1.

```cpp
#include <iostream>
#include <map>
using namespace std; // Add this line to use the std namespace
// Function to print all key-value pairs in the dictionary
void printDictionary(const map<int, int>& dictionary) {
  cout << "Dictionary Contents (Sorted by Keys):" << endl;

  for (const auto& pair : dictionary) {
    cout << "Key: " << pair.first << ", Value: " << pair.second << endl;
  }
}

int main() {
  map<int, int> dictionary;

  int choice, key, value;

  while (true) {
    cout << "\n1. Insert" << endl;
    cout << "2. Delete" << endl;
    cout << "3. Print Dictionary" << endl;
    cout << "4. Quit" << endl;
    cout << "Enter your choice: ";
    cin >> choice;

    switch (choice) {
      case 1:
        cout << "Enter key (integer): ";
        cin >> key;
        cout << "Enter value (integer): ";
        cin >> value;
        dictionary[key] = value;
        cout << "Key '" << key << "' inserted successfully." << endl;
        break;
      case 2:
        cout << "Enter key to delete (integer): ";
        cin >> key;
        if (dictionary.find(key) != dictionary.end()) {
          dictionary.erase(key);
          cout << "Key '" << key << "' deleted successfully." << endl;
        } else {
          cout << "Key '" << key << "' not found. Cannot delete." << endl;
        }
        break;
      case 3:
        printDictionary(dictionary);
        break;
      case 4:
        return 0;
```

```
        default:
          cout << "Invalid choice. Please try again." << endl;
      }
    }

    return 0;
}
```

2.

```
#include <iostream>
#include <stack>

using namespace std; // Add this line to use the std namespace

// Define a structure for a binary tree node
struct Node {
    int data;
    Node* left;
    Node* right;
};

// Function to create a new node
Node* newNode(int data) {
    Node* node = new Node;
    node->data = data;
    node->left = nullptr;
    node->right = nullptr;
    return node;
}

// Stack structure for non-recursive traversal
struct StackNode {
    Node* data;
    StackNode* next;
};

struct Stack {
    StackNode* top;
};

// Function to create an empty stack
Stack* createStack() {
    Stack* stack = new Stack;
    stack->top = nullptr;
    return stack;
}

// Function to push a node onto the stack
void push(Stack* stack, Node* data) {
    StackNode* newNode = new StackNode;
    newNode->data = data;
```

```
    newNode->next = stack->top;
    stack->top = newNode;
}

// Function to pop a node from the stack
Node* pop(Stack* stack) {
    if (stack->top == nullptr)
        return nullptr;

    StackNode* temp = stack->top;
    stack->top = stack->top->next;
    Node* popped = temp->data;
    delete temp;
    return popped;
}

// Function to check if the stack is empty
bool isEmpty(Stack* stack) {
    return (stack->top == nullptr);
}

// Iterative in-order traversal
void iterativeInorder(Node* root) {
    if (root == nullptr)
        return;

    Stack* stack = createStack();
    Node* current = root;

    while (current || !isEmpty(stack)) {
        while (current) {
            push(stack, current);
            current = current->left;
        }
        current = pop(stack);
        cout << current->data << " ";
        current = current->right;
    }

    delete stack;
}

// Iterative pre-order traversal
void iterativePreorder(Node* root) {
    if (root == nullptr)
        return;

    Stack* stack = createStack();
    push(stack, root);

    while (!isEmpty(stack)) {
```

```
        Node* current = pop(stack);
        cout << current->data << " ";

        if (current->right)
            push(stack, current->right);
        if (current->left)
            push(stack, current->left);
    }

    delete stack;
}

// Iterative post-order traversal
void iterativePostorder(Node* root) {
    if (root == nullptr)
        return;

    Stack* stack1 = createStack();
    Stack* stack2 = createStack();
    push(stack1, root);

    while (!isEmpty(stack1)) {
        Node* current = pop(stack1);
        push(stack2, current);

        if (current->left)
            push(stack1, current->left);
        if (current->right)
            push(stack1, current->right);
    }

    while (!isEmpty(stack2)) {
        Node* current = pop(stack2);
        cout << current->data << " ";
    }

    delete stack1;
    delete stack2;
}

int main() {
    // Construct a sample binary tree
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    cout << "Iterative In-order traversal: ";
    iterativeInorder(root);
    cout << endl;
```

```
  cout << "Iterative Pre-order traversal: ";
  iterativePreorder(root);
  cout << endl;

  cout << "Iterative Post-order traversal: ";
  iterativePostorder(root);
  cout << endl;

  return 0;
}
```

3.

```
#include <stdio.h>
#include <stdlib.h>

// Define a structure for a binary tree node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
struct Node* newNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return node;
}

// Recursive function to perform in-order traversal
void inorder(struct Node* root) {
    if (root == NULL)
        return;
    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
}
```

```
// Recursive function to perform pre-order traversal
void preorder(struct Node* root) {
  if (root == NULL)
    return;
  printf("%d ", root->data);
  preorder(root->left);
  preorder(root->right);
}

// Recursive function to perform post-order traversal
void postorder(struct Node* root) {
  if (root == NULL)
    return;
  postorder(root->left);
  postorder(root->right);
  printf("%d ", root->data);
}

int main() {
  // Construct a sample binary tree
  struct Node* root = newNode(1);
  root->left = newNode(2);
  root->right = newNode(3);
  root->left->left = newNode(4);
  root->left->right = newNode(5);

  printf("In-order traversal: ");
  inorder(root);

  printf("\nPre-order traversal: ");
  preorder(root);

  printf("\nPost-order traversal: ");
  postorder(root);

  return 0;
```

}

4.

```python
class TreeNode:
    def __init__(self, info):
        self.info = info
        self.left = None
        self.lthread = True
        self.right = None
        self.rthread = True


def insert(root, ikey):
    tmp = None
    par = None
    ptr = root
    found = False

    while ptr:
        if ikey == ptr.info:
            found = True
            break
        par = ptr
        if ikey < ptr.info:
            if not ptr.lthread:
                ptr = ptr.left
            else:
                break
        else:
            if not ptr.rthread:
                ptr = ptr.right
            else:
                break

    if found:
        print("\nDuplicate key")
    else:
        tmp = TreeNode(ikey)
        tmp.lthread = True
        tmp.rthread = True
        if not par:
            root = tmp
            tmp.left = None
            tmp.right = None
        elif ikey < par.info:
            tmp.left = par.left
            tmp.right = par
            par.lthread = False
            par.left = tmp
        else:
            tmp.left = par
            tmp.right = par.right
```

```
      par.rthread = False
      par.right = tmp
   return root


def delete(root, dkey):
   ptr = root
   par = None
   found = False

   while ptr:
      if dkey == ptr.info:
         found = True
         break
      par = ptr
      if dkey < ptr.info:
         if not ptr.lthread:
            ptr = ptr.left
         else:
            break
      else:
         if not ptr.rthread:
            ptr = ptr.right
         else:
            break

   if not found:
      print("\ndkey not present in tree")
   elif not ptr.lthread and not ptr.rthread:
      root = case_c(root, par, ptr)
   elif not ptr.lthread or not ptr.rthread:
      root = case_b(root, par, ptr)
   else:
      root = case_a(root, par, ptr)
   return root


def case_a(root, par, ptr):
   if not par:  # root node to be deleted
      root = None
   elif ptr == par.left:  # node is left child of its parent
      par.lthread = True
      par.left = ptr.left
   else:  # node is right child of its parent
      par.rthread = True
      par.right = ptr.right
   return root


def case_b(root, par, ptr):
   child = None
```

```
    s = None
    p = None

    if not ptr.lthread:  # node to be deleted has left child
        child = ptr.left
    else:  # node to be deleted has right child
        child = ptr.right

    if not par:  # node to be deleted is root node
        root = child
    elif ptr == par.left:  # node is left child of its parent
        par.left = child
    else:  # node is right child of its parent
        par.right = child

    s = in_succ(ptr)
    p = in_pred(ptr)

    if not ptr.lthread:  # if ptr has left subtree
        p.right = s
    else:
        if not ptr.rthread:  # if ptr has right subtree
            s.left = p

    return root


def case_c(root, par, ptr):
    succ = None
    parsucc = ptr

    succ = ptr.right
    while succ.left:
        parsucc = succ
        succ = succ.left

    ptr.info = succ.info

    if succ.lthread and succ.rthread:
        root = case_a(root, parsucc, succ)
    else:
        root = case_b(root, parsucc, succ)

    return root


def in_succ(ptr):
    if ptr.rthread:
        return ptr.right
    else:
        ptr = ptr.right
```

```
        while not ptr.lthread:
            ptr = ptr.left
        return ptr


def in_pred(ptr):
    if ptr.lthread:
        return ptr.left
    else:
        ptr = ptr.left
        while not ptr.rthread:
            ptr = ptr.right
        return ptr


def inorder(root):
    ptr = root
    if not root:
        print("Tree is empty")
        return

    while not ptr.lthread:
        ptr = ptr.left

    while ptr:
        print(ptr.info, end=" ")
        ptr = in_succ(ptr)
    print()


def preorder(root):
    ptr = root
    if not root:
        print("Tree is empty")
        return

    while ptr:
        print(ptr.info, end=" ")
        if not ptr.lthread:
            ptr = ptr.left
        elif not ptr.rthread:
            ptr = ptr.right
        else:
            while ptr and ptr.rthread:
                ptr = ptr.right
            if ptr:
                ptr = ptr.right


if __name__ == "__main__":
    root = None
```

```
while True:
  print("\n1.Insert")
  print("2.Delete")
  print("3.Inorder Traversal")
  print("4.Preorder Traversal")
  print("5.Quit")
  choice = int(input("Enter your choice : "))

  if choice == 1:
    num = int(input("Enter the number to be inserted : "))
    root = insert(root, num)
  elif choice == 2:
    num = int(input("Enter the number to be deleted : "))
    root = delete(root, num)
  elif choice == 3:
    inorder(root)
  elif choice == 4:
    preorder(root)
  elif choice == 5:
    break
  else:
    print("\nWrong choice")
```