

**1. Write a C program to Implement a 2-3-4 tree as an extension of the 2-3 tree.**

```
#include <stdio.h>

#include <stdlib.h>

#define MAX 3

#define MIN 1

typedef struct Node {
    int keys[MAX + 1], count;
    struct Node *child[MAX + 2];
}
Node;

Node* createNode(int key, Node *child) {
    Node *newNode;
    newNode = (Node*)malloc(sizeof(Node));
    newNode->keys[1] = key;
    newNode->count = 1;
    newNode->child[0] = child;
    newNode->child[1] = NULL;
    return newNode;
}

void insertValue(int key, int pos, Node *node, Node *child)
{
    int j = node->count;
    while (j > pos) {
        node->keys[j + 1] = node->keys[j];
        node->child[j + 1] = node->child[j];
        j--;
    }
    node->keys[j + 1] = key;
    node->child[j + 1] = child;
    node->count++;
}
```

```

void splitNode(int key, int *pval, int pos, Node *node, Node *child, Node **newNode)
{
    int median, j;

    if (pos > MIN)
        median = MIN + 1;
    else
        median = MIN;

    *newNode = (Node*)malloc(sizeof(Node));

    j = median + 1;
    while (j <= MAX)
    {
        (*newNode)->keys[j - median] = node->keys[j];
        (*newNode)->child[j - median] = node->child[j];

        j++;
    }

    node->count = median;
    (*newNode)->count = MAX - median;
    if (pos <= MIN)
        insertValue(key, pos, node, child);
    else
        insertValue(key, pos - median, *newNode, child);

    *pval = node->keys[node->count];
    (*newNode)->child[0] = node->child[node->count];
    node->count--;
}

int setValue(int key, int *pval, Node *node, Node **child) {
    int pos;

    if (!node)
    {
        *pval = key;
        *child = NULL;
    }
}

```

```

        return 1;
    }
    if (key < node->keys[1]) {
        pos = 0;
    }
else
{
    for (pos = node->count; (key < node->keys[pos] && pos > 1); pos--);
    if (key == node->keys[pos]) {
        return 0;
    }
}
if (setValue(key, pval, node->child[pos], child)) {
    if (node->count < MAX) {
        insertValue(*pval, pos, node, *child);
        return 0;
    }
Else
{
    splitNode(*pval, pval, pos, node, *child, child);
    return 1;
}
}
return 0;
}

void insert(int key, Node **root) {
    int flag, i;
    Node *child;
    flag = setValue(key, &i, *root, &child);
    if (flag)
    {

```

```

        Node *newNode = createNode(i, child);

        newNode->child[0] = *root;

        *root = newNode;
    }
}

void inorder(Node *node)
{
    int i;
    if (node)
    {
        for (i = 0; i < node->count; i++) {
            inorder(node->child[i]);
            printf("%d ", node->keys[i + 1]);
        }
        inorder(node->child[i]);
    }
}

int main()
{
    Node *root = NULL;

    insert(10, &root);
    insert(20, &root);
    insert(5, &root);
    insert(6, &root);
    insert(12, &root);
    insert(30, &root);
    insert(7, &root);
    insert(17, &root);

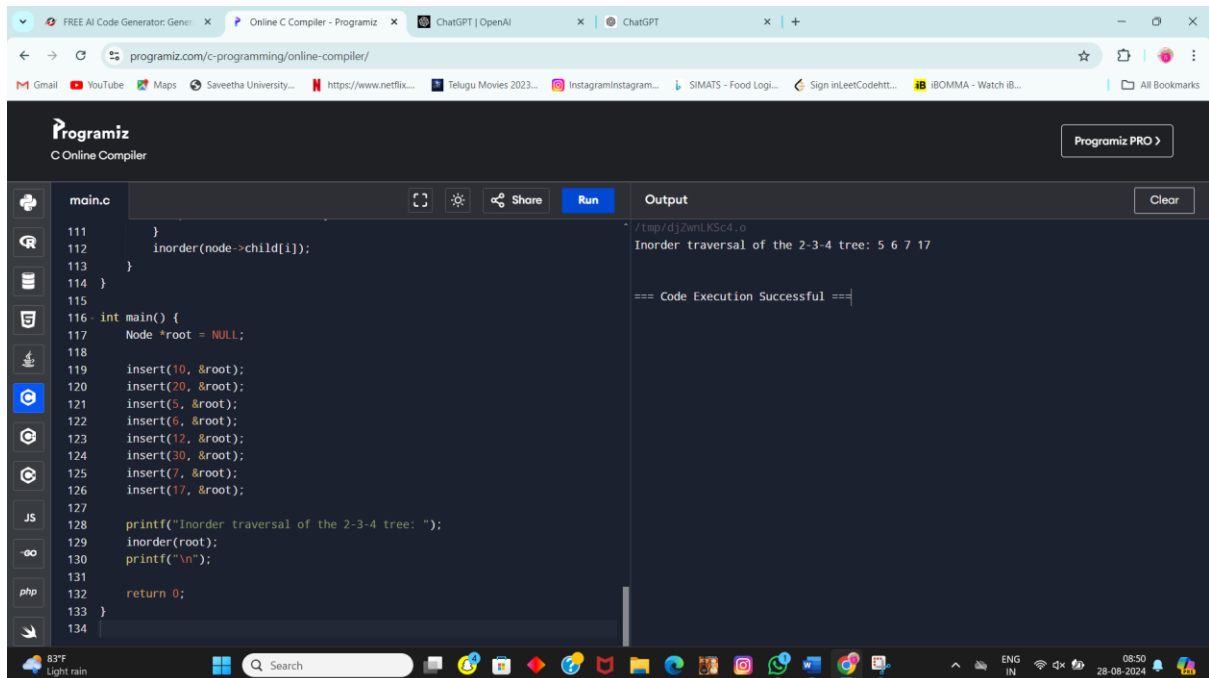
    printf("Inorder traversal of the 2-3-4 tree: ");
    inorder(root);
    printf("\n");
}

```

```

return 0;
}

```



**2. Write a C program to perform the following operations:**

- Insert an element into a Splay tree.
- Delete an element from a Splay tree.
- Search for a key element in a Splay tree.

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

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

```
typedef struct Node
```

```

{
    int key;
    struct Node *left, *right;
}

```

```
Node;
```

```
Node* newNode(int key)
```

```

{
    Node* node = (Node*)malloc(sizeof(Node));
    node->key = key;
}

```

```

    node->left = node->right = NULL;

    return node;
}

Node* rightRotate(Node* x) {
    Node* y = x->left;
    x->left = y->right;
    y->right = x;
    return y;
}

Node* leftRotate(Node* x)
{
    Node* y = x->right;
    x->right = y->left;
    y->left = x;
    return y;
}

Node* splay(Node* root, int key)
{
    if (!root || root->key == key)
        return root;
    if (key < root->key)
    {
        if (!root->left) return root;
        if (key < root->left->key)
        {
            root->left->left = splay(root->left->left, key);
            root = rightRotate(root);
        }
        else if (key > root->left->key) {
            root->left->right = splay(root->left->right, key);
            if (root->left->right)

```

```

        root->left = leftRotate(root->left);
    }
    return root->left ? rightRotate(root) : root;
}
else
{
    if (!root->right) return root;
    if (key < root->right->key)
    {
        root->right->left = splay(root->right->left, key);
        if (root->right->left)
            root->right = rightRotate(root->right);
    }
    else if (key > root->right->key)
    {
        root->right->right = splay(root->right->right, key);
        root = leftRotate(root);
    }
    return root->right ? leftRotate(root) : root;
}
}

```

```

Node* insert(Node* root, int key) {
    if (!root)
        return newNode(key);
    root = splay(root, key);
    if (root->key == key) return root;
    Node* new_node = newNode(key);
    if (key < root->key) {
        new_node->right = root;
        new_node->left = root->left;
        root->left = NULL;
    }
}

```

```
}
```

```
Else
```

```
{
```

```
    new_node->left = root;
```

```
    new_node->right = root->right;
```

```
    root->right = NULL;
```

```
}
```

```
    return new_node;
```

```
}
```

```
Node* deleteNode(Node* root, int key) {
```

```
    if (!root) return NULL;
```

```
    root = splay(root, key);
```

```
    if (root->key != key) return root;
```

```
    Node* temp;
```

```
    if (!root->left)
```

```
{
```

```
        temp = root;
```

```
        root = root->right;
```

```
}
```

```
Else
```

```
{
```

```
    temp = root;
```

```
    root = splay(root->left, key);
```

```
    root->right = temp->right;
```

```
}
```

```
    free(temp);
```

```
    return root;
```

```
}
```

```
Node* search(Node* root, int key) {
```

```
    return splay(root, key);
```



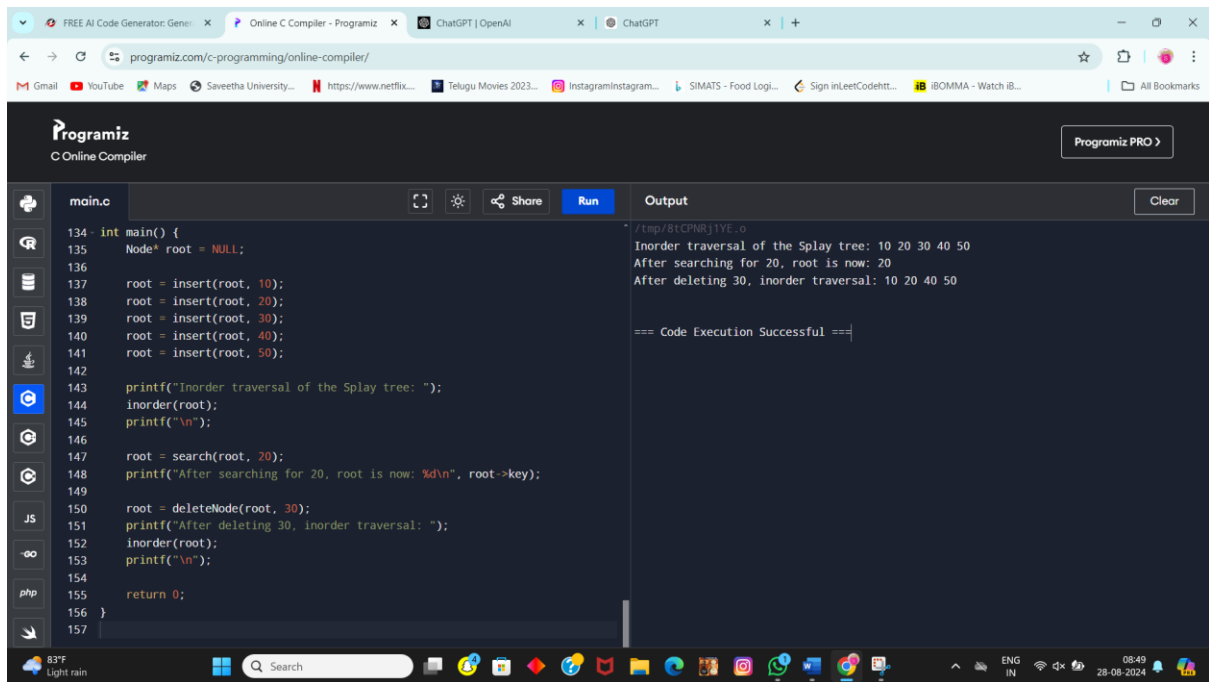
```

}

void inorder(Node* root) {
    if (root) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}

int main()
{
    Node* root = NULL;
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    printf("Inorder traversal of the Splay tree: ");
    inorder(root);
    printf("\n");
    root = search(root, 20);
    printf("After searching for 20, root is now: %d\n", root->key);
    root = deleteNode(root, 30);
    printf("After deleting 30, inorder traversal: ");
    inorder(root);
    printf("\n");
    return 0;
}

```



```
134 int main() {
135     Node* root = NULL;
136
137     root = insert(root, 10);
138     root = insert(root, 20);
139     root = insert(root, 30);
140     root = insert(root, 40);
141     root = insert(root, 50);
142
143     printf("Inorder traversal of the Splay tree: ");
144     inorder(root);
145     printf("\n");
146
147     root = search(root, 20);
148     printf("After searching for 20, root is now: %d\n", root->key);
149
150     root = deleteNode(root, 30);
151     printf("After deleting 30, inorder traversal: ");
152     inorder(root);
153     printf("\n");
154
155     return 0;
156 }
```

Output

```
/tmp/81CPNRj1YE.o
Inorder traversal of the Splay tree: 10 20 30 40 50
After searching for 20, root is now: 20
After deleting 30, inorder traversal: 10 20 40 50

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

**3. Implement a Trie data structure to store a set of strings. The Trie should support the following operations:**

**Insert:** Insert a string into the Trie.

**Search:** Search for a given string in the Trie.

**Delete:** Delete a string from the Trie.

**Prefix Search:** Find all words in the Trie that start with a given prefix.

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

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

```
#include <string.h>
```

```
#define ALPHABET_SIZE 26
```

```
typedef struct TrieNode
```

```
{
```

```
    struct TrieNode* children[ALPHABET_SIZE];
```

```
    int isEndOfWord;
```

```
}
```

```
TrieNode;
```

```
TrieNode* createNode()
```

```
{
```

```
    TrieNode* node = (TrieNode*)malloc(sizeof(TrieNode));
```

```

node->isEndOfWord = 0;
for (int i = 0; i < ALPHABET_SIZE; i++)
    node->children[i] = NULL;
return node;
}

void insert(TrieNode* root, const char* key)
{
    TrieNode* current = root;
    while (*key) {
        int index = *key - 'a';
        if (!current->children[index])
            current->children[index] = createNode();
        current = current->children[index];
        key++;
    }
    current->isEndOfWord = 1;
}

int search(TrieNode* root, const char* key)
{
    TrieNode* current = root;
    while (*key) {
        int index = *key - 'a';
        if (!current->children[index])
            return 0;
        current = current->children[index];
        key++;
    }
    return current && current->isEndOfWord;
}

int hasChildren(TrieNode* node)
{

```

```

    for (int i = 0; i < ALPHABET_SIZE; i++)
        if (node->children[i])
            return 1;
    return 0;
}

int delete(TrieNode* root, const char* key, int depth)
{
    if (!root) return 0;
    if (*key)
    {
        int index = *key - 'a';
        if (delete(root->children[index], key + 1, depth + 1))
        {
            free(root->children[index]);
            root->children[index] = NULL;
            return !hasChildren(root) && !root->isEndOfWord;
        }
    }
    Else
    {
        if (root->isEndOfWord)
        {
            root->isEndOfWord = 0;
            return !hasChildren(root);
        }
    }
    return 0;
}

```

```

void prefixSearchHelper(TrieNode* root, char* prefix)
{

```

```

    if (root->isEndOfWord) printf("%s\n", prefix);
    for (int i = 0; i < ALPHABET_SIZE; i++)
    {
        if (root->children[i])
        {
            char nextChar = 'a' + i;
            size_t len = strlen(prefix);
            prefix[len] = nextChar;
            prefix[len + 1] = '\0';
            prefixSearchHelper(root->children[i], prefix);
            prefix[len] = '\0';
        }
    }
}

void prefixSearch(TrieNode* root, const char* prefix) {
    TrieNode* current = root;
    while (*prefix) {
        int index = *prefix - 'a';
        if (!current->children[index])
            return;
        current = current->children[index];
        prefix++;
    }
    char buffer[100] = "";
    strcpy(buffer, prefix);
    prefixSearchHelper(current, buffer);
}

int main()
{
    TrieNode* root = createNode();

```

```

insert(root, "hello");

insert(root, "hell");

insert(root, "heaven");

insert(root, "heavy");

printf("Search for 'hell': %s\n", search(root, "hell") ? "Found" : "Not Found");

printf("Search for 'hello': %s\n", search(root, "hello") ? "Found" : "Not Found");

printf("Search for 'heavens': %s\n", search(root, "heavens") ? "Found" : "Not Found");

printf("\nWords with prefix 'he':\n");

prefixSearch(root, "he");

delete(root, "hell", 0);

printf("\nAfter deleting 'hell':\n");

printf("Search for 'hell': %s\n", search(root, "hell") ? "Found" : "Not Found");

printf("\nWords with prefix 'he':\n");

prefixSearch(root, "he");

return 0;
}

```

The screenshot displays the Programiz C Online Compiler interface. The code editor on the left contains a C program for a trie data structure. The code includes functions for inserting words, searching for words, finding words with a given prefix, and deleting a word. The output window on the right shows the results of running the program. The output confirms that 'hell', 'hello', and 'heaven' were successfully inserted and found. The prefix search for 'he' returned 'aven', 'avy', 'll', and 'llo'. After deleting 'hell', the search for 'hell' returned 'Not Found', and the prefix search for 'he' still returned the same words, indicating that the deletion operation was successful and did not affect the other words in the trie.

```

main.c
76 // Helper function for prefix search
77 void prefixSearchHelper(TrieNode* root, char* prefix) {
78     if (root->isEndOfWord) printf("%s\n", prefix);
79
80     for (int i = 0; i < ALPHABET_SIZE; i++) {
81         if (root->children[i]) {
82             char nextChar = 'a' + i;
83             size_t len = strlen(prefix);
84             prefix[len] = nextChar;
85             prefix[len + 1] = '\0';
86             prefixSearchHelper(root->children[i], prefix);
87             prefix[len] = '\0';
88         }
89     }
90 }
91
92 // Find all words in the Trie that start with a given prefix
93 void prefixSearch(TrieNode* root, const char* prefix) {
94     TrieNode* current = root;
95     while (*prefix) {
96         int index = *prefix - 'a';
97         if (!current->children[index])
98             return;
99         current = current->children[index];
100         prefix++;

```

Output

```

/tmp/1em1THGkLh.o
Search for 'hell': Found
Search for 'hello': Found
Search for 'heavens': Not Found

Words with prefix 'he':
aven
avy
ll
llo

After deleting 'hell':
Search for 'hell': Not Found

Words with prefix 'he':
aven
avy
llo

=== Code Execution Successful ===

```