IMPLEMENTATION OF TRIE

```
#include <stdio.h>

#include <string.h>

#include <stdlib.h>


#define ALPHABET_SIZE 26


struct node {

        int data;

        struct node* link[ALPHABET_SIZE];

};


struct node* root = NULL;


struct node* create_node() {

        struct node *q = (struct node*) malloc(sizeof(struct node));

        int x;

        for (x = 0; x < ALPHABET_SIZE; x++)

                q->link[x] = NULL;

        q->data = -1;

        return q;

}


void insert_node(char key[]) {

        int length = strlen(key);

        int index;
```

```
	int level = 0;
	if (root == NULL)
		root = create_node();
	struct node *q = root; // For insertion of each String key, we will start from the root

	for (; level < length; level++) {
		index = key[level] - 'a';

		if (q->link[index] == NULL) {
			q->link[index] = create_node(); // which is : struct node *p = create_node(); q->link[index]
= p;
		}

		q = q->link[index];
	}
	q->data = level; // Assuming the value of this particular String key is 11
}

int search(char key[]) {
	struct node *q = root;
	int length = strlen(key);
	int level = 0;
	for (; level < length; level++) {
		int index = key[level] - 'a';
		if (q->link[index] != NULL)
			q = q->link[index];
```

```
            else
                break;
        }
        if (key[level] == '\0' && q->data != -1)
            return q->data;
        return -1;
}

int main(int argc, char **argv) {
        insert_node("by");
        insert_node("program");
        insert_node("programming");
        insert_node("data structure");
        insert_node("coding");
        insert_node("code");
        printf("Searched value: %d\n", search("code"));
        printf("Searched value: %d\n", search("geeks"));
        printf("Searched value: %d\n", search("coding"));
        printf("Searched value: %d\n", search("programming"));
        return 0;
}
```

OUTPUT

Searched value: 4

Searched value: -1

Searched value: 6

Searched value: 11

IMPLEMENTATION OF TRIE 2-3 tree

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

struct node {
	int n;
	int keys[M-1];
	struct node *p[M];
}*root=NULL;

enum KeyStatus { Duplicate,SearchFailure,Success,InsertIt,LessKeys };

void insert(int key);
void display(struct node *root,int);
void DelNode(int x);
enum KeyStatus ins(struct node *r, int x, int* y, struct node** u);
int searchPos(int x,int *key_arr, int n);
enum KeyStatus del(struct node *r, int x);
void eatline(void);


int main()
{
	int key;
	int choice;
```

```
while(1)
{
        printf("1.insert\n");
        printf("2.delete\n");
        printf("3.display\n");
        printf("4.exit\n");
        printf("enter the choice: ");
        scanf("%d",&choice); eatline();
        switch(choice)
        {
        case 1:
                printf("enter the choice: ");
                scanf("%d",&key); eatline();
                insert(key);
                break;
        case 2:
                printf("enter the choice: ");
                scanf("%d",&key); eatline();
                DelNode(key);
                break;
        case 3:
                printf("23 tree :\n");
                display(root,0);
                break;
        case 4:
                exit(1);
```

```
            default:
                printf("no option avilable.\n");
                break;
            }
        }
        return 0;
}

void insert(int key)
{
        struct node *newnode;
        int upKey;
        enum KeyStatus value;
        value = ins(root, key, &upKey, &newnode);
        if (value == Duplicate)
            printf("the numbers already exists\n");
        if (value == InsertIt)
        {
            struct node *uproot = root;
            root=malloc(sizeof(struct node));
            root->n = 1;
            root->keys[0] = upKey;
            root->p[0] = uproot;
            root->p[1] = newnode;
        }
}
```

```
enum KeyStatus ins(struct node *ptr, int key, int *upKey,struct node **newnode)
{
	struct node *newPtr, *lastPtr;
	int pos, i, n,splitPos;
	int newKey, lastKey;
	enum KeyStatus value;
	if (ptr == NULL)
	{
		*newnode = NULL;
		*upKey = key;
		return InsertIt;
	}
	n = ptr->n;
	pos = searchPos(key, ptr->keys, n);
	if (pos < n && key == ptr->keys[pos])
		return Duplicate;
	value = ins(ptr->p[pos], key, &newKey, &newPtr);
	if (value != InsertIt)
		return value;
	if (n < M - 1)
	{
		pos = searchPos(newKey, ptr->keys, n);
		for (i=n; i>pos; i--)
		{
			ptr->keys[i] = ptr->keys[i-1];
```

```
            ptr->p[i+1] = ptr->p[i];
        }
        ptr->keys[pos] = newKey;
        ptr->p[pos+1] = newPtr;
        ++ptr->n;
        return Success;
}
if (pos == M - 1)
{
        lastKey = newKey;
        lastPtr = newPtr;
}
else
{
        lastKey = ptr->keys[M-2];
        lastPtr = ptr->p[M-1];
        for (i=M-2; i>pos; i--)
        {
            ptr->keys[i] = ptr->keys[i-1];
            ptr->p[i+1] = ptr->p[i];
        }
        ptr->keys[pos] = newKey;
        ptr->p[pos+1] = newPtr;
}
splitPos = (M - 1)/2;
(*upKey) = ptr->keys[splitPos];
```

```
        (*newnode)=malloc(sizeof(struct node));
        ptr->n = splitPos;
        (*newnode)->n = M-1-splitPos;
        for (i=0; i < (*newnode)->n; i++)
        {
                (*newnode)->p[i] = ptr->p[i + splitPos + 1];
                if(i < (*newnode)->n - 1)
                        (*newnode)->keys[i] = ptr->keys[i + splitPos + 1];
                else
                        (*newnode)->keys[i] = lastKey;
        }
        (*newnode)->p[(*newnode)->n] = lastPtr;
        return InsertIt;
}

void display(struct node *ptr, int blanks)
{
        if (ptr)
        {
                int i;
                for(i=1; i<=blanks; i++)
                        printf(" ");
                for (i=0; i < ptr->n; i++)
                        printf("%d ",ptr->keys[i]);
                printf("\n");
```

```
        for (i=0; i <= ptr->n; i++)
            display(ptr->p[i], blanks+10);
    }
}


int searchPos(int key, int *key_arr, int n)
{
    int pos=0;
    while (pos < n && key > key_arr[pos])
        pos++;
    return pos;
}

void DelNode(int key)
{
    struct node *uproot;
    enum KeyStatus value;
    value = del(root,key);
    switch (value)
    {
    case SearchFailure:
        printf("number %d not found\n",key);
        break;
    case LessKeys:
        uproot = root;
```

```
            root = root->p[0];
            free(uproot);
            break;
        }
}
enum KeyStatus del(struct node *ptr, int key)
{
        int pos, i, pivot, n ,min;
        int *key_arr;
        enum KeyStatus value;
        struct node **p,*lptr,*rptr;

        if (ptr == NULL)
            return SearchFailure;
        n=ptr->n;
        key_arr = ptr->keys;
        p = ptr->p;
        min = (M - 1)/2;

        pos = searchPos(key, key_arr, n);
        if (p[0] == NULL)
        {
            if (pos == n || key < key_arr[pos])
                return SearchFailure;
            for (i=pos+1; i < n; i++)
            {
```

```
            key_arr[i-1] = key_arr[i];
            p[i] = p[i+1];
        }
        return --ptr->n >= (ptr==root ? 1 : min) ? Success : LessKeys;
}

if (pos < n && key == key_arr[pos])
{
        struct node *qp = p[pos], *qp1;
        int nkey;
        while(1)
        {
            nkey = qp->n;
            qp1 = qp->p[nkey];
            if (qp1 == NULL)
                break;
            qp = qp1;
        }
        key_arr[pos] = qp->keys[nkey-1];
        qp->keys[nkey - 1] = key;
}
value = del(p[pos], key);
if (value != LessKeys)
        return value;

if (pos > 0 && p[pos-1]->n > min)
```

```
{
        pivot = pos - 1;
        lptr = p[pivot];
        rptr = p[pos];

        rptr->p[rptr->n + 1] = rptr->p[rptr->n];
        for (i=rptr->n; i>0; i--)
        {
                rptr->keys[i] = rptr->keys[i-1];
                rptr->p[i] = rptr->p[i-1];
        }
        rptr->n++;
        rptr->keys[0] = key_arr[pivot];
        rptr->p[0] = lptr->p[lptr->n];
        key_arr[pivot] = lptr->keys[--lptr->n];
        return Success;
}

if (pos < n && p[pos + 1]->n > min)
{
        pivot = pos;
        lptr = p[pivot];
        rptr = p[pivot+1];

        lptr->keys[lptr->n] = key_arr[pivot];
        lptr->p[lptr->n + 1] = rptr->p[0];
```

```
        key_arr[pivot] = rptr->keys[0];
        lptr->n++;
        rptr->n--;
        for (i=0; i < rptr->n; i++)
        {
                rptr->keys[i] = rptr->keys[i+1];
                rptr->p[i] = rptr->p[i+1];
        }
        rptr->p[rptr->n] = rptr->p[rptr->n + 1];
        return Success;
}

if(pos == n)
        pivot = pos-1;
else
        pivot = pos;

lptr = p[pivot];
rptr = p[pivot+1];

lptr->keys[lptr->n] = key_arr[pivot];
lptr->p[lptr->n + 1] = rptr->p[0];
for (i=0; i < rptr->n; i++)
{
        lptr->keys[lptr->n + 1 + i] = rptr->keys[i];
        lptr->p[lptr->n + 2 + i] = rptr->p[i+1];
```

```
        }
        lptr->n = lptr->n + rptr->n +1;
        free(rptr);
        for (i=pos+1; i < n; i++)
        {
                key_arr[i-1] = key_arr[i];
                p[i] = p[i+1];
        }
        return --ptr->n >= (ptr == root ? 1 : min) ? Success : LessKeys;
}


void eatline(void) {
    char c;
    printf("");
    while (c=getchar()!='\n') ;
}
```

OUTPUT

```
1.insert
2.delete
3.display
4.exit
enter the choice: 1
enter the choice: 2     3     4
1.insert
2.delete
3.display
```

4.exit

enter the choice: 2

enter the choice: 2

1.insert

2.delete

3.display

4.exit

enter the choice: 3

23 tree

IMPLEMENTATION 2-3-4 TREE

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

#define MAX_KEYS 3
#define MAX_CHILDREN 4

struct Node {
    int num_keys;
    int keys[MAX_KEYS];
    struct Node* children[MAX_CHILDREN];
};

struct Node* createNode(int key);
void insert(int key, struct Node** root);
void splitChild(struct Node* parent, int childIndex);
void insertNonFull(struct Node* node, int key);
```

```
void display(struct Node* root);

// Function to create a new node
struct Node* createNode(int key) {
	struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
	newNode->num_keys = 1;
	newNode->keys[0] = key;
	for (int i = 0; i < MAX_CHILDREN; i++) {
		newNode->children[i] = NULL;
	}
	return newNode;
}

// Function to insert a key into the tree
void insert(int key, struct Node** root) {
	if (*root == NULL) {
		*root = createNode(key);
	} else {
		if ((*root)->num_keys == MAX_KEYS) {
			struct Node* newRoot = createNode((*root)->keys[1]);
			newRoot->children[0] = *root;
			splitChild(newRoot, 0);
			insertNonFull(newRoot, key);
			*root = newRoot;
		} else {
			insertNonFull(*root, key);
```

```
            }
        }
}

// Function to split a child node
void splitChild(struct Node* parent, int childIndex) {
        struct Node* child = parent->children[childIndex];
        struct Node* newChild = createNode(child->keys[2]);
        parent->children[childIndex + 1] = newChild;
        child->num_keys = 1;
        newChild->num_keys = 1;
        newChild->keys[0] = child->keys[2];
        child->keys[2] = 0;
}

// Function to insert a key into a non-full node
void insertNonFull(struct Node* node, int key) {
        int i = node->num_keys - 1;
        if (node->children[0] == NULL) {
                while (i >= 0 && key < node->keys[i]) {
                        node->keys[i + 1] = node->keys[i];
                        i--;
                }
                node->keys[i + 1] = key;
                node->num_keys++;
        } else {
```

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

// Function to display the tree
void display(struct Node* root) {
    if (root != NULL) {
        for (int i = 0; i < root->num_keys; i++) {
            display(root->children[i]);
            printf("%d ", root->keys[i]);
        }
        display(root->children[root->num_keys]);
    }
}

int main() {
```

```
    struct 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);
    insert(3, &root);

    display(root);

    return 0;
}
```

OUTPUT

3 5 10 7

=== Code Execution Successful ===