Application of Linked List : Addition of Two Polynomial Equation

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

// Node structure for the linked list
struct Node {
    int coefficient;
    int exponent;
    struct Node* next;
};

// Function to create a new node
struct Node* createNode(int coeff, int exp) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->coefficient = coeff;
    newNode->exponent = exp;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a term into the polynomial
void insertTerm(struct Node** poly, int coeff, int exp) {
    struct Node* temp = createNode(coeff, exp);
    temp->next = *poly;
    *poly = temp;
}

// Function to add two polynomials
struct Node* addPolynomials(struct Node* poly1, struct Node* poly2) {
    struct Node* result = NULL;

    while (poly1 != NULL || poly2 != NULL) {
        int coeff1 = (poly1 != NULL) ? poly1->coefficient : 0;
        int exp1 = (poly1 != NULL) ? poly1->exponent : 0;

        int coeff2 = (poly2 != NULL) ? poly2->coefficient : 0;

        int sumCoeff = coeff1 + coeff2;

        insertTerm(&result, sumCoeff, exp1);

        if (poly1 != NULL) poly1 = poly1->next;
        if (poly2 != NULL) poly2 = poly2->next;
    }

    return result;
}

// Function to display a polynomial
void displayPolynomial(struct Node* poly) {
    while (poly != NULL) {
        printf("%dx^%d ", poly->coefficient, poly->exponent);
        if (poly->next != NULL)
            printf("+ ");
```

```
        poly = poly->next;
    }
    printf("\n");
}

int main() {
    struct Node* poly1 = NULL;
    struct Node* poly2 = NULL;

    // Taking input for the first polynomial
    int n1, coeff, exp;
    printf("Enter the number of terms in Polynomial 1: ");
    scanf("%d", &n1);

    printf("Enter the coefficients and exponents for Polynomial 1:\n");
    int i; // Declare i outside the loop
    for (i = 0; i < n1; ++i) {
        printf("Term %d: ", i + 1);
        scanf("%d %d", &coeff, &exp);
        insertTerm(&poly1, coeff, exp);
    }

    // Taking input for the second polynomial
    int n2;
    printf("Enter the number of terms in Polynomial 2: ");
    scanf("%d", &n2);

    printf("Enter the coefficients and exponents for Polynomial 2:\n");
    for (i = 0; i < n2; ++i) {
        printf("Term %d: ", i + 1);
        scanf("%d %d", &coeff, &exp);
        insertTerm(&poly2, coeff, exp);
    }

    // Displaying the input polynomials
    printf("Polynomial 1: ");
    displayPolynomial(poly1);
    printf("Polynomial 2: ");
    displayPolynomial(poly2);

    // Adding two polynomials
    struct Node* result = addPolynomials(poly1, poly2);
    printf("Resultant Polynomial: ");
    displayPolynomial(result);

    return 0;
}
```

Conversion of Infix expression to Postfix expression

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
int isOperator(char ch) {
    return (ch == '+' || ch == '-' || ch == '*' || ch == '/');
}
int getPrecedence(char op) {
    if (op == '+' || op == '-')
        return 1;
    else if (op == '*' || op == '/')
        return 2;
    else
        return 0;
}
void infixToPostfix(char infix[], char postfix[]) {
    char stack[100];
    int top = -1;
    int i, j;
    for (i = 0, j = 0; infix[i] != '\0'; i++) {
        char ch = infix[i];
        if (isalnum(ch)) {
            postfix[j++] = ch;
        } else if (ch == '(') {
            stack[++top] = ch;
        } else if (ch == ')') {
            while (top >= 0 && stack[top] != '(') {
                postfix[j++] = stack[top--];
            }
            top--;
        } else if (isOperator(ch)) {
            while (top >= 0 && getPrecedence(stack[top]) >= getPrecedence(ch)) {
                postfix[j++] = stack[top--];
            }
            stack[++top] = ch;
        }
    }
    while (top >= 0) {
        postfix[j++] = stack[top--];
    }
    postfix[j] = '\0';
}
int main() {
    char infix[100], postfix[100];
    printf("Enter an infix expression: ");
    gets(infix);
    infixToPostfix(infix, postfix);
    printf("Postfix expression: %s\n", postfix);
    return 0;
}
```

Evaluation of Postfix Expression

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#define MAX 100
int stack[MAX];
int top = -1;
void push(int item) {
    if (top == MAX - 1) {
        printf("Stack Overflow\n");
        exit(1);
    }
    stack[++top] = item;
}
int pop() {
    if (top == -1) {
        printf("Stack Underflow\n");
        exit(1);
    }
    return stack[top--];
}
int evaluatePostfix(char* exp) {
    int i, op1, op2, len;
    char ch;
    len = strlen(exp);
    for (i = 0; i < len; i++) {
        ch = exp[i];
        if (isdigit(ch)) {
            push(ch - '0');
        } else {
            op2 = pop();
            op1 = pop();
            switch (ch) {
                case '+':
                    push(op1 + op2);
                    break;
                case '-':
                    push(op1 - op2);
                    break;
                case '*':
                    push(op1 * op2);
                    break;
                case '/':
                    push(op1 / op2);
                    break;
            }
        }
    }
    return pop();
}
int main() {
    char exp[MAX];
    printf("Enter postfix expression: ");
    gets(exp);
    printf("Result: %d\n", evaluatePostfix(exp));
    return 0;
}
```

Quick Sort (Recursion)

```c
#include <stdio.h>

void quicksort(int [], int, int);
int partition(int [], int, int);

int main() {
    int arr[10], n, i;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    printf("Enter the elements:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    printf("Before sorting:\n");
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    quicksort(arr, 0, n - 1);
    printf("\nAfter sorting:\n");
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}

void quicksort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quicksort(arr, low, pi - 1);
        quicksort(arr, pi + 1, high);
    }
}

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
    return (i + 1);
}
```

Application of Queue: Breadth First Search Graph Traversal Technique

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

#define MAX 100

int adj[MAX][MAX];
int visited[MAX];
int queue[MAX];
int front = -1, rear = -1;

void BFS(int start, int vertices) {
    visited[start] = 1;
    queue[++rear] = start;
    while (front != rear) {
        int current = queue[++front];
        printf("%d ", current);
        for (int i = 0; i < vertices; i++) {
            if (adj[current][i] == 1 && visited[i] == 0) {
                visited[i] = 1;
                queue[++rear] = i;
            }
        }
    }
}

int main() {
    int vertices;
    printf("Enter the number of vertices: ");
    scanf("%d", &vertices);
    printf("Enter the adjacency matrix:\n");
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
            scanf("%d", &adj[i][j]);
        }
    }
    int start;
    printf("Enter the starting vertex: ");
    scanf("%d", &start);
    BFS(start, vertices);
    return 0;
}
```

6. Insertion, Deletion and Traversal operations on a Binary Search Tree.

```
#include <stdio.h>
#include <stdlib.h>  // Include for malloc
#include <conio.h>
struct node {
    int data;
    struct node* left;
    struct node* right;
};
struct node* createNode(int data) {
    struct node* newNode = (struct node*)malloc(sizeof(struct node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
struct node* insert(struct node* 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 node* findMin(struct node* root) {
    while (root->left != NULL) {
        root = root->left;
    }
    return root;
}
struct node* deleteNode(struct node* root, int data) {
    if (root == NULL) {
        return root;
    }
    if (data < root->data) {
        root->left = deleteNode(root->left, data);
    } else if (data > root->data) {
        root->right = deleteNode(root->right, data);
    } else {
        if (root->left == NULL) {
            struct node* temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            struct node* temp = root->left;
            free(root);
            return temp;
        }
        struct node* temp = findMin(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}
void inorderTraversal(struct node* root) {
```

```c
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d ", root->data);
        inorderTraversal(root->right);
    }
}
int main() {
    struct node* root = NULL;
    int choice, data;
    do {
        printf("\nBinary Search Tree Operations:\n");
        printf("1. Insert\n");
        printf("2. Delete\n");
        printf("3. Inorder Traversal\n");
        printf("4. 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);
                break;
            case 2:
                printf("Enter data to delete: ");
                scanf("%d", &data);
                root = deleteNode(root, data);
                break;
            case 3:
                printf("Inorder Traversal: ");
                inorderTraversal(root);
                printf("\n");
                break;
            case 4:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice. Please enter a valid option.\n");
        }
    } while (choice != 4);
    getch();  // For Turbo C++ 3.2
    return 0;
}
```

7. Insertion and Traversal operations on an AVL Tree.

```
#include <stdio.h>
#include <stdlib.h>
struct node {
    int data;
    struct node *left;
    struct node *right;
    int height;
};
int max(int a, int b) {
    return (a > b) ? a : b;
}

int height(struct node *N) {
    if (N == NULL)
        return 0;
    return N->height;
}
struct node *newNode(int data) {
    struct node *node = (struct node *)malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    node->height = 1;
    return (node);
}
struct node *rightRotate(struct node *y) {
    struct node *x = y->left;
    struct node *T2 = x->right;

    x->right = y;
    y->left = T2;

    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;

    return x;
}
struct node *leftRotate(struct node *x) {
    struct node *y = x->right;
    struct node *T2 = y->left;

    y->left = x;
    x->right = T2;

    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;

    return y;
}
int getBalance(struct node *N) {
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

struct node *insert(struct node *node, int data) {
    if (node == NULL)
        return (newNode(data));
```

```
    if (data < node->data)
        node->left = insert(node->left, data);
    else if (data > node->data)
        node->right = insert(node->right, data);
    else
        return node;

    node->height = 1 + max(height(node->left), height(node->right));

    int balance = getBalance(node);

    if (balance > 1 && data < node->left->data)
        return rightRotate(node);

    if (balance < -1 && data > node->right->data)
        return leftRotate(node);

    if (balance > 1 && data > node->left->data) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    if (balance < -1 && data < node->right->data) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;
}
void preOrder(struct node *root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preOrder(root->left);
        preOrder(root->right);
    }
}
int main() {
    struct node *root = NULL;
    int n, data;

    printf("Enter the number of elements to be inserted: ");
    scanf("%d", &n);

    printf("Enter %d elements to be inserted:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &data);
        root = insert(root, data);
    }
    printf("Preorder traversal of the constructed AVL tree is \n");
    preOrder(root);
    return 0;
}
```

9. Implementation of Hashing Techniques -
Linear Probing , Quadratic Probing and Separate Chaining method.

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 10
struct Node {
    int data;
    struct Node* next;
};
void linearProbing(int hashTable[], int key);
void quadraticProbing(int hashTable[], int key);
void separateChaining(struct Node* hashTable[], int key);
void displayHashTable(int hashTable[]);
void displayHashTableSeparateChaining(struct Node* hashTable[]);
int main() {
    int hashTableLinear[SIZE] = {0};
    int hashTableQuadratic[SIZE] = {0};
    struct Node* hashTableSeparateChaining[SIZE] = {NULL};
    int choice, key;
    do {
        printf("\n1. Linear Probing\n2. Quadratic Probing\n3. Separate Chaining\n4. Display Hash Tables\n5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter key to insert using Linear Probing: ");
                scanf("%d", &key);
                linearProbing(hashTableLinear, key);
                break;
            case 2:
                printf("Enter key to insert using Quadratic Probing: ");
                scanf("%d", &key);
                quadraticProbing(hashTableQuadratic, key);
                break;
            case 3:
                printf("Enter key to insert using Separate Chaining: ");
                scanf("%d", &key);
                separateChaining(hashTableSeparateChaining, key);
                break;
            case 4:
                printf("\nLinear Probing Hash Table:\n");
                displayHashTable(hashTableLinear);
                printf("\nQuadratic Probing Hash Table:\n");
                displayHashTable(hashTableQuadratic);
                printf("\nSeparate Chaining Hash Table:\n");
                displayHashTableSeparateChaining(hashTableSeparateChaining);
                break;
            case 5:
                printf("Exiting program...\n");
                break;
            default:
                printf("Invalid choice! Please enter a valid option.\n");
        }
    } while (choice != 5);
    return 0;
}
void linearProbing(int hashTable[], int key) {
    int index = key % SIZE;
    int i = index;
```

```
    while (hashTable[i] != 0) {
        i = (i + 1) % SIZE;
        if (i == index) {
            printf("Hash table is full. Unable to insert %d.\n", key);
            return;
        }
    }
    hashTable[i] = key;
    printf("%d inserted at index %d using Linear Probing.\n", key, i);
}
void quadraticProbing(int hashTable[], int key) {
    int index = key % SIZE;
    int i = index;
    int count = 1;
    while (hashTable[i] != 0) {
        i = (index + count * count) % SIZE;
        count++;
        if (count > SIZE) {
            printf("Hash table is full. Unable to insert %d.\n", key);
            return;
        }
    }
    hashTable[i] = key;
    printf("%d inserted at index %d using Quadratic Probing.\n", key, i);
}
void separateChaining(struct Node* hashTable[], int key) {
    int index = key % SIZE;
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = key;
    newNode->next = NULL;
    if (hashTable[index] == NULL) {
        hashTable[index] = newNode;
    } else {
        struct Node* temp = hashTable[index];
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
    printf("%d inserted at index %d using Separate Chaining.\n", key, index);
}
void displayHashTable(int hashTable[]) {
    for (int i = 0; i < SIZE; i++) {
        printf("%d\t", hashTable[i]);
    }
    printf("\n");
}
void displayHashTableSeparateChaining(struct Node* hashTable[]) {
    for (int i = 0; i < SIZE; i++) {
        printf("Index %d: ", i);
        struct Node* temp = hashTable[i];
        while (temp != NULL) {
            printf("%d -> ", temp->data);
            temp = temp->next;
        }
        printf("NULL\n");
    }
}
```

8. Application of Binary Heap: Heap Sort.

```
#include <stdio.h>
void heapify(int arr[], int n, int i) {
    int temp, maximum, left_index, right_index;
    maximum = i;
    right_index = 2 * i + 2;
    left_index = 2 * i + 1;
    if (left_index < n && arr[left_index] > arr[maximum])
        maximum = left_index;
    if (right_index < n && arr[right_index] > arr[maximum])
        maximum = right_index;
    if (maximum != i) {
        temp = arr[i];
        arr[i] = arr[maximum];
        arr[maximum] = temp;
        heapify(arr, n, maximum);
    }
}
void heapsort(int arr[], int n) {
    int i, temp;
    for (i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }
    for (i = n - 1; i > 0; i--) {
        temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;
        heapify(arr, i, 0);
    }
}
int main() {
    int arr[100], n, i;
    printf("Enter the number of elements in the array: ");
    scanf("%d", &n);
    printf("Enter the elements of the array: ");
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    printf("Original Array: ");
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    heapsort(arr, n);
    printf("Array after performing heap sort: ");
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

10. Implementation of Brute force String searching technique.

```
#include <stdio.h>
#include <string.h>
void search(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);
    for (int i = 0; i <= N - M; i++) {
        int j;
        for (j = 0; j < M; j++)
            if (txt[i + j] != pat[j])
                break;
        if (j == M)
            printf("Pattern found at index %d\n", i);
    }
}
int main()
{
    char txt[100];
    char pat[100];
    printf("Enter the text: ");
    scanf("%s", txt);
    printf("Enter the pattern to search: ");
    scanf("%s", pat);
    search(pat, txt);
    return 0;
}
```

Uday..