1.
code.

```
#include <iostream>
#include <vector>
#include <algorithm>

class BPlusTreeNode {
public:
    bool leaf;
    std::vector<int> keys;
    std::vector<BPlusTreeNode*> children;

    BPlusTreeNode(bool isLeaf) : leaf(isLeaf) {}
};

class BPlusTree {
private:
    BPlusTreeNode* root;
    int order;

    void splitChild(BPlusTreeNode* parent, int index);
    void insertNonFull(BPlusTreeNode* node, int key);
    void printTree(BPlusTreeNode* node, int level);
    int findKey(BPlusTreeNode* node, int key);
    void deleteNode(BPlusTreeNode* node, int key);
    void deleteInternalNode(BPlusTreeNode* node, int index);
    int getPredecessor(BPlusTreeNode* node);
    int getSuccessor(BPlusTreeNode* node);
    void fill(BPlusTreeNode* node, int index);
    void borrowFromPrev(BPlusTreeNode* node, int index);
    void borrowFromNext(BPlusTreeNode* node, int index);
    void merge(BPlusTreeNode* node, int index);

public:
    BPlusTree(int order) : root(new BPlusTreeNode(true)), order(order) {}

    void insert(int key);
    void print();
    void deleteKey(int key);
};

void BPlusTree::splitChild(BPlusTreeNode* parent, int index) {
    BPlusTreeNode* nodeToSplit = parent->children[index];
```

```
    BPlusTreeNode* newNode = new
    BPlusTreeNode(nodeToSplit->leaf);
    int midIndex = nodeToSplit->keys.size() / 2;
    int midKey = nodeToSplit->keys[midIndex];

    parent->keys.insert(parent->keys.begin() + index, midKey);
    parent->children.insert(parent->children.begin() + index + 1, newNode);

    newNode->keys.assign(nodeToSplit->keys.begin() + midIndex + 1, nodeToSplit->keys.end());
    nodeToSplit->keys.resize(midIndex);

    if (!nodeToSplit->leaf) {
        newNode->children.assign(nodeToSplit->children.begin() + midIndex + 1,
nodeToSplit->children.end());
        nodeToSplit->children.resize(midIndex + 1);
    }
}

void BPlusTree::insertNonFull(BPlusTreeNode* node, int key) {
    if (node->leaf) {
        node->keys.push_back(key);
        std::sort(node->keys.begin(), node->keys.end());
    } else {
        int index = node->keys.size() - 1;
        while (index >= 0 && key < node->keys[index]) {
            index--;
        }
        index++;

        if (node->children[index]->keys.size() == order - 1) {
            splitChild(node, index);
            if (key > node->keys[index]) {
                index++;
            }
        }
        insertNonFull(node->children[index], key);
    }
}

void BPlusTree::insert(int key) {
    if (root->keys.size() == order - 1) {
        BPlusTreeNode* newRoot = new BPlusTreeNode(false);
        newRoot->children.push_back(root);
```

```
        splitChild(newRoot, 0);
        root = newRoot;
        insertNonFull(newRoot, key);
    } else {
        insertNonFull(root, key);
    }
}

void BPlusTree::printTree(BPlusTreeNode* node, int level) {
    std::cout << "Level " << level << ": ";
    for (int key : node->keys) {
        std::cout << key << " ";
    }
    std::cout << std::endl;
    if (!node->leaf) {
        for (auto child : node->children) {
            printTree(child, level + 1);
        }
    }
}

void BPlusTree::print() {
    printTree(root, 0);
}

void BPlusTree::deleteKey(int key) {
    deleteNode(root, key);
    if (root->keys.empty()) {
        BPlusTreeNode* oldRoot = root;
        root = root->children[0];
        delete oldRoot;
    }
}

void BPlusTree::deleteNode(BPlusTreeNode* node, int key) {
    int index = findKey(node, key);

    if (index < node->keys.size() && node->keys[index] == key) {
        if (node->leaf) {
            node->keys.erase(node->keys.begin() + index);
        } else {
            deleteInternalNode(node, index);
        }
```

```
    } else {
        if (node->leaf) {
            std::cout << "Key not found: " << key << std::endl;
            return;
        }

        if (node->children[index]->keys.size() < (order / 2)) {
            fill(node, index);
        }

        if (index >= node->keys.size()) {
            index--;
        }
        deleteNode(node->children[index], key);
    }
}

int BPlusTree::findKey(BPlusTreeNode* node, int key) {
    for (int i = 0; i < node->keys.size(); i++) {
        if (key < node->keys[i]) {
            return i;
        }
    }
    return node->keys.size();
}

void BPlusTree::deleteInternalNode(BPlusTreeNode* node, int index) {
    int key = node->keys[index];
    if (node->children[index]->keys.size() >= (order / 2)) { int
        predecessor = getPredecessor(node->children[index]);
        node->keys[index] = predecessor;
        deleteNode(node->children[index], predecessor);
    } else if (node->children[index + 1]->keys.size() >= (order / 2)) {
        int successor = getSuccessor(node->children[index + 1]);
        node->keys[index] = successor;
        deleteNode(node->children[index + 1], successor);
    } else {
        merge(node, index);
    }
}

int BPlusTree::getPredecessor(BPlusTreeNode* node) {
    while (!node->leaf) {
```

```
        node = node->children.back();
    }
    return node->keys.back();
}
int BPlusTree::getSuccessor(BPlusTreeNode* node) {
    while (!node->leaf) {
        node = node->children.front();
    }
    return node->keys.front();
}

void BPlusTree::fill(BPlusTreeNode* node, int index) {
    if (index != 0 && node->children[index - 1]->keys.size() >= (order / 2)) {
        borrowFromPrev(node, index);
    } else if (index != node->keys.size() && node->children[index + 1]->keys.size() >= (order / 2))
{
        borrowFromNext(node, index);
    } else {
        if (index != node->keys.size()) {
            merge(node, index);
        } else {
            merge(node, index - 1);
        }
    }
}

void BPlusTree::borrowFromPrev(BPlusTreeNode* node, int index) {
    BPlusTreeNode* child = node->children[index];
    BPlusTreeNode* sibling = node->children[index - 1];

    child->keys.insert(child->keys.begin(), node->keys[index - 1]);
    node->keys[index - 1] = sibling->keys.back();
    sibling->keys.pop_back();

    if (!child->leaf) {
        child->children.insert(child->children.begin(), sibling->children.back());
        sibling->children.pop_back();
    }
}

void BPlusTree::borrowFromNext(BPlusTreeNode* node, int index) {
    BPlusTreeNode* child = node->children[index];
    BPlusTreeNode* sibling = node->children[index + 1];
```

```
    child->keys.push_back(node->keys[index]);
    node->keys[index] = sibling->keys.front();
    sibling->keys.erase(sibling->keys.begin());
    if (!child->leaf) {
        child->children.push_back(sibling->children.front());
        sibling->children.erase(sibling->children.begin());
    }
}

void BPlusTree::merge(BPlusTreeNode* node, int index) {
    BPlusTreeNode* child = node->children[index];
    BPlusTreeNode* sibling = node->children[index + 1];

    child->keys.push_back(node->keys[index]);
    child->keys.insert(child->keys.end(), sibling->keys.begin(), sibling->keys.end());

    if (!child->leaf) {
        child->children.insert(child->children.end(), sibling->children.begin(),
sibling->children.end());
    }

    node->keys.erase(node->keys.begin() + index);
    node->children.erase(node->children.begin() + index + 1);
}

int main() {
    BPlusTree bpt(3);

    std::vector<int> keysToInsert = {1, 3, 5, 7, 9, 2, 4, 6, 8, 10};
    for (int key : keysToInsert) {
        bpt.insert(key);
    }

    std::cout << "B+ Tree after insertion:" << std::endl;
    bpt.print();

    std::vector<int> keysToDelete = {9, 7, 8};
    for (int key : keysToDelete) {
        std::cout << "\nDeleting key: " << key << std::endl;
        bpt.deleteKey(key);
        bpt.print();
    }
```

```
    return 0;
}
```

Output

```
B+ Tree after insertion:
Level 0: 7
Level 1: 3 5
Level 2: 1 2
Level 2: 4
Level 2: 6
Level 1: 9
Level 2: 8
Level 2: 10

Deleting key: 9
Key not found: 9
Level 0: 7
Level 1: 3 5
Level 2: 1 2
Level 2: 4
Level 2: 6
Level 1: 9
Level 2: 8
Level 2: 10
```

```
Deleting key: 7
Key not found: 7
Level 0: 7
Level 1: 3 5
Level 2: 1 2
Level 2: 4
Level 2: 6
Level 1: 9
Level 2: 8
Level 2: 10

Deleting key: 8
Key not found: 8
Level 0: 7
Level 1: 3 5
Level 2: 1 2
Level 2: 4
Level 2: 6
Level 1: 9
Level 2: 8
Level 2: 10
```

Ques 2

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

struct BPTreeNode {
    int *keys;
    int order;
    BPTreeNode **children;
    bool isLeaf;
    int keyCount;

    BPTreeNode(int order, bool isLeaf) {
        this->order = order;
        this->isLeaf = isLeaf;
        keys = new int[order - 1];
        children = new BPTreeNode *[order];
        keyCount = 0;
    }
};
```

```
BPTreeNode* search(BPTreeNode *root, int k) {
    if (!root) return nullptr;

    int i = 0;
    while (i < root->keyCount && k > root->keys[i]) {
        i++;
    }

    if (i < root->keyCount && root->keys[i] == k && root->isLeaf) {
        return root;
    }

    if (root->isLeaf) {
        return nullptr;
    }

    return search(root->children[i], k);
}

BPTreeNode* createTree() {
    BPTreeNode *root = new BPTreeNode(3, false);
    BPTreeNode *child1 = new BPTreeNode(3, true);
    BPTreeNode *child2 = new BPTreeNode(3, true);
    BPTreeNode *child3 = new BPTreeNode(3, true);

    root->keys[0] = 25;
    root->keyCount = 1;
    root->children[0] = child1;
    root->children[1] = child2;

    child1->keys[0] = 5;
    child1->keys[1] = 15;
    child1->keys[2] = 20;
    child1->keyCount = 3;

    child2->keys[0] = 25;
    child2->keys[1] = 30;
    child2->keyCount = 2;

    child3->keys[0] = 35;
    child3->keys[1] = 40;
    child3->keys[2] = 45;
    child3->keyCount = 3;
```

```
    return root;
}

int main() {
    int key = 45;
    BPTreeNode *root = createTree();
    BPTreeNode *result = search(root, key);

    if (result != nullptr) {
        cout << "Key " << key << " found in B+ Tree." << endl;
    } else {
        cout << "Key " << key << " not found in B+ Tree." << endl;
    }

    return 0;
}
```

Output:

```
Output
/tmp/SQywYNBoP1.o
Key 45 not found in B+ Tree.


=== Code Execution Successful ===
```

Ques 3,

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

struct BPTreeNode {
    int *keys;
    int order;
    BPTreeNode **children;
    bool isLeaf;
    int keyCount;

    BPTreeNode(int order, bool isLeaf) {
        this->order = order;
        this->isLeaf = isLeaf;
```

```
        keys = new int[order - 1];
        children = new BPTreeNode *[order];
        keyCount = 0;
    }
};

BPTreeNode* search(BPTreeNode *root, int k) {
    if (!root) return nullptr;

    int i = 0;
    while (i < root->keyCount && k > root->keys[i]) {
        i++;
    }

    if (i < root->keyCount && root->keys[i] == k && root->isLeaf) {
        return root;
    }

    if (root->isLeaf) {
        return nullptr;
    }

    return search(root->children[i], k);
}

void deleteKey(BPTreeNode *node, int key) {
    if (!node) return;

    int i;
    for (i = 0; i < node->keyCount; i++) {
        if (node->keys[i] == key) {
            break;
        }
    }

    if (i < node->keyCount) {
        for (int j = i; j < node->keyCount - 1; j++) {
            node->keys[j] = node->keys[j + 1];
        }
        node->keyCount--;
    }
}
```

```
BPTreeNode* createTree() {
    BPTreeNode *root = new BPTreeNode(3, false);
    BPTreeNode *child1 = new BPTreeNode(3, true);
    BPTreeNode *child2 = new BPTreeNode(3, true);
    BPTreeNode *child3 = new BPTreeNode(3, true);

    root->keys[0] = 25;
    root->keyCount = 1;
    root->children[0] = child1;
    root->children[1] = child2;

    child1->keys[0] = 5;
    child1->keys[1] = 15;
    child1->keys[2] = 20;
    child1->keyCount = 3;

    child2->keys[0] = 25;
    child2->keys[1] = 30;
    child2->keyCount = 2;

    child3->keys[0] = 35;
    child3->keys[1] = 40;
    child3->keys[2] = 45;
    child3->keyCount = 3;

    return root;
}

void printTree(BPTreeNode *root) {
    if (!root) return;

    for (int i = 0; i < root->keyCount; i++) {
        cout << root->keys[i] << " ";
    }
    cout << endl;

    if (!root->isLeaf) {
        for (int i = 0; i <= root->keyCount; i++) {
            printTree(root->children[i]);
        }
    }
}
```

```
int main() {
    int key1 = 35, key2 = 40;
    BPTreeNode *root = createTree();

    BPTreeNode *leafNode1 = search(root, key1);
    if (leafNode1) deleteKey(leafNode1, key1);

    BPTreeNode *leafNode2 = search(root, key2);
    if (leafNode2) deleteKey(leafNode2, key2);

    cout << "Updated B+ Tree after deleting keys 35 and 40:" << endl;
    printTree(root);

    return 0;
}
```

Output:

```
Updated B+ Tree after deleting keys 35 and 40:
25
5 15 20
25 30


--- Code Execution Successful ---
```