19Z310-DATA STRUCTURES LABORATORY

Sreeraghavan R
(22z261)
(Batch: 2022-2026)

BACHELOR OF ENGINEERING

Branch: COMPUTER SCIENCE AND ENGINEERING

Of Anna University

PSG COLLEGE OF TECHNOLOGY

(Autonomous Institution)

COIMBATORE - 641 004

PSG COLLEGE OF TECHNOLOGY

(Autonomous Institution)

COIMBATORE - 641 004

19Z310 – DATA STRUCTURES LABORATORY

Bona fide record of work done by

Sreeraghavan R

(22z261)

Dissertation submitted in partial fulfilment of the requirements for the degree of

BACHELOR OF ENGINEERING

Branch: COMPUTER SCIENCE AND ENGINEERING

of Anna University

……..

Dr. S. Lovelyn Rose

Faculty In-charge

Certified that the candidate was examined in the viva-voce examination held on ……..

…….. ……..……

(Internal Examiner) (External Examiner)

19Z310-DATA STRUCTURES LABORATORY

LAB EXPERIMENTS

NAME : Sreeraghavan R **ROLL NO:** 22z261

CLASS :B.E. CSE- G1 **BATCH:** 2022-2026

TABLE OF CONTENTS

S.No.	Title	Page No
1.	**Stacks**	3
2.	**Queues**	9
3.	**Deque**	13
4.	**Priority Queue**	18
5.	**Binary Search Tree**	20
6.	**Heaps**	26
7.	**AVL Trees**	31
8.	**Expression Trees**	34
9.	**Graphs**	36
10.	**Leet Code Problems**	40
11.	**CA2 Report**	44

Stacks:

Implementation of Stack using array:

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

void peek(int* array,int top){
    printf("%d\n",array[top]);
}

int push(int* arr,int size,int top,int val){
    if(++top<=size){
        arr[top] = val;
        return top;
    }
    printf("Stack Overflow\n");
    return top;
}

int pop(int* arr,int* top){
    if((*top)<0){
        printf("Stack Overflow");
        return *top;
    }
    int temp = arr[*top];
    (*top)--;
    return temp;
}


void display(int* arr,int top){
    for(int i = 0;i<=top;i++) printf("%d ",arr[i]);
    printf("\n");
}
```

Implementation of Stack using Linked Lists:

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

```c
struct node{
    int data;
    struct node* below;
};

typedef struct node node;

int isEmpty(node* s){
    return !s;
}

node* getNode(int val){
    node* newNode = (node *)malloc(sizeof(node));
    newNode->data = val;
    newNode->below = NULL;
    return newNode;
}

node* push(node* s,int val){
    node* new = getNode(val);
    new->below = s;
    return new;
}

void print(node* s){
    node* iter = s;
    while(iter){
        printf("%d ",iter->data);
        iter = iter->below;
    }
}


int pop(node* s){
    if(isEmpty(s)){
        printf("UNDERFLOW\n");
        return -999999;
    }
    int tbReturned = s->data;
    s = s->below;
    return tbReturned;
}
```

Infix to Postfix:

```
int intPeek(int* array,int top){
    return array[top];
}
int intPush(int* arr,int size,int top,float val){
    if(++top<=size){
        arr[top] = val;
        return top;
    }
    printf("Stack Overflow\n");
    return top;
}

int intPop(int* arr,int* top){
    if((*top)<0){
        printf("Stack Overflow");
        return *top;
    }
    int temp = arr[*top];
    (*top)--;
    return temp;
}

int precedence(char c){
    switch(c){
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
        default:
            return -1;
    }
}

int isOperator(char c){
    switch(c){
        case '+':
        case '-':
        case '*':
        case '/':
            return 1;
        default:
```

```
            return 0;
    }
}

float* conversion(char* exp){
    int size = strlen(exp);
    int stk[size];
    int i = 0;
    int top = -1;



    while(exp[i] != '\0'){
        //for number
        if(exp[i] > 47 && exp[i] < 58){
            int num = 0;
            while(exp[i] != ' '){
                num *= 10;
                num += exp[i]-48;
                i++;
            }
           printf("%d ",num);
        }

        if(exp[i] == '('){
            top = intPush(stk,size,top,exp[i]);
        }

        if(exp[i] == ')'){
            while(intPeek(stk,top) != '(') printf("%c ",intPop(stk,&top));
            intPop(stk,&top);
        }

        if(isOperator(exp[i])){
            while(precedence(intPeek(stk,top)) >= precedence(exp[i]))
printf("%c ",intPop(stk,&top));
            top = intPush(stk,size,top,exp[i]);
        }

        i++;
    }
}
```

Evaluation of Postfix:

```
float floatPeek(float* array,int top){
    return array[top];
}
int floatPush(float* arr,int size,int top,float val){
    if(++top<=size){
        arr[top] = val;
        return top;
    }
    printf("Stack Overflow\n");
    return top;
}

float floatPop(float* arr,int* top){
    if((*top)<0){
        printf("Stack Overflow");
        return *top;
    }
    float temp = arr[*top];
    (*top)--;
    return temp;
}
void floatDisplay(float* arr,int top){
    for(int i = 0;i<=top;i++) printf("%f ",arr[i]);
    printf("\n");
}

void charDisplay(int* arr,int top){
    for(int i = 0;i<=top;i++) printf("%c ",arr[i]);
    printf("\n");
}

void evalPostFix(char* exp){
    int i = 0;
    float op1,op2;
    int size = strlen(exp);
    float array[size];
    int top = -1;
    while(exp[i] != '\0'){
        //for number
        if(exp[i] > 47 && exp[i] < 58){
            int num = 0;
            while(exp[i] != ' '){
                num *= 10;
                num += exp[i]-48;
                i++;
            }
            top = floatPush(array,size,top,num);
        }
```

```
        //+
        if(exp[i] == 43){
            op2 = floatPop
        (array,&top);
            op1 = floatPop
        (array,&top);
            float temp = op1 + op2;
            top = floatPush(array,size,top,temp);

        }

        //-
        if(exp[i] == 45){
            op2 = floatPop
        (array,&top);
            op1 = floatPop
        (array,&top);
            float temp = op1 - op2;
            top = floatPush(array,size,top,temp);

        }

        //*
        if(exp[i] == 42){
            op2 = floatPop
        (array,&top);
            op1 = floatPop
        (array,&top);
            float temp = op1 * op2;
            top = floatPush(array,size,top,temp);

        }

        ///
        if(exp[i] == 47){
            op2 = floatPop
        (array,&top);
            op1 = floatPop
        (array,&top);
            float temp = op1 / op2;
            top = floatPush(array,size,top,temp);
        }
        i++;
    floatDisplay(array,top);
    }
}
```

Queue:

Implementation of Queue using Arrays:

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

void enqueue(int* q, int *front, int *rear, int size, int val){
    if(*front == -1){
        *front = 0;
        q[++(*rear)] = val;
        return;
    }
    if((*rear == size-1 && *front == 0)|| *rear == *front-1){
        printf("OVERFLOW\n");
        return;
    }
    if(*rear == size-1){
      *rear = 0;
      q[0] = val;
      return;
    }
    q[++(*rear)] = val;
}

int dequeue(int* q, int *front, int *rear, int size){
   if(*front == -1){
      printf("UNDERFLOW");
   }
   if(*front == *rear){
      *front = *rear = -1;
   }
   return q[(*front)++];
}

void display(int *arr,int *front , int *rear,int size){

   printf("Size is %d\n",size);
   if(*front ==-1 && *rear ==-1){
      printf("the list is empty ....! ");
      return ;
   }
   if(*rear<*front){
      for(int i=*front;i<size;i++){
         printf("%d ",arr[i]);
      }
      for(int i=0;i<=*rear;i++){
```

```
            printf("%d ",arr[i]);
        }
        printf("\n");
        return;
    }
    else{
        for(int i=*front;i<=*rear;i++){
            printf("%d ",arr[i]);
        }
        return;
    }
}
```

Implementation of Queue using Linked Lists:

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

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

struct Queue{
    struct node* front;
    struct node* rear;
};

typedef struct node node;
typedef struct Queue Queue;

node* createNode(int val){
    node* new = (node*)malloc(sizeof(node));
    new->data = val;
    new->next = NULL;
    return new;
}

Queue* createQueue(){
    Queue* new = (Queue*)malloc(sizeof(Queue));
    new->front = new->rear = NULL;
    return new;
}

bool isEmpty(Queue* queue){
    return !queue->front;
```

```
}


void enqueue(Queue* queue,int val){
    //queue is empty
    if(isEmpty(queue)){
        queue->front = queue->rear = createNode(val);
        return;
    }
    //not empty
    node* new = createNode(val);
    queue->rear->next = new;
    queue->rear = queue->rear->next;
}

int dequeue(Queue* queue){
    //queue is empty
    if(isEmpty(queue)){
        printf("UNDERFLOW\n");
        return -999999;
    }
    //not empty
    node* temp = queue->front;
    int tbReturned = temp->data;
    queue->front = queue->front->next;
    return tbReturned;
}

int front(Queue* queue){
    if(isEmpty(queue)){
        printf("EMPTY QUEUE\n");
        return -999999;
    }
    return queue->front->data;
}

void displayQueue(Queue* queue){
    if(isEmpty(queue)){
        printf("EMPTY QUEUE\n");
        return;
    }
    node* iter = queue->front;
    while(iter){
        printf("%d-->",iter->data);
        iter = iter->next;
    }
    printf("NULL\n");
}
```

```
void printFront(Queue* queue){
    printf("%d\n",front(queue));
}
```

Deque:

Implementation of Deque with Arrays:

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

void enqueue_front(int *queue, int *front, int *rear, int data, int size){
    // empty check
    if (*front == -1 && *rear == -1)
    {
        queue[0] = data;
        *front = 0;
        *rear = 0;
        return;
    }
    //overflow
    if((*front == 0 && *rear == size - 1) || (*front - *rear == 1)){
        printf("Overflow!\n");
        return;
    }
    // not overflow, but front is at start
    if(*front == 0){
        *front = size - 1;
        queue[*front] = data;
        return;
    }
    // normal case
    queue[--(*front)] = data;
}

void enqueue_rear(int *queue, int *front, int *rear, int data, int size){
    // empty check
    if (*front == -1 && *rear == -1)
    {
        queue[0] = data;
        *front = 0;
        *rear = 0;
        return;
    }
    //overflow
    if((*front == 0 && *rear == size - 1) || (*front - *rear == 1)){
        printf("Overflow!\n");
        return;
    }
    // not overflow, but rear is at end
    if(*rear == size - 1){
```

```
        *rear = 0;
        queue[*rear] = data;
        return;
    }
    // normal case
    queue[++(*rear)] = data;
}

int dequeue_front(int *queue, int *front, int *rear, int size){
    // empty check
    if (*front == -1 && *rear == -1)
    {
        printf("Empty queue\n");
        return -999999;
    }
    // one element
    if (*front == *rear)
    {
        int data = queue[*front];
        *front = -1;
        *rear = -1;
        return data;
    }
    // front at end
    if (*front == size - 1)
    {
        int data = queue[*front];
        *front = 0;
        return data;
    }
    //normal case
    return queue[(*front)++];
}

int dequeue_rear(int *queue, int *front, int *rear, int size){
    // empty check
    if (*front == -1 && *rear == -1)
    {
        printf("Empty queue\n");
        return -999999;
    }
    // one element
    if (*front == *rear)
    {
        int data = queue[*front];
        *front = -1;
        *rear = -1;
        return data;
```

```
    }
    // rear at start
    if (*rear == 0)
    {
        int data = queue[*rear];
        *rear = size - 1;
        return data;
    }
    //normal case
    return queue[(*rear)--];
}

void printArray(int * arr, int front, int rear, int size){
    if (front == -1 && rear == -1)
    {
        printf("Empty queue\n");
        return;
    }
    if (rear < front)
    {
        for (int i = front; i < size; i++)
        {
            printf("%d\t", arr[i]);
        }
        for (int i = 0; i <= rear; i++)
        {
            printf("%d\t", arr[i]);
        }
        printf("\n");
        return;
    }
    for (int i = front; i <= rear; i++)
    {
        printf("%d\t", arr[i]);
    }
    printf("\n");
}
```

Implementation of Deque with Linked Lists:

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

struct node {
    int data;
    struct node* prev;
```

```
    struct node* next;
};

typedef struct node node;

struct Deque {
    node* front;
    node* rear;
};
typedef struct Deque Deque;

node* createNode(int data) {
    node* newNode = (node*)malloc(sizeof(node));
    newNode->data = data;
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}

Deque* createDeque() {
    Deque* deque = (Deque*)malloc(sizeof(Deque));
    deque->front = NULL;
    deque->rear = NULL;
    return deque;
}

int isEmpty(Deque* deque) {
    return (deque->front == NULL);
}

void insertFront(Deque* deque, int data) {
    node* newNode = createNode(data);
    if (isEmpty(deque)) {
        deque->front = newNode;
        deque->rear = newNode;
    } else {
        newNode->next = deque->front;
        deque->front->prev = newNode;
        deque->front = newNode;
    }
}

void insertRear(Deque* deque, int data) {
    node* newNode = createNode(data);
    if (isEmpty(deque)) {
        deque->front = newNode;
        deque->rear = newNode;
    } else {
```

```
        newNode->prev = deque->rear;
        deque->rear->next = newNode;
        deque->rear = newNode;
    }
}

void deleteFront(Deque* deque) {
    if (isEmpty(deque)) {
        printf("UNDERFLOW\n");
    } else {
        node* temp = deque->front;
        deque->front = deque->front->next;
        if (deque->front == NULL) {
            deque->rear = NULL;
        } else {
            deque->front->prev = NULL;
        }
        free(temp);
    }
}

void deleteRear(Deque* deque) {
    if (isEmpty(deque)) {
        printf("UNDERFLOW\n");
    } else {
        node* temp = deque->rear;
        deque->rear = deque->rear->prev;
        if (deque->rear == NULL) {
            deque->front = NULL;
        } else {
            deque->rear->next = NULL;
        }
        free(temp);
    }
}

void display(Deque* deque) {
    if (isEmpty(deque)) {
        printf("EMPTY QUEUE\n");
    } else {
        node* current = deque->front;
        while (current != NULL) {
            printf("%d ", current->data);
            current = current->next;
        }
        printf("\n");
    }
}
```

Priority Queue:

Implementation of Priority Queue:

```
struct node{
    int data;
    struct node * next;
};

typedef struct node node;

struct Queue{
    node * front;
    node * rear;
};

typedef struct Queue Queue;


Queue ** createPriorityQueue(int size){
    Queue **pq = (Queue **)malloc(sizeof(Queue *) * size);
    for (int i = 0; i < size; i++)
    {
        pq[i] = (Queue *)malloc(sizeof(Queue));
        pq[i]->front = NULL;
        pq[i]->rear = NULL;
    }
    return pq;
}

void enqueue(Queue **pq, int data, int priority){
    node *new = (node *)malloc(sizeof(node));
    new->data = data;
    if (pq[priority]->front == NULL && pq[priority]->rear == NULL)
    {
        new->next = NULL;
        pq[priority]->front = new;
        pq[priority]->rear = new;
        return;
    }
    pq[priority]->rear->next = new;
    pq[priority]->rear = new;
    return;
}

int dequeue(Queue **pq, int size){
    for(int i = 0; i < size; i++){
```

```
        if(pq[i]->front != NULL){
            node *temp = pq[i]->front;
            int data = temp->data;
            pq[i]->front = pq[i]->front->next;
            free(temp);
            return data;
        }
    }
    printf("EMPTY QUEUE\n");
    return -999999;
}
```

Binary Search Trees:

Implementation of BST with Arrays:

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

void swap(int*l, int i, int j){
    int temp = l[i];
    l[i] = l[j];
    l[j] = temp;
}


int power2(int a){
    int i = 1;
    int j = 1;
    for(i;i<=a;i++){
        j *= 2;
    }
    return j;
}

int greater(int a,int b){
    if(a>=b){
        return 1;
    }
    return 0;
}

void insertElement(int* tree,int x){
    int i = 0;
    while(tree[i] != 0){
        if(greater(x,tree[i])) i = 2*i + 2;
        else i = 2*i + 1;
    }
    tree[i] = x;
}

void insertElements(int* tree,int* arr,int size){
    for(int i = 0;i<size;i++){
        insertElement(tree,arr[i]);
    }
}

int* createBinaryTree(int size){
```

```
    int max = power2(size);
    int* tree = (int*)calloc(max,sizeof(int));
    return tree;
}


int numLevels(int size){
    int i = 0;
    while(size != 0){
        size /= 2;
        i++;
    }
    return i;
}

//traverse indorder the left subtree
//process root
//traverse indored the right subtree
void inorder(int* tree,int i){
    if(tree[2*i+1] != 0) inorder(tree,2*i+1);
    printf("%d ",tree[i]);
    if(tree[2*i+2] != 0) inorder(tree,2*i+2);
}

//traverse postdorder the left subtree+
//traverse postorder the right subtree
//process root
void postorder(int* tree,int i){
    if(tree[2*i+1]) postorder(tree,2*i+1);
    if(tree[2*i+2]) postorder(tree,2*i+2);
    printf("%d ",tree[i]);
}

//process root
//traverse predorder the left subtree
//traverse preorder the right subtree
void preorder(int* tree,int i){
    printf("%d ",tree[i]);
    if(tree[2*i+1]) preorder(tree,2*i+1);
    if(tree[2*i+2]) preorder(tree,2*i+2);
}

int searchNode(int* tree,int val){
    int i = 0;
    while(tree[i] != 0 && tree[i] != val){
        if(greater(val,tree[i])) i = 2*i + 2;
        else i = 2*i + 1;
    }
    if(tree[i] == val) return i;
```

```
    return -1;
}
```

Implementation of BST with Linked Lists:

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


struct node{
    int data;
    struct node* lc;
    struct node* rc;
};

typedef struct node node;

enum nodeType{BOTH,LEFT,RIGHT,NONE};
enum childType{LC,RC,HEAD};

int typeNode(node* tree){
    if(tree->lc && tree->rc) return BOTH;
    if(tree->lc) return LEFT;
    if(tree->rc) return RIGHT;
    return NONE;
}


node* createNode(int val){
    node* newNode = (node*)malloc(sizeof(node));
    newNode-> data = val;
    newNode-> lc = NULL;
    newNode->rc = NULL;
    return newNode;
}

node* emptyNode(){
    return createNode(-1);
}

void printNode(node* cur){
    printf("%d\n",cur->data);
}

node *insertNodeBST(node *tree,node *newNode)
{
```

```
    //printf("test");
    if (tree == NULL)
        return newNode;
    if (newNode->data < tree->data)
        tree->lc = insertNodeBST(tree->lc, newNode);
    else
        tree->rc = insertNodeBST(tree->rc, newNode);
    return tree;
}

node *insertElement(node *tree,int val){
    return insertNodeBST(tree,createNode(val));
}

node *insertElements(node* tree,int* arr,int len){
    int i = 0;
    if(!tree) tree = createNode(arr[i++]);
    for(i;i<len;i++) tree = insertElement(tree,arr[i]);
    return tree;
}

node *searchNode(node* tree,int val){
    if(tree->data == val) return tree;
    if(tree->lc) if(val < tree->data) return searchNode(tree->lc,val);
    if(tree->rc) if(val > tree->data) return searchNode(tree->rc,val);
    return emptyNode();
}

node *searchParent(node* tree,int val,node* parent){
    if(tree->data == val) return parent;
    if(tree->lc) if(val < tree->data) return searchParent(tree->lc,val,tree);
    if(tree->rc) if(val > tree->data) return searchParent(tree->rc,val,tree);
    return emptyNode();
}

int typeChild(node* tree,int val){
    node* parent = searchParent(tree,val,tree);
    if(parent->data == val) return HEAD;
    if(val > parent->data) return RC;
    if(val < parent->data) return LC;
}
void inorder(node* tree){
    if(tree->lc) inorder(tree->lc);
    printf("%d ",tree->data);
    if(tree->rc) inorder(tree->rc);
}

void inordernl(node* tree){
```

```
    inorder(tree);
    printf("\n");
}

int nodeExists(node* cur){
    return cur->data != -1;
}

node* inorderSuccessor(node* cur){
    node* succ = cur;
    if(cur->rc) succ = cur->rc; else return succ;
    while(succ->lc) succ = succ->lc;
    return succ;
}

void deleteNode(node* tree,int val){
    node* cur = searchNode(tree,val);
    node* parent = searchParent(tree,val,tree);
    node* temp;

    if(typeNode(cur) == NONE){
        if(typeChild(tree,val) == LC){
            parent->lc = NULL;
            return;
        }
        parent->rc = NULL;

    }

    node* child;
    if(typeNode(cur) == LEFT){
        child = cur->lc;
        if(typeChild(tree,val) == LC){
            parent->lc = child;

            return;
        }
        parent->rc = child;
    }

    if(typeNode(cur) == RIGHT){
        child = cur->rc;
        if(typeChild(tree,val) == LC){
            parent->lc = child;

            return;
        }
        parent->rc = child;
```

```
    }

    if(typeNode(cur) == BOTH){
        int temp = inorderSuccessor(cur)->data;
        deleteNode(tree,temp);
        cur->data = temp;
    }
}
```

Heaps:

Implementation of Min Heap:

```
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#define left(i) (2*i+1)
#define right(i) (2*i+2)


struct heap{
    int* arr;
    int last;
};

typedef struct heap heap;

void swap(int*l, int i, int j){
    int temp = l[i];
    l[i] = l[j];
    l[j] = temp;
}


void percolateDown(heap* heap,int i){

    if(heap->arr[i]>heap->arr[left(i)] && heap->arr[i]>heap->arr[right(i)])
return;

    switch (heap->arr[left(i)]>heap->arr[right(i)])
    {
    case true:
        swap(heap->arr,i,left(i));
        percolateDown(heap,left(i));
        break;
    default:
        swap(heap->arr,i,right(i));
        percolateDown(heap,right(i));
        break;
    }


}

void percolateUp(heap* heap,int i){
    // in correct position
```

```
    int prevIndex = (i-1)/2;
    if(prevIndex < 0 ) return;
    if(heap->last <= 0) return;
    if(heap->arr[i] > heap->arr[prevIndex]) return;
    printf("Hello World!\n");

    swap(heap->arr,i,prevIndex);
    i = prevIndex;
    percolateUp(heap,prevIndex);

}

void insert(heap* heap,int val){
    heap->arr[heap->last] = val;
    percolateUp(heap,heap->last);
    heap->last++;
}

heap* createHeap(int size){
    heap* new = (heap*)malloc(sizeof(heap));
    new->arr = (int*) calloc(size,sizeof(int));
    new->last = 0;
    return new;
}

void inorder(heap* heap,int i){
    printf("%d %d\n",left(i),right(i));
    if(heap->arr[left(i)] != 0) inorder(heap,left(i));
    printf("%d ",heap->arr[i]);
    if(heap->arr[right(i)] != 0) inorder(heap,right(i));
}

void inordernl(heap* heap){
    inorder(heap,0);
    printf("\n");
}


int extractMin(heap* heap){
    return heap->arr[0];
}
```

Implementation of Max Heap:

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

```
#define left(i) (2*i+1)
#define right(i) (2*i+2)


struct heap{
    int* arr;
    int last;
};

typedef struct heap heap;

void swap(int*l, int i, int j){
    int temp = l[i];
    l[i] = l[j];
    l[j] = temp;
}


void percolateDown(heap* heap,int i){

    if(heap->arr[i]<heap->arr[left(i)] && heap->arr[i]<heap->arr[right(i)])
return;

    switch (heap->arr[left(i)]<heap->arr[right(i)])
    {
    case true:
        swap(heap->arr,i,left(i));
        percolateDown(heap,left(i));
        break;
    default:
        swap(heap->arr,i,right(i));
        percolateDown(heap,right(i));
        break;
    }


}

void percolateUp(heap* heap,int i){
    // in correct position
    int prevIndex = (i-1)/2;
    if(prevIndex < 0 ) return;
    if(heap->last <= 0) return;
    if(heap->arr[i] < heap->arr[prevIndex]) return;
    printf("Hello World!\n");

    swap(heap->arr,i,prevIndex);
    i = prevIndex;
    percolateUp(heap,prevIndex);
```

```
}

void insert(heap* heap,int val){
    heap->arr[heap->last] = val;
    percolateUp(heap,heap->last);
    heap->last++;
}

heap* createHeap(int size){
    heap* new = (heap*)malloc(sizeof(heap));
    new->arr = (int*) calloc(size,sizeof(int));
    new->last = 0;
    return new;
}

void inorder(heap* heap,int i){
    printf("%d %d\n",left(i),right(i));
    if(heap->arr[left(i)] != 0) inorder(heap,left(i));
    printf("%d ",heap->arr[i]);
    if(heap->arr[right(i)] != 0) inorder(heap,right(i));
}

void inordernl(heap* heap){
    inorder(heap,0);
    printf("\n");
}


int extractMax(heap* heap){
    return heap->arr[0];
}
```

Heap Sort:

```
void swap(int*l, int i, int j){
    int temp = l[i];
    l[i] = l[j];
    l[j] = temp;
}


void percolateDown(heap* heap,int i){

    if(heap->arr[i]>heap->arr[left(i)] && heap->arr[i]>heap->arr[right(i)])
return;
```

```
    switch (heap->arr[left(i)]>heap->arr[right(i)])
    {
    case true:
        swap(heap->arr,i,left(i));
        percolateDown(heap,left(i));
        break;
    default:
        swap(heap->arr,i,right(i));
        percolateDown(heap,right(i));
        break;
    }


}

void heapSort(int* arr, int size) {
    for (int i = size - 1; i > 0; i--) {
        swap(arr, 0, i);
        percolateDown(arr,i);
    }
}
```

AVL Trees:

Implementation of AVL Trees:

```
#include<stdio.h>
#include<stdlib.h>
#define max(a,b) ((a>b)?a:b)

struct node{
    int data;
    struct node *lc;
    struct node *rc;
    int height;
};

typedef struct node node;

int height(node *N) {
    if (!N)
        return 0;
    return N->height;
}


node* newNode(int data){
    node* new = (node*)malloc(sizeof(node));
    new->data = data;
    new->lc = NULL;
    new->rc = NULL;
    new->height = 1;
    return(new);
}


node *rightRotate(node *y) {
    node *x = y->lc;
    node *T2 = x->rc;

    x->rc = y;
    y->lc = T2;

    y->height = max(height(y->lc),height(y->rc)) + 1;
    x->height = max(height(x->lc),height(x->rc)) + 1;

    return x;
}
```

```
node *leftRotate(node *x) {
    node *y = x->rc;
    node *T2 = y->lc;

    y->lc = x;
    x->rc = T2;

    x->height = max(height(x->lc), height(x->rc)) + 1;
    y->height = max(height(y->lc),height(y->rc)) + 1;

    return y;
}

int getBalance(node *N){
    if (N == NULL)
        return 0;
    return height(N->lc) - height(N->rc);
}

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

    if (data < node->data)
        node->lc = insert(node->lc, data);
    else if (data > node->data)
        node->rc = insert(node->rc, data);
    else return node;

    node->height = 1 + max(height(node->lc), height(node->rc));

    int balance = getBalance(node);



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

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

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

    // Right Left Case
```

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

    return node;
}


void inorder(node* tree){
    if(tree->lc) inorder(tree->lc);
    printf("%d ",tree->data);
    if(tree->rc) inorder(tree->rc);
}

void inordernl(node* tree){
    inorder(tree);
    printf("\n");
}
```

Expression Trees:

Implementation of Expression Trees:

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

struct node {
    char data;
    struct node* lc;
    struct node* rc;
};

typedef struct node node;


node* createNode(char data) {
    node* new = (node*)malloc(sizeof(node));
    new->data = data;
    new->lc = new->rc = NULL;
    return new;
}

int isOperator(char c){
    switch(c){
        case '+':
        case '-':
        case '*':
        case '/':
            return 1;
        default:
            return 0;
    }
}

node* constructExpressionTree(char postfix[]) {
    int i = 0;
    node* stack[100];
    int top = -1;
    while (postfix[i] != '\0') {
        char symbol = postfix[i];
        if (!isOperator(symbol)) {
            stack[++top] = createNode(symbol);
        } else {
```

```
            node* new = createNode(symbol);
            new->rc = stack[top--];
            new->lc = stack[top--];
            stack[++top] = new;
        }
        i++;
    }
    return stack[top];
}


int evalExp(node* root) {
    if (root == NULL) {
        return 0;
    }
    if (!isOperator(root->data)) {
        return root->data - '0';
    } else {
        int leftValue = evalExp(root->lc);
        int rightValue = evalExp(root->rc);
        switch (root->data) {
            case '+':
                return leftValue + rightValue;
            case '-':
                return leftValue - rightValue;
            case '*':
                return leftValue * rightValue;
            case '/':
                if (rightValue != 0) {
                    return leftValue / rightValue;
                } else {
                    printf("Error: Division by zero.\n");

                }
            default:
                printf("Error: Invalid operator.\n");

        }
    }
}
```

Graphs:

Implementation of Graphs:

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

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

typedef struct node node;

void printNode(node* obj){
    printf("%d <-- %d --> %d\n",(obj->prev)->data,obj->data,(obj->next)-
>data);
}

node* getEnd(node* start){
    node* iter = start;
    while(1){
        if(iter->next == NULL){
            break;
        }
        iter = iter->next;
    }

    return iter;
}

node* insertAtEnd(node* start,int val){
    node* end = getEnd(start);
    node* new = (node*)malloc(sizeof(node));
    end->next = new;
    new->prev = end;
    new->data = val;
    new->next = NULL;

    return new;
}

void displayList(node* start){
    node* iter = start;
    while(1){
```

```
        printf(" %d <-->",iter->data);
        if(iter->next == NULL){
            printf(" NULL\n");
            break;
        }
        iter = iter->next;
    }
}

void deleteNode(node* start,node* toDelete){
    node* previous = toDelete->prev;
    node* nextobj = toDelete->next;
    printNode(previous);
    printNode(nextobj);
    free(toDelete);
}

node* searchNode(node* start,int val){
    if(start == NULL) return NULL;
    if(start->data == val) return start;
    searchNode(start->next,val);
}

node* createPath(int val){
    node* new = (node*)malloc(sizeof(node));
    new->prev = new->next = NULL;
    new->data = val;
    return new;
}

node** createAdjacencyList(int numNodes){
    node** new = (node**)malloc(sizeof(node*)*numNodes);
    for(int i = 0;i<numNodes;i++){
        new[i] = createPath(i);
    }
    return new;
}

void addEdge(node** adjList,int source, int destination){
    insertAtEnd(adjList[source],destination);
}

void displayAdjlist(node** adjList,int numNodes){
    for(int i = 0;i<numNodes;i++){
        displayList(adjList[i]);
    }
}
```

```
int push(int* arr,int size,int top,int val){
    if(++top<=size){
        arr[top] = val;
        return top;
    }
    printf("Stack Overflow\n");
    return top;
}

int pop(int* arr,int* top){
    if((*top)<0){
        printf("Stack Overflow");
        return *top;
    }
    int temp = arr[*top];
    (*top)--;
    return temp;
}


void bfs(node** adjList, int s){
    bool visited[100];
    for (int i = 0; i < 100; i++) {
        visited[i] = false;
    }

    int queue[100];
    int front = 0, rear = 0;

    visited[s] = true;
    queue[rear++] = s;

    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 start){
    bool visited[100];
    int stack[100];
```

```
    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;
        }
    }
}
```

Leetcode Problems:

Univalued Binary Tree:

```
/**
 * Definition for a binary tree struct TreeNode.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */
// bool isUnivalTree(struct TreeNode* root) {

// }

enum type{BOTH,LEFT,RIGHT,NONE};


int typeNode(struct TreeNode* tree){
    if(tree->left && tree->right) return BOTH;
    if(tree->left) return LEFT;
    if(tree->right) return RIGHT;
    return NONE;
}

bool isUnivalTree(struct TreeNode* tree){
    switch(typeNode(tree)){
        case BOTH:
            if((tree->val == tree->left->val)&&(tree->val == tree->right-
>val)){
                return isUnivalTree(tree->left) && isUnivalTree(tree->right);
            }
            else{
                return false;
            }
            break;
        case LEFT:
            if(tree->val == tree->left->val){
                return isUnivalTree(tree->left);
            }
            else{
                return false;
```

```
            }
            break;
        case RIGHT:
            if(tree->val == tree->right->val){
                return isUnivalTree(tree->right);
            }
            else{
                return false;
            }
            break;
        case NONE:
            return true;
        default:
            return true;
    }
}
```

Kth Smallest Element in a BST:

```
/**
 * Definition for a binary tree struct TreeNode.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */

struct TreeNode* createNode(int val){
    struct TreeNode* newNode = (struct TreeNode*)malloc(sizeof(struct
TreeNode));
    newNode-> val = val;
    newNode-> left = NULL;
    newNode->right = NULL;
    return newNode;
}
struct TreeNode* emptyNode(){
    return createNode(-1);
}
struct TreeNode *searchNode(struct TreeNode* tree,int val){
    if(tree->val == val) return tree;
    if(tree->left) if(val < tree->val) return searchNode(tree->left,val);
    if(tree->right) if(val > tree->val) return searchNode(tree->right,val);
    return emptyNode();
}
```

```
struct TreeNode *searchParent(struct TreeNode* tree,int val,struct TreeNode*
parent){
    if(tree->val == val) return parent;
    if(tree->left) if(val < tree->val) return searchParent(tree-
>left,val,tree);
    if(tree->right) if(val > tree->val) return searchParent(tree-
>right,val,tree);
    return emptyNode();
}



enum childType{LC,RC,HEAD};

int typeChild(struct TreeNode* tree,int val){
    struct TreeNode* parent = searchParent(tree,val,tree);
    if(parent->val == val) return HEAD;
    if(val > parent->val) return RC;
    if(val < parent->val) return LC;
    return -1;
}


struct TreeNode* inorderSuccessor(struct TreeNode* tree,struct TreeNode* cur){
    struct TreeNode* succ = cur;
    if(cur->right){
        succ = cur->right;
        while(succ->left) succ = succ->left;
        return succ;
    }
    int temp = succ->val;
    while(typeChild(tree,succ->val) != HEAD){
        succ = searchParent(tree,succ->val,tree);
        if(succ->val > temp) return succ;
    }
    return searchNode(tree,temp);
}


struct TreeNode* findMin(struct TreeNode* tree){
    if(tree->left) return findMin(tree->left);
    return tree;
}

int kthSmallest(struct TreeNode* root, int k) {
    struct TreeNode* smallest = findMin(root);
    struct TreeNode* kthSmallestNode = smallest;
    for(int i = 0;i < k-1;i++){
        kthSmallestNode = inorderSuccessor(root,kthSmallestNode);
```

```
    }
    return kthSmallestNode->val;
}
```