TRIE

TRIE

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define ALPHABET_SIZE 26
typedef struct TrieNode {
    struct TrieNode *children[ALPHABET_SIZE];
    bool isEndOfWord;
} TrieNode;
TrieNode* createNode() {
    TrieNode *node = (TrieNode *)malloc(sizeof(TrieNode));
    for (int i = 0; i < ALPHABET_SIZE; i++) {
        node->children[i] = NULL;
    }
    node->isEndOfWord = false;
    return node;
}
void insert(TrieNode *root, const char *word) {
    TrieNode *current = root;
    while (*word) {
        int index = *word - 'a';
        if (!current->children[index]) {
            current->children[index] = createNode();
        }
        current = current->children[index];
        word++;
    }
    current->isEndOfWord = true;
}
bool search(TrieNode *root, const char *word) {
    TrieNode *current = root;
    while (*word) {
        int index = *word - 'a';
```

```
        if (!current->children[index]) return false;
        current = current->children[index];
        word++;
    }
    return current->isEndOfWord;
}
bool hasChildren(TrieNode *node) {
    for (int i = 0; i < ALPHABET_SIZE; i++) {
        if (node->children[i]) return true;
    }
    return false;
}
bool deleteHelper(TrieNode *root, const char *word) {
    if (!root) return false;
    if (*word) {
        int index = *word - 'a';
        if (deleteHelper(root->children[index], word + 1)) {
            free(root->children[index]);
            root->children[index] = NULL;
            return !root->isEndOfWord && !hasChildren(root);
        }
    } else {
        if (root->isEndOfWord) {
            root->isEndOfWord = false;
            return !hasChildren(root);
        }
    }
    return false;
}
void deleteWord(TrieNode *root, const char *word) {
    deleteHelper(root, word);
}
void freeTrie(TrieNode *root) {
    if (!root) return;
    for (int i = 0; i < ALPHABET_SIZE; i++) {
        freeTrie(root->children[i]);
    }
```

```
    free(root);
}
int main() {
    TrieNode *root = createNode();
    insert(root, "hello");
    insert(root, "world");
    printf("Search 'hello': %s\n", search(root, "hello") ? "Found" : "Not Found");
    printf("Search 'world': %s\n", search(root, "world") ? "Found" : "Not Found");
    printf("Search 'helloworld': %s\n", search(root, "helloworld") ? "Found" : "Not Found");
    deleteWord(root, "world");
    printf("Search 'world' after deletion: %s\n", search(root, "world") ? "Found" : "Not Found");
    freeTrie(root);
    return 0;
}
```

OUTPUT:

```
Search 'hello': Found
Search 'world': Found
Search 'helloworld': Not Found
Search 'world' after deletion: Not Found
```

2-3 TREE:

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_KEYS 2
#define MAX_CHILDREN 3
typedef struct Node {
    int keys[MAX_KEYS];
    struct Node *children[MAX_CHILDREN];
    int numKeys;
    int isLeaf;
} Node;
Node* createNode(int isLeaf) {
    Node *newNode = (Node *)malloc(sizeof(Node));
    newNode->numKeys = 0;
    newNode->isLeaf = isLeaf;
    for (int i = 0; i < MAX_CHILDREN; i++) {
```

```
        newNode->children[i] = NULL;
    }
    return newNode;
}
void splitChild(Node *parent, int childIndex) {
    Node *fullChild = parent->children[childIndex];
    Node *newChild = createNode(fullChild->isLeaf);
    parent->children[childIndex] = createNode(0);
    Node *newParentChild = parent->children[childIndex];
    newParentChild->numKeys = 1;
    newParentChild->keys[0] = fullChild->keys[1];
    newChild->numKeys = 1;
    newChild->keys[0] = fullChild->keys[2];
    if (!fullChild->isLeaf) {
        newChild->children[0] = fullChild->children[2];
        newChild->children[1] = fullChild->children[3];
    }
    for (int i = parent->numKeys; i > childIndex; i--) {
        parent->children[i + 1] = parent->children[i];
    }
    parent->children[childIndex + 1] = newChild;
    fullChild->numKeys = 1;
}
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 == 2) {
            splitChild(node, i);
            if (key > node->keys[i]) {
                i++;
            }
        }
        insertNonFull(node->children[i], key);
    }
}
void insert(Node **root, int key) {
    Node *r = *root;
    if (r->numKeys == 2) {
        Node *newRoot = createNode(0);
        *root = newRoot;
        newRoot->children[0] = r;
        splitChild(newRoot, 0);
        insertNonFull(newRoot, key);
    } else {
        insertNonFull(r, key);
    }
}
void inOrder(Node *node) {
    int i;
    if (node != NULL) {
        for (i = 0; i < node->numKeys; i++) {
            if (!node->isLeaf) {
                inOrder(node->children[i]);
            }
            printf("%d ", node->keys[i]);
        }
        if (!node->isLeaf) {
            inOrder(node->children[i]);
        }
    }
}
void freeTree(Node *node) {
    if (node != NULL) {
```

```
        for (int i = 0; i <= node->numKeys; i++) {
            freeTree(node->children[i]);
        }
        free(node);
    }
}
int main() {
    Node *root = createNode(1);
    insert(&root, 10);
    insert(&root, 20);
    insert(&root, 5);
    insert(&root, 6);
    insert(&root, 15);
    insert(&root, 30);
    printf("In-order traversal of the 2-3 tree:\n");
    inOrder(root);
    printf("\n");
    freeTree(root);
    return 0;
}
```

OUTPUT:

```
In-order traversal of the 2-3 tree:
5 6 10 15 20 30
```

2-3-4 TREE:

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_KEYS 3
#define MAX_CHILDREN 4
typedef struct Node {
    int keys[MAX_KEYS];
    struct Node *children[MAX_CHILDREN];
    int numKeys;
    int isLeaf;
} Node;
Node* createNode(int isLeaf) {
```

```
    Node *newNode = (Node *)malloc(sizeof(Node));
    newNode->numKeys = 0;
    newNode->isLeaf = isLeaf;
    for (int i = 0; i < MAX_CHILDREN; i++) {
        newNode->children[i] = NULL;
    }
    return newNode;
}
void splitChild(Node *parent, int index) {
    Node *fullChild = parent->children[index];
    Node *newChild = createNode(fullChild->isLeaf);
    parent->children[index] = createNode(0);
    Node *newParentChild = parent->children[index];
    newParentChild->numKeys = 1;
    newParentChild->keys[0] = fullChild->keys[1];

    newChild->numKeys = 2;
    newChild->keys[0] = fullChild->keys[2];
    newChild->keys[1] = fullChild->keys[3];
    if (!fullChild->isLeaf) {
        newChild->children[0] = fullChild->children[2];
        newChild->children[1] = fullChild->children[3];
        newChild->children[2] = fullChild->children[4];
    }
    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; i > index; i--) {
        parent->keys[i] = parent->keys[i - 1];
    }
    parent->keys[index] = fullChild->keys[0];

    parent->numKeys++;
    fullChild->numKeys = 1;
}
```

```
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 == MAX_KEYS) {
            splitChild(node, i);
            if (key > node->keys[i]) {
                i++;
            }
        }
        insertNonFull(node->children[i], key);
    }
}
void insert(Node **root, int key) {
    Node *r = *root;
    if (r->numKeys == MAX_KEYS) {
        Node *newRoot = createNode(0);
        *root = newRoot;
        newRoot->children[0] = r;
        splitChild(newRoot, 0);
        insertNonFull(newRoot, key);
    } else {
        insertNonFull(r, key);
    }
}
void inOrder(Node *node) {
    if (node != NULL) {
```

```
        int i;
        for (i = 0; i < node->numKeys; i++) {
            if (!node->isLeaf) {
                inOrder(node->children[i]);
            }
            printf("%d ", node->keys[i]);
        }
        if (!node->isLeaf) {
            inOrder(node->children[i]);
        }
    }
}
void freeTree(Node *node) {
    if (node != NULL) {
        for (int i = 0; i <= node->numKeys; i++) {
            freeTree(node->children[i]);
        }
        free(node);
    }
}

int main() {
    Node *root = createNode(1);
    insert(&root, 10);
    insert(&root, 20);
    insert(&root, 5);
    insert(&root, 6);
    insert(&root, 15);
    insert(&root, 30);
    insert(&root, 25);
    printf("In-order traversal of the 2-3-4 tree:\n");
    inOrder(root);
    printf("\n");
    freeTree(root);
    return 0;
}
```

OUTPUT:

In-order traversal of the 2-3-4 tree:

5 6 10 15 20 25 30