        if (C[idx]->curr_n >= t)
        {
                int pred = Predecessor(idx);
                keys[idx] = pred;
                C[idx]->DELETE(pred);
        }

        else if (C[idx+1]->curr_n >= t)
        {
                int succ = Successor(idx);
                keys[idx] = succ;
                C[idx+1]->DELETE(succ);
        }

        else
        {
                MERGE(idx);
                C[idx]->DELETE(k);
        }
        return;
}


int BTreeNode::Predecessor(int idx)
```

```
{

	BTreeNode *cur=C[idx];
	while (!cur->leaf)
		cur = cur->C[cur->curr_n];


	return cur->keys[cur->curr_n-1];
}

int BTreeNode::Successor(int idx)
{


	BTreeNode *cur = C[idx+1];
	while (!cur->leaf)
		cur = cur->C[0];

	return cur->keys[0];
}

void BTreeNode::fill(int idx)
{

	if (idx!=0 && C[idx-1]->curr_n>=t)
		BorrowLeft(idx);

	else if (idx!=curr_n && C[idx+1]->curr_n>=t)
		BorrowRight(idx);


	else
	{
		if (idx != curr_n)
			MERGE(idx);
		else
			MERGE(idx-1);
	}
	return;
}


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


        for (int i=child->curr_n-1; i>=0; --i)
                child->keys[i+1] = child->keys[i];

        if (!child->leaf)
        {
                for(int i=child->curr_n; i>=0; --i)
                        child->C[i+1] = child->C[i];
        }


        child->keys[0] = keys[idx-1];


        if(!child->leaf)
                child->C[0] = sibling->C[sibling->curr_n];


        keys[idx-1] = sibling->keys[sibling->curr_n-1];

        child->curr_n += 1;
        sibling->curr_n -= 1;

        return;
}

void BTreeNode::BorrowRight(int idx)
{

        BTreeNode *child=C[idx];
        BTreeNode *sibling=C[idx+1];


        child->keys[(child->curr_n)] = keys[idx];


        if (!(child->leaf))
                child->C[(child->curr_n)+1] = sibling->C[0];
        keys[idx] = sibling->keys[0];

        for (int i=1; i<sibling->curr_n; ++i)
                sibling->keys[i-1] = sibling->keys[i];


        if (!sibling->leaf)
        {
                for(int i=1; i<=sibling->curr_n; ++i)
                        sibling->C[i-1] = sibling->C[i];
        }

        child->curr_n += 1;
        sibling->curr_n -= 1;

        return;
}

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


        child->keys[t-1] = keys[idx];


        for (int i=0; i<sibling->curr_n; ++i)
                child->keys[i+t] = sibling->keys[i];

        if (!child->leaf)
        {
                for(int i=0; i<=sibling->curr_n; ++i)
                        child->C[i+t] = sibling->C[i];
        }

        for (int i=idx+1; i<curr_n; ++i)
                keys[i-1] = keys[i];


        for (int i=idx+2; i<=curr_n; ++i)
                C[i-1] = C[i];
        child->curr_n += sibling->curr_n+1;
        curr_n--;

        delete(sibling);
        return;
}


void BTree::INSERT(int k)
{

        if (root == NULL)
        {

                root = new BTreeNode(t, true);
                root->keys[0] = k;
                root->curr_n = 1;
        }
        else
        {

                if (root->curr_n == 2*t-1)
                {

                        BTreeNode *s = new BTreeNode(t, false);


                        s->C[0] = root;

                        s->splitChild(0, root);


                        int i = 0;
                        if (s->keys[0] < k)
                                i++;
                        s->C[i]->insertNonFull(k);


                        root = s;
                }
                else
                        root->insertNonFull(k);
        }
}
```

```
void BTreeNode::insertNonFull(int k)
{

    int i = curr_n-1;


    if (leaf == true)
    {

        while (i >= 0 && keys[i] > k)
        {
            keys[i+1] = keys[i];
            i--;
        }

        keys[i+1] = k;
        curr_n = curr_n+1;
    }
    else
    {

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


        if (C[i+1]->curr_n == 2*t-1)
        {

            splitChild(i+1, C[i+1]);

            if (keys[i+1] < k)
                i++;
        }
        C[i+1]->insertNonFull(k);
    }
}

void BTreeNode::splitChild(int i, BTreeNode *y)
{

    BTreeNode *z = new BTreeNode(y->t, y->leaf);
    z->curr_n = t - 1;
```

```
        for (int j = 0; j < t-1; j++)
                z->keys[j] = y->keys[j+t];


        if (y->leaf == false)
        {
                for (int j = 0; j < t; j++)
                        z->C[j] = y->C[j+t];
        }


        y->curr_n = t - 1;

        for (int j = curr_n; j >= i+1; j--)
                C[j+1] = C[j];

        C[i+1] = z;


        for (int j = curr_n-1; j >= i; j--)
                keys[j+1] = keys[j];


        keys[i] = y->keys[t-1];

        curr_n = curr_n + 1;
}

void BTreeNode::PRINT()
{
        int i;
        for (i = 0; i < curr_n; i++)
        {
                if (leaf == false)
                        C[i]->PRINT();
                cout << keys[i]<<" ";
        }

        if (leaf == false)
                C[i]->PRINT();
}
```

```
BTreeNode *BTreeNode::SEARCH(int k)
{

        int i = 0;
        while (i < curr_n && k > keys[i])
                i++;


        if (keys[i] == k)
                return this;


        if (leaf == true)
                return NULL;


        return C[i]->SEARCH(k);
}



void BTree::DELETE(int k)
{
        if (!root)
        {
                cout << "The tree is empty\n";
                return;
        }


        root->DELETE(k);

        if (root->curr_n==0)
        {
                BTreeNode *tmp = root;
                if (root->leaf)
                        root = NULL;
                else
                        root = root->C[0];


                delete tmp;
        }
        return;
```

```
}

int main()
{
        int ord;
        char op;
        int key_count;
        int ins;

        cout<<"Enter the order of the tree: ";
        cin>>ord;
        //cout<<int(ord/2);
        BTree t(int(ord/2));
        cout<<"Number of keys to be entered: ";
        cin>>key_count;
        cout<<"Enter the keys: ";
        for(int i=1;i<=key_count;i++)
        {
                cin>>ins;
                t.INSERT(ins);
        }
        cout<<"MENU\n";
        cout<<"i:INSERT\n";
        cout<<"d:DELETE\n";
        cout<<"s:SEARCH\n";
        cout<<"p:PRINT\n";
        cout<<"e:EXIT\n";
        cout<<"Enter your option: ";
        cin>>op;

        while(op!='e')
        {
                switch(op)
                {
                        case 'i':cout<<"Enter the roll number to be inserted: ";

                                cin>>ins;
                                t.INSERT(ins);
                                break;
                        case 'd':       cout<<"Enter roll number to be removed: ";

                                cin>>ins;
                                t.DELETE(ins);
                                break;
                case 'p':       t.PRINT();
                                cout<<endl;
                                break;
                case 's':       cout<<"Enter roll number to be searched: ";

                                cin>>ins;
                                if(t.SEARCH(ins)==NULL)
                                        cout<<"FALSE\n";
                                else
                                        cout<<"TRUE\n";

                                break;

        }
        cout<<"Enter your option: ";
        cin>>op;
    }


    return 0;
}
```