Experiment 18 **Date: 09-12-2024**

Binary Search Tree Operations

Aim: Menu Driven program to implement Binary Search Tree (BST) Operations- Insertion of node, Deletion of a node, In-order traversal, Pre-order traversal and post-order traversal.

Algorithm

main()

1. Start
2. Declare and Initialize:
    a. Variables:
        i. root = NULL
        ii. choice, data
3. Repeat Until choice == 6:
4. Display Menu Options:
    1. Insert
    ii. 2. Delete
    iii. 3. In-order Traversal
    iv. 4. Pre-order Traversal
    v. 5. Post-order Traversal
    vi. 6. Exit
5. Read user choice into choice.
    a. Case 1:
        i. Read data to insert into BST
        ii. Call insert(root, data) to insert node into BST
        iii. Print "Node <data> inserted."
    b. Case 2:
        i. Read data to delete from BST
        ii. Call delete(root, data) to delete node from BST
        iii. Print "Node <data> deleted (if it existed)."
    c. Case 3:
        i. Print "In-order Traversal: "
        ii. Call inorder(root) to display the in-order traversal of the BST.
    d. Case 4:
        i. Print "Pre-order Traversal: "
        ii. Call preorder(root) to display the pre-order traversal of the BST.
    e. Case 5:
        i. Print "Post-order Traversal: "
        ii. Call postorder(root) to display the post-order traversal of the BST.
    f. Case 6:
        i. Print "Exiting..."
6. Exit.

createNode()

1. Start

2. Allocate memory for a new node.
3. Set node->data to the provided data.
4. Set node->left and node->right to NULL.
5. Return the newly created node.
6. End

insert(root, data)

1. Start
2. If root == NULL:
    a. Create a new node with data.
    b. Return the new node.
3. If data < root->data:
    a. Call insert(root->left, data) and assign it to root->left.
4. Else If data > root->data:
    a. Call insert(root->right, data) and assign it to root->right.
5. Return root.
6. End

findMin(root)

1. Start
2. While root and root->left != NULL:
    a. Set root = root->left.
3. Return root.
4. End

delete(root, data)

1. Start
2. If root == NULL:
    a. Return root.
3. If data < root->data:
    a. Call delete(root->left, data) and assign it to root->left.
4. Else If data > root->data:
    a. Call delete(root->right, data) and assign it to root->right.
5. Else:
    a. If root->left == NULL:
        i. Set temp = root->right
        ii. Free root
        iii. Return temp
    b. Else If root->right == NULL:
        i. Set temp = root->left
        ii. Free root
        iii. Return temp.
    c. Set temp = findMin(root->right)
    d. Set root->data = temp->data
    e. Call delete(root->right, temp->data) and assign it to root->right.
6. Return root.
7. End

inorder(root)

1. Start
2. If root != NULL:

a. Print root->data
b. Call preorder(root->left)
c. Call preorder(root->right).

3. End

preorder(root)

1. Start
2. If root != NULL:
    a. Call postorder(root->left)
    b. Call postorder(root->right)
    c. Print root->data.
3. End

postorder(root)

1. Start
2. If root != NULL:
    a. Call postorder(root->left)
    b. Call postorder(root->right)
    c. Print root->data.
3. End

Program

```
#include <stdio.h>
#include <stdlib.h>
struct BSTNode {
int data;
struct BSTNode *left, *right;};
struct BSTNode* createNode(int data) {
struct BSTNode* node = (struct BSTNode*)malloc(sizeof(struct
BSTNode));
node->data = data;
node->left = node->right = NULL;
return node;}
struct BSTNode* insert(struct BSTNode* root, int data) {
if (root == NULL) {
return createNode(data);}
if (data < root->data) {
root->left = insert(root->left, data);
} else if (data > root->data) {
root->right = insert(root->right, data);}
return root;}
struct BSTNode* findMin(struct BSTNode* root) {
while (root && root->left != NULL) {
root = root->left;}
return root;}
struct BSTNode* delete(struct BSTNode* root, int data) {
if (root == NULL) {
return root;}
if (data < root->data) {
root->left = delete(root->left, data);
```

```
} else if (data > root->data) {
root->right = delete(root->right, data);
} else {
if (root->left == NULL) {
struct BSTNode* temp = root->right;
free(root);
return temp;
} else if (root->right == NULL) {
struct BSTNode* temp = root->left;
free(root);
return temp;}
struct BSTNode* temp = findMin(root->right);
root->data = temp->data;
root->right = delete(root->right, temp->data);}
return root;}
void inorder(struct BSTNode* root) {
if (root != NULL) {
inorder(root->left);
printf("%d ", root->data);
inorder(root->right);}}
void preorder(struct BSTNode* root) {
if (root != NULL) {
printf("%d ", root->data);
preorder(root->left);
preorder(root->right);}}
void postorder(struct BSTNode* root) {
if (root != NULL) {
postorder(root->left);
postorder(root->right);
printf("%d ", root->data);}}
int main() {
struct BSTNode* root = NULL;
int choice, data;
do {
printf("\nBinary Search Tree Operations:\n");
printf("1. Insert\n2. Delete\n3. In-order Traversal\n4. Pre-order
Traversal\n5. Post-order Traversal\n6. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);
switch (choice) {
case 1:printf("Enter data to insert: ");
scanf("%d", &data);
root = insert(root, data);
printf("Node %d inserted.\n", data);break;
case 2:printf("Enter data to delete: ");
scanf("%d", &data);
root = delete(root, data);
printf("Node %d deleted (if it existed).\n", data);break;
case 3:printf("In-order Traversal: ");
```

```
inorder(root);
printf("\n");break;
case 4:printf("Pre-order Traversal: ");
preorder(root);
printf("\n");break;
case 5:printf("Post-order Traversal: ");
postorder(root);
printf("\n");break;
case 6:printf("Exiting...\n");break;
default:printf("Invalid choice! Please try again.\n");
}} while (choice != 6);
return 0;}
```

Output

```
Binary Search Tree Operations:
1. Insert
2. Delete
3. In-order Traversal
4. Pre-order Traversal
5. Post-order Traversal
6. Exit
Enter your choice: 1
Enter data to insert: 20
Node 20 inserted.

Binary Search Tree Operations:
1. Insert
2. Delete
3. In-order Traversal
4. Pre-order Traversal
5. Post-order Traversal
6. Exit
Enter your choice: 1
Enter data to insert: 16
Node 16 inserted.

Binary Search Tree Operations:
1. Insert
2. Delete
3. In-order Traversal
4. Pre-order Traversal
5. Post-order Traversal
6. Exit
Enter your choice: 1
Enter data to insert: 4
Node 4 inserted.

Binary Search Tree Operations:
```

```
1. Insert
2. Delete
3. In-order Traversal
4. Pre-order Traversal
5. Post-order Traversal
6. Exit
Enter your choice: 1
Enter data to insert: 17
Node 17 inserted.

Binary Search Tree Operations:
1. Insert
2. Delete
3. In-order Traversal
4. Pre-order Traversal
5. Post-order Traversal
6. Exit
Enter your choice: 1
Enter data to insert: 35
Node 35 inserted.

Binary Search Tree Operations:
1. Insert
2. Delete
3. In-order Traversal
4. Pre-order Traversal
5. Post-order Traversal
6. Exit
Enter your choice: 1
Enter data to insert: 45
Node 45 inserted.

Binary Search Tree Operations:
1. Insert
2. Delete
3. In-order Traversal
4. Pre-order Traversal
5. Post-order Traversal
6. Exit
Enter your choice: 1
Enter data to insert: 50
Node 50 inserted.

Binary Search Tree Operations:
1. Insert
2. Delete
3. In-order Traversal
4. Pre-order Traversal
5. Post-order Traversal
```

```
6. Exit
Enter your choice: 3
In-order Traversal: 4 16 17 20 35 45 50

Binary Search Tree Operations:
1. Insert
2. Delete
3. In-order Traversal
4. Pre-order Traversal
5. Post-order Traversal
6. Exit
Enter your choice: 4
Pre-order Traversal: 20 16 4 17 35 45 50

Binary Search Tree Operations:
1. Insert
2. Delete
3. In-order Traversal
4. Pre-order Traversal
5. Post-order Traversal
6. Exit
Enter your choice: 5
Post-order Traversal: 4 17 16 50 45 35 20

Binary Search Tree Operations:
1. Insert
2. Delete
3. In-order Traversal
4. Pre-order Traversal
5. Post-order Traversal
6. Exit
Enter your choice: 2
Enter data to delete: 4
Node 4 deleted (if it existed).

Binary Search Tree Operations:
1. Insert
2. Delete
3. In-order Traversal
4. Pre-order Traversal
5. Post-order Traversal
6. Exit
Enter your choice: 3
In-order Traversal: 16 17 20 35 45 50

Binary Search Tree Operations:
1. Insert
2. Delete
3. In-order Traversal
```

4. Pre-order Traversal
5. Post-order Traversal
6. Exit
Enter your choice: 6
Exiting...

Experiment 19 **Date: 09-12-2024**

Red Black Tree Operations

Aim: Create Red Black Tree and perform the following operation.

i. Create
ii. Insert a new node
iii. Left rotate
iv. Right rotate
v. Inorder traversal

Algorithm
main()

1. Start
2. Declare and Initialize: root = NULL,choice, data
3. Repeat Until choice == 6:
4. Display Menu Options:
   1. Create
   2. Insert a new node
   3. Perform Left Rotate
   4. Perform Right Rotate
   5. In-order Traversal
   6. Exit
5. Read user choice into choice
   a. Case 1:
      i. tree=call createRedBlackTree()
   b. Case 2:
      i. Read data to insert into the Red-Black Tree
      ii. Call insert(root, data) to insert a node
      iii. Print "Node <data> inserted."
   c. Case 3:
      i. Read data for the node to perform a left rotation
      ii. Find the node in the tree
      iii. Call leftRotate(tree, node) to perform a left rotation
      iv. Print "Left rotation performed on node <data>"
   d. Case 4:
      i. Read data for the node to perform a right rotation
      ii. Find the node in the tree
      iii. Call rightRotate(tree, node) to perform a right rotation
      iv. Print "Right rotation performed on node <data>"
   e. Case 5:
      i. Print "In-order Traversal: "
      ii. Call inOrderTraversal(root) to display the in-order traversal of the tree
   f. Case 6:
      i.Print "Exiting..."
6. Stop

createNode()

1. Start
2. Allocate memory for a new node.
3. Set node->data to the provided data.
4. Set node->left and node->right to NULL.
5. Set node->color to 1 (Red).
6. Return new node
7. End

insert(root, data)

1. Start
2. Create z =createNode(data).
3. Set y = NULL
4. Set x = root.
5. While x != NULL:
   a. Set y = x
   b. If data < x->data
      i. set x = x->left
   c. Else
      i. set x = x->right.
6. Set z->parent = y
7. If y == NULL,
   a. Set tree->root = z.
8. Else If z->data < y->data
   a. set y->left = z.
9. Else
   a. set y->right = z.
10. Call fixInsert(tree,z)
11. End

leftRotate(tree, x)

1. Start
2. Set y = x->right.
3. Set x->right = y->left.
4. If y->left != NULL,
   a. set y->left->parent = x.
5. Set y->parent = x->parent.
6. If x->parent == NULL,
   a. set tree->root = y.
7. Else If x == x->parent->left,
   a. set x->parent->left = y.
8. Else,
   a. set x->parent->right = y.
9. Set y->left = x and x->parent = y.
10. End

rightRotate(tree, y)

1. Start
2. Set x = y->left.
3. Set y->left = x->right
4. If x->right != NULL,

a. set x->right->parent = y
5. Set x->parent = y->parent.
6. If y->parent == NULL,
   a. set tree->root = x
7. Else If y == y->parent->left,
   a. set y->parent->left = x
8. Else,
   a. set y->parent->right = x
9. Set x->right = y and y->parent = x
10. End

fixInsert(tree, z)

1. Start
2. While z != tree->root and z->parent->color == 1
   a. If z->parent == z->parent->parent->left,
      i. Set y = z->parent->parent->right
      If y != NULL and y->color == 1,
      Set z->parent->color = 0
      Set y->color = 0
      Set z->parent->parent->color = 1
      Set z = z->parent->parent
   Else:
      If z == z->parent->right
      z = z->parent
      call leftRotate(tree, z)
      Set z->parent->color = 0
      Set z->parent->parent->color = 1
      Call rightRotate(tree, z->parent->parent);
   b. Else
      i. Set y = z->parent->parent->left
      ii. If y != NULL and y->color == 1:
         Set z->parent->color = 0.
         Set y->color = 0.
         Set z->parent->parent->color = 1.
         Set z = z->parent->parent
      iii. Else:
         If z == z->parent->left, set z = z->parent and call rightRotate(tree, z).
         Set z->parent->color = 0.
         Set z->parent->parent->color = 1.
         Call leftRotate(tree, z->parent->parent).
3. Set tree->root->color = 0.
4. End

inOrderTraversal(root)

1. Start
2. If root != NULL,
   a. Call inOrderTraversal(root->left)
   b. Print root->data and root->color
   c. Call inOrderTraversal(root->right).

3. End

Program

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node {
int data;
struct Node* parent;
struct Node* left;
struct Node* right;
int color;
} Node;
typedef struct RedBlackTree {
Node* root;
} RedBlackTree;
Node* createNode(int data) {
Node* newNode = (Node*)malloc(sizeof(Node));
if (newNode == NULL) {
printf("Memory allocation error\n");
exit(1);}
newNode->data = data;
newNode->parent = newNode->left = newNode->right = NULL;
newNode->color = 1;
return newNode;}
RedBlackTree* createRedBlackTree() {
RedBlackTree* newTree = (RedBlackTree*)malloc(sizeof(RedBlackTree));
if (newTree == NULL) {
printf("Memory allocation error\n");
exit(1);}
newTree->root = NULL;
return newTree;}
void leftRotate(RedBlackTree* tree, Node* x) {
Node* y = x->right;
x->right = y->left;
if (y->left != NULL) y->left->parent = x;
y->parent = x->parent;
if (x->parent == NULL) tree->root = y;
else if (x == x->parent->left) x->parent->left = y;
else x->parent->right = y;
y->left = x;
x->parent = y;
}
void rightRotate(RedBlackTree* tree, Node* y) {
Node* x = y->left;
y->left = x->right;
if (x->right != NULL) x->right->parent = y;
x->parent = y->parent;
if (y->parent == NULL) tree->root = x;
else if (y == y->parent->left) y->parent->left = x;
```

```
else y->parent->right = x;
x->right = y;
y->parent = x;
}
void fixInsert(RedBlackTree* tree, Node* z) {
while (z != tree->root && z->parent->color == 1) {
if (z->parent == z->parent->parent->left) {
Node* y = z->parent->parent->right;
if (y != NULL && y->color == 1) {
z->parent->color = 0;
y->color = 0;
z->parent->parent->color = 1;
z = z->parent->parent;
} else {
if (z == z->parent->right) {
z = z->parent;
leftRotate(tree, z);}
z->parent->color = 0;
z->parent->parent->color = 1;
rightRotate(tree, z->parent->parent);}
} else {
Node* y = z->parent->parent->left;
if (y != NULL && y->color == 1) {
z->parent->color = 0;
y->color = 0;
z->parent->parent->color = 1;
z = z->parent->parent;
} else {
if (z == z->parent->left) {
z = z->parent;
rightRotate(tree, z);}
z->parent->color = 0;
z->parent->parent->color = 1;
leftRotate(tree, z->parent->parent);
}}}
tree->root->color = 0;}
void insert(RedBlackTree* tree, int data) {
Node* z = createNode(data);
Node* y = NULL;
Node* x = tree->root;
while (x != NULL) {
y = x;
if (z->data < x->data) x = x->left;
else x = x->right;}
z->parent = y;
if (y == NULL) tree->root = z;
else if (z->data < y->data) y->left = z;
else y->right = z;
fixInsert(tree, z);
```

```
}
void inOrderTraversal(Node* root) {
if (root != NULL) {
inOrderTraversal(root->left);
printf("%d (%s) -> ", root->data, root->color == 0 ? "BLACK" : "RED");
inOrderTraversal(root->right);
}}
int main() {
int choice, value;
RedBlackTree* tree = createRedBlackTree();
do {
printf("\n1. Create Red-Black Tree\n");
printf("2. Insert a new node\n");
printf("3. Left Rotate\n");
printf("4. Right Rotate\n");
printf("5. Inorder Traversal\n");
printf("6. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);
switch(choice) {
case 1:
tree = createRedBlackTree();
printf("Red-Black Tree created successfully!\n");
break;
case 2:
printf("Enter the value to be inserted: ");
scanf("%d", &value);
insert(tree, value);
printf("Node inserted successfully!\n");
break;
case 3:
{int rotateValue;
printf("Enter the node value to perform left rotation: ");
scanf("%d", &rotateValue);
Node* nodeToRotate = tree->root;
while (nodeToRotate != NULL && nodeToRotate->data != rotateValue) {
if (rotateValue < nodeToRotate->data)
nodeToRotate = nodeToRotate->left;
else
nodeToRotate = nodeToRotate->right;
}
if (nodeToRotate != NULL) {
leftRotate(tree, nodeToRotate);
printf("Left rotation performed on node %d\n", rotateValue);
} else {
printf("Node with value %d not found!\n", rotateValue);
}}
break;
case 4:{
```

```
int rotateValue;
printf("Enter the node value to perform right rotation: ");
scanf("%d", &rotateValue);
Node* nodeToRotate = tree->root;
while (nodeToRotate != NULL && nodeToRotate->data != rotateValue) {
if (rotateValue < nodeToRotate->data)
nodeToRotate = nodeToRotate->left;
else
nodeToRotate = nodeToRotate->right;}
if (nodeToRotate != NULL) {
rightRotate(tree, nodeToRotate);
printf("Right rotation performed on node %d\n", rotateValue);
} else {
printf("Node with value %d not found!\n", rotateValue);}}
break;
case 5:printf("Inorder Traversal: ");
inOrderTraversal(tree->root);
printf("NULL\n");
break;
case 6:printf("Exiting...\n");
break;
default:
printf("Invalid choice. Please try again!\n");
}} while (choice != 6);
return 0;}
```

Output

```
1. Create Red-Black Tree
2. Insert a new node
3. Left Rotate
4. Right Rotate
5. Inorder Traversal
6. Exit
Enter your choice: 1
Red-Black Tree created successfully!

1. Create Red-Black Tree
2. Insert a new node
3. Left Rotate
4. Right Rotate
5. Inorder Traversal
6. Exit
Enter your choice: 2
Enter the value to be inserted: 14
Node inserted successfully!
```

```
1. Create Red-Black Tree
2. Insert a new node
3. Left Rotate
4. Right Rotate
5. Inorder Traversal
6. Exit
Enter your choice: 2
Enter the value to be inserted: 18
Node inserted successfully!

1. Create Red-Black Tree
2. Insert a new node
3. Left Rotate
4. Right Rotate
5. Inorder Traversal
6. Exit
Enter your choice: 2
Enter the value to be inserted: 7
Node inserted successfully!

1. Create Red-Black Tree
2. Insert a new node
3. Left Rotate
4. Right Rotate
5. Inorder Traversal
6. Exit
Enter your choice: 2
Enter the value to be inserted: 61
Node inserted successfully!

1. Create Red-Black Tree
2. Insert a new node
3. Left Rotate
4. Right Rotate
5. Inorder Traversal
6. Exit
Enter your choice: 5
Inorder Traversal: 7 (BLACK) -> 14 (BLACK) -> 18 (BLACK) -> 61 (RED) ->
NULL

1. Create Red-Black Tree
2. Insert a new node
3. Left Rotate
4. Right Rotate
5. Inorder Traversal
6. Exit
Enter your choice: 3
Enter the node value to perform left rotation: 14
Left rotation performed on node 14
```

1. Create Red-Black Tree
2. Insert a new node
3. Left Rotate
4. Right Rotate
5. Inorder Traversal
6. Exit

Enter your choice: 4
Enter the node value to perform right rotation: 7

Experiment 20 **Date: 11-12-2024**

B-Tree Operation

Aim Write a program to implement the following operation on B Tree

i. Creation
ii. Insertion
iii. Searching
iv. Inorder traversal

Algorithm

main()

1. Start
2. Declare and initialize: tree = createBTree(DEGREE), choice, value.
3. Repeat until choice == 4:
4. Display menu options:
   1: Insert a new node
   2: Search for a node
   3: Perform in-order traversal
   4: Exit
5. Read user choice from input.
6. Perform action based on choice:
   a. Case 1:
      i. Read value
      ii. Call insert(tree, value)
      iii. Print: "Node <value> inserted."
   b. Case 2:
      i. Read value
      ii. Call search(tree, value)
      iii. Print: "Value <value> found in the tree" or "Value <value> not found in the tree."
   c. Case 3:
      i. Print: "In-order Traversal:"
      ii. Call inOrderTraversal(tree) to display the traversal.
   d. Case 4: Print: "Exiting..."
7. Stop

createNode(t, leaf)

1. Start
2. Allocate memory for a new node.
3. Set node->t = t.
4. Set node->leaf = leaf.
5. Allocate memory for node->keys with size (2 * t - 1).
6. Allocate memory for node->C with size 2 * t.
7. Set node->n = 0.
8. Return the created node.
9. End

createBTree(t)

1. Start
2. Call createNode(t, 1) to create a new node (leaf).
3. Return the created node as the root of the tree.
4. End

inOrderTraversal(root)

1. Start
2. If root != NULL: i. For each i from 0 to root->n - 1:
    a. Call inOrderTraversal(root->C[i])
    b. Print root->keys[i] ii. Call inOrderTraversal(root->C[root->n])
3. End

search(root, key)

1. Start
2. Set i = 0.
3. While i < root->n and key > root->keys[i],
    a. increment i.
4. If i < root->n and key == root->keys[i],
    a. return root (found).
5. If root->leaf,
    a. return NULL (key not found).
6. Call search(root->C[i], key) recursively.
7. End

splitChild(parent, i, child)

1. Start
2. Set t = child->t.
3. Create a new node newNode = createNode(t, child->leaf).
4. Set newNode->n = t - 1.
5. For j = 0 to t – 2:
    a. Set newNode->keys[j] = child->keys[j + t].
6. If child->leaf is false, for j = 0 to t - 1:
    a. Set newNode->C[j] = child->C[j + t].
7. Set child->n = t - 1.
8. For j = parent->n - 1 to i:
    a. Set parent->C[j + 1] = parent->C[j].
9. Set parent->C[i + 1] = newNode
10. For j = parent->n - 1 to i:
    a. Set parent->keys[j + 1] = parent->keys[j]
11. Set parent->keys[i] = child->keys[t - 1]
12. Increment parent->n by 1
13. End

insertNonFull(node, key)

1. Start
2. Set i = node->n - 1.
3. If node->leaf:
    a. While i >= 0 and key < node->keys[i],
        i. move node->keys[i] to node->keys[i + 1] and

decrement i
ii. Insert key at position i + 1
iii. Increment node->n by 1.

4. Else:
   a. While i >= 0 and key < node->keys[i]
      i. decrement i.
      ii. If node->C[i + 1]->n == 2 * node->t – 1
          call splitChild(node, i + 1, node->C[i + 1]).
      iii. If key > node->keys[i + 1],
          increment i
      iv. Call insertNonFull(node->C[i + 1], key).
5. End

insert(root, key)

1. Start
2. If root->n == 2 * root->t - 1:
   a. Create newRoot = createNode(root->t, 0)
   b. Set newRoot->C[0] = root
   c. Call splitChild(newRoot, 0, root)
   d. Set i = 0. If key > newRoot->keys[0],
      increment i
   e. Call insertNonFull(newRoot->C[i], key)
   f. Return newRoot.
3. Else:
   a. Call insertNonFull(root, key)
   b. Return root.
4. End

Program

```
#include <stdio.h>
#include <stdlib.h>
#define DEGREE 3

typedef struct BTreeNode {
int *keys;
int t;
struct BTreeNode **C;
int n;
int leaf;
} BTreeNode;
BTreeNode *createNode(int t, int leaf) {
BTreeNode *node = (BTreeNode *)malloc(sizeof(BTreeNode));
node->t = t;
node->leaf = leaf;
node->keys = (int *)malloc((2 * t - 1) * sizeof(int));
node->C = (BTreeNode **)malloc(2 * t * sizeof(BTreeNode *));
node->n = 0;
return node;}
BTreeNode *createBTree(int t) {
```

```
return createNode(t, 1);}
void inOrderTraversal(BTreeNode *root) {
if (root != NULL) {
for (int i = 0; i < root->n; i++) {
inOrderTraversal(root->C[i]);
printf("%d ", root->keys[i]);}
inOrderTraversal(root->C[root->n]);
}}
BTreeNode *search(BTreeNode *root, int key) {
int i = 0;
while (i < root->n && key > root->keys[i])
i++;
if (i < root->n && key == root->keys[i])
return root;
if (root->leaf)
return NULL;
return search(root->C[i], key);}
void splitChild(BTreeNode *parent, int i, BTreeNode *child) {
int t = child->t;
BTreeNode *newNode = createNode(t, child->leaf);
newNode->n = t - 1;
for (int j = 0; j < t - 1; j++)
newNode->keys[j] = child->keys[j + t];
if (!child->leaf) {
for (int j = 0; j < t; j++)
newNode->C[j] = child->C[j + t];}
child->n = t - 1;
for (int j = parent->n; j >= i + 1; j--)
parent->C[j + 1] = parent->C[j];
parent->C[i + 1] = newNode;
for (int j = parent->n - 1; j >= i; j--)
parent->keys[j + 1] = parent->keys[j];
parent->keys[i] = child->keys[t - 1];
parent->n++;}
void insertNonFull(BTreeNode *node, int key) {
int i = node->n - 1;
if (node->leaf) {
while (i >= 0 && key < node->keys[i]) {
node->keys[i + 1] = node->keys[i];
i--;}
node->keys[i + 1] = key;
node->n++;
} else {
while (i >= 0 && key < node->keys[i])
i--;
if (node->C[i + 1]->n == 2 * node->t - 1) {
splitChild(node, i + 1, node->C[i + 1]);
if (key > node->keys[i + 1])
i++;}
```

```
insertNonFull(node->C[i + 1], key);
}}BTreeNode *insert(BTreeNode *root, int key) {
if (root->n == 2 * root->t - 1) {
BTreeNode *newRoot = createNode(root->t, 0);
newRoot->C[0] = root;
splitChild(newRoot, 0, root);
int i = 0;
if (key > newRoot->keys[0])
i++;
insertNonFull(newRoot->C[i], key);
return newRoot;
} else {
insertNonFull(root, key);
return root;}}
int main() {
int choice, value;
BTreeNode *tree = createBTree(DEGREE);
do {
printf("\n1. Insertion\n2. Searching\n3. Display (Inorder Traversal)\n4.
Exit\nEnter your choice: ");
scanf("%d", &choice);
switch (choice) {
case 1:
printf("Enter the value to insert: ");
scanf("%d", &value);
tree = insert(tree, value);
break;
case 2:printf("Enter the value to search: ");
scanf("%d", &value);
BTreeNode *result = search(tree, value);
if (result != NULL)
printf("Value %d found in the tree.\n", value);
else
printf("Value %d not found in the tree.\n", value);
break;
case 3:printf("Inorder Traversal: ");
inOrderTraversal(tree);
printf("\n");
break;
case 4:printf("Exiting...\n");
break;
default:printf("Invalid choice!\n");
}} while (choice != 4);
return 0;}
```

Output

1. Insertion
2. Searching
3. Display (Inorder Traversal)

```
4. Exit
Enter your choice: 1
Enter the value to insert: 10

1. Insertion
2. Searching
3. Display (Inorder Traversal)
4. Exit
Enter your choice: 1
Enter the value to insert: 20

1. Insertion
2. Searching
3. Display (Inorder Traversal)
4. Exit
Enter your choice: 1
Enter the value to insert: 30

1. Insertion
2. Searching
3. Display (Inorder Traversal)
4. Exit
Enter your choice: 1
Enter the value to insert: 40

1. Insertion
2. Searching
3. Display (Inorder Traversal)
4. Exit
Enter your choice: 3
Inorder Traversal: 10 20 30 40

1. Insertion
2. Searching
3. Display (Inorder Traversal)
4. Exit
Enter your choice: 2
Enter the value to search: 40
Value 40 found in the tree.

1. Insertion
2. Searching
3. Display (Inorder Traversal)
4. Exit
Enter your choice: 4
Exiting...
```