1)**WRITE C PROGRAM FOR 2-3-4 TREE.**

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

typedef struct Node {
    int keys[3];
    struct Node *children[4];
    int numKeys;
    int isLeaf;
} Node;

Node* createNode(int isLeaf) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->isLeaf = isLeaf;
    newNode->numKeys = 0;
    for (int i = 0; i < 4; i++) newNode->children[i] = NULL;
    return newNode;
}

void splitChild(Node* parent, int index, Node* child) {
```

```
    Node* newChild = createNode(child->isLeaf);
    newChild->numKeys = 1;
    newChild->keys[0] = child->keys[1];
    if (!child->isLeaf) {
        for (int i = 0; i < 2; i++) newChild->children[i] = child->children[i + 2];
    }
    child->numKeys = 1;
    for (int i = parent->numKeys; i > index; i--) parent->children[i + 1] =
parent->children[i];
    parent->children[index + 1] = newChild;
    for (int i = parent->numKeys - 1; i >= index; i--) parent->keys[i + 1] =
parent->keys[i];
    parent->keys[index] = child->keys[0];
    parent->numKeys++;
}

void insertNonFull(Node* node, int key) {
    int i = node->numKeys - 1;
    if (node->isLeaf) {
        while (i >= 0 && key < node->keys[i]) {
            node->keys[i + 1] = node->keys[i];
            i--;
```

```
            }
            node->keys[i + 1] = key;
            node->numKeys++;
        } else {
            while (i >= 0 && key < node->keys[i]) i--;
            i++;
            if (node->children[i]->numKeys == 3) {
                splitChild(node, i, node->children[i]);
                if (key > node->keys[i]) i++;
            }
            insertNonFull(node->children[i], key);
        }
}

void insert(Node** root, int key) {
        if ((*root)->numKeys == 3) {
            Node* newRoot = createNode(0);
            newRoot->children[0] = *root;
            splitChild(newRoot, 0, *root);
            int i = (newRoot->keys[0] < key) ? 1 : 0;
            insertNonFull(newRoot->children[i], key);
            *root = newRoot;
```

```
        } else {
            insertNonFull(*root, key);
        }
}

void printTree(Node* root, int level) {
        if (root) {
            printf("Level %d: ", level);
            for (int i = 0; i < root->numKeys; i++) {
                printf("%d ", root->keys[i]);
            }
            printf("\n");
            for (int i = 0; i <= root->numKeys; i++) {
                printTree(root->children[i], level + 1);
            }
        }
}

int main() {
        Node* root = createNode(1);
        insert(&root, 10);
        insert(&root, 20);
```

```
    insert(&root, 5);
    insert(&root, 6);
    insert(&root, 12);
    insert(&root, 30);
    insert(&root, 25);

    printTree(root, 0);
    return 0;
}
```

OUTPUT:

```
/tmp/tIqvPEqFi4.o
Level 0: 5 6
Level 1: 5
Level 1: 6
Level 1: 10 25 30


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

2.WRITE C PROGRAM FOR SPLAY TREE

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

typedef struct Node {
```

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

Node* rightRotate(Node* root) {
    Node* newRoot = root->left;
    root->left = newRoot->right;
    newRoot->right = root;
    return newRoot;
}

Node* leftRotate(Node* root) {
    Node* newRoot = root->right;
    root->right = newRoot->left;
    newRoot->left = root;
    return newRoot;
}

Node* splay(Node* root, int key) {
    if (!root) return NULL;
    if (key < root->key) {
        if (!root->left) return root;
```

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

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

```
Node* delete(Node* root, int key) {
	if (!root) return NULL;
	root = splay(root, key);
	if (key != root->key) return root;
	Node* temp;
	if (!root->left) {
		temp = root->right;
		free(root);
		return temp;
	} else {
		temp = root->right;
		root = splay(root->left, key);
		root->right = temp;
		free(root);
		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 = delete(root, 20);
	root = search(root, 30);
	inorder(root);
	return 0;
}
```

output:

```
/tmp/EG4sdx5rYP.o
1 0 30

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

3. WRITE C PROGRAME FOR TRIE DATA STRUCTURE.

```
#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 *word) {
```

```
    TrieNode *node = root;
    while (*word) {
        int index = *word - 'a';
        if (!node->children[index])
            node->children[index] = createNode();
        node = node->children[index];
        word++;
    }
    node->isEndOfWord = 1;
}

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

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

int deleteHelper(TrieNode *node, const char *word, int depth) {
    if (!node) return 0;
    if (depth == strlen(word)) {
        if (node->isEndOfWord) {
            node->isEndOfWord = 0;
            return !hasChildren(node);
        }
        return 0;
    }
    int index = word[depth] - 'a';
    if (deleteHelper(node->children[index], word, depth + 1)) {
        free(node->children[index]);
        node->children[index] = NULL;
        return !node->isEndOfWord && !hasChildren(node);
    }
```

```
    return 0;
}

void prefixSearch(TrieNode *node, char *prefix, int level) {
    if (node->isEndOfWord) {
        prefix[level] = '\0';
        printf("%s\n", prefix);
    }
    for (int i = 0; i < ALPHABET_SIZE; i++) {
        if (node->children[i]) {
            prefix[level] = i + 'a';
            prefixSearch(node->children[i], prefix, level + 1);
        }
    }
}

void findWordsWithPrefix(TrieNode *root, const char *prefix) {
    TrieNode *node = root;
    while (*prefix) {
        int index = *prefix - 'a';
        if (!node->children[index]) return;
        node = node->children[index];
```

```
            prefix++;
        }
        char buffer[100];
        strcpy(buffer, prefix);
        prefixSearch(node, buffer, strlen(prefix));
}

int main() {
        TrieNode *root = createNode();
        insert(root, "hello");
        insert(root, "helium");
        insert(root, "hero");
        printf("Search 'hello': %d\n", search(root, "hello"));
        printf("Search 'he': %d\n", search(root, "he"));
        deleteHelper(root, "hello", 0);
        printf("Search 'hello' after deletion: %d\n", search(root, "hello"));
        printf("Words with prefix 'he':\n");
        findWordsWithPrefix(root, "he");
        return 0;
}
```

OUTPUT:

```
/tmp/ApMhnB8Q3t.o
Search 'hello': 1
Search 'he': 0
Search 'hello' after deletion: 0
Words with prefix 'he':
lium
ro


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