Arrays

Bubble Sort

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

int main(){
    int n;
    printf("Enter the number of elements to be sorted: ");
    scanf("%d", &n);
    int *arr = (int*)malloc(n * sizeof(int));

    // Input
    for(int i = 0; i < n; ++i){
        printf("Enter element number %d: ", i + 1);
        scanf("%d", &arr[i]);
    }

    bubbleSort(arr, n);

    // Printing
    printf("Sorted elements:\n");
    for(int i = 0; i < n; ++i){
        printf("%d ", arr[i]);
    }

    free(arr);

    return 0;
}

void bubbleSort(int *arr, int n){
    int r=0;
    for (int c = 0; c < n - 1; ++c) {
        for (int x = 0; x < n - 1 - c; ++x) {
            if (arr[x] > arr[x + 1]) {
                swap(&arr[x], &arr[x + 1]);
                r++;
                printf(" Number of Iterations: %d\n",r);
            }
            else{
                continue;
            }
        }
        break;
    }
}
```

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

Insertion Sort

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

void insertionSort(int sampleLengthVar, int* sampleArrayPtr) {
    for (int i = 1; i < sampleLengthVar; i++) {
        int key = sampleArrayPtr[i];
        int j = i - 1;

        while (j >= 0 && sampleArrayPtr[j] > key) {
            sampleArrayPtr[j + 1] = sampleArrayPtr[j];
            j = j - 1;
        }

        sampleArrayPtr[j + 1] = key;
    }
}

int* readFile(char filename[], int* sampleLengthVar) {
    FILE* fp;
    fp = fopen(filename, "r");

    if (fp == NULL) {
        printf("Error opening file.\n");
        return NULL;
    }

    fscanf(fp, "%d", sampleLengthVar);
    printf("Scanning a sample of length: %d\n", *sampleLengthVar);

    int* sampleArrayPtr = (int*)malloc(*sampleLengthVar * sizeof(int));
    if (sampleArrayPtr == NULL) {
        printf("Memory allocation failed.\n");
        fclose(fp);
        return NULL;
    }

    for (int i = 0; i < *sampleLengthVar; i++) {
        fscanf(fp, "%d", &sampleArrayPtr[i]);
    }
```

```
    fclose(fp);
    return sampleArrayPtr;
}

int main() {
    char filename[] = "./number files/descending.txt";
    int sampleLengthVar;
    int* sampleArrayPtr = readFile(filename, &sampleLengthVar);

    if (sampleArrayPtr != NULL) {
        clock_t start_time = clock();

        insertionSort(sampleLengthVar, sampleArrayPtr);

        clock_t end_time = clock();
        double time_taken = (double)(end_time - start_time) /
CLOCKS_PER_SEC;

        printf("Sorted Array:\n");
        for (int i = 0; i < sampleLengthVar; i++) {
            printf("%d ", sampleArrayPtr[i]);
        }
        printf("\n");
printf("Execution Time: %f seconds\n", time_taken);
        free(sampleArrayPtr);
    }

    return 0;
}
```

Linear Search

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

int linearsearch(int *arr, int size, int val) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == val)
            return 1;
    }
    return 0;
}

int main() {
    int n, i, v;
    printf("Enter the value to be searched:\n");
    scanf("%d", &v);

    FILE *file = fopen("array.txt", "r");
```

```
    if (file == NULL) {
        printf("Failed to open the file.\n");
        return 1;
    }

    fscanf(file, "%d", &n);
    int *a = (int *)malloc(n * sizeof(int));

    printf("Array read from the file:\n");
    for (i = 0; i < n; i++) {
        fscanf(file, "%d", &a[i]);
        printf("%d ", a[i]);
    }
    printf("\n");

    fclose(file);

    clock_t start_time = clock();

    if (linearsearch(a, n, v))
        printf("Value %d is in the array.\n", v);
    else
        printf("Value %d is not in the array.\n", v);

    clock_t end_time = clock();
    double time_taken = (double)(end_time - start_time) /
CLOCKS_PER_SEC;
    printf("Execution Time: %f seconds\n", time_taken);

    free(a);
    return 0;
}
```

Stacks

Infix to Postfix

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

#define SIZE 100

char push();
char pop();
void display();
int precedence(char);
int isOperator(char);
```

```
char stack[SIZE];
int top = -1;

int main(){
    char infix[SIZE], postfix[SIZE];
    int i, j;
    char ch, elem;
    printf("Enter Infix Expression : ");
    scanf("%s", infix);
    i = 0;
    j = 0;
    while (infix[i] != '\0'){
        ch = infix[i];
        if (ch == '('){
        push(ch);
        }
        else if (isalnum(ch)){
        postfix[j] = ch;
        j++;
        }
        else if (isOperator(ch) == 1){
        elem = pop();
        while (isOperator(elem) == 1 && precedence(elem) >=
precedence(ch)){
            postfix[j] = elem;
            j++;
            elem = pop();
        }
        push(elem);
        push(ch);
        }
        else if (ch == ')'){
        elem = pop();
        while (elem != '('){
            postfix[j] = elem;
            j++;
            elem = pop();
        }
        }
        else{
        printf("\nInvalid Arithmetic Expression.\n");
        exit(0);
        }
        i++;
    }
    while (top > -1){
        postfix[j] = pop();
        j++;
    }
    postfix[j] = '\0';
    printf("\nArithmetic expression in Postfix notation : ");
    puts(postfix);
```

```
    return 0;
}


char push(char elem){
    top++;
    stack[top] = elem;
    return elem;
}

char pop(){
    if (top == -1){
        printf("\nStack is Empty\n");
        return -1;
    }
    char elem;
    elem = stack[top];
    top--;
    return elem;
}

void display(){
    int i;
    if (top == -1)
    printf("\nStack is Empty\n");
    else{
    printf("\nStack Elements :\n");
    for (i = top; i >= 0; i--)
        printf(" %c\n", stack[i]);
    }
}

int precedence(char elem){
    switch (elem){
        case '#':
        return 0;
        case '(':
        return 1;
        case '+':
        case '-':
        return 2;
        case '*':
        case '/':
        return 3;
        case '^':
        return 4;
    }
}

int isOperator(char elem){
    switch (elem){
        case '+':
        case '-':
```

```
        case '*':
        case '/':
        case '^':
        return 1;
        default:
        return 0;
    }
}
```

Queue

Basics

```
// functions to create update read and delete queue

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

struct Node{
  int value;
  struct Node *next;
};

struct Node* createNode(int value){
  struct Node* newNode=malloc(sizeof(struct Node));
  newNode->value=value;
  newNode->next=NULL;
  return newNode;
}

struct Node* enqueue(struct Node* root,int value){
  struct Node* newNode=createNode(value);
  if(root==NULL){
    root=newNode;
    return root;
  }
  struct Node* temp=root;
  while(temp->next!=NULL){
    temp=temp->next;
  }
  temp->next=newNode;
  return root;
}

struct Node* dequeue(struct Node* root){
  if(root==NULL){
```

```
    printf("Queue is empty\n");
    return root;
  }
  struct Node* temp=root;
  root=root->next;
  free(temp);
  return root;
}

void printQueue(struct Node* root){
  if(root==NULL){
    printf("Queue is empty\n");
    return;
  }
  struct Node* temp=root;
  while(temp!=NULL){
    printf("%d ",temp->value);
    temp=temp->next;
  }
  printf("\n");
}

int main(){
  struct Node* root=NULL;
  root=enqueue(root,10);
  root=enqueue(root,20);
  root=enqueue(root,30);
  root=enqueue(root,40);
  root=enqueue(root,50);
  printQueue(root);
  root=dequeue(root);
  root=dequeue(root);
  printQueue(root);
  return 0;
}
```

Linked List

Singly Linked List

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

struct Node {
    int data;
    struct Node* next;
```

```
};

void insertAtBeginning(struct Node** head, int newData) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        return;
    }

    newNode->data = newData;

    newNode->next = *head;

    *head = newNode;
}


// Function to print the elements of the linked list
void printList(struct Node* head) {
    struct Node* current = head;
    while (current != NULL) {
        printf("%d -> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}

struct Node *getData(int data) {
    struct Node *p = (struct Node *)malloc(sizeof(struct Node));
    p->data = data;
    p->next = NULL;
    return p;
}

int main() {
    empty linked list testcase
    struct Node* head = NULL;
    printf("Empty Linked List: ");
    printList(head);
    insertAtBeginning(&head, 3);
    printf("Linked List: ");
    printList(head);
    // inserted element 1 into empty linked list


Linked List with one element
    struct Node* head1 = (struct Node*)malloc(sizeof(struct Node));
    if (head1 == NULL) {
        printf("Memory allocation failed.\n");
        return 1; //  error code
    }
    head1->data = 1; //  data of the head node
    head1->next = NULL;
```

```
    insertAtBeginning(&head1, 3);
    printf("Linked List with one element insert with function: ");
    printList(head1);
    printf("Linked List with one element above");


Linked list with 3 elements
    struct Node* head2_1,*head2_2,*head2_3;
    head2_1=getData(1);
    head2_2=getData(2);
    head2_3=getData(3);
    head2_1->next=head2_2;
    head2_2->next=head2_3;
    printf("Linked List: ");
    printList(head2_1);

    insertAtBeginning(&head2_1, 3);
    printf("Linked List with one element insert with function: ");
    printList(head2_1);
    printf("Linked List with one element above");


// Linked list with 5 elements
    struct Node* head3_1,*head3_2,*head3_3,*head3_4,*head3_5;
    head3_1=getData(1);
    head3_2=getData(2);
    head3_3=getData(3);
    head3_4=getData(4);
    head3_5=getData(5);
    head3_1->next=head3_2;
    head3_2->next=head3_3;
    head3_3->next=head3_4;
    head3_4->next=head3_5;

    printf("Linked List: ");
    printList(head3_1);

    insertAtBeginning(&head3_1, 3);
    printf("Linked List with five element insert with function: ");
    printList(head3_1);
    printf("Linked List with one element above");

//Linked List with 10 elements
    struct Node*
head4_1,*head4_2,*head4_3,*head4_4,*head4_5,*head4_6,*head4_7,*head4_8,*
head4_9,*head4_10;;
    head4_1=getData(1);
    head4_2=getData(2);
    head4_3=getData(3);
    head4_4=getData(4);
    head4_5=getData(5);
    head4_6=getData(6);
    head4_7=getData(7);
```

```
    head4_8=getData(8);
    head4_9=getData(9);
    head4_10=getData(10);
    head4_1->next=head4_2;
    head4_2->next=head4_3;
    head4_3->next=head4_4;
    head4_4->next=head4_5;
    head4_5->next=head4_6;
    head4_6->next=head4_7;
    head4_7->next=head4_8;
    head4_8->next=head4_9;
    head4_9->next=head4_10;

    printf("Linked List: ");
    printList(head4_1);

    insertAtBeginning(&head4_1, 3);

    printf("Linked List with one element insert with function: ");
    printList(head4_1);
    printf("Linked List with one element above");



    printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n");



    return 0;
}
```

Doubly Linked List

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

struct Node{
    int data;
    struct Node *next;
    struct Node *prev;
};

struct Node* getNode(int data){
    struct Node *p = (struct Node *)malloc(sizeof(struct Node));
    p->data = data;
    p->next = NULL;
    p->prev = NULL;
    return p;
```

```c
}

void insertAtBeginning(struct Node** head, int newData) {
    struct Node* newNode = getNode(newData);

    newNode->next = *head;
    if (*head != NULL) {
        (*head)->prev = newNode;
    }

    *head = newNode;
}

void insertAtEnd(struct Node** head, int newData) {
    struct Node* newNode = getNode(newData);

    if (*head == NULL) {
        // If the list is empty, make the new node the head
        *head = newNode;
    } else {
        struct Node* current = *head;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = newNode;
        newNode->prev = current;
    }

    printf("Node with data %d inserted at the end\n", newData);
}


void insertAfterNodex(int x,struct Node**head, int newData){
    struct Node* newNode = getNode(newData);

    if (*head == NULL) {
        *head = newNode;
    } else {
        struct Node* current = *head;
        while (current != NULL && current->data != x) {
            current = current->next;
        }
        if (current == NULL) {
            printf("Node with data %d not found\n", x);
            free(newNode);
            return;
        }
        newNode->next=current->next;
        if (current->next != NULL) {
            current->next->prev = newNode;
        }
        current->next=newNode;
        newNode->prev = current;
```

```
    }

    printf("Node with data %d inserted after %d\n", newData,x);
}

void insertBeforeNodex(int x,struct Node**head, int newData){
    struct Node* newNode = getNode(newData);

    if (*head == NULL) {
        *head = newNode;
    } else {
        struct Node* current = *head;
        while (current != NULL && current->data != x) {
            current = current->next;
        }
        if (current == NULL) {
            printf("Node with data %d not found\n", x);
            free(newNode);
            return;
        }
        newNode->next=current;
        newNode->prev = current->prev;
        if (current->prev != NULL) {
            current->prev->next = newNode;
        } else {
            *head = newNode;
        }
        current->prev=newNode;
    }

    printf("Node with data %d inserted before %d\n", newData,x);
}

void deleteNode(int x,struct Node**head){
    struct Node* current = *head;
    while (current != NULL && current->data != x) {
        current = current->next;
    }
    if (current == NULL) {
        printf("Node with data %d not found\n", x);
        return;
    }
    if (current->prev != NULL) {
        current->prev->next = current->next;
    } else {
        *head = current->next;
    }
    if (current->next != NULL) {
        current->next->prev = current->prev;
    }
    free(current);
    printf("Node with data %d deleted\n", x);
}
```

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

    //testing
    insertAtBeginning(&head, 1);
    insertAtBeginning(&head, 2);
    insertAtBeginning(&head, 3);
    insertAtBeginning(&head, 4);

    insertAtEnd(&head, 5);
    insertAtEnd(&head, 6);
    insertAtEnd(&head, 7);

    insertAfterNodex(3,&head, 8);
    insertAfterNodex(5,&head, 9);
    insertAfterNodex(7,&head, 10);

    insertBeforeNodex(3,&head, 11);
    insertBeforeNodex(5,&head, 12);
    insertBeforeNodex(7,&head, 13);

    deleteNode(3,&head);
    deleteNode(5,&head);
    deleteNode(7,&head);
    return 0;
}
```

Delete Start Node

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

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

void deletefirstNode(struct Node ** head){
    struct Node *temp = *head;
    *head = (*head)->next;
    free(temp);
}

void deletelastElement(struct Node **head){
    struct Node* current = *head;
    struct Node* after = current->next;
    while (1) {
        if(after->next==NULL){
            current->next=NULL;
```

```
            break;
        }
        current = after;
        after = after->next;
    }
    printf("NULL\n");
}

void insertAtBeginning(struct Node** head, int newData) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        return;
    }

    newNode->data = newData;

    newNode->next = *head;

    *head = newNode;
}

// Function to print the elements of the linked list
void printList(struct Node* head) {
    struct Node* current = head;
    while (current != NULL) {
        printf("%d -> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}

struct Node *getData(int data) {
    struct Node *p = (struct Node *)malloc(sizeof(struct Node));
    p->data = data;
    p->next = NULL;
    return p;
}

int main() {
    Linked List with one element
    struct Node* head1 = (struct Node*)malloc(sizeof(struct Node));
    if (head1 == NULL) {
        printf("Memory allocation failed.\n");
        return 1; //  error code
    }
    head1->data = 1; //  data of the head node
    head1->next = NULL;
    insertAtBeginning(&head1, 3);
    printf("Linked List with one element insert with function: ");
    printList(head1);
    printf("Linked List with one element above");
```

```
    printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n");

    return 0;
}
```

Circular Linked List

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

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

struct Node *getData(int data) {
    struct Node *p = (struct Node *)malloc(sizeof(struct Node));
    p->data = data;
    p->next = NULL;
    return p;
}

void insertAtBeginning(struct Node** head, int newData){
    struct Node* newNode = getData(newData);

    newNode->next = *head;
    if (*head != NULL) {
        struct Node* temp = *head;
        while (temp->next != *head) {
            temp = temp->next;
        }
        temp->next = newNode;
    } else {
        newNode->next = newNode;
    }

    *head = newNode;
}

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

int main(){
```

```
    struct Node* head = NULL;

    insertAtBeginning(&head, 1);
    insertAtBeginning(&head, 2);
    insertAtBeginning(&head, 3);

    // Print the circular linked list
    struct Node* current = head;
    if (head != NULL) {
        do {
            printf("Data in the node: %d\n", current->data);
            current = current->next;
        } while (current != head);
    }

    return 0;
}
```

Polynomial Addition

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

struct node
{
    int coeff;
    int exp;
    struct node *next;
};


struct node *create(struct node *head)
{
    struct node *newnode,*temp;
    int n,i;
    printf("Enter the number of terms: ");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        newnode=(struct node *)malloc(sizeof(struct node));
        printf("Enter the coefficient and exponent of term %d: ",i+1);
        scanf("%d%d",&newnode->coeff,&newnode->exp);
        newnode->next=NULL;
        if(head==NULL)
        {
            head=newnode;
            temp=head;
        }
        else
```

```
        {
            temp->next=newnode;
            temp=newnode;
        }
    }
    return head;
}

struct node* addNode(struct node *head,int coeff,int exp){
    struct node* temp=NULL, *newNode;
    temp=head;
    while(temp->next!=NULL){
        temp=temp->next;
    }
    newNode=(struct node *)malloc(sizeof(struct node));
    temp->next=newNode;
    newNode->coeff=coeff;
    newNode->exp=exp;
    newNode->next=NULL;

}

void Display(struct node* head){
    struct node *temp;
    temp=head;
    printf("The polynomial is: ");
    while(temp!=NULL){

        printf("%dx^%d + ",temp->coeff,temp->exp);
        temp=temp->next;
    }
    printf("\n");
}

void PolynomialAdditon(struct node *head1, struct node *head2, struct
node *finalList){
    struct node* temp1 = NULL;
    temp1 = head1;
    struct node* temp2 = NULL;
    temp2 = head2;
    struct node* final=NULL;
    final=finalList;

    while(temp1->next!=NULL && temp2->next!=NULL){
        if(temp1->exp>temp2->exp){
            addNode(final,temp1->coeff,temp1->exp);
            temp1=temp1->next;
            }
            else if(temp2->exp>temp1->exp){
                addNode(final,temp2->coeff,temp2->exp);
            temp2=temp2->next;
        }
        else if(temp1->exp==temp2->exp){
```

```
            int coeffNew= temp1->coeff+temp2->coeff;
            addNode(final,coeffNew,temp1->exp);
        }
    }

    return final;
}

int main(){
    struct node *head1=NULL,*head2=NULL,*head3=NULL,
*final=NULL,*temp,*newnode;
    head1=create(head1);
    head2=create(head2);
    Display(head1);
    Display(head2);
    temp=head1;
    //initializing the final list
    final=(struct node *)malloc(sizeof(struct node));
    PolynomialAdditon(head1,head2,final);
    Display(final);
    return 0;

}
```

Trees

Tree Construction from Pre and In order

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

struct Node{
    int key;
    struct Node* left;
    struct Node* right;
};

struct Node *createNode(int x){
    struct Node *temp=(struct Node*)malloc(sizeof(struct Node));
    temp->key=x;
    temp->left=NULL;
    temp->right=NULL;
    return temp;
}

//construct tree from pre order and in order
int search(int arr[],int start,int end,int key){
    int i;
```

```
    for(i=start;i<=end;i++){
        if(arr[i]==key){
            return i;
        }
    }
    return -1;
}

struct Node* constructTree(int pre[], int in[], int start, int end, int*
preIndex) {
    if (start > end) {
        return NULL;
    }
    struct Node* temp = createNode(pre[(*preIndex)++]);
    if (start == end) {
        return temp;
    }
    int inIndex = search(in, start, end, temp->key);
    temp->left = constructTree(pre, in, start, inIndex - 1, preIndex);
    temp->right = constructTree(pre, in, inIndex + 1, end, preIndex);
    return temp;
}



void printInorder(struct Node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    printf("%d ", node->key);
    printInorder(node->right);
}

int main() {
    struct Node* root = NULL;
    int n;
    printf("Enter the number of elements in the tree: ");
    scanf("%d", &n);
    int pre[n], in[n];
    printf("Enter the preorder traversal of the tree: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &pre[i]);
    }
    printf("Enter the inorder traversal of the tree: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &in[i]);
    }
    int preIndex = 0;  // Initialize preIndex
    root = constructTree(pre, in, 0, n - 1, &preIndex);
    printf("Inorder traversal of the constructed tree is: ");
    printInorder(root);
    return 0;
```

```
}
```

Binary Search Tree using Arrays

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

#define MAX_NODES 100

int binaryTree[MAX_NODES];
int root = -1; // Index of the root node

//  Initialize all pos in tree
void initialize() {
    for (int i = 0; i < MAX_NODES; i++) {
        binaryTree[i] = -1; //giving empty value as -1
    }
}

void insert(int data) {
    if (root == -1) {
        root = 0;
        binaryTree[root] = data;
    } else {
        int current = root;
        while (1) {
            if (data < binaryTree[current]) {
                if (binaryTree[2 * current + 1] == -1) {
                    binaryTree[2 * current + 1] = data;
                    break;
                } else {
                    current = 2 * current + 1;
                }
            } else {
                if (binaryTree[2 * current + 2] == -1) {
                    binaryTree[2 * current + 2] = data;
                    break;
                } else {
                    current = 2 * current + 2;
                }
            }
        }
    }
}
//inorder print
void printInOrder(int current) {
    if (binaryTree[current] != -1) {
        printInOrder(2 * current + 1);
        printf("%d ", binaryTree[current]);
```

```
        printInOrder(2 * current + 2);
    }
}

//preorder print
void printPreOrder(int current) {
    if (binaryTree[current] != -1) {
        printf("%d ", binaryTree[current]);
        printPreOrder(2 * current + 1);
        printPreOrder(2 * current + 2);
    }
}

//postorder print
void printPostOrder(int current) {
    if (binaryTree[current] != -1) {
        printPostOrder(2 * current + 1);
        printPostOrder(2 * current + 2);
        printf("%d ", binaryTree[current]);
    }
}


int searchTree(int *arr,int data){
    int i=0;
    while(arr[i]!=-1){
        if(arr[i]==data){
        printf("Found\n");
        return i;
        }
        else if(arr[i]>data){
        i=2*i+1;
        }
        else{
        i=2*i+2;
        }
    }
    printf("Not Found\n");
    return -1;
}

//delete the node using the search function
void deleteNode(int *arr,int data){
    int i=searchTree(arr,data);
    if(i==-1){
    printf("Not Found");
    }
    else{

        //if there is not chlid
        if(arr[2*i+1]==-1 && arr[2*i+2]==-1){
            arr[i]=-1;
        }
```

```
        // right child alone
        else if(arr[2*i+1]==-1 && arr[2*i+2]!=-1){
            arr[i]=arr[2*i+2];
            arr[2*i+2]=-1;
        }

        //left child alone
        else if(arr[2*i+2]==-1 && arr[2*i+1]!=-1){
            arr[i]=arr[2*i+1];
            arr[2*i+1]=-1;
        }

        //both child
        else{
            int j=2*i+2;
            while(arr[2*j+1]!=-1){
                j=2*j+1;
            }
            arr[i]=arr[j];
            arr[j]=-1;
        }
    }

}

int main() {
    initialize();

    // Insert nodes into the binary tree
    insert(50);
    insert(30);
    insert(70);
    insert(20);
    insert(40);
    insert(60);
    insert(80);

    // Print the binary tree in in-order
    printf("In-order traversal: ");
    printInOrder(root);
    printf("\n");

    return 0;
}
```

Binary Search Tree using Linked List

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

struct Node {
  int value;
  struct Node *left;
  struct Node *right;
};

struct Node* createNode(int value) {
  struct Node* newNode = malloc(sizeof(struct Node));
  newNode->value = value;
  newNode->left = NULL;
  newNode->right = NULL;
  return newNode;
}

struct Node* insert(struct Node* root, int value) {
  if (root == NULL) {
    return createNode(value);
  }

  if (value < root->value) {
    root->left = insert(root->left, value);
  } else {
    root->right = insert(root->right, value);
  }

  return root;
}

void inOrder(struct Node* root) {
  if (root == NULL) return;

  inOrder(root->left);
  printf("%d ", root->value);
  inOrder(root->right);
}


//create a function to traverse the tree and find the node
struct Node* findNode(struct Node *root,int data){
  if(root==NULL){
    return NULL;
  }
  if(root->value==data){
    return root;
  }
  else if(root->value>data){
    return findNode(root->left,data);
  }
```

```
  else{
    return findNode(root->right,data);
  }
}


void deleteNode(struct Node *element,int data){
  struct Node *root=findNode(element,data);
  if(root==NULL){
    return;
  }
  if(root->value==data){
    if(root->left==NULL && root->right==NULL){
      free(root);
      root=NULL;
    }
    else if(root->left==NULL && root->right!=NULL){
      struct Node *temp=root;
      root=root->right;
      free(temp);
      printf("Right Child is promoted\n");
    }
    else if(root->right==NULL && root->left!=NULL){
      struct Node *temp=root;
      root=root->left;
      free(temp);
      printf("Left Child is promoted\n");
    }
    else{
      struct Node *temp=root->right;
      while(temp->left!=NULL){
        temp=temp->left;
      }
      root->value=temp->value;
      deleteNode(root->right,temp->value);
    }
  }
  else if(root->value>data){
    deleteNode(root->left,data);
  }
  else{
    deleteNode(root->right,data);
}

}

// ! Notes
// ! Find the in-order successor of the node that needs to be deleted
// ! It will always be a leaf node or a node with only one child
// ! It will be in the left-most position of the right subtree of the
node to be delete
// ! Replace the node to be deleted with the in-order successor
// ! Delete the in-order successor
```

```c
// * Find the in-order successor
struct Node* findInorderSuccessor(struct Node *root){
  struct Node *temp=root->right;
  while(temp->left!=NULL){
    temp=temp->left;
  }
  return temp;
}


int main() {
  struct Node* root = NULL;
  root = insert(root, 15);
  root = insert(root, 10);
  root = insert(root, 20);

  inOrder(root);

  return 0;
}
```

Maximum Path Sum

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

struct Node{
    int key;
    struct Node* left;
    struct Node* right;
};

struct Node *createNode(int x){
    struct Node *temp=(struct Node*)malloc(sizeof(struct Node));
    temp->key=x;
    temp->left=NULL;
    temp->right=NULL;
    return temp;
}

struct Node *insert(struct Node *root,int x){
    if(root==NULL){
        root=createNode(x);
        return root;
    }
    else if(x<=root->key){
```

```
        root->left=insert(root->left,x);
    }
    else{
        root->right=insert(root->right,x);
    }
    return root;
}

int greater(int a , int b){
    return (a>b)?a:b;
}

int maxPathSum(struct Node *root,int *res){
    if(root==NULL){
        return 0;
    }
    //the sum of the elements in the left subtree
    int l=maxPathSum(root->left,res);
    //the sum of the elements in the right subtree
    int r=maxPathSum(root->right,res);
    //max path for parent call of root. This path must include at most
one child of root
    int max_single=greater(greater(l,r)+root->key,root->key);
    //max top represents the sum when the node under consideration is
the root of the maxSum path and no ancestor of root are there in max sum
path
    int max_top=greater(max_single,l+r+root->key);
    *res=greater(*res,max_top);
    return max_single;
}

int main() {
    struct Node* root = NULL;
    int n;
    printf("Enter the number of elements in the tree: ");
    scanf("%d", &n);
    int a[n];
    for (int i = 0; i < n; i++) {
        printf("Enter the element: ");
        scanf("%d", &a[i]);
        root = insert(root, a[i]);
    }
    int res = 0;
    maxPathSum(root, &res);
    printf("Maximum Path Sum: %d\n", res);
    return 0;
}
```

Graphs

Basics of Graphs

```c
// bfs and dfs traversal for graphs

#include<stdio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node *next;
};
struct node *createNode(int data)
{
    struct node *newNode = (struct node *)malloc(sizeof(struct node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}
void addEdge(struct node **adjList, int u, int v)
{
    struct node *newNode = createNode(v);
    newNode->next = adjList[u];
    adjList[u] = newNode;
    newNode = createNode(u);
    newNode->next = adjList[v];
    adjList[v] = newNode;
}
void printGraph(struct node **adjList, int V)
{
    for (int i = 0; i < V; i++)
    {
        struct node *temp = adjList[i];
        printf("Adjacency list of vertex %d\n", i);
        while (temp)
        {
            printf("%d -> ", temp->data);
            temp = temp->next;
        }
        printf("NULL\n");
    }
}

void bfs(struct node **adjList, int V, int start)
{
    int *visited = (int *)calloc(V, sizeof(int));
    int *queue = (int *)malloc(V * sizeof(int));
    int front = -1, rear = -1;
    queue[++rear] = start;
    visited[start] = 1;
    while (front != rear)
    {
```

```
        int u = queue[++front];
        printf("%d ", u);
        struct node *temp = adjList[u];
        while (temp)
        {
            if (!visited[temp->data])
            {
                queue[++rear] = temp->data;
                visited[temp->data] = 1;
            }
            temp = temp->next;
        }
    }
}

void dfs(struct node **adjList, int V, int start)
{
    int *visited = (int *)calloc(V, sizeof(int));
    int *stack = (int *)malloc(V * sizeof(int));
    int top = -1;
    stack[++top] = start;
    visited[start] = 1;
    while (top != -1)
    {
        int u = stack[top--];
        printf("%d ", u);
        struct node *temp = adjList[u];
        while (temp)
        {
            if (!visited[temp->data])
            {
                stack[++top] = temp->data;
                visited[temp->data] = 1;
            }
            temp = temp->next;
        }
    }
}

int main()
{
    int V = 5;
    struct node **adjList = (struct node **)malloc(V * sizeof(struct
node *));
    for (int i = 0; i < V; i++)
        adjList[i] = NULL;
    addEdge(adjList, 0, 1);
    addEdge(adjList, 0, 4);
    addEdge(adjList, 1, 2);
    addEdge(adjList, 1, 3);
    addEdge(adjList, 1, 4);
    addEdge(adjList, 2, 3);
    addEdge(adjList, 3, 4);
```

```
    printGraph(adjList, V);
    return 0;
}
```

Heaps

Min Heap Programs

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

#define MAX_HEAP_SIZE 100

// min-heap
struct MinHeap {
    int arr[MAX_HEAP_SIZE];
    int size;
};

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

//heapify at index
void minHeapify(struct MinHeap *heap, int index) {
    int smallest = index;
    int left = 2 * index + 1;
    int right = 2 * index + 2;

    if (left < heap->size && heap->arr[left] < heap->arr[smallest])
        smallest = left;

    if (right < heap->size && heap->arr[right] < heap->arr[smallest])
        smallest = right;

    if (smallest != index) {
        swap(&heap->arr[index], &heap->arr[smallest]);
        minHeapify(heap, smallest);
    }
}


void insert(struct MinHeap *heap, int key) {
    if (heap->size == MAX_HEAP_SIZE) {
        printf("Heap is full. Cannot insert.\n");
```

```
        return;
    }

    int i = heap->size++;
    heap->arr[i] = key;


    while (i != 0 && heap->arr[(i - 1) / 2] > heap->arr[i]) {
        swap(&heap->arr[i], &heap->arr[(i - 1) / 2]);
        i = (i - 1) / 2;
    }
}

//extractMin
int extractMin(struct MinHeap *heap) {
    if (heap->size <= 0) {
        printf("Heap is empty. Cannot extract minimum.\n");
        return -1;
    }


    int min = heap->arr[0];


    heap->arr[0] = heap->arr[heap->size - 1];
    heap->size--;


    minHeapify(heap, 0);

    return min;
}

// Function to delete a key at a given index from the heap
void deleteKey(struct MinHeap *heap, int index) {
    if (index >= heap->size) {
        printf("Index out of range. Cannot delete key.\n");
        return;
    }


    while (index != 0 && heap->arr[(index - 1) / 2] > heap->arr[index])
{
        swap(&heap->arr[index], &heap->arr[(index - 1) / 2]);
        index = (index - 1) / 2;
    }

    extractMin(heap);
}

// show heap
void printHeap(struct MinHeap *heap) {
    printf("Heap: ");
```

```
    for (int i = 0; i < heap->size; i++)
        printf("%d ", heap->arr[i]);
    printf("\n");
}

int main() {
    struct MinHeap heap;
    heap.size = 0;

    // Insert  keys into the heap
    insert(&heap, 3);
    insert(&heap, 2);
    insert(&heap, 15);
    insert(&heap, 5);
    insert(&heap, 4);
    insert(&heap, 45);

    printHeap(&heap);

    printf("Extracted Min: %d\n", extractMin(&heap));

    printHeap(&heap);

    deleteKey(&heap, 2);

    printHeap(&heap);

    return 0;
}
```

Max Heap Program

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

struct Heap {
    int* arr;
    int size;
};

void propagateUp(struct Heap* mHeap, int i) {
    if (i == 0)
        return;

    int tmp = 0;
    int* heap = mHeap->arr;

    while ((*(heap + (i - 1) / 2) < *(heap + i)) && (i != 0)) {
        tmp = *(heap + (i - 1) / 2);
```

```
        *(heap + (i - 1) / 2) = *(heap + i);
        *(heap + i) = tmp;

        i = (i - 1) / 2;
    }
    return;
}

struct Heap* createHeap() {
    struct Heap* heap = malloc(sizeof(struct Heap));
    printf("Enter the number of elements:");
    scanf("%d", &(heap->size));
    heap->arr = malloc(heap->size * sizeof(int));

    int val;
    for (int i = 0; i < heap->size; i++) {
        printf("Enter a number:");
        scanf("%d", &val);
        heap->arr[i] = val;
        propagateUp(heap, i);
    }
    return heap;
}

int extractMax(struct Heap* h) {
    return h->arr[0];
}

void heapify(struct Heap* h, int i) {
    // curr is out of bounds
    if (i >= h->size)
        return;

    // children are out of bounds
    if (((2 * i + 1) >= h->size) || ((2 * i + 2) >= h->size))
        return;

    // both children are smaller than curr
    if ((*(h->arr + 2 * i + 1) < *(h->arr + i)) && (*(h->arr + 2 * i +
2) < *(h->arr + i)))
        return;

    // parent is smaller than both children
    if ((*(h->arr + 2 * i + 1) > *(h->arr + i)) && (*(h->arr + 2 * i +
2) > *(h->arr + i))) {

        // swapping curr with the larger child
        if (*(h->arr + 2 * i + 1) > *(h->arr + 2 * i + 2)) {
            int tmp = *(h->arr + 2 * i + 1);
            *(h->arr + 2 * i + 1) = *(h->arr + i);
            *(h->arr + i) = tmp;
            i = 2 * i + 1;
        } else {
```

```
            int tmp = *(h->arr + 2 * i + 2);
            *(h->arr + 2 * i + 2) = *(h->arr + i);
            *(h->arr + i) = tmp;
            i = 2 * i + 2;
        }
        heapify(h, i);
        return;
    }

    // left child is larger, right is smaller
    if (*(h->arr + 2 * i + 1) > *(h->arr + i)) {
        int tmp = *(h->arr + 2 * i + 1);
        *(h->arr + 2 * i + 1) = *(h->arr + i);
        *(h->arr + i) = tmp;
        i = 2 * i + 1;
        heapify(h, i);
        return;
    }

    // left child is smaller, right is larger
    if (*(h->arr + 2 * i + 2) > *(h->arr + i)) {
        int tmp = *(h->arr + 2 * i + 2);
        *(h->arr + 2 * i + 2) = *(h->arr + i);
        *(h->arr + i) = tmp;
        i = 2 * i + 2;
        heapify(h, i);
        return;
    }
}

void deleteMax(struct Heap* h) {
    h->arr[0] = h->arr[--h->size];
    for (int i = 0; i < h->size; i++)
        printf("%d, ", h->arr[i]);
    printf("\n");
    heapify(h, 0);
    return;
}

int main(void) {
    struct Heap* maxHeap = createHeap();
    for (int i = 0; i < maxHeap->size; i++)
        printf("%d, ", maxHeap->arr[i]);
    printf("\nRoot: %d\n", extractMax(maxHeap));
    deleteMax(maxHeap);
    for (int i = 0; i < maxHeap->size; i++)
        printf("%d, ", maxHeap->arr[i]);
    return 0;
}
```

Heap Sort

```c
#include <stdio.h>

void heapify(int arr[], int n, int i, int isMaxHeap) {
    int largestOrSmallest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (isMaxHeap) {
        // For max-heap, compare with left and right children
        if (left < n && arr[left] > arr[largestOrSmallest])
            largestOrSmallest = left;

        if (right < n && arr[right] > arr[largestOrSmallest])
            largestOrSmallest = right;
    } else {
        // For min-heap, compare with left and right children
        if (left < n && arr[left] < arr[largestOrSmallest])
            largestOrSmallest = left;

        if (right < n && arr[right] < arr[largestOrSmallest])
            largestOrSmallest = right;
    }

    // If the largest/smallest is not the root, swap them and
recursively heapify the affected sub-tree
    if (largestOrSmallest != i) {
        int temp = arr[i];
        arr[i] = arr[largestOrSmallest];
        arr[largestOrSmallest] = temp;

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largestOrSmallest, isMaxHeap);
    }
}

void heapSort(int arr[], int n, int isMaxHeap) {
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i, isMaxHeap);

    // One by one extract an element from the heap
    for (int i = n - 1; i >= 0; i--) {
        // Move current root to end
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        // Call heapify on the reduced heap
        heapify(arr, i, 0, isMaxHeap);
```

```
    }
}

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

int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: \n");
    printArray(arr, n);

    // Sort as Max Heap (Ascending Order)
    heapSort(arr, n, 1);
    printf("Sorted array (Max Heap): \n");
    printArray(arr, n);

    // Sort as Min Heap (Descending Order)
    heapSort(arr, n, 0);
    printf("Sorted array (Min Heap): \n");
    printArray(arr, n);

    return 0;
}
```

AVL Trees

```
//create an avl tree with 29
//insert 30,5

#include<stdio.h>
#include<stdlib.h>
#include<math.h>
struct node
{
    int data;
    struct node *left;
    struct node *right;
    // int height;
};


struct node *createNode(int data)
{
```

```
    struct node *newNode = (struct node *)malloc(sizeof(struct node));
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    // newNode->height = 1;
    return newNode;
}

int height(struct node *root)
{
    if (root == NULL)
        return 0;
    return 1 + fmax(height(root->left), height(root->right));
}

int getBalanceFactor(struct node *root)
{
    if (root == NULL)
        return 0;
    return height(root->left) - height(root->right);
}

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

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

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);
    else
        return root;

    int balanceFactor = getBalanceFactor(root);

    if (balanceFactor > 1 && data < root->left->data)
        return rightRotate(root);
    if (balanceFactor < -1 && data > root->right->data)
```

```
        return leftRotate(root);
    if (balanceFactor > 1 && data > root->left->data)
    {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }
    if (balanceFactor < -1 && data < root->right->data)
    {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }
    return root;
}
```