INSERTION SORT

```
#include<stdio.h>

void Insertion_Sort(int arr[], int size){
    for (int i = 1; i < size; i++){
        int temp = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > temp){
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = temp;
    }
}
int main(){
    int arr[9] = {3, 1, 4, 1, 5, 9, 2, 6, 5};
    int size = sizeof(arr) / sizeof(int);
    Insertion_Sort(arr, size);
    for (int i = 0; i < size; i++){
        printf("%d ", arr[i]);
    }
    return 0;
}
```

MERGE SORT

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
void Merge(int A[], int TmpArray[], int Lpos, int Rpos, int RightEnd){
    int LeftEnd, NumElements, TmpPos;

    LeftEnd = Rpos - 1;
    TmpPos = Lpos;
    NumElements = RightEnd - Lpos + 1;


    while (Lpos <= LeftEnd && Rpos <= RightEnd){
        if (A[Lpos] <= A[Rpos]){
            TmpArray [TmpPos] = A[Lpos];
            TmpPos++;
            Lpos++;
        }
        else {
            TmpArray[TmpPos] = A[Rpos];
            TmpPos++;
            Rpos++;
        }
    }

    while (Lpos <= LeftEnd){
        TmpArray[TmpPos] = A[Lpos];
        TmpPos++;
        Lpos++;
    }

    while (Rpos <= RightEnd)
    {
        TmpArray[TmpPos] = A[Rpos];
        TmpPos++;
        Rpos++;
    }

    for (int i = 0; i < NumElements; i++, RightEnd--){
        A[RightEnd] = TmpArray[RightEnd];
    }
}
void MSort(int A[], int TmpArray[], int Left, int Right) {
    // Function implementation goes here
    int Center;

    if (Left < Right){
        Center = (Left + Right) / 2;
        MSort(A, TmpArray, Left, Center);
        MSort(A, TmpArray, Center + 1, Right);
        Merge(A, TmpArray, Left, Center + 1, Right);
    }
}

void Mergesort(int A[], int N){
    int *TmpArray;

    TmpArray = malloc (N * sizeof(int));
    if (TmpArray != NULL){
        MSort(A, TmpArray, 0, N-1);
        free (TmpArray);
    }
    else {
        printf("No space for tmp array!!!");
    }
}

void printArray(int A[], int N){
    for (int i = 0; i < N; i++){
        printf("%d ", A[i]);
    }
    printf("\n");
}
```

```
int main() {
    int array[] = {3, 1, 4, 1, 5, 9, 2, 6, 5};

    // int array[] = {3, 1, 4, 1, 5, 9, 2, 6};
    int size = sizeof(array) / sizeof(int);
    printf("The original array is: ");
    printArray(array, size);
    Mergesort(array, size);
    printf("The sorted array is: ");
    printArray(array, size);
    return 0;
}
```

QUICK SORT (1ST ELEMENT)

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

void swap(int *a, int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Partition function
int Partition(int A[], int Left, int Right){
    // initialize pivot to be the first element
    int pivot = A[Left];
    int i = Left;
    int j = Right;
    while (i < j){
        // condition 1: find the first element greater than
        // the pivot (from starting)
        do {
            i++;
        } while (A[i] <= pivot);

        // condition 2: find the first element smaller than
        // the pivot (from last)
        do {
            j--;
        } while (A[j] > pivot);

        if (i < j){
            swap(&A[i], &A[j]);
        }
    }
    swap(&A[Left], &A[j]);
    return j;
}

// QuickSort function
void QuickSort(int A[], int Left, int Right){
    if (Left < Right){

        // call Partition function to find Partition Index
        int pivotIndex = Partition(A, Left, Right);

        // Recursively call quickSort() for left and right
        // half based on partition Index
        QuickSort(A, Left, pivotIndex);
        QuickSort(A, pivotIndex + 1, Right);
    }
}

void PrintArray(int A[], int N){
    for (int i = 0; i < N; i++){
        printf("%d ", A[i]);
    }
}
int main(){
    // int A[] = {3, 1, 4, 1, 5, 9, 2, 6, 5};
    int A[] = {0,1,1,0,1,2,1,2,0,0,0,1};
    int N = sizeof(A) / sizeof(int);
    printf("Before Sorting: ");
    PrintArray(A, N);
    QuickSort(A, 0, N);
    printf("\n");
    printf("After Sorting: ");
    PrintArray(A, N);
    return 0;
}
```

QUICK SORT (MEDIAN OF THREE)

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

```
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Median of three pivot selection function
int medianOfThree(int A[], int left, int right) {
    int mid = left + (right - left) / 2;
    if (A[left] > A[mid]) {
        swap(&A[left], &A[mid]);
    }
    if (A[left] > A[right]) {
        swap(&A[left], &A[right]);
    }
    if (A[mid] > A[right]) {
        swap(&A[mid], &A[right]);
    }
    return mid; // Return the median element (middle index) as the pivot
}

// Partition function (using median-of-three pivot)
int Partition(int A[], int Left, int Right) {
    // Select median of three as the pivot
    int pivotIndex = medianOfThree(A, Left, Right);
    int pivot = A[pivotIndex];

    swap(&A[pivotIndex], &A[Right]); // Move pivot to the end for easier partitioning

    int i = Left - 1; // Index of element smaller than pivot
    int j = Left;     // Index to iterate through the array

    while (j < Right) {
        if (A[j] <= pivot) {
            i++;
            swap(&A[i], &A[j]);
        }
        j++;
    }

    swap(&A[i + 1], &A[Right]); // Move pivot to its final position
    return i + 1;
}

// QuickSort function
void QuickSort(int A[], int Left, int Right) {
    if (Left < Right) {
        int pivotIndex = Partition(A, Left, Right);

        // Recursively call quickSort() for left and right subarrays
        QuickSort(A, Left, pivotIndex - 1);
        QuickSort(A, pivotIndex + 1, Right);
    }
}

int main() {
    int A[] = {3, 1, 4, 1, 5, 9, 2, 6, 5};
    int n = sizeof(A) / sizeof(A[0]);

    printf("Unsorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", A[i]);
    }
    printf("\n");

    QuickSort(A, 0, n - 1);

    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", A[i]);
    }
    printf("\n");

    return 0;
}
```

BUCKET SORT (BINSORT)

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

// Function to sort using bucket sort
void bucketSort(int array[], int n) {
    // Create n empty buckets
    int max_value = array[0];
    for (int i = 1; i < n; i++) {
        if (array[i] > max_value) {
            max_value = array[i];
        }
    }
    int bucket_count = max_value / 10 + 1;
    int** buckets = (int**)malloc(bucket_count * sizeof(int*));
    int* bucket_sizes = (int*)calloc(bucket_count, sizeof(int));

    // Initialize each bucket as empty
    for (int i = 0; i < bucket_count; i++) {
        buckets[i] = (int*)malloc(n * sizeof(int));
```

```
    }

    // Put elements into the buckets
    for (int i = 0; i < n; i++) {
        int bucket_index = array[i] / 10;
        buckets[bucket_index][bucket_sizes[bucket_index]] = array[i];
        bucket_sizes[bucket_index]++;
    }

    // Sort elements within each bucket using insertion sort
    for (int i = 0; i < bucket_count; i++) {
        for (int j = 1; j < bucket_sizes[i]; j++) {
            int key = buckets[i][j];
            int k = j - 1;
            while (k >= 0 && buckets[i][k] > key) {
                buckets[i][k + 1] = buckets[i][k];
                k--;
            }
            buckets[i][k + 1] = key;
        }
    }

    // Concatenate the sorted buckets
    int index = 0;
    for (int i = 0; i < bucket_count; i++) {
        for (int j = 0; j < bucket_sizes[i]; j++) {
            array[index++] = buckets[i][j];
        }
        free(buckets[i]);
    }
    free(buckets);
    free(bucket_sizes);
}

// Function to print an array
void printArray(int array[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");
}

// Driver code
int main() {
    int array[] = {3, 1, 4, 1, 5, 9, 2, 6, 5};
    int n = sizeof(array) / sizeof(array[0]);

    printf("Original array: ");
    printArray(array, n);
```

```
    bucketSort(array, n);

    printf("Sorted array: ");
    printArray(array, n);

    return 0;
}
```

SORT THIS ARRAY BY GPA OF THE STUDENT IN DESCENDING ORDER USING QUICK SORT WITH MEDIAN-OF-THREE

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAXSTUDENTS 100  // maximum number of students

typedef struct {
    int student_id;
    char *full_name;
    int age;
    float gpa;
} Student;

void inputStudent(Student *s){
    printf("Enter the id: ");
    scanf("%d", &s->student_id);
    printf("Enter the name: ");
    char name[100];
    scanf("%s", name);
    s->full_name = (char *)malloc(strlen(name) + 1);
    strcpy(s->full_name, name);
    printf("Enter the age: ");
    scanf("%d", &s->age);
    printf("Enter the gpa: ");
    scanf("%f", &s->gpa);
}

void printStudent(Student *s){
    printf("ID: %d\t", s->student_id);
    printf("Name: %s\t", s->full_name);
    printf("Age: %d\t", s->age);
    printf("GPA: %.3f\n", s->gpa);
}
```

```
void inputMultipleStudents(Student* students, int *N){
    for (int i = 0; i < *N; i++){
        printf("Enter the information of student %d: \n", i+1);
        inputStudent(&students[i]);
    }
}


void swap(Student *a, Student *b) {
    Student temp = *a;
    *a = *b;
    *b = temp;
}

// Quick Sort Algorithm for descending order

// Median of three pivot selection function
int medianOfThree(Student A[], int left, int right) {
    int mid = left + (right - left) / 2;
    if (A[left].gpa < A[mid].gpa) {
        swap(&A[left], &A[mid]);
    }
    if (A[left].gpa < A[right].gpa) {
        swap(&A[left], &A[right]);
    }
    if (A[mid].gpa < A[right].gpa) {
        swap(&A[mid], &A[right]);
    }
    return mid; // Return the median element (middle index) as the pivot
}

// Partition function (using median-of-three pivot)
int Partition(Student A[], int Left, int Right) {
    // Select median of three as the pivot
    int pivotIndex = medianOfThree(A, Left, Right);
    float pivot = A[pivotIndex].gpa;

    swap(&A[pivotIndex], &A[Right]); // Move pivot to the end for easier partitioning

    int i = Left - 1; // Index of element smaller than pivot
    int j = Left;     // Index to iterate through the array

    while (j < Right) {
        if (A[j].gpa >= pivot) {
            i++;
            swap(&A[i], &A[j]);
        }
        j++;
    }
    swap(&A[i + 1], &A[Right]); // Move pivot to its final position
    return i + 1;
}

// QuickSort function
void QuickSort(Student A[], int Left, int Right) {
    if (Left < Right) {
        int pivotIndex = Partition(A, Left, Right);

        // Recursively call quickSort() for left and right subarrays
        QuickSort(A, Left, pivotIndex - 1);
        QuickSort(A, pivotIndex + 1, Right);
    }
}


int main(){
    Student students[MAXSTUDENTS];
    int N;
    printf("Enter the number of students: ");
    scanf("%d", &N);
    inputMultipleStudents(students, &N);
    for (int i = 0; i < N; i++){
        printf("Information of student %d: \n", i+1);
        printStudent(&students[i]);
    }

    printf("Before Sorting: \n");
    for (int i = 0; i < N; i++){
        printStudent(&students[i]);
    }
    printf("\n");

    // Sort the students based on their GPA in descending order
    QuickSort(students, 0, N - 1);

    printf("After Sorting: \n");
    for (int i = 0; i < N; i++){
        printStudent(&students[i]);
    }
    return 0;
}
```

SINGLE LINKED LIST

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

```
struct Node{
    int data;
    struct Node* next;
};
typedef struct Node* PtrToNode;
typedef PtrToNode List;
typedef PtrToNode Position;

List head = NULL;

List createEmptyList(){
    return NULL;
}

PtrToNode createNode(int data){
    struct Node* new_node = (struct Node*) malloc (sizeof(struct Node));
    new_node->data = data;
    new_node->next = NULL;
    return new_node;
}

List insertAtBeginning(int data){
    struct Node* new_node = createNode(data);
    if (head == NULL){
        head = new_node;
    }
    else{
        new_node->next = head;
        head = new_node;
    }
    return head;
}


List insertAtEnd(int data){
    struct Node* new_node = createNode(data);
    struct Node* temp = head;
    if (head == NULL){
        head = new_node;
    }
    else{
        while(temp->next != NULL){
            temp = temp->next;
        }
        temp->next = new_node;
    }
    return head;
}
```

```
void deleteNode(int position){
    struct Node* temp = head;
    if (head == NULL){
        return;
    }
    if (position == 0){
        head = head->next;
        free(temp);
        return;
    }
    for (int i = 0; temp != NULL && i < position - 1; i++){
        temp = temp->next;
    }
    if (temp == NULL || temp->next == NULL){
        return;
    }
    struct Node* next = temp->next->next;
    free(temp->next);
    temp->next = next;
}

void access_pos(int position){
    struct Node* temp = head;
    for (int i = 0; temp != NULL && i < position; i++){
        temp = temp->next;
    }
    if (temp == NULL){
        printf("Invalid position\n");
    }
    else{
        printf("Data at position %d is %d\n", position, temp->data);
    }
}

void access_element(int data){
    struct Node* temp = head;
    int position = 0;
    while (temp != NULL){
        if (temp->data == data){
            printf("Data %d found at position %d\n", data, position);
            return;
        }
        temp = temp->next;
        position++;
    }
    printf("Data %d not found\n", data);
}

void printList(){
    struct Node* temp = head;
```

```
    while (temp != NULL){
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main(){
    head = createEmptyList();
    head = insertAtBeginning(1);
    head = insertAtBeginning(2);
    head = insertAtEnd(3);
    head = insertAtEnd(4);
    printList();        // 2 1 3 4
    deleteNode(1);
    printList();        // 2 3 4
    return 0;
}
```

ADDITION OF POLYNOMIALS USING SINGLE LINKED LIST

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

struct Node{
    int coef;
    int exp;
    struct Node* next;
};

typedef struct Node* PtrToNode;
struct Node* createNode(int coef, int exp){
    struct Node* new_node = (struct Node*) malloc (sizeof(struct Node));
    new_node->coef = coef;
    new_node->exp = exp;
    new_node->next = NULL;
    return new_node;
}
void insert(PtrToNode *poly, int coef, int exp){
    PtrToNode new_node = createNode(coef, exp);
    if (*poly == NULL){
        *poly = new_node;
    }
    else{
        PtrToNode temp = *poly;
        while (temp->next != NULL){
            temp = temp->next;
        }
        temp->next = new_node;
    }
}

void print(PtrToNode poly){
    PtrToNode temp = poly;
    while (temp != NULL){
        printf("%dx^%d", temp->coef, temp->exp);
        temp = temp->next;
        if (temp != NULL){
            printf(" + ");
        }
    }
    printf("\n");
}

PtrToNode add(PtrToNode poly1, PtrToNode poly2){
    PtrToNode result = NULL;
    PtrToNode temp1 = poly1;
    PtrToNode temp2 = poly2;
    while (temp1 != NULL && temp2 != NULL){
        if (temp1->exp == temp2->exp){
            insert(&result, temp1->coef + temp2->coef, temp1->exp);
            temp1 = temp1->next;
            temp2 = temp2->next;
        }
        else if (temp1->exp > temp2->exp){
            insert(&result, temp1->coef, temp1->exp);
            temp1 = temp1->next;
        }
        else {
            insert(&result, temp2->coef, temp2->exp);
            temp2 = temp2->next;
        }
    }

    while (temp1 != NULL){
        insert(&result, temp1->coef, temp1->exp);
        temp1 = temp1->next;
    }

    while (temp2 != NULL){
        insert(&result, temp2->coef, temp2->exp);
        temp2 = temp2->next;
    }

    return result;
}
```

```
//
int main(){
    PtrToNode poly1 = NULL;

    PtrToNode poly2 = NULL;


    insert(&poly1, 5, 2);
    insert(&poly1, 4, 1);
    insert(&poly1, 2, 0);

    insert(&poly2, 5, 1);
    insert(&poly2, 5, 0);

    print(poly1);
    print(poly2);

    PtrToNode result =  NULL;
    result = add(poly1, poly2);
    print(result);


    return 0;
}
```

DOUBLY LINKED LIST

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};
typedef struct Node* PtrToNode;
typedef struct Node* List;

List head = NULL;
List tail = NULL;
List createEmptyList(){
   return NULL;
}

PtrToNode createNode(int data){
    struct Node* new_node = (struct Node*) malloc (sizeof(struct Node));
    new_node->data = data;
    new_node->next = NULL;
    new_node->prev = NULL;
    return new_node;
}
List insertAtBeginning(int data){
    struct Node* new_node = createNode(data);
    if (head == NULL){
        head = new_node;
        tail = new_node;
    }
    else{
        new_node->next = head;
        head->prev = new_node;
        head = new_node;
    }
    return head;
}

List insertAtEnd(int data){
    struct Node* new_node = createNode(data);
    if (tail == NULL){
        head = new_node;
        tail = new_node;
    }
    else{
        new_node->prev = tail;
        tail->next = new_node;
        tail = new_node;
    }
    return tail;
}

void deleteNode(int position){
    struct Node* temp = head;
    if (head == NULL){
        return;
    }
    if (position == 0){
        head = head->next;
        head->prev = NULL;
        free(temp);
        return;
    }
    for (int i = 0; temp != NULL && i < position - 1; i++){
        temp = temp->next;
    }
    if (temp == NULL || temp->next == NULL){
        return;
```

```
    }
    struct Node* next = temp->next->next;
    free(temp->next);
    temp->next = next;
    next->prev = temp;
}

void access_pos(int position){
    struct Node* temp = head;
    for (int i = 0; temp != NULL && i < position; i++){
        temp = temp->next;
    }
    if (temp == NULL){
        printf("Out of range\n");
    }
    else{
        printf("Data at position %d is %d\n", position, temp->data);
    }
}

void access_element(int data){
    struct Node* temp = head;
    int position = 0;
    while (temp != NULL){
        if (temp->data == data){
            printf("Data %d found at position %d\n", data, position);
            return;
        }
        temp = temp->next;
        position++;
    }
    printf("Data %d not found\n", data);
}

void printList_forward(){
    struct Node* temp = head;
    while(temp != NULL){
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

void printList_backward(){
    struct Node* temp = tail;
    while(temp != NULL){
        printf("%d ", temp->data);
        temp = temp->prev;
    }
    printf("\n");
```

```
}

void findNode(int data){
    struct Node* temp = head;
    int position = 0;
    while(temp != NULL){
        if (temp->data == data){
            printf("Found at position %d\n", position);
        }
        temp = temp->next;
        position++;
    }
}


int main(){
    List head = createEmptyList();
    List tail = createEmptyList();
    head = insertAtBeginning(1);
    head = insertAtBeginning(4);
    head = insertAtBeginning(3);
    tail = insertAtEnd(4);
    tail = insertAtEnd(5);
    tail = insertAtEnd(6);
    printList_forward();
    printList_backward();
    deleteNode(2);
    printList_forward();
    printList_backward();
    findNode(4);
    return 0;
}
```

CIRCULAR LINKED LIST

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

// Node structure
struct Node {
    int data;
    struct Node* next;
};

// Function to insert a node at the beginning of the circular linked list
void insertAtBeginning(struct Node** head, int data) {
    // Create a new node
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
```

```
    // If the list is empty, make the new node as the head and point its next to itself
    if (*head == NULL) {
        *head = newNode;
        newNode->next = newNode;
    } else {
        // Find the last node in the list
        struct Node* last = *head;
        while (last->next != *head) {
            last = last->next;
        }

        // Make the new node as the new head and point its next to the previous head
        newNode->next = *head;

        // Make the last node point to the new head
        last->next = newNode;

        // Update the head pointer
        *head = newNode;
    }
}

// Function to display the circular linked list
void display(struct Node* head) {
    if (head == NULL) {
        printf("Circular linked list is empty.\n");
        return;
    }

    struct Node* temp = head;
    do {
        printf("%d ", temp->data);
        temp = temp->next;
    } while (temp != head);
    printf("\n");
}

int main() {
    struct Node* head = NULL;

    // Insert nodes at the beginning of the circular linked list
    insertAtBeginning(&head, 5);
    insertAtBeginning(&head, 4);
    insertAtBeginning(&head, 3);
    insertAtBeginning(&head, 2);
    insertAtBeginning(&head, 1);

    // Display the circular linked list
    printf("Circular linked list: ");
    display(head);
```

```
    return 0;
}
```

MERGE SORT LINKED LIST

```
// C code for linked list merged sort
#include <stdio.h>
#include <stdlib.h>

/* Link list node */
struct Node {
    int data;
    struct Node* next;
};

/* function prototypes */
struct Node* SortedMerge(struct Node* a, struct Node* b);
void FrontBackSplit(struct Node* source,
                    struct Node** frontRef, struct Node** backRef);

/* sorts the linked list by changing next pointers (not data) */
void MergeSort(struct Node** headRef)
{
    struct Node* head = *headRef;
    struct Node* a;
    struct Node* b;

    /* Base case -- length 0 or 1 */
    if ((head == NULL) || (head->next == NULL)) {
        return;
    }

    /* Split head into 'a' and 'b' sublists */
    FrontBackSplit(head, &a, &b);

    /* Recursively sort the sublists */
    MergeSort(&a);
    MergeSort(&b);

    /* answer = merge the two sorted lists together */
    *headRef = SortedMerge(a, b);
}

/* See https://www.geeksforgeeks.org/merge-two-sorted-linked-lists/
for details of this function */
struct Node* SortedMerge(struct Node* a, struct Node* b)
{
    struct Node* result = NULL;
```

```
    /* Base cases */
    if (a == NULL)
        return (b);
    else if (b == NULL)
        return (a);

    /* Pick either a or b, and recur */
    if (a->data <= b->data) {
        result = a;
        result->next = SortedMerge(a->next, b);
    }
    else {
        result = b;
        result->next = SortedMerge(a, b->next);
    }
    return (result);
}

/* UTILITY FUNCTIONS */
/* Split the nodes of the given list into front and back halves,
    and return the two lists using the reference parameters.
    If the length is odd, the extra node should go in the front list.
    Uses the fast/slow pointer strategy. */
void FrontBackSplit(struct Node* source,
                    struct Node** frontRef, struct Node** backRef)
{
    struct Node* fast;
    struct Node* slow;
    slow = source;
    fast = source->next;

    /* Advance 'fast' two nodes, and advance 'slow' one node */
    while (fast != NULL) {
        fast = fast->next;
        if (fast != NULL) {
            slow = slow->next;
            fast = fast->next;
        }
    }

    /* 'slow' is before the midpoint in the list, so split it in two
    at that point. */
    *frontRef = source;
    *backRef = slow->next;
    slow->next = NULL;
}

/* Function to print nodes in a given linked list */
void printList(struct Node* node)
{
    while (node != NULL) {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Function to insert a node at the beginning of the linked list */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list of the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Driver program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct Node* res = NULL;
    struct Node* a = NULL;

    /* Let us create a unsorted linked lists to test the functions
Created lists shall be a: 2->3->20->5->10->15 */
    push(&a, 15);
    push(&a, 10);
    push(&a, 5);
    push(&a, 20);
    push(&a, 3);
    push(&a, 2);

    /* Sort the above created Linked List */
    MergeSort(&a);

    printf("Sorted Linked List is: \n");
    printList(a);

    getchar();
    return 0;
}
```

STACK IMPLEMENTATION

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

struct Node {
    int data;
    struct Node* next;
};
typedef struct Node* PtrToNode;
typedef struct Node* Stack;
struct Node *top = NULL;

Stack CreateStack(){
    Stack S = malloc(sizeof(struct Node));
    if (S == NULL){
        printf("Out of space!!!\n");
    }
    S->next = NULL;
    return S;
}

void Push(Stack S, int x){
    PtrToNode TmpCell;
    TmpCell = malloc(sizeof(struct Node));
    if (TmpCell == NULL){
        printf("Out of space!!!\n");
    }
    else {
        TmpCell->data = x;
        TmpCell->next = S->next;
        S->next = TmpCell;
    }
}

int isEmpty(Stack S){
    return S->next == NULL;
}

int Pop(Stack S){
    PtrToNode FirstCell;
    int x;
    if (isEmpty(S)){
        printf("Empty stack\n");
        return 0;
    }
    else {
        FirstCell = S->next;
        x = FirstCell->data;
        S->next = S->next->next;
        free(FirstCell);
    }
    return x;
}

int Top(Stack S){
    if (!isEmpty(S)){
        return S->next->data;
    }
    else {
        printf("Empty stack\n");
        return 0;
    }
}

int main(){
    Stack S = CreateStack();
    Push(S, 1);
    Push(S, 2);
    Push(S, 3);
    printf("%d\n", Pop(S));
    printf("%d\n", Pop(S));
    printf("%d\n", Pop(S));
    printf("%d\n", Pop(S));
    return 0;
}
```

QUEUE IMPLEMENTATION

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

struct Node {
    int data;
    struct Node* next;
};
typedef struct Node* PtrToNode;

struct queue {
    PtrToNode Head;
    PtrToNode Tail;
};
typedef struct queue *Queue;

Queue CreateQueue(){
    Queue Q = malloc(sizeof(struct queue));
    Q->Head = malloc(sizeof(struct Node));
    Q->Head->next = NULL;
    Q->Tail = Q->Head;
```

```
    return Q;
}

// Enqueue is similar to push
void Enqueue(Queue Q, int x){
    PtrToNode TmpCell = malloc(sizeof(struct Node));
    if (TmpCell == NULL){
        printf("Out of space!!!");
    }
    else{
        TmpCell->data = x;
        TmpCell->next = Q->Tail->next;
        Q->Tail->next = TmpCell;
        Q->Tail = TmpCell;
    }
}

int IsEmpty(Queue Q){
    return Q->Head->next == NULL;
}

int top_Queue(Queue Q){
    if (IsEmpty(Q)){
        printf("Empty queue\n");
        return -1;
    }
    return Q->Head->next->data;
}
// Dequeue is similar to pop
int Dequeue(Queue Q){
    PtrToNode FirstCell;
    int x;
    if (IsEmpty(Q)){
        printf("Empty queue\n");
    }
    else{
        FirstCell = Q->Head->next;
        x = FirstCell->data;
        Q->Head->next = Q->Head->next->next;
        free(FirstCell);
    }
    return x;
}



int main(){
    Queue Q = CreateQueue();
    Enqueue(Q, 3);
    Enqueue(Q, 5);
    Enqueue(Q, 2);
    Enqueue(Q, 15);
    Enqueue(Q, 42);
    Dequeue(Q);
    Dequeue(Q);
    Enqueue(Q, 14);
    Enqueue(Q, 7);
    Dequeue(Q);
    Enqueue(Q, 9);
    Dequeue(Q);
    Dequeue(Q);
    Enqueue(Q, 51);
    Dequeue(Q);
    //Dequeue(Q);
    // printf("%d\n", Dequeue(Q));
    // printf("%d\n", Dequeue(Q));
    // printf("%d\n", Dequeue(Q));
    printf("%d\n", Dequeue(Q));
    return 0;
}
```

INFIX TO POSTFIX

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

struct Node {
    char data;
    struct Node* next;
};
typedef struct Node* Stack;
typedef struct Node* PtrToNode;

void Push(Stack S, char c){
    PtrToNode TmpCell;
    TmpCell = (struct Node*)malloc(sizeof(struct Node));
    if (TmpCell == NULL){
        printf("Out of space!!!");
    }
    else {
        TmpCell->data = c;
        TmpCell->next = S->next;
        S->next = TmpCell;
    }
}

int IsEmpty(Stack S){
```

```
    return S->next == NULL;
}

char Pop(Stack S){
    PtrToNode FirstCell = (struct Node*)malloc(sizeof(struct Node));
    char c;
    if (IsEmpty(S)){
        printf("Empty stack!!!");
    }
    else{
        FirstCell = S->next;
        c = FirstCell->data;
        S->next = S->next->next;
    }
    return c;
}

int prec(char c){
    if (c == '^') return 3;
    else if (c == '*' || c == '/') return 2;
    else if (c == '+' || c == '-') return 1;
    return -1;
}

char associativity(char c){
    if (c == '^') return 'r';
    return 'l';
}
void infixToPostfix(char* str, char* postfix){
    Stack S = (struct Node*)malloc(sizeof(struct Node));
    S->next = NULL;
    int k = 0;
    for (int i = 0; i < strlen(str); i++){
        char c = str[i];
        if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c >= '0' && c <= '9')){
            postfix[k] = c;
            k++;
        }
        else if (c == '('){
            Push(S, c);
        }
        else if (c == ')'){
            while (!IsEmpty(S) && S->next->data != '('){
                postfix[k] = Pop(S);
                k++;
            }
            if (!IsEmpty(S) && S->next->data == '('){
                Pop(S);
            }
        }
```

```
        else {
            while (!IsEmpty(S) && (prec(c) < prec(S->next->data) || (prec(c) ==
prec(S->next->data) && associativity(c) == 'l'))){
                postfix[k] = Pop(S);
                k++;
            }
            Push(S, c);
        }
    }
    while (!IsEmpty(S)){
        postfix[k] = Pop(S);
        k++;
    }
    postfix[k] = '\0';
}

int calculate(char* postfix) {
    Stack S = (struct Node*)malloc(sizeof(struct Node));
    S->next = NULL;
    int i = 0;
    while (postfix[i] != '\0') {
        char c = postfix[i];
        if (c >= '0' && c <= '9') {
            Push(S, c);
        } else {
            int operand2 = Pop(S) - '0';
            int operand1 = Pop(S) - '0';
            int result;
            switch (c) {
                case '+':
                    result = operand1 + operand2;
                    break;
                case '-':
                    result = operand1 - operand2;
                    break;
                case '*':
                    result = operand1 * operand2;
                    break;
                case '/':
                    result = operand1 / operand2;
                    break;
                case '^':
                    result = 1;
                    for (int j = 0; j < operand2; j++) {
                        result *= operand1;
                    }
                    break;
            }
            Push(S, result + '0');
        }
```

```
        i++;
    }
    int finalResult = Pop(S) - '0';
    return finalResult;
}

int main(){
    char str[] = "5*(6+2)-12/4";
    char postfix[strlen(str)];
    infixToPostfix(str, postfix);
    printf("%s", postfix);
    printf("\n");
    printf("%d", calculate(postfix));
    return 0;
}
```

SYMBOL BALANCING

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

struct Node {
    char data;
    struct Node* next;
};
typedef struct Node* Stack;
typedef struct Node* PtrToNode;

void Push(Stack S, char c){
    PtrToNode TmpCell;
    TmpCell = malloc(sizeof(struct Node));
    if (TmpCell == NULL){
        printf("Out of space!!!");
    }
    else {
        TmpCell->data = c;
        TmpCell->next = S->next;
        S->next = TmpCell;
    }
}

int IsEmpty(Stack S){
    return S->next == NULL;
}
```

```
char Pop(Stack S){
    PtrToNode FirstCell = malloc(sizeof(struct Node));
    char c;
    if (IsEmpty(S)){
        printf("Empty stack!!!");
    }
    else{
        FirstCell = S->next;
        c = FirstCell->data;
        S->next = S->next->next;
    }
    return c;
}
void symbol(char *str){
    Stack S = malloc(sizeof(struct Node));
    S->next = NULL;
    int check = 0;
    for (int i = 0; i < strlen(str); i++){
        if (str[i] == '[' || str[i] == '{' || str[i] == '('){
            Push(S, str[i]);
        }
        else if (str[i] == ']' || str[i] == '}' || str[i] == ')'){
            if (IsEmpty(S)){
                printf("Not balancing\n");
                return;
            }
            else {
                char c;
                c = Pop(S);
                if ((c == '[' && str[i] == ']') || (c == '{' && str[i] == '}') || (c == '(' && str[i] == ')')){
                    check = 1;
                }
                else {
                    printf("Not balancing\n");
                    return;
                }
            }
        }
    }
    if (IsEmpty(S) && check == 1){
        printf("Symbol balancing\n");
    }
    else{
        printf("Not balancing\n");
    }
}
```

```
int main(){
    char str1[] = "[a+b{1*2]9*1}+(2-1)";
    char str2[] = "[a+b{1*2}9*1]+(2-1)";
    char str3[] = "[ { [ ( )] } ]";
    char str4[] = "[{]}()";
    symbol(str1);
    symbol(str2);
    symbol(str3);
    symbol(str4);
    return 0;
}
```

BINARY TREE

```
#include <stdio.h>
#include<stdlib.h>
// struct TreeNode {
//     int number;
//     PtrToNode FirstChild;
//     PtrToNode NextSibling;
// };
// typedef struct TreeNode *PtrToNode;

struct BinaryTreeNode {
    int data;
    struct BinaryTreeNode * left;
    struct BinaryTreeNode * right;
};
typedef struct BinaryTreeNode * PtrToNode;

void PreOrder(struct BinaryTreeNode * root){
    if (root){
        printf("%d -> ", root->data);
        PreOrder(root->left);
        PreOrder(root->right);
    }
}

void InOrder(struct BinaryTreeNode * root){
    if (root){
        InOrder(root->left);
        printf("%d -> ", root->data);
        InOrder(root->right);
    }
}

void PostOrder(struct BinaryTreeNode * root){
    if (root){
        PostOrder(root->left);
        PostOrder(root->right);
        printf("%d -> ", root->data);
    }
}

int main(){
    PtrToNode root = (PtrToNode)malloc(sizeof(struct BinaryTreeNode));
    root->data = 1;
    root->left = (PtrToNode)malloc(sizeof(struct BinaryTreeNode));
    root->left->data = 2;
    root->left->left = NULL;
    root->left->right = NULL;
    root->right = (PtrToNode)malloc(sizeof(struct BinaryTreeNode));
    root->right->data = 3;
    root->right->left = NULL;
    root->right->right = NULL;
    PreOrder(root);
    printf("\n");
    InOrder(root);
    printf("\n");
    PostOrder(root);
    return 0;
}
```

BINARY SEARCH TREE

```
#include <stdio.h>
#include <stdlib.h>
struct BinaryTreeNode {
    int data;
    struct BinaryTreeNode * left;
    struct BinaryTreeNode * right;
};
typedef struct BinaryTreeNode * PtrToNode;

//31, 45, 36, 14, 52, 42, 6, 21, 73, 47, 26, 37, 33, 8
PtrToNode InsertNode(PtrToNode root, int data) {
    if (root == NULL) {
        // Create a new node
        PtrToNode newNode = (PtrToNode)malloc(sizeof(struct BinaryTreeNode));
        newNode->data = data;
        newNode->left = NULL;
        newNode->right = NULL;
        return newNode;
    }
```

```
    else {
        // Recursively insert the node in the left or right subtree
        if (data < root->data) {
            root->left = InsertNode(root->left, data);
        } else {
            root->right = InsertNode(root->right, data);
        }
        return root;
    }
}

// print the tree in-order
void displayTree(PtrToNode root) {
    if (root) {
        displayTree(root->left);
        printf("%d -> ", root->data);
        displayTree(root->right);
    }
}

void search(PtrToNode root, int data) {
    if (root == NULL) {
        printf("Not found\n");
    } else if (root->data == data) {
        printf("Found\n");
    } else if (data < root->data) {
        search(root->left, data);
    } else {
        search(root->right, data);
    }
}

struct Node* deleteNode(PtrToNode root, int k) {
    // Base case
    if (root == NULL)
        return root;

    // If the key to be deleted is smaller than the root's key, then it lies in the left subtree
    if (k < root->data) {
        root->left = deleteNode(root->left, k);
        return root;
    }
    // If the data to be deleted is greater than the root's data, then it lies in the right subtree
    else if (k > root->data) {
        root->right = deleteNode(root->right, k);
        return root;
    }

    // If data is same as root's data, then this is the node to be deleted
    // Node with only one child or no child
```

```
    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;
    }

    // Node with two children: Get the inorder successor (smallest in the right subtree)
    PtrToNode succParent = root;
    PtrToNode succ = root->right;
    while (succ->left != NULL) {
        succParent = succ;
        succ = succ->left;
    }

    // Copy the inorder successor's content to this node
    root->data = succ->data;

    // Delete the inorder successor
    if (succParent->left == succ)
        succParent->left = succ->right;
    else
        succParent->right = succ->right;

    free(succ);
    return root;
}

int main(){
    PtrToNode root = NULL;
    int array[14] = {31, 45, 36, 14, 52, 42, 6, 21, 73, 47, 26, 37, 33, 8};
    for (int i = 0; i < 14; i++) {
        root = InsertNode(root, array[i]);
    }
    search(root, 100);
    displayTree(root);
    printf("\n");
    deleteNode(root, 14);
    displayTree(root);
    return 0;
}
```

PREORDER NON-RECURSIVE TREE

```
#include <stdio.h>
#include <stdlib.h>
struct BinaryTreeNode {
    int data;
    struct BinaryTreeNode * left;
    struct BinaryTreeNode * right;
};
typedef struct BinaryTreeNode * PtrToNode;
typedef struct BinaryTreeNode * Stack;

void PreOrder(struct BinaryTreeNode * root){
    Stack S = malloc(sizeof(struct BinaryTreeNode));
    S->right = NULL;
    S->left = NULL;
    S->data = 0;
    PtrToNode TmpCell;
    TmpCell = malloc(sizeof(struct BinaryTreeNode));
    TmpCell = root;
    while (TmpCell != NULL || S->data != 0){
        while (TmpCell != NULL){
            printf("%d -> ", TmpCell->data);
            S->data++;
            S->left = TmpCell;
            S = S->left;
            TmpCell = TmpCell->left;
        }
        if (S->data != 0){
            TmpCell = S->right;
            S = S->right;
            S->data--;
        }
    }
}

int main(){
    PtrToNode root = (PtrToNode)malloc(sizeof(struct BinaryTreeNode));
    root->data = 1;
    root->left = (PtrToNode)malloc(sizeof(struct BinaryTreeNode));
    root->left->data = 2;
    root->left->left = NULL;
    root->left->right = NULL;
    root->right = (PtrToNode)malloc(sizeof(struct BinaryTreeNode));
    root->right->data = 3;
    root->right->left = NULL;
    root->right->right = NULL;
    PreOrder(root);
    return 0;
}
```

AVL TREE

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

// Structure for a node
struct AVLNode {
    int key;
    struct AVLNode *left;
    struct AVLNode *right;
    int height;
};
typedef struct AVLNode* PtrToNode;
// Function to get the height of a node
int height(PtrToNode node) {
    if (node == NULL)
        return 0;
    return node->height;
}

// Function to get the maximum of two integers
int max(int a, int b) {
    return (a > b) ? a : b;
}

// Function to create a new node
struct AVLNode *newNode(int key) {
    struct AVLNode *node = (struct AVLNode *)malloc(sizeof(struct AVLNode));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1;
    return node;
}
struct AVLNode * minValueNode(struct AVLNode* node)
{
    struct AVLNode* current = node;

    /* loop down to find the leftmost leaf */
    while (current->left != NULL)
        current = current->left;

    return current;
}
// Function to right rotate subtree rooted with y
struct AVLNode *rightRotate(PtrToNode y) {
    PtrToNode x = y->left;
    PtrToNode T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

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

    // Return new root
    return x;
}

// Function to left rotate subtree rooted with x
```

```
struct AVLNode *leftRotate(PtrToNode x) {
    PtrToNode y = x->right;
    PtrToNode T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

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

    // Return new root
    return y;
}

// Function to get the balance factor of a node
int getBalance(PtrToNode node) {
    if (node == NULL)
        return 0;
    return height(node->left) - height(node->right);
}

// Function to insert a node into the AVL tree
struct AVLNode *insert(PtrToNode node, int key) {
    // Perform the normal BST insertion
    if (node == NULL)
        return newNode(key);

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else // Duplicate keys are not allowed
        return node;

    // Update the height of the ancestor node
    node->height = 1 + max(height(node->left), height(node->right));

    // Get the balance factor of the ancestor node
    int balance = getBalance(node);

    // If the node becomes unbalanced, there are four cases

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    // Return the unchanged node pointer
    return node;
}

// Function to print the AVL tree in inorder traversal
void inorder(PtrToNode root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}

struct AVLNode* deleteNode(PtrToNode root, int key)
{
    // STEP 1: PERFORM STANDARD BST DELETE

    if (root == NULL)
        return root;

    // If the key to be deleted is smaller than the
    // root's key, then it lies in left subtree
    if ( key < root->key )
        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater than the
    // root's key, then it lies in right subtree
    else if( key > root->key )
        root->right = deleteNode(root->right, key);

    // if key is same as root's key, then This is
    // the node to be deleted
    else
    {
        // node with only one child or no child
        if( (root->left == NULL) || (root->right == NULL) )
        {
            struct AVLNode *temp = root->left ? root->left :
                                   root->right;

            // No child case
            if (temp == NULL)
            {
                temp = root;
                root = NULL;
            }
            else // One child case
             *root = *temp; // Copy the contents of
                            // the non-empty child
            free(temp);
        }
        else
        {
            // node with two children: Get the inorder
            // successor (smallest in the right subtree)
            struct AVLNode* temp = minValueNode(root->right);

            // Copy the inorder successor's data to this node
```

```
            root->key = temp->key;

            // Delete the inorder successor
            root->right = deleteNode(root->right, temp->key);
        }
    }

    // If the tree had only one node then return
    if (root == NULL)
      return root;

    // STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
    root->height = 1 + max(height(root->left),
                           height(root->right));

    // STEP 3: GET THE BALANCE FACTOR OF THIS NODE (to
    // check whether this node became unbalanced)
    int balance = getBalance(root);

    // If this node becomes unbalanced, then there are 4 cases

    // Left Left Case
    if (balance > 1 && getBalance(root->left) >= 0)
        return rightRotate(root);

    // Left Right Case
    if (balance > 1 && getBalance(root->left) < 0)
    {
        root->left =  leftRotate(root->left);
        return rightRotate(root);
    }

    // Right Right Case
    if (balance < -1 && getBalance(root->right) <= 0)
        return leftRotate(root);

    // Right Left Case
    if (balance < -1 && getBalance(root->right) > 0)
    {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}
// Driver program to test the AVL tree functions
int main() {
    struct Node *root = NULL;

    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

    printf("Inorder traversal of the constructed AVL tree is: ");
    inorder(root);

    return 0;
}
```

DIJKSTRA ALGORITHM

Pseudocode

```
function Dijkstra(graph, source):
    dist = array of infinity values, size of graph
    dist[source] = 0
    priorityQueue = empty priority queue
    priorityQueue.insert(source, 0)

    while priorityQueue is not empty:
        u = priorityQueue.extractMin()

        for each neighbor v of u:
            alt = dist[u] + weight(u, v)
            if alt < dist[v]:
                dist[v] = alt
                priorityQueue.insert(v, alt)

    return dist
```

C Program

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

#define V 9

int minDistance(int dist[], bool sptSet[]) {
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++)
        if (sptSet[v] == false && dist[v] <= min)
            min = dist[v], min_index = v;
    return min_index;
}

void printSolution(int dist[]) {
    printf("Vertex \t Distance from Source\n");
    for (int i = 0; i < V; i++)
        printf("%d \t\t %d\n", i, dist[i]);
}

void dijkstra(int graph[V][V], int src) {
    int dist[V];
    bool sptSet[V];

    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;

    dist[src] = 0;

    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, sptSet);
        sptSet[u] = true;

        for (int v = 0; v < V; v++)
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }
```

```
    printSolution(dist);
}

int main() {
    int graph[V][V] = {{0, 4, 0, 0, 0, 0, 0, 8, 0},
                       {4, 0, 8, 0, 0, 0, 0, 11, 0},
                       {0, 8, 0, 7, 0, 4, 0, 0, 2},
                       {0, 0, 7, 0, 9, 14, 0, 0, 0},
                       {0, 0, 0, 9, 0, 10, 0, 0, 0},
                       {0, 0, 4, 14, 10, 0, 2, 0, 0},
                       {0, 0, 0, 0, 0, 2, 0, 1, 6},
                       {8, 11, 0, 0, 0, 0, 1, 0, 7},
                       {0, 0, 2, 0, 0, 0, 6, 7, 0}};
    dijkstra(graph, 0);
    return 0;
}
```

A* ALGORITHM

Pseudocode

```
function A*(graph, start, goal, heuristic):
    openSet = priority queue containing start
    cameFrom = empty map
    gScore = map with default value infinity
    gScore[start] = 0
    fScore = map with default value infinity
    fScore[start] = heuristic(start, goal)

    while openSet is not empty:
        current = openSet.extractMin()

        if current == goal:
            return reconstructPath(cameFrom, current)

        for each neighbor of current:
            tentative_gScore = gScore[current] + weight(current, neighbor)
            if tentative_gScore < gScore[neighbor]:
                cameFrom[neighbor] = current
                gScore[neighbor] = tentative_gScore
                fScore[neighbor] = gScore[neighbor] + heuristic(neighbor, goal)
                if neighbor not in openSet:
                    openSet.insert(neighbor, fScore[neighbor])

    return failure

function reconstructPath(cameFrom, current):
    totalPath = [current]
    while current in cameFrom:
        current = cameFrom[current]
        totalPath.prepend(current)
    return totalPath
```

C Program

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

#define N 5

typedef struct {
    int x, y;
} Point;

typedef struct {
    Point point;
    int cost;
    int heuristic;
} Node;

typedef struct {
    Node *nodes;
    int size;
    int capacity;
} PriorityQueue;

int manhattanDistance(Point a, Point b) {
    return abs(a.x - b.x) + abs(a.y - b.y);
}

PriorityQueue *createQueue(int capacity) {
    PriorityQueue *queue = (PriorityQueue *)malloc(sizeof(PriorityQueue));
    queue->nodes = (Node *)malloc(sizeof(Node) * capacity);
    queue->size = 0;
    queue->capacity = capacity;
    return queue;
}

void insert(PriorityQueue *queue, Node node) {
    queue->nodes[queue->size++] = node;
    int i = queue->size - 1;
    while (i > 0 && queue->nodes[i].cost + queue->nodes[i].heuristic < queue->nodes[(i - 1) / 2].cost +
queue->nodes[(i - 1) / 2].heuristic) {
        Node temp = queue->nodes[i];
        queue->nodes[i] = queue->nodes[(i - 1) / 2];
        queue->nodes[(i - 1) / 2] = temp;
        i = (i - 1) / 2;
    }
}

Node extractMin(PriorityQueue *queue) {
    Node root = queue->nodes[0];
    queue->nodes[0] = queue->nodes[--queue->size];
    int i = 0;
    while (true) {
        int left = 2 * i + 1;
        int right = 2 * i + 2;
        int smallest = i;

        if (left < queue->size && queue->nodes[left].cost + queue->nodes[left].heuristic <
queue->nodes[smallest].cost + queue->nodes[smallest].heuristic)
            smallest = left;
        if (right < queue->size && queue->nodes[right].cost + queue->nodes[right].heuristic <
queue->nodes[smallest].cost + queue->nodes[smallest].heuristic)
```

```
            smallest = right;
        if (smallest == i)
            break;

        Node temp = queue->nodes[i];
        queue->nodes[i] = queue->nodes[smallest];
        queue->nodes[smallest] = temp;
        i = smallest;
    }
    return root;
}

bool isValid(int x, int y) {
    return (x >= 0 && x < N && y >= 0 && y < N);
}

bool isDestination(Point point, Point dest) {
    return point.x == dest.x && point.y == dest.y;
}

void aStarSearch(int grid[N][N], Point start, Point end) {
    if (!grid[start.x][start.y] || !grid[end.x][end.y]) {
        printf("Start or end is blocked\n");
        return;
    }

    PriorityQueue *openList = createQueue(N * N);
    bool closedList[N][N] = {false};

    Node startNode = {start, 0, manhattanDistance(start, end)};
    insert(openList, startNode);

    while (openList->size > 0) {
        Node current = extractMin(openList);
        if (isDestination(current.point, end)) {
            printf("Path found with cost %d\n", current.cost);
            return;
        }
        closedList[current.point.x][current.point.y] = true;

        int rowNum[] = {-1, 0, 0, 1};
        int colNum[] = {0, -1, 1, 0};

        for (int i = 0; i < 4; i++) {
            int newX = current.point.x + rowNum[i];
            int newY = current.point.y + colNum[i];

            if (isValid(newX, newY) && grid[newX][newY] && !closedList[newX][newY]) {
                Point newPoint = {newX, newY};
                int newCost = current.cost + 1;
                int newHeuristic = manhattanDistance(newPoint, end);
                Node neighborNode = {newPoint, newCost, newHeuristic};

                insert(openList, neighborNode);
            }
        }
    }

    printf("No path found\n");
}
```

```
int main() {
    int grid[N][N] = {
        {1, 1, 1, 1, 1},
        {1, 0, 1, 0, 1},
        {1, 1, 1, 1, 1},
        {1, 0, 1, 0, 1},
        {1, 1, 1, 1, 1}
    };

    Point start = {0, 0};
    Point end = {4, 4};

    aStarSearch(grid, start, end);

    return 0;
}
```

OPTIMIZED PRIM ALGORITHM

Pseudocode

```
function Prim(graph):
    startNode = any node in graph
    mstSet = set containing startNode
    priorityQueue = priority queue of edges connected to startNode
    mst = empty list of edges

    while priorityQueue is not empty:
        edge = priorityQueue.extractMin()
        if edge connects a node in mstSet to a node not in mstSet:
            mst.add(edge)
            mstSet.add(edge.otherNode())
            for each edge from newly added node:
                if edge leads to a node not in mstSet:
                    priorityQueue.insert(edge)

    return mst
```

C Program

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

#define V 5

int minKey(int key[], bool mstSet[]) {
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;
    return min_index;
}

void printMST(int parent[], int graph[V][V]) {
    printf("Edge \tWeight\n");
    for (int i = 1; i < V; i++)
        printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
}
```

```
void primMST(int graph[V][V]) {
    int parent[V];
    int key[V];
    bool mstSet[V];

    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

    key[0] = 0;
    parent[0] = -1;

    for (int count = 0; count < V - 1; count++) {
        int u = minKey(key, mstSet);
        mstSet[u
```

OPTIMIZED KRUSKAL ALGORITHM

Pseudocode

```
function Kruskal(graph):
    mst = empty list of edges
    edgeList = list of all edges in graph, sorted by weight
    disjointSet = DisjointSet(graph.nodes())

    for each edge in edgeList:
        u, v = edge.endpoints()
        if disjointSet.find(u) != disjointSet.find(v):
            disjointSet.union(u, v)
            mst.add(edge)

    return mst

class DisjointSet:
    function __init__(nodes):
        parent = map with each node pointing to itself
        rank = map with each node having rank 0

    function find(node):
        if parent[node] != node:
            parent[node] = find(parent[node])
        return parent[node]

    function union(node1, node2):
        root1 = find(node1)
        root2 = find(node2)
        if root1 != root2:
            if rank[root1] > rank[root2]:
                parent[root2] = root1
            else if rank[root1] < rank[root2]:
                parent[root1] = root2
            else:
                parent[root2] = root1
                rank[root1] += 1
```

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

```
// Edge structure
typedef struct Edge {
    int src, dest, weight;
} Edge;

// Graph structure
typedef struct Graph {
    int V, E;
    Edge* edge;
} Graph;

// Subset structure for union-find
typedef struct subset {
    int parent;
    int rank;
} subset;

// Function to create a graph
Graph* createGraph(int V, int E) {
    Graph* graph = (Graph*)malloc(sizeof(Graph));
    graph->V = V;
    graph->E = E;
    graph->edge = (Edge*)malloc(graph->E * sizeof(Edge));
    return graph;
}

// Find function with path compression
int find(subset subsets[], int i) {
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);
    return subsets[i].parent;
}

// Union function with union by rank
void Union(subset subsets[], int x, int y) {
    int rootX = find(subsets, x);
    int rootY = find(subsets, y);

    if (subsets[rootX].rank < subsets[rootY].rank)
        subsets[rootX].parent = rootY;
    else if (subsets[rootX].rank > subsets[rootY].rank)
        subsets[rootY].parent = rootX;
    else {
        subsets[rootY].parent = rootX;
        subsets[rootX].rank++;
    }
}

// Comparison function for qsort
int compareEdges(const void* a, const void* b) {
    Edge* a1 = (Edge*)a;
    Edge* b1 = (Edge*)b;
    return a1->weight > b1->weight;
}

// Kruskal's algorithm
void KruskalMST(Graph* graph) {
    int V = graph->V;
    Edge result[V];
    int e = 0;
```

```
    int i = 0;

    // Sort all edges in non-decreasing order of their weight
    qsort(graph->edge, graph->E, sizeof(graph->edge[0]), compareEdges);

    subset* subsets = (subset*)malloc(V * sizeof(subset));
    for (int v = 0; v < V; ++v) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    // Number of edges to be taken is equal to V-1
    while (e < V - 1 && i < graph->E) {
        // Pick the smallest edge
        Edge next_edge = graph->edge[i++];
        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);

        // If including this edge doesn't cause a cycle, include it
        if (x != y) {
            result[e++] = next_edge;
            Union(subsets, x, y);
        }
    }

    // Print the result
    printf("Following are the edges in the constructed MST\n");
    for (i = 0; i < e; ++i)
        printf("%d -- %d == %d\n", result[i].src, result[i].dest, result[i].weight);

    free(subsets);
}

int main() {
    int V = 4; // Number of vertices
    int E = 5; // Number of edges
    Graph* graph = createGraph(V, E);

    // Adding edges
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = 10;

    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 6;

    graph->edge[2].src = 0;
    graph->edge[2].dest = 3;
    graph->edge[2].weight = 5;

    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;
    graph->edge[3].weight = 15;

    graph->edge[4].src = 2;
    graph->edge[4].dest = 3;
    graph->edge[4].weight = 4;

    // Run Kruskal's algorithm
    KruskalMST(graph);

    // Free allocated memory
    free(graph->edge);
    free(graph);

    return 0;
}
```

0/1 KNAPSACK - DYNAMIC PROGRAMMING

```
// A Dynamic Programming based
// solution for 0-1 Knapsack problem
#include <stdio.h>

// A utility function that returns
// maximum of two integers
int max(int a, int b) { return (a > b) ? a : b; }

// Returns the maximum value that
// can be put in a knapsack of capacity W
int knapSack(int W, int wt[], int val[], int n)
{
    int i, w;
    int K[n + 1][W + 1];

    // Build table K[][] in bottom up manner
    for (i = 0; i <= n; i++) {
        for (w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                K[i][w] = 0;
            else if (wt[i - 1] <= w)
                K[i][w] = max(val[i - 1]
                              + K[i - 1][w - wt[i - 1]],
                          K[i - 1][w]);
            else
                K[i][w] = K[i - 1][w];
        }
    }

    return K[n][W];
}

// Driver Code
int main()
{
    int profit[] = { 60, 100, 120 };
    int weight[] = { 10, 20, 30 };
    int W = 50;
    int n = sizeof(profit) / sizeof(profit[0]);
    printf("%d", knapSack(W, weight, profit, n));
    return 0;
}
```

OUTPUT: 220