UCS 2312 Data Structures Lab

Exercise 1: Array ADT and its application

Create an ADT for the array data structure with the following functions. *arrADT* will have the integer array and size. [CO1, K3]

a. create(arrADT,size, array) – Create the array with the required elements
b. deleteAt(arrADT, pos) – Delete the specified element
c. insertAtEvery(arrADT,data) – Insert data before every element
d. changing the order of the elements.
e. printArray(arrADT) – prints the elements of the array

Write an application to use the arrADT to do the following:

- findPeek(arrADT) – return the peek elements. The element greater than its predecessor and successor
  Input: [3,5,2,6,4,11,2,3] Output: 5, 6, 11
- moveNegativesEnd(arrADT) – Move the negative elements to the end of the array by preserving the order of elements
  Input: [2,-11,-8,8,15,12,4,-3] Output: [2,8,15,12,4,-11,-8,-3]
- find pairs in an array that sum up to a specific target Input: [2, 5, 12, 11] Output: 17

Test the operations of arrADT with the following test cases

Operation	**Expected Output**
create(arrADT,20,[2,4,6,8,10])	2,4,6,8,10
deleteAt(arrADT, 3)	2,4,6,10
insertAtEvery(arrADT,1)	1,2,1,4,1,6,1,10
search(arrADT,1)	2
search(arrADT,2)	-1
printArray(arrADT)	1,2,1,4,1,6,1,10

Best practices to be followed:

- Design before coding
- Usage of algorithm notation
- Use of multi-file C program
- Versioning of code

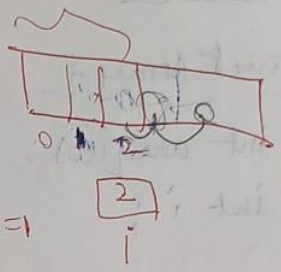
algorithm for insert middle.

define function insertmiddle (struct Array *A, n, p)

for j ← A->i to p, j--

A -> arr[j+1] ← A-> arr[j]

A -> arr[p] = n
A -> i + = 1
display (A)



2 > = 1

Code

```
Void insertmiddle (struct Array *A, int n, int p ) {
for (int j ← A->i ; j >=p ; j--) {
   A->arr[j+1] ← A-> arr[j];
}
A -> arr[p] = n ;
A->i + ← 1;
}
```

Algorithm for insertfront

define function insertfront (struct Array *A, n)

for j = A->i to 0, j--

A -> arr[j+1] ← A -> arr[j]

A -> arr[0] n

A -> i + = 1

Algorithm for insert end

```
define insertend (struct array *A, n){
A -> arr[A->i] = n
A->i += 1
}
```

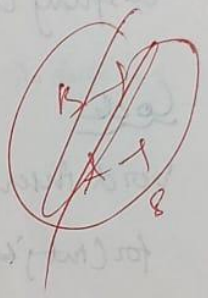
30/7/24

Ex 1 - Array ADT Implementation

ArrayADT.h

```
struct Array {
    int arr[100];
    int i;
};
void insert(struct Array *A, int x){
    A->arr[i++] = x;
}
void display(Struct Array *A) {
    for(int i=0; i<A->i; i++)
        printf("%d", A->arr[i]);
}
```

find the 3rd occurence of a given elt.



Program.c

```
#include<stdio.h>
#include "ArrayADT.h"
void main() {
 struct Array *A;
 A = (Struct Array*) malloc(sizeof(StructArray));
 insert(A, 10);
 insert(A, 20);
 display(A);
}
```

Algorithm for occourence

```
define function occurence (struct Array *A, int x)
    initialize count ← 0
    for j ← 0 to A->i
        if (A->arr[j] = x)
            count ++
        if (count = oc)
            display "Occurence at index " j
            return

    display "Not enough repetitions"
    return
```

x is the value

Algorithm for Two Sum

```
function TwoSum(Struct Array * A, sum)

for j = 0 to A->i-1 {
    for k = j+1 to k < A->i {
        if A->arr[j] + A->arr[k] = sum {
            printf("pair found at %d & %d", j, k)
            return 1;
}}
printf("No pairs")
return
}
```

find the peak element

4 10 7 16 22

```
for(j=len to len+1, j++)
    A->arr[j] = A->arr[j+1]
A->i = len
}
```

Algorithm to find peaks

```
define fn peak (struct Array *A) {
for (j=0 ; j < A->i ;) {
    if (j==0)
        if ( A->arr[j] > A->arr[j+1])
            print A->arr[j]
            j = j+2
    if (j == A->i-1)
        if ( A->arr[j] > A->arr[j-1]
            print A->arr[j]
            j=j+2
    else if ( A->arr[j] > both A->arr[j-1]
                and A->arr[j+1])
            print A->arr[j]
            j=j+2
    j=j+1 }}
```

Algorithm for deleting

```
define function deleteAt(Struct Array *A, pos)
    for(j = pos ; j < A->i ; j++)
        A->arr[j] = A->arr[j+1]
    A->i = A->i - 1 ;
    display(A)
```

Algorithm for inserting at every space

```
define function InsertAtEvery(Struct Array *A, n)
    for(j = A->i ; j >= 0 ; j--) {
        A->arr[j*2] = A->arr[j] ;
        A->arr[j*2 - 1] = n ;
    }
    A->i = A->i * 2
    display(A)
```

Algorithm for having negatives at end

```
define NegEnd(Struct Array *A) {
    len = A->i
    for(j = 0 to len , len++)
        if A->arr[j] < 0
            insertend(A, A->arr[j]), c+=1
    m = 0
    for(j = 0 to len ; j++)
        if A->arr[m] > 0
            m++
        else deleteAt(A, m)
```

Learning Outcomes: Week 1

Technical Outcomes:

Category	Rating	Comments
Design	2	Good
Understanding of DS	2	Good
Use of DS	1	Algorithm Utilization
Debugging	1	Need to do it faster

Best Practices

Category	Rating	Comments
Design before Coding	2	More planning
Usage of Algorithm notation	2	-
Usage of Multi file	3	-
Versioning	2	More progression needed

```c
#define ARRADT_H

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

typedef struct {
    int *array;
    int size;
} arrADT;

void create(arrADT *arr, int size, int *input_array) {
    arr->size = size;
    arr->array = (int *)malloc(size * sizeof(int));
    for (int i = 0; i < size; i++) {
        arr->array[i] = input_array[i];
    }
}

void deleteAt(arrADT *arr, int pos) {
    if (pos < 0 || pos >= arr->size) return;
    for (int i = pos; i < arr->size - 1; i++) {
        arr->array[i] = arr->array[i + 1];
    }
    arr->size--;
    arr->array = (int *)realloc(arr->array, arr->size * sizeof(int));
}

void insertAtEvery(arrADT *arr, int data) {
    int new_size = arr->size * 2;
    int *new_array = (int *)malloc(new_size * sizeof(int));
    for (int i = 0, j = 0; i < arr->size; i++) {
        new_array[j++] = data;
        new_array[j++] = arr->array[i];
    }
    free(arr->array);
    arr->array = new_array;
    arr->size = new_size;
}
```

```c
void reverseArray(arrADT *arr) {
    for (int i = 0; i < arr->size / 2; i++) {
        int temp = arr->array[i];
        arr->array[i] = arr->array[arr->size - 1 - i];
        arr->array[arr->size - 1 - i] = temp;
    }
}

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

void findPeek(arrADT *arr) {
    for (int i = 1; i < arr->size - 1; i++) {
        if (arr->array[i] > arr->array[i - 1] && arr->array[i] > arr->array[i + 1]) {
            printf("%d ", arr->array[i]);
        }
    }
    printf("\n");
}

void moveNegativesEnd(arrADT *arr) {
    int *temp = (int *)malloc(arr->size * sizeof(int));
    int j = 0;
    for (int i = 0; i < arr->size; i++) {
        if (arr->array[i] >= 0) {
            temp[j++] = arr->array[i];
        }
    }
    for (int i = 0; i < arr->size; i++) {
        if (arr->array[i] < 0) {
            temp[j++] = arr->array[i];
        }
    }
    free(arr->array);
    arr->array = temp;
}
```

```c
void findPairs(arrADT *arr, int target) {
    for (int i = 0; i < arr->size; i++) {
        for (int j = i + 1; j < arr->size; j++) {
            if (arr->array[i] + arr->array[j] == target) {
                printf("(%d, %d) ", arr->array[i], arr->array[j]);
            }
        }
    }
    printf("\n");
}

int search(arrADT *arr, int value) {
    for (int i = 0; i < arr->size; i++) {
        if (arr->array[i] == value) {
            return i;
        }
    }
    return -1;
}
```

```c
#include "arrADT.h"

int main() {
    int array1[] = {2, 4, 6, 8, 10};
    arrADT arr1;
    create(&arr1, 5, array1);
    printf("Original Array: ");
    printArray(&arr1);

    printf("After deleting at index 3: ");
    deleteAt(&arr1, 3);
    printArray(&arr1);

    printf("After inserting 1 at every position: ");
    insertAtEvery(&arr1, 1);
    printArray(&arr1);

    int index1 = search(&arr1, 1);
    printf("Search for 1: %d\n", index1);

    int index2 = search(&arr1, 2);
    printf("Search for 2: %d\n", index2);

    printf("Final Array: ");
    printArray(&arr1);

    // Free the allocated memory
    free(arr1.array);

    return 0;
}
```

```
Original Array: 3 5 2 6 4 11 2 3
Peek Elements: 5 6 11
Array before moving negatives to end: 2 -11 -8 8 15 12 4 -3
Array after moving negatives to end: 2 8 15 12 4 -11 -8 -3
Pairs that sum to 17: (5, 12)
```

Technical Outcomes

Design	2	good
Understanding of DS	3	good
Use of DS	2	bad
Debugging	1	Extremely bad

Best Practices

Design before coding	2	Needs improvement
Usage of Algo	3	Very good
Multifile	3	Very good
Versioning	0	

UCS 2312 Data Structures Lab

Assignment 2: ListADT and its application

Create an ADT for the linked list data structure with the following functions. list*ADT* will have the integer array and size. [CO1, K3]

a. insert(header,data) – Insert data into the list using inserting at front
b. display(header) – Display the elements of the list
c. insertAtEnd(header,data) – Insert data at the end of the list
d. searchElt(header, key) – return the value if found, otherwise return -1
e. deleteElt(header,data) – Deletes the element data
f. findMiddleElt(header) – find the middle element in the list
g. reverseList(header) – Reverse the list
h. length(header) – find the length of the list

Write an application to use the listADT to do the following:

Test the operations of listADT with the following test cases

Operation	**Expected Output**
length(header)	0
insert(header,2)	2
insert(header,4)	4, 2
insert(header,6)	6, 4, 2
insert(header,8)	8, 6, 4, 2
length(header)	4
insertLast(header,1)	8, 6, 4, 2, 1
insertLast(header,3)	8, 6, 4, 2, 1, 3
length(header)	6
findMiddleElt(header)	2 or 4
reverseList(header)	3, 1, 2, 4, 6, 8
searchElt(4)	4
searchElt(5)	-1
deleteElt(2)	8, 6, 4, 1, 3

Best practices to be followed:

- Design before coding
- Usage of algorithm notation
- Use of multi-file C program
- Versioning of code

Write a program to add the given two polynomials Example:

Poly1: $6x^7+8x^3+7x^2+9$,
Poly2: $7x^3+6$
Resultant Poly: $6x^7+15x^3+7x^2+15$

```
Struct node*
search (Struct node * header, int key )
{
  Struct node *ptr
  ptr = header -> next
  while (ptr ≠ Null)
      if (ptr -> data = key)
          return ptr
      else
          ptr ⇐ ptr -> next
  return - NULL
}

Void insertAfter(Struct node *header, int key, int n) {
  Struct node *ptr, *temp ;
  ptr = find(header, key)
  temp = ( ) malloc( )
  if (ptr = = NULL)
      print(Not found)
  else
      temp -> next = ptr -> next
      ptr -> next = temp
```

LinkedList_ADT.h

```
Struct Node { int data; Struct Node *next; };
Void display (Struct Node *header) {
    Struct Node * ptr;
    ptr = header -> next
    while (ptr != NULL) {
        printf ("%d ", ptr -> data);
        ptr = ptr -> next;
    }
}

Void insert (Struct Node * header, int x) {
    Struct Node * ptr, *temp;
    ptr = header -> next;
    temp = (Struct Node*) malloc (sizeof (Struct Node));
    temp -> data = x;
    header -> next = temp;
    temp -> next = ptr;
}
```

LinkedList_APP.c

```
#include <stdio.h>
#include <stdlib.h>
#include "LinkedList_ADT.h"
main () {
    Struct Node * header;
    header = (Struct Node*) malloc (sizeof (Struct Node)); header -> next
                                                           = NULL
```

6/8/24

Ex-2 Linked Lists ADT

write ADT ASAP.

```
Struct Node {
    int data
    Struct node *next;
};
// have function prototypes before main
main() {
    Struct node * header;
    header = (Struct node*) Malloc (Size of(Struct node));
    header -> next = NULL;
}
```

In short ()malloc() for convenience

```
insert (Struct node * header, int x) {
    Struct node * temp, *ptr;
    temp = () Malloc ();
    temp -> data = x;
    header -> next = temp;
    temp -> next = ptr;
}

display (Struct node *header) {
    Struct node * ptr
    ptr = header -> next
    ~~printf("%d", header->data);~~
    while (ptr ≠ NULL) {
        printf(ptr->data)
        ptr = ptr -> next
    }
}
```

```
insert(header,10);
insert(header,20);
insert(header,30);
display(header);
}
```

insert before find

20 30 40

Algorithm to find previous

```
define fn findPrevious(struct node *header, key)
    struct node *before, *ptr
    before ← header
    ptr ← header->next
    while(ptr ≠ NULL)
        if(ptr->data = key)
            return before
        else
            before ← ptr
            ptr ← ptr->next
    return NULL
}
```

Algorithm for insert Before

Define fn insertBefore (struct node *header, key, val)

struct node * ptr, *temp

ptr = findPrevious (header, key)

temp = (*)malloc() temp->data = n

if (ptr = NULL)

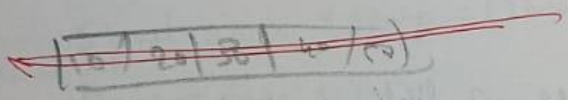
print "key not found"

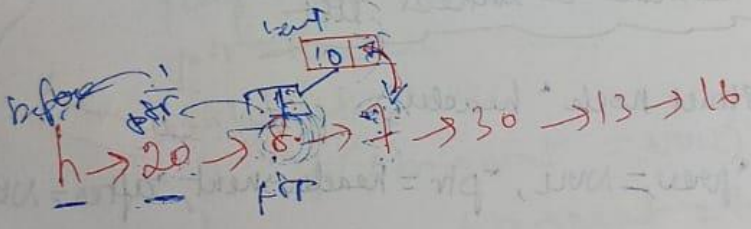
else

{ temp -> next = ptr -> next

ptr -> next = temp

}





for

Algorithm for Insert After (* header, key, n)

Struct node * ptr, * temp

temp = (*)malloc()

if (ptr = NULL)

Display invalid

else

temp->next = ptr->next

ptr -> next = temp

temp -> data = n

```
Void del (Struct node * header, val)
    Struct node * ptr, * prev
    ptr = header -> next
    prev = header
    while (ptr != NULL)
        if ptr -> data = val
            prev -> next = ptr -> next
            free(ptr)
        else
            ptr = ptr -> next
            prev = prev -> next
```

Algorithm to Reverse a linked list

```
void reverse (Struct node * header)
    Struct node *prev = NULL, *ptr = header -> next, *after = N
    while (ptr != NULL)
        after = ptr -> next
        ptr -> next = prev
        prev = ptr
        ptr = after
    header -> next = prev
```

```
int search (struct node *header, val) {
    Struct node *ph = header->next
    while (ph != NULL)
        if(ph->data = val)
            return 1
        ph = ph->next
    return -1
}
```

Algorithm for Addition of polynomials

```
struct
Node* addpolys (Struct Node* P1, (Struct node* P2)
    Struct node* ptr1 = P1->next, *ptr2 = P2->next
        , result
    result = ()malloc
    result->next = NULL
    while (ptr1 != NULL and ptr2 != NULL)
        if (ptr1->exp = ptr2->exp)
            insert (result, ptr1->exp* ptr1->coeff+ptr2->c
            ptr1 = ptr1->next; ptr2 = ptr2->next
        else if (ptr1->exp > ptr2->exp)
            insert(result, ptr1->exp, ptr1->coff)
            ptr1 = ptr1->next
        else if (ptr2->exp > ptr1->exp)
            insert(result, ptr2->exp, ptr2->coeff)
            ptr2 = ptr2->next
```

```
while (ptr1 != NULL)
    insert(result, ptr1->emp, ptr1->coeff)
    ptr1 = ptr1->next
while (ptr2 != NULL)
    insert(result, ptr2->emp, ptr2->coeff)

return *result
}
```

earning Outcomes : Technical (Week 2)

ategory	Rating	Comments
Design	3	Did tracing well
nderstanding DS	3	Understood properly
per Usage	2	Could utilise better
bugging	2	Do fast & Do right

aning Outcomes : Best Practices (Week 2)

tegory	Rating	Comments
ign before Coding	3	Did tracing we
ge & Algo Notation	1	Better notation & Han writi
ng Multi files	3	Used APP & ADT
rsioning	1	Must do Bette

Algorithm to find length

```
int length (struct node *header){
    count = 0
    Struct node * ptr = header -> next
    while (ptr != NULL)
        count += 1
        ptr = ptr -> next
    return count
```

Algorithm to insert at end

```
void insert end (node* header)
    Struct node * temp = () malloc (), ptr
    temp -> data = n
    ptr = header
    while (ptr ≠ NULL)
        if (ptr -> next = NULL)
            ptr -> next = temp
            temp -> next = NULL
```

Algorithm to find middle

```
int middle find (node * header)
    Struct Node ptr =
    index = length (header) / 2 ; ptr
    int count
    while (ptr ≠ NULL)
        count += 1
        if (count == index)
            display ptr -> data
            break
        ptr = ptr -> next
```

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

// ---------------------------- List ADT ----------------------------

// Node structure for List ADT
typedef struct ListNode {
    int data;
    struct ListNode *next;
} ListNode;

// Function to insert at the front of the list
void insert(ListNode **header, int data) {
    ListNode *newNode = (ListNode *)malloc(sizeof(ListNode));
    newNode->data = data;
    newNode->next = *header;
    *header = newNode;
}

// Function to display the elements of the list
void display(ListNode *header) {
    ListNode *temp = header;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

// Function to insert data at the end of the list
void insertAtEnd(ListNode **header, int data) {
    ListNode *newNode = (ListNode *)malloc(sizeof(ListNode));
    newNode->data = data;
    newNode->next = NULL;
```

```c
    if (*header == NULL) {
        *header = newNode;
    } else {
        ListNode *temp = *header;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

// Function to search an element in the list
int searchElt(ListNode *header, int key) {
    ListNode *temp = header;
    while (temp != NULL) {
        if (temp->data == key) {
            return key;
        }
        temp = temp->next;
    }
    return -1;
}

// Function to delete an element from the list
void deleteElt(ListNode **header, int data) {
    ListNode *temp = *header, *prev = NULL;

    if (temp != NULL && temp->data == data) {
        *header = temp->next;
        free(temp);
        return;
    }
```

```
    while (temp != NULL && temp->data != data) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == NULL) return;

    prev->next = temp->next;
    free(temp);
}

// Function to find the middle element of the list
int findMiddleElt(ListNode *header) {
    ListNode *slow = header;
    ListNode *fast = header;

    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;
    }

    return slow != NULL ? slow->data : -1;
}

// Function to reverse the list
void reverseList(ListNode **header) {
    ListNode *prev = NULL, *current = *header, *next = NULL;

    while (current != NULL) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
```

```
    *header = prev;
}

// Function to find the length of the list
int length(ListNode *header) {
    int len = 0;
    ListNode *temp = header;
    while (temp != NULL) {
        len++;
        temp = temp->next;
    }
    return len;
}

// --------------------------- Polynomial ADT ---------------------------

// Node structure for Polynomial
typedef struct PolyNode {
    int coeff;
    int exp;
    struct PolyNode *next;
} PolyNode;

// Function to create a new polynomial node
PolyNode* createPolyNode(int coeff, int exp) {
    PolyNode *newNode = (PolyNode *)malloc(sizeof(PolyNode));
    newNode->coeff = coeff;
    newNode->exp = exp;
    newNode->next = NULL;
    return newNode;
}
```

```c
// Function to insert a node in the polynomial
void insertPolyNode(PolyNode **poly, int coeff, int exp) {
    PolyNode *newNode = createPolyNode(coeff, exp);
    if (*poly == NULL || (*poly)->exp < exp) {
        newNode->next = *poly;
        *poly = newNode;
    } else {
        PolyNode *temp = *poly;
        while (temp->next != NULL && temp->next->exp > exp) {
            temp = temp->next;
        }
        newNode->next = temp->next;
        temp->next = newNode;
    }
}

// Function to display polynomial
void displayPoly(PolyNode *poly) {
    while (poly != NULL) {
        printf("%dx^%d", poly->coeff, poly->exp);
        poly = poly->next;
        if (poly != NULL) {
            printf(" + ");
        }
    }
    printf("\n");
}

// Function to add two polynomials
PolyNode* addPolynomials(PolyNode *poly1, PolyNode *poly2) {
    PolyNode *result = NULL;
    while (poly1 != NULL && poly2 != NULL) {
        if (poly1->exp == poly2->exp) {
            int sum = poly1->coeff + poly2->coeff;
```

```c
    while (poly1 != NULL && poly2 != NULL) {
        if (poly1->exp == poly2->exp) {
            int sum = poly1->coeff + poly2->coeff;
            if (sum != 0) {
                insertPolyNode(&result, sum, poly1->exp);
            }
            poly1 = poly1->next;
            poly2 = poly2->next;
        } else if (poly1->exp > poly2->exp) {
            insertPolyNode(&result, poly1->coeff, poly1->exp);
            poly1 = poly1->next;
        } else {
            insertPolyNode(&result, poly2->coeff, poly2->exp);
            poly2 = poly2->next;
        }
    }

    while (poly1 != NULL) {
        insertPolyNode(&result, poly1->coeff, poly1->exp);
        poly1 = poly1->next;
    }

    while (poly2 != NULL) {
        insertPolyNode(&result, poly2->coeff, poly2->exp);
        poly2 = poly2->next;
    }

    return result;
}
```

```c
#include "ListADT.h"

int main() {
    // Testing List ADT operations
    printf("Testing List ADT:\n");
    ListNode *header = NULL;
    printf("Length: %d\n", length(header));

    insert(&header, 2);
    display(header);

    insert(&header, 4);
    display(header);

    insert(&header, 6);
    display(header);

    insert(&header, 8);
    display(header);

    printf("Length: %d\n", length(header));

    insertAtEnd(&header, 1);
    display(header);

    insertAtEnd(&header, 3);
    display(header);

    printf("Length: %d\n", length(header));

    printf("Middle Element: %d\n", findMiddleElt(header));

    reverseList(&header);
    display(header);
```

```c
    deleteElt(&header, 2);
    display(header);

    // Testing Polynomial ADT operations
    printf("\nTesting Polynomial ADT:\n");
    PolyNode *poly1 = NULL, *poly2 = NULL, *result = NULL;

    // Polynomial 1: 6x^7 + 8x^3 + 7x^2 + 9
    insertPolyNode(&poly1, 6, 7);
    insertPolyNode(&poly1, 8, 3);
    insertPolyNode(&poly1, 7, 2);
    insertPolyNode(&poly1, 9, 0);

    // Polynomial 2: 7x^3 + 6
    insertPolyNode(&poly2, 7, 3);
    insertPolyNode(&poly2, 6, 0);

    // Display the polynomials
    printf("Polynomial 1: ");
    displayPoly(poly1);

    printf("Polynomial 2: ");
    displayPoly(poly2);

    // Add the two polynomials
    result = addPolynomials(poly1, poly2);

    // Display the result
    printf("Resultant Polynomial: ");
    displayPoly(result);

    return 0;
}
```

```
Testing List ADT:
Length: 0
2
4 2
6 4 2
8 6 4 2
Length: 4
8 6 4 2 1
8 6 4 2 1 3
Length: 6
Middle Element: 2
3 1 2 4 6 8
Search 4: 4
Search 5: -1
3 1 4 6 8

Testing Polynomial ADT:
Polynomial 1: 6x^7 + 8x^3 + 7x^2 + 9x^0
Polynomial 2: 7x^3 + 6x^0
Resultant Polynomial: 6x^7 + 15x^3 + 7x^2 + 15x^0
```

Technical Outcomes

Design	3	nice
Understanding of DS	3	Understood prprly
Use of DS	2	Could do better
Debugging	1	Was hard

Best Practices

Design before coding	3	nice
Usage of Algo	1	Better notation
Multifile	3	did
Versioning	1	didnt

UCS 2312 Data Structures Lab
Exercise 3: Doubly Linked List and its applications

Create an ADT for the doubly linked list data structure with the following functions. Each node which consists of integer data, address of left and right nodes [CO1, K3]

Create a ListADT which has implementations for the following operations

1. Insert an item in the front of the list void insertFront(listADT L, int c)
2. Display the items from the list void displayItems(listADT L)
3. Delete the item present in the list void deleteItem(listADT L, int c)
4. Search an element in the list and return the number of occurrences int searchItem(listADT L, int c)

Write a program in C to test the ListADT for its operations with the following test cases.

Testcase:
Initially L is Empty insertFront(L,6) ❼
header←❼6
insertEnd(L,2) ❼ header←❼2←❼6 insertMiddle(L,2,1) ❼
header←❼2←❼1←❼6
insertMiddle(L,2,1) ❼ header←❼2←❼1←❼1←❼6 search(L,1) ❼
2

In addition, do the following operations:

1. Delete the nodes with odd numbers in the list.
2. Replace the nodes with odd sum digit values with random odd numbers
3. Add a single digit to a DLL.

Example: Input:
header←❼1←❼9←❼9←❼9
digit =6
Output: header←❼2←❼0←❼0←❼5
Input:
header←❼1←❼9←❼9←❼3
digit =6
Output: header←❼1←❼9←❼9←❼9

Input:
header←❼9←❼9←❼9←❼9
digit =6
Output:
header←❼1←❼0←❼0←❼0←❼5

```
void display(struct node * header)
{
    struct node *ptr, *end;
    ptr = header->right;
    while(ptr != NULL)
    {
        printf("%d", ptr->data);
        end = ptr;
        ptr = ptr->right;
    }
    while(end != header)
    {
        printf("%d", ptr->data);
        end = end->left;
    }
```

check whether the list is Bitonic / Strictly increasing decreasing

~~1 2 3~~

1 → 2 → 3 → 4 → 5

5 → 4 → 3 → 2 → 1

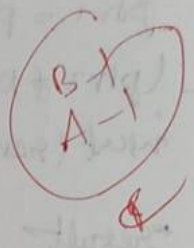
5 4 3 11 12 13

20/8/24

Ex 3

Doubly Linked list - Implementation & Applcn

Doubly Linked List - ADT file



```
struct node {
    int data ;
    struct node *left ;
    struct node *right ;
}
void insertf(struct node * header, int x) {
    struct node * temp, * ptr ;
    temp = (struct node*) malloc (sizeof(struct node)) ;
    temp -> data = x ;
    ptr = header -> right
    if (ptr == NULL) {
        header -> right = temp ;
        temp -> left = header ;
        temp -> right = NULL }
    else {
        temp -> right = ptr ;
        ptr -> left = temp ;
        temp -> left = header ;
        header -> right = temp ; } }
```

Algorithm for increasing / decreasing / Bitonic

```
Void checkBitonic (struct node * header){
    inc = 0, dec = 0, change = 0
    ptr = header -> right
    if (ptr = NULL or ptr -> right = NULL){
        print "List too short"
        return }
    while (ptr ≠ NULL){
        if (ptr -> right -> data > ptr -> data){
            inc = 1
            if (dec = 1){
                dec = 0
                change += 1 }}
        if (ptr -> right -> data < ptr -> data){
            dec = 1
            if (inc = 1){
                inc = 0
                change += 1 }}
        ptr = ptr -> right }
    if (change = 1)
        print "Bitonic"
    if (change = 0 and inc = 1)
        print "Strictly increasing"
    if (change = 0 and dec = 1)
        print "Strictly decreasing"
```

```
if (change > 1)
    print "Random"
return
}
```

1 2 3 4 3 2 1

Learning Outcomes

Tech Category	Rating	Comments
Design	2	Can utilise Data Structure Better
DSA	2	
Usage	2	
Debug	3	Struggled a lot

Best Practice		
Design	2	Can be utilised better
Algo Notation	2	
Modularity	3	Perfect
Version	0	Didn't do

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

// Structure for a Doubly Linked List node
typedef struct Node {
    int data;
    struct Node *prev;
    struct Node *next;
} Node;

// Structure for the ListADT
typedef struct {
    Node *head;
} ListADT;

// Function to create a new node
Node* createNode(int data) {
    Node *newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}

// Function to initialize the list
void initializeList(ListADT *L) {
    L->head = NULL;
}

// Insert an item in the front of the list
void insertFront(ListADT *L, int c) {
    Node *newNode = createNode(c);
    if (L->head == NULL) {
        L->head = newNode;
    } else {
        newNode->next = L->head;
        L->head->prev = newNode;
        L->head = newNode;
    }
}

// Insert an item at the end of the list
void insertEnd(ListADT *L, int c) {
    Node *newNode = createNode(c);
    if (L->head == NULL) {
        L->head = newNode;
    } else {
        Node *temp = L->head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->prev = temp;
    }
}

// Insert an item in the middle after a specific value
void insertMiddle(ListADT *L, int after, int c) {
    Node *temp = L->head;
    while (temp != NULL && temp->data != after) {
        temp = temp->next;
    }
    if (temp != NULL) {
        Node *newNode = createNode(c);
        newNode->next = temp->next;
        if (temp->next != NULL) {
            temp->next->prev = newNode;
        }
```

```
        temp->next = newNode;
        newNode->prev = temp;
    } else {
        printf("Element %d not found.\n", after);
    }
}

// Display the items from the list
void displayItems(ListADT *L) {
    Node *temp = L->head;
    printf("List: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

// Delete an item from the list
void deleteItem(ListADT *L, int c) {
    Node *temp = L->head;
    while (temp != NULL && temp->data != c) {
        temp = temp->next;
    }
    if (temp != NULL) {
        if (temp->prev != NULL) {
            temp->prev->next = temp->next;
        } else {
            L->head = temp->next;
        }
        if (temp->next != NULL) {
            temp->next->prev = temp->prev;
        }
        free(temp);
    } else {
        printf("Element %d not found.\n", c);
    }
}

// Search for an element and return the number of occurrences
int searchItem(ListADT *L, int c) {
    Node *temp = L->head;
    int count = 0;
    while (temp != NULL) {
        if (temp->data == c) {
            count++;
        }
        temp = temp->next;
    }
    return count;
}
```

```
List: 6
List: 6 2
List: 6 2 1
List: 6 2 1 1
Occurrences of 1: 2
List: 6 2 1
```

Technical Outcomes

Design	2	Needs improvement
Understanding of DS	2	Needs improvement
Use of DS	2	
Debugging	3	

Best Practices

Design before coding	2	Needs improvement
Usage of Algo	2	
Multifile	3	
Versioning	0	

UCS 2312 Data Structures Lab

Assignment 4: StackADT and its application

Create an ADT for the stack data structure with the following functions. stack*ADT* will have the integer array, top and size. [CO1, K3]

a. createStack(top) – initialize size and top with -1
b. push(top,data) – push data into the stack if stack is not full. Print a message when stack is
full
c. pop(top) – decrements the top by 1
d. peek(top)– returns the element at top, if stack is not empty, otherwise returns -1
e. isEmpty(top) – returns 1 if stack empty, otherwise returns 0
f. isFull(top) – returns 1 if stack full, otherwise returns 0

Test the operations of stackADT with the following test cases

Operation	**Expected Output**
peek(top)	Empty
push(top,2)	2
push(top,4)	4, 2
push(top,6)	6, 4, 2
push(top,8)	Full
pop(top)	
peek(top)	4
peek(top)	4
pop(top)	
pop(top)	
peek(top)	Empty
pop(top)	
pop(top)	
push(top,11)	11
peek(top)	11

Best practices to be followed:

- Design before coding
- Usage of algorithm notation
- Use of multi-file C program
- Versioning of code

Application using Stack

1. Apply the stack to check whether the given input is balanced or not?
   Examples:
   [[ ), [()()], (([][]))
2. Convert the given decimal number into binary using stack Example: 14
   Ans: 1110

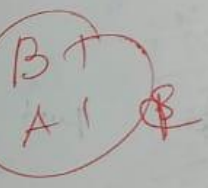
3. Understand the following procedure to use Stack and trace it for the input: input: [34, 3, 31, 98, 92, 23] and write an algorithm.
    - Create a temporary stack say **tmpStack**.
    - While input stack is NOT empty do this:
        - Pop an element from input stack call it **temp**
        - while temporary stack is NOT empty and top of temporary stack is greater than temp,
          pop from temporary stack and push it to the input stack
        - push **temp** in temporary stack • Print the numbers in tmpStack Predict the output ??? = [ ]
4. Given an array of integers temperatures represents the daily temperatures, return *an array* answer *such that* answer[i] *is the number of days you have to wait after the* i^{th} *day to get a warmer temperature*. If there is no future day for which this is possible, keep answer[i] == 0 instead. (use stack)

   Example 1:
   Input: temperatures = [73,74,75,71,69,72,76,73]
   Output: [1,1,4,2,1,1,0,0] **Example 2:**
   Input: temperatures = [30,40,50,60]
   Output: [1,1,1,0]

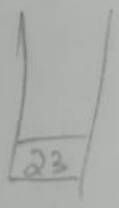
3/9/24

Implementation of Application of Stack & Queue

```
fn1(struct stack *s){
struct stack *t; t=malloc;
while(peek(s)!=-1){
    int temp = pop(s)
    while(peek(t)!=-1 && peek(t)>temp
        push(s, pop(t))
    push(t, temp)
}
for(int i=0; i<s->size; i++){
    print s->arr[i]}
```



24, 3, 31, 96, 92, 23



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

struct Stack{
    int size;
    int top;
    int a[100];
};

void createStack(struct Stack * s, int maxEle){
    s->size = maxEle-1;
    s->top = -1;
}

int isEmpty(struct Stack * s){
    if(s->top == -1)
        return 1;
    return 0;
}

int isFull(struct Stack * s){
    if(s->top == s->size)
        return 1;
    return 0;
}

void Push(struct Stack * s, int x){
    if(isFull(s) == 1)
        printf("\nStack is full\n");
    else{
        s->top ++;
        s->a[s->top] = x;
    }
}
```

```c
void pop (struct Stack * s){
    if(isEmpty(s) == 1)
        printf("\nStack empty\n");
    else{
        s->top --;
    }
}

int top (struct Stack * s){
    if(isEmpty(s) == 1)
        return 9999999;
    else
        return s-> a[s->top];
}

int isBalanced(char *input) {
    struct Stack s;
    createStack(&s, 100);

    for (int i = 0; input[i] != '\0'; i++) {
        char ch = input[i];

        if (ch == '(' || ch == '[' || ch == '{') {
            Push(&s, ch);
        } else if (ch == ')' || ch == ']' || ch == '}') {
            if (isEmpty(&s)) return 0;

            char topChar = top(&s);
            if ((ch == ')' && topChar != '(') ||
                (ch == ']' && topChar != '[') ||
                (ch == '}' && topChar != '{')) {
                return 0;
            }
```

```
            pop(&s);
        }
    }

    return isEmpty(&s);
}

void decimalToBinary(int num) {
    struct Stack s;
    createStack(&s, 100);

    while (num > 0) {
        Push(&s, num % 2);
        num /= 2;
    }

    while (!isEmpty(&s)) {
        printf("%d", top(&s));
        pop(&s);
    }
    printf("\n");
}


void sortStack(struct Stack *input) {
    struct Stack tmpStack;
    createStack(&tmpStack, 100);

    while (!isEmpty(input)) {
        int temp = top(input);
```

```
        pop(input);

        while (!isEmpty(&tmpStack) && top(&tmpStack) > temp) {
            Push(input, top(&tmpStack));
            pop(&tmpStack);
        }

        Push(&tmpStack, temp);
    }

    while (!isEmpty(&tmpStack)) {
        printf("%d ", top(&tmpStack));
        pop(&tmpStack);
    }
    printf("\n");
}

void dailyTemperatures(int temperatures[], int n, int result[]) {
    struct Stack stack;
    createStack(&stack, 100);

    for (int i = n - 1; i >= 0; i--) {
        while (!isEmpty(&stack) && temperatures[i] >= temperatures[top(&stack)]) {
            pop(&stack);
        }

        result[i] = isEmpty(&stack) ? 0 : top(&stack) - i;
        Push(&stack, i);
    }
}
```

```c
int main()
{
    char input[] = "{[()]}";
    if (isBalanced(input)) {
        printf("The input is balanced.\n");
    } else {
        printf("The input is not balanced.\n");
    }

    int num = 14;
    printf("Binary of %d: ", num);
    decimalToBinary(num);

    struct Stack stack;
    createStack(&stack, 100);

    int arr[] = {34, 3, 31, 98, 92, 23};
    int n = sizeof(arr) / sizeof(arr[0]);

    for (int i = 0; i < n; i++) {
        Push(&stack, arr[i]);
    }

    printf("Sorted stack: ");
    sortStack(&stack);

    int temperatures[] = {73, 74, 75, 71, 69, 72, 76, 73};
    int n = sizeof(temperatures) / sizeof(temperatures[0]);
    int result[n];

    dailyTemperatures(temperatures, n, result);

    printf("Output: ");
```

```c
    printf("Output: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", result[i]);
    }
    printf("\n");

    return 0;
}
```

```
The input is balanced.
Binary of 14: 1110
Sorted stack: 98 92 34 31 23 3
Output: 1 1 4 2 1 1 0 0
```

Technical Outcomes

Design	2	Needs improvement
Understanding of DS	2	Needs improvement
Use of DS	3	
Debugging	3	

Best Practices

Design before coding	2	Needs improvement
Usage of Algo	3	
Multifile	3	
Versioning	3	

UCS 2312 Data Structures Lab

Assignment 5: QueueADT and its applications

Create an ADT for the circular queue data structure with the following functions. queue*ADT* will have the integer array, front and rear. [CO1, K3]

a. createQueue(Q) – initialize size and front and rear with -1
b. enqueue(Q,data) – push data into the queue if queue is not full.
c. Dequeue(Q) – returns the data if queue is not empty. otherwise, the function returns -1
d. isEmpty(Q) – returns 1 if queue empty, otherwise returns 0
e. isFull(Q) – returns 1 if queue full, otherwise returns 0

Test the operations of queueADT with the following test cases

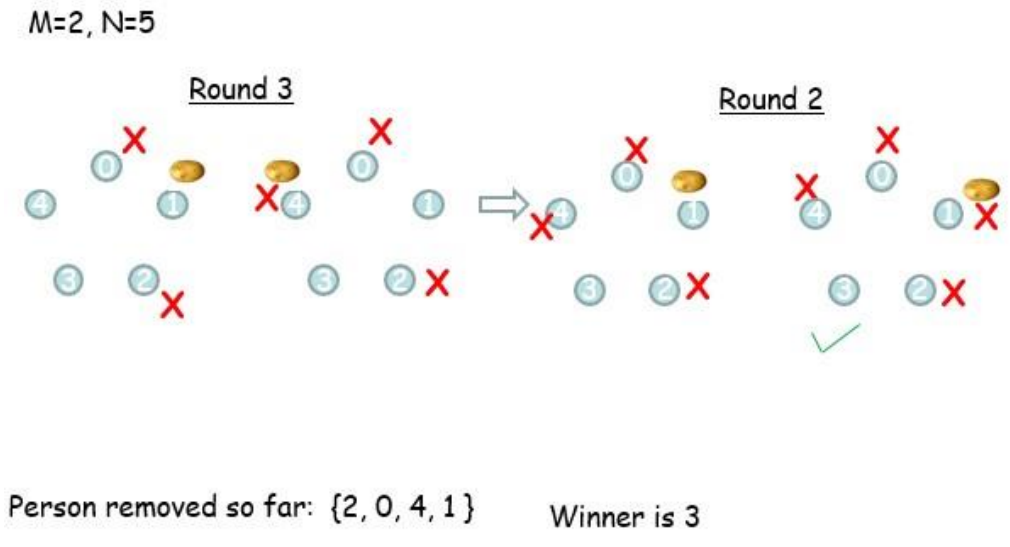
Operation	**Expected Output**
createQueue(Q, 2)	
enqueue(Queue,10)	10
enqueue(Queue,20)	20
enqueue(Queue,30)	Full
dequeue(Queue)	10
dequeue(Queue)	20
dequeue(Queue)	Empty

Best practices to be followed:

- Design before coding
- Usage of algorithm notation
- Use of multi-file C program
- Versioning of code

Application using Queue

Implement an **integer Circular Queue ADT** and driver to use the queue. Write an algorithm to play **Hot Potato Game** to find the winner.



Learning Outcomes

Technical

* Did test
* Utilized these skill in all fi
Data structure

Design

* Can make design with dif
Architecture

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

struct Queue {
    int f, r, size;
    int a[100];
};

void createQ(struct Queue * Q, int s) {
    Q->f = -1;
    Q->r = -1;
    Q->size = s;
}

int isFull(struct Queue * Q) {
    return (Q->r + 1) % Q->size == Q->f;
}

int isEmpty(struct Queue * Q) {
    return Q->f == -1;
}

void enqueue(struct Queue * Q, int ele) {
    if (isFull(Q)) {
        printf("\nQueue is full");
    } else {
        if (isEmpty(Q)) {
            Q->f = 0;
        }
        Q->r = (Q->r + 1) % Q->size;
        Q->a[Q->r] = ele;
    }
}
```

```c
void dequeue(struct Queue * Q) {
    if (isEmpty(Q)) {
        printf("\nQueue underflow");
    } else if (Q->f == Q->r) {
        printf("\n %d is the element dequeued", Q->a[Q->f]);
        Q->f = -1;
        Q->r = -1;
    } else {
        printf("\n %d is the element dequeued", Q->a[Q->f]);
        Q->f = (Q->f + 1) % Q->size;
    }
}
```

```c
#include "CQ.h"
int main() {
    struct Queue *Q;
    Q = (struct Queue*)malloc(sizeof(struct Queue));
    createQ(Q, 3);
    enqueue(Q, 1);
    enqueue(Q, 2);
    enqueue(Q, 3);
    dequeue(Q);
    dequeue(Q);
    enqueue(Q, 4);
    enqueue(Q, 5);
    dequeue(Q);
    dequeue(Q);
    dequeue(Q);
    dequeue(Q);
    return 0;
}
```

```c
void josephus(int n, int k) {
    struct Queue Q;
    createQ(&Q, n);

    // Enqueue all people
    for (int i = 1; i <= n; i++) {
        enqueue(&Q, i);
    }

    while (!isSizeOne(&Q)) {
        // Skip k-1 people
        for (int i = 0; i < k - 1; i++) {
            enqueue(&Q, dequeue(&Q));
        }
        // Eliminate the k-th person
        printf("Person %d is eliminated\n", dequeue(&Q));
    }

    // The last remaining person is the winner
    printf("The last remaining person is %d\n", dequeue(&Q));
}
```

```c
int n, k;
printf("Enter the number of people: ");
scanf("%d", &n);
printf("Enter the step size (k): ");
scanf("%d", &k);

josephus(n, k);
```

```
 1 is the element dequeued
 2 is the element dequeued
 3 is the element dequeued
 4 is the element dequeued
 5 is the element dequeued
Queue underflow%
```

```
Enter the number of people: 5
Enter the step size (k): 2
Person 2 is eliminated
Person 4 is eliminated
Person 1 is eliminated
Person 5 is eliminated
The last remaining person is 3
```

Technical Outcomes

Design	2	Needs improvement
Understanding of DS	3	
Use of DS	3	
Debugging	3	

Best Practices

Design before coding	2	Needs improvement
Usage of Algo	3	
Multifile	3	
Versioning	3	

UCS 2312 Data Structures Lab

Assignment 5: BSTADT and its application

Create an ADT for the binary search tree data structure with the following functions. Each node which consists of integer data, address of left and right children.

[CO2, K3]

a. insertBST(t,data) – insert data into BST
b. inorder(t) – display the tree using inorder traversal
c. preorder(t) – display the tree using preorder traversal
d. postorder(t) – display the tree using postorder traversal
e. levelorder(t) – display the tree hierarchically
f. findmin(t)– returns the minimum element in the tree
g. search(t,key) – returns the element found, otherwise returns NULL
h. delete(t,elt) – delete the given elt from tree

1. Demonstrate the BSTADT with the following test case

Insert(t,29)
Insert(t,23)
Insert(t,4)
Insert(t,13)
Insert(t,39)
Insert(t,31)
Insert(t,45)
Insert(t,56)
Insert(t,49)
Inorder(t) ❼ 4,13,23,29,31,39,45,49,56 Levelorder(t)❼ 1st Level ❼ 29
2nd level ❼ 23, 39
3rd Level ❼4, 31, 45
4th Level ❼13, 56
5th Level ❼ 49 Findmin(t) ❼
4
Find(t, 13) ❼ Found, value is
3 Find(t,3) ❼ Not found

2. Write an application to do the following

a. Given a sorted array. Write a function that creates a Balanced Binary Search Tree using array elements.

Examples:

Input: arr[] = {1, 2, 3}
Output: A Balanced BST

Explanation: all elements less than 2 are on the left side of 2 , and all the elements greater than 2 are on the right side

Input: arr[] = {1, 2, 3, 4}
Output: A Balanced BST

```
      3
    /  \
  2    4
 /
1
```

b. Given a Binary Search Tree (BST), find the second largest element.

Example:

Input: *Root of below BST*

```
  10
 /
5
```

Output: *5*

Input: *Root of below BST*

```
    10
   /  \
  5    20
        \
         30
```

Output: *20*

c. Count the number of nodes in tree within the given range

d. Find sum of k smallest elements in the given BST

e.

f. Test case for the Application (a)

Input: Tree 1

```
        10
      /    \
     5      50
    /      /  \
   1      40  100
```

Input: Tree2

```
        10
      /    \
     5      50
    /      /  \
   1      40  100
```

Tree1 and Tree2 are identical with a set of elements (b) Tree1 not complete (c) Tree1

Range: [**5**, 45] Output: 3

Nodes are 5, 10, 40

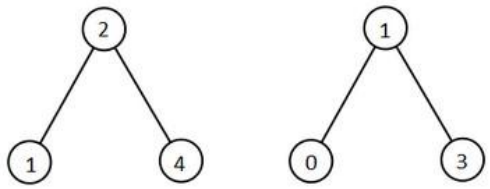
Tree2 Range: [**1**, 45]

Output: 4

Nodes are 1,5, 10, 40

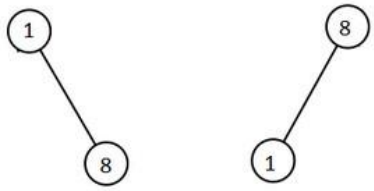
Given two binary search trees `root1` and `root2`, return *a list containing all the integers from both trees sorted in* **ascending**

Example 1:




```
Input: root1 = [2,1,4], root2 = [1,0,3]
Output: [0,1,1,2,3,4]
```

Example 2:




```
Input: root1 = [1,null,8], root2 = [8,1]
Output: [1,1,8,8]
```

Write a function to check whether the given tree is BST

Best practices to be followed:

- Design before coding
- Usage of algorithm notation
- Use of multi-file C program
- Versioning of code

Learning Outcome

Tech Category	Rating	Comments
Design	3	} Nice design
DSA	3	
Version	2	} could try hard
Debugging	3	} Struggled as usual

Best Practices

Design	3	Best
Algo Matching	2	Could do better
Multiple	3	Did
~~Algo Optimize~~		
Version	0	Didn't

Exp 6

10/9/24

BST - ADT & Applications

BST-ADT.h

```
struct tree{
    int data;
    struct tree *left, *right;
};
struct tree * insert (struct tree *t, int x){
① if (t == NULL){
        t = (...) malloc()
        t->data = x;
        t->left = NULL;
        t->right = NULL;
        printf ("inserted");
    }
② if (x < t->data)
        t->left = insert (t->left, x)
③ if (x > t->data)
        t->right = insert (t->right, x)
④ return t;
}
void inorder (struct tree *t){
    if (t->left != NULL)
        inorder (t->left)
    printf (t->data)
    if (t->right != NULL)
        inorder (t->right);
}
```

```
struct tree * delete (struct tree *t, int x) {
    struct tree *temp;
    if (x < t->data)
        t->left = delete (t->left, x);
    else if (x > t->data)
        t->right = delete (t->right, x)
    else if (t->left == NULL || t->right == NULL) {
        temp = t
        if (t->right == NULL)
            t = t->left;
        else if (t->left == NULL)
            t = t->right;
    }
    else if (t->left && t->right) {
        temp = findMin (t->right)
        t->data = temp->data
        t->right = delete (t->right, t->data
    }
    return t;
}
```

```
int findMin(struct tree *t) {
    while (t->left != NULL)
        t = t->left;
    return t->data;
}
int findMax(struct tree *t)
{
    while (t->right != NULL)
        t = t->right;
    return t->data;
}
int find(struct tree *t, int x) {
    if (t == NULL)
        return 0;
    else {
        if (t->data == x)
            return 1;
        if (x < t->data)
            return find(t->left, x);
        if (x > t->data)
            return find(t->right, x);
    }
}
```

```c
struct tree{
    int data;
    struct tree *left, *right;
};

struct tree *insert(struct tree *t, int x) {
    if (t == NULL) {
        t = (struct tree *)malloc(sizeof(struct tree));
        t->data = x;
        t->left = NULL;
        t->right = NULL;
    }
    if (x < t->data)
        t->left = insert(t->left, x);
    if (x > t->data)
        t->right = insert(t->right, x);
    return t;
}

struct tree* newNode(int data) {
    struct tree* node = (struct tree*)malloc(sizeof(struct tree));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return node;
}

// Function to find the node with the minimum value
struct tree *findMin(struct tree *t) {
    while (t->left != NULL)
        t = t->left;
    return t; // Now returning the node, not the int value
}
```

```c
// Function to find the node with the maximum value
struct tree *findMax(struct tree *t) {
    while (t->right != NULL)
        t = t->right;
    return t;
}

// Delete a node from the BST
struct tree *deleteNode(struct tree *t, int x) {
    if (t == NULL) return t;

    if (x < t->data)
        t->left = deleteNode(t->left, x);
    else if (x > t->data)
        t->right = deleteNode(t->right, x);
    else {
        // Node to be deleted found
        if (t->left == NULL || t->right == NULL) {
            struct tree *temp = t->left ? t->left : t->right;

            // No child case
            if (temp == NULL) {
                temp = t;
                t = NULL;
            } else { // One child case
                *t = *temp;
            }
            free(temp);
        } else {
            // Node with two children
            struct tree *temp = findMin(t->right); // Find the minimum in the right subtree
            t->data = temp->data; // Copy the inorder successor's value to this node
            t->right = deleteNode(t->right, temp->data); // Delete the inorder successor
```

```
        }
    }
    return t;
}

// Traversals
void inorder(struct tree *t) {
    if (t != NULL) {
        inorder(t->left);
        printf("%d   ", t->data);
        inorder(t->right);
    }
}

void preorder(struct tree *t) {
    if (t != NULL) {
        printf("%d   ", t->data);
        preorder(t->left);
        preorder(t->right);
    }
}

void postorder(struct tree *t) {
    if (t != NULL) {
        postorder(t->left);
        postorder(t->right);
        printf("%d   ", t->data);
    }
}

// Function to create a balanced BST from a sorted array
struct tree* createBalancedBST(int arr[], int start, int end) {
```

```
    if (start > end)
        return NULL;

    int mid = (start + end) / 2;
    struct tree* root = newNode(arr[mid]);
    root->left = createBalancedBST(arr, start, mid - 1);
    root->right = createBalancedBST(arr, mid + 1, end);

    return root;
}

// Function to find the second largest element
void findSecondLargestUtil(struct tree* root, int* count) {
    if (root == NULL || *count >= 2)
        return;

    findSecondLargestUtil(root->right, count);
    (*count)++;

    if (*count == 2) {
        printf("Second Largest: %d\n", root->data);
        return;
    }

    findSecondLargestUtil(root->left, count);
}

void findSecondLargest(struct tree* root) {
    int count = 0;
    findSecondLargestUtil(root, &count);
}

// Function to count nodes in a range
int countNodesInRange(struct tree* root, int low, int high) {
```

```c
int countNodesInRange(struct tree* root, int low, int high) {
    if (root == NULL)
        return 0;

    if (root->data >= low && root->data <= high)
        return 1 + countNodesInRange(root->left, low, high) + countNodesInRange(root->right, low, high);
    else if (root->data < low)
        return countNodesInRange(root->right, low, high);

    return countNodesInRange(root->left, low, high);
}

// Function to find sum of k smallest elements
void sumKSmallestUtil(struct tree* root, int* count, int k, int* sum) {
    if (root == NULL || *count >= k)
        return;

    sumKSmallestUtil(root->left, count, k, sum);

    if (*count < k) {
        *sum += root->data;
        (*count)++;
    }

    sumKSmallestUtil(root->right, count, k, sum);
}

int sumKSmallest(struct tree* root, int k) {
    int count = 0;
    int sum = 0;
    sumKSmallestUtil(root, &count, k, &sum);
    return sum;
}
```

```c
int main() {
    int arr[] = {1, 2, 3, 4};
    int n = sizeof(arr) / sizeof(arr[0]);

    // Creating a balanced BST from the sorted array
    struct tree* root = createBalancedBST(arr, 0, n - 1);
    printf("Inorder traversal of the Balanced BST:\n");
    inorder(root);
    printf("\n");

    // Finding second largest element
    findSecondLargest(root);

    // Counting nodes in range [1, 3]
    int low = 1, high = 3;
    printf("Count of nodes in range [%d, %d]: %d\n", low, high, countNodesInRange(root, low, high));

    // Finding sum of k smallest elements
    int k = 2;
    printf("Sum of %d smallest elements: %d\n", k, sumKSmallest(root, k));

    return 0;
}
```

```
Inorder traversal of the Balanced BST:
1   2   3   4
Second Largest: 3
Count of nodes in range [1, 3]: 3
Sum of 2 smallest elements: 3
```

Technical Outcomes

Design	2	Needs improvement
Understanding of DS	2	Needs improvement
Use of DS	3	
Debugging	3	

Best Practices

Design before coding	2	Needs improvement
Usage of Algo	3	
Multifile	1	Needs improvement
Versioning	3	

UCS 2312 Data Structures Lab
Assignment 7: Implementation of AVL Tree

Design an ADT for the AVL Tree data structure with the following functions. Each node consists of a character data, address of left, right and parent nodes [CO1, K3]

a. insertAVL(t, data) – insert data into BST
b. hierarchical(t) – display the tree in hierarchical fashion
c. findParent(t, key) – will return the parent of the given data
d. findLeaf(t) – returns the number of leaf nodes
e. findDepth(t,x) – returns the depth of the node x

Demonstrate the AVL ADT with the insertion of the following character data one at a time.

H, I, J, B, A, E, C, F, D, G, K, L

Ex 7 17/7/24

AVL Tree - ADT & Implementation

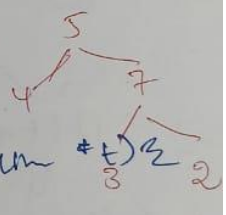
AVLADT.h

```
struct AVLtree {
   int data, height;
   struct AVLtree * left, *right;
};
int height(struct AVLtree *t){
   if(t==NULL)
      return -1;
   else
      return t->height
}
truct AVLtree *insert(struct AVLtree *t, int x){
   if (t==NULL) {
      t=(malloc());
      t->data=x;
      t->height=0;
      t->left = t->right = NULL;
   }
   else if(x<t->data) {
      t->left= insert(t->left,x);
      if(height(t->left)-height(t->right)==2)
         if(x<t->left->data)
            t=singlerotatewithleft(t);
         else doublerotatewithleft(t); }
   else if(x>t->data) {
      t->right = insert(t->right);
      if(height(t->right)-height(t->left)==2)
         if(x>t->left->data) t=singlerotatewithright
         else t=doublerotatewithright(t); }
```

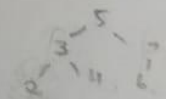
```
t->height = max(height(t->left), height(t->right)) + 1;
return t;
}

struct AVLNode * singleRotateWithLeft(struct AVLNode * k2)
{
    struct AVLNode *k1;
    k1 = k2->left;
    k1->right = k2; k2->height = max(height(k2->left), height(k2->
    k2->left = k1->right; k1->height = max height k1 left, Hig
    return k1;
}
struct AVLNode * singleRotateWithright(struct AVLNode * k1){
    struct AVLNode *k2;
    k2 = k1->right
    k1->right = k2->left;
    k2->left = k1;
    k1->height = max(h(k1->l), h(k1->r)) + 1;
    k2->height = max(h(k2->l), h(k2->r)) + 1;
    return k2;
}

struct AVLNode* doubleleft(struct AVLNode *k3){
    k3->left = singleRotateWithRight(k3->left);
    k3 = singleRotateWithleft(k3);
    return k3; }
struct AVLNode* doubleright(struct AVLNode *k1){
    k1->right = singleRotateWithleft(k1->right);
    k1 = singleRotatewithright(k1);
    return k1; }
```

```
void hierarchy (struct AVLtree *t, int lvl) {
  for (int i=0; i<lvl; i++) {
      printf ("\t");
  }
  if (t==NULL) {
      printf ("-1\n");
      return;
  }
  printf ("%d ------ \n", t->data);
  hierarchy (t->left, lvl+1);
  hierarchy (t->right, lvl+1);
}
```

Find the inverse of BST



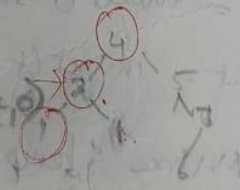

```
struct AVLtree * invert (struct AVLtree *t) {
  if (t==NULL)
      return NULL;
  t->left = invert (t->left);
  t->right = invert (t->right);
  struct AVLtree *temp = t->left;
  t->left = t->right;
  t->right = temp;
  return t;
}
```



Check whether BST / not

```
int CheckBst (struct AVL *t)
{
  static int a = -1;
  if (t->left != NULL)
    if CheckBST (t->left) return -
  if (t->data > a)
      a = t->data;
  else
      return -1; // NOT BST
  if (t->right != NULL)
    if CheckBST (t->right) rel
  if (a == 0) a = -1;
  return 1; // is BST
}
```

then in main CheckBST(t);



ind height of the tree

```
t maxheight(struct AVLtree *t, int r){
    static int max=0, a=0;
    if(a>=max)
        max=a
    if (t->left!=NULL){
        a++
        maxheight(t->left, r);
    if (t->right!=NULL){
        a++
        maxheight(t->right, r);
    a--;
    if(r==0)
        max=a=0;
    return max;
}
```

for recursion for the like in

in main, call with r=0

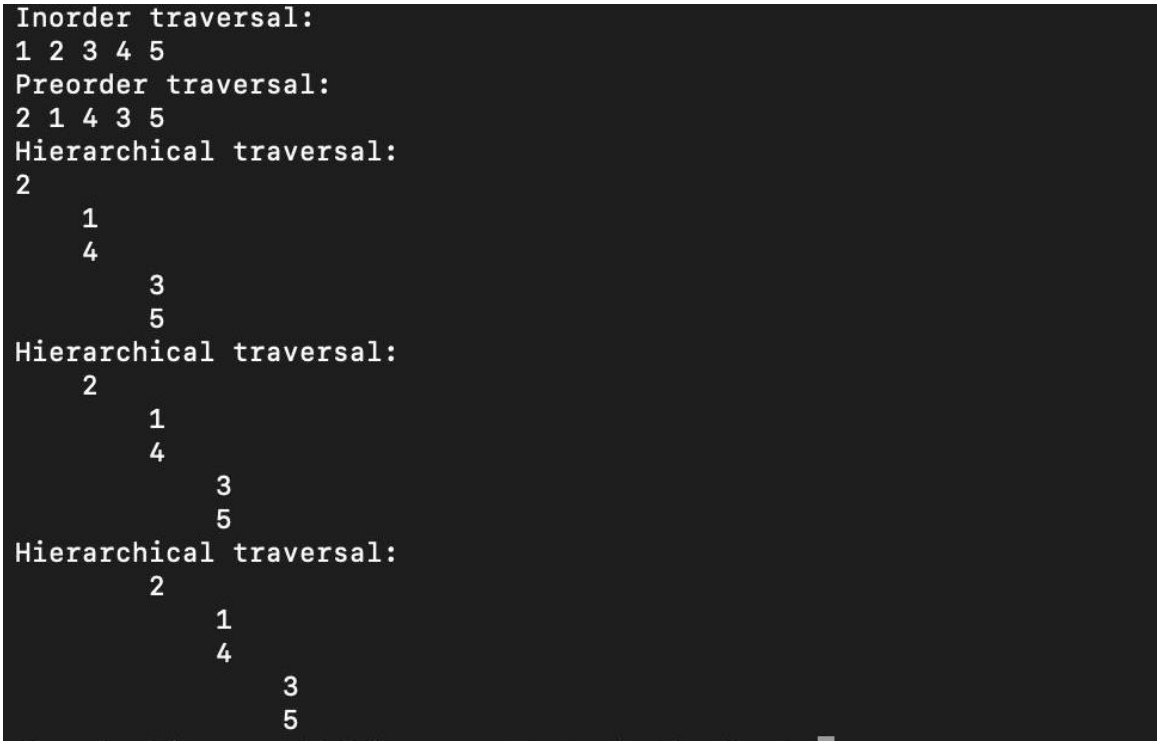
Learning Outcomes

Algo Dt.	2
Design	3
Usability	3
Version	0
Usage	2
DSA	2

```c
#include <stdlib.h>
#include <stdio.h>
struct AVLTree {
    int data, height;
    struct AVLTree *left, *right;
};

int height (struct AVLTree *t) {
    if (t == NULL)
        return -1;
    else
        return t->height;
}

int max (int a, int b) {
    if (a > b)
        return a;
    else
        return b;
}

struct AVLTree *singlerotatewithleft(struct AVLTree *K2) {
    struct AVLTree *K1;
    K1 = K2->left;
    K2->left = K1->right;
    K1->right = K2;
    K2->height = max(height(K2->left), height(K2->right)) + 1;
    K1->height = max(height(K1->left), height(K1->right)) + 1;
    return K1;
}

struct AVLTree *singlerotatewithright(struct AVLTree *K1) {
    struct AVLTree *K2;
    K2 = K1->right;
    K1->right = K2->left;
    K2->left = K1;
    K2->height = max(height(K2->left), height(K2->right)) + 1;
    K1->height = max(height(K1->left), height(K1->right)) + 1;
    return K2;
}

struct AVLTree *doublerotatewithleft(struct AVLTree *K3) {
    K3->left = singlerotatewithright(K3->left);
    return singlerotatewithleft(K3);
}

struct AVLTree *doublerotatewithright(struct AVLTree *K1) {
    K1->right = singlerotatewithleft(K1->right);
    return singlerotatewithright(K1);
}

struct AVLTree *insert(struct AVLTree *t, int x) {
    if (t == NULL) {
        t = (struct AVLTree *)malloc(sizeof(struct AVLTree));
        t->data = x;
        t->height = 0;
        t->left = t->right = NULL;
    }
    else if (x < t->data) {
        t->left = insert(t->left, x);
        if (height(t->left) - height(t->right) == 2) {
            if (x < t->left->data)
                t = singlerotatewithleft(t);
            else
                t = doublerotatewithleft(t);
        }
    }
    else if (x > t->data) {
```

```
        t->right = insert(t->right, x);
        if (height(t->right) - height(t->left) == 2) {
            if (x > t->right->data)
                t = singlerotatewithright(t);
            else
                t = doublerotatewithright(t);
        }
    }
    t->height = max(height(t->left), height(t->right)) + 1;
    return t;
}

void inorder(struct AVLTree *t) {
    if (t->left != NULL)
        inorder(t->left);
    printf("%d ", t->data);
    if (t->right != NULL)
        inorder(t->right);
}

void preorder(struct AVLTree *t) {
    printf("%d ", t->data);
    if (t->left != NULL)
        preorder(t->left);
    if (t->right != NULL)
        preorder(t->right);
}

void hierarchy(struct AVLTree *t, int level) {
    if (t != NULL) {
        for (int i = 0; i < level; i++)
            printf("    ");

        printf("%d\n", t->data);

        if (t->left != NULL)
            hierarchy(t->left, level + 1);
        if (t->right != NULL)
            hierarchy(t->right, level + 1);
    }
}
```

```
int main() {
    struct AVLTree *t = NULL;
    t = insert(t, 1);
    t = insert(t, 2);
    t = insert(t, 3);
    t = insert(t, 4);
    t = insert(t, 5);

    printf("Inorder traversal: \n");
    inorder(t);
    printf("\n");

    printf("Preorder traversal: \n");
    preorder(t);
    printf("\n");

    printf("Hierarchical traversal: \n");
    hierarchy(t, 0);

    printf("Hierarchical traversal: \n");
    hierarchy(t, 1);

    printf("Hierarchical traversal: \n");
    hierarchy(t, 2);
    return 0;
}
```



Technical Outcomes

Design	2	
Understanding of DS	2	Needs improvement
Use of DS	3	
Debugging	2	

Best Practices

Design before coding	2	Needs improvement
Usage of Algo	2	
Multifile	3	
Versioning	0	didnt

UCS 2312 Data Structures Lab
Assignment 8: Binary Heap and its applications

Implement Priority Queue using Binary Heap. priorityQueueADT consists of integer element. Implement the following methods.

[CO2, K3]

- void insert(struct priorityQueueADT *P, int x) – Insertion new item into priority queue using Max Heap property
- int delete(struct priorityQueueADT *P) – Will remove the root of binary heap
- void display(struct priorityQueueADT *P) – Will display the contents pf Priority Queue

1. Demonstrate ADT with the following testcase insert(p,14); insert(p,16); insert(p,22); insert(p,11); insert(p,9); insert(p,18); insert(p,10); insert(p,7); insert(p,4);
   insert(p,1);

2. Write an application to design a priority queue using max binary heap. An item in the priority queue consists of employee id and salary amount. The queue supports two operations, namely, insertion and deletion.

Test the application with the following

```
insert(p,('A',15000));   insert(p,('K',12000));
insert(p,('R',4000));    insert(p,('T',3500));
insert(p,('L',4600));    insert(p,('P',6000));
insert(p,('Y',8600));
```

Output:
Employees are removed in the following order

('A',15000), ('K',12000), ('Y',8600), ('P',6000), ('L',4600), ('R',4000), ('T',3500),

Gr 8

1/10/24

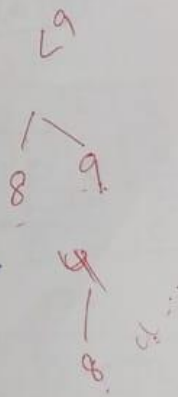
Implementation of Binary Heap * Applen

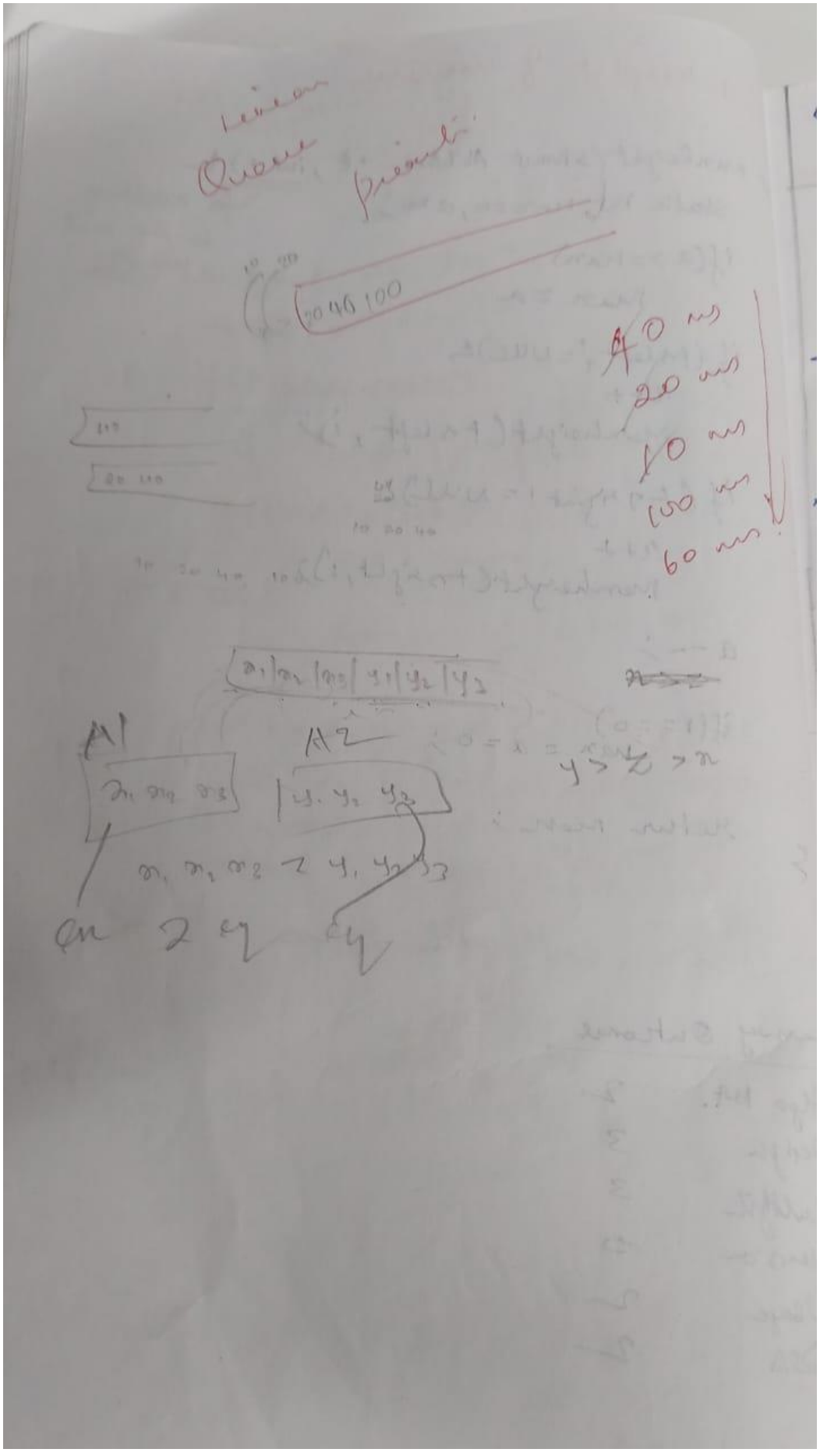
Minheap_ADT.h

```
struct PriQueue {
    int arr[100];
    int size;
};
```

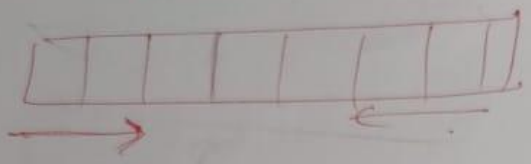
min Heap array is Visualized

```
void init(struct PriQueue *q) {
    q->size = 0;
    q->arr[0] = -10;   // very small no.
}
void insert(struct PriQueue *q, int x) {
    int i;
    for(i = ++q->size; q->arr[i/2] > x; i = i/2) {
        q->arr[i] = q->arr[i/2];
    }
    q->arr[i] = x;
}
int deletemin(struct PriQueue *q) {
    int i, child, min, last;
    min = q->arr[1];
    last = q->arr[q->size]; q->size--;
    for(i = 1; i*2 <= q->size; i = child) {
        child = i*2
        if(q->arr[child+1] < q->arr[child]) child++;
        if(last > q->arr[child]) q->arr[i] = q->arr[child];
        else break;
    }
    q->arr[i] = last; return min;
}
```

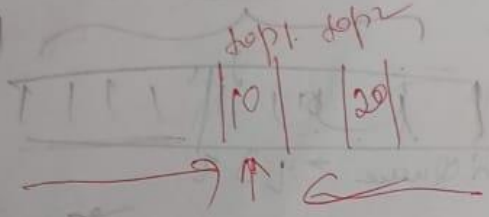



Queue
40 ms
20 ms
10 ms
100 ms
60 ms
A1
A2

Construct Queue using stacks

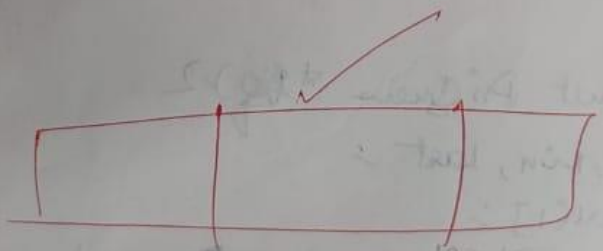


design



Push

pop



```c
#include <stdio.h>

struct PriorityQueue {
    int arr[100];
    int size;
};

void configure(struct PriorityQueue * PQ){
    PQ->size = 0;
    PQ->arr[0] = -10;
}

void insert(struct PriorityQueue * PQ, int x){
    int i;
    for(i = ++PQ->size; PQ->arr[i/2] > x; i /= 2){
        PQ->arr[i] = PQ->arr[i/2];
    }
    PQ->arr[i] = x;
}

int deletemin(struct PriorityQueue * PQ){
    int i, child, minelement, lastelement;
    minelement = PQ->arr[1];
    lastelement = PQ->arr[PQ->size--];

    for(i = 1; i * 2 <= PQ->size; i = child){
        child = i * 2;
        if(child != PQ->size && PQ->arr[child + 1] < PQ->arr[child]) {
            child++;
        }
        if(lastelement > PQ->arr[child]){
            PQ->arr[i] = PQ->arr[child];
        } else {
            break;
        }
        }
    }
    PQ->arr[i] = lastelement;
    return minelement;
}

void display(struct PriorityQueue * PQ){
    for(int i = 1; i <= PQ->size; i++)
        printf("%d\n ", PQ->arr[i]);
}
```

```c
int main() {
    struct PriorityQueue pq;

    configure(&pq);

    insert(&pq, 15);
    insert(&pq, 10);
    insert(&pq, 30);
    insert(&pq, 40);
    insert(&pq, 5);

    display(&pq);

    printf("Minimum element: %d\n", deletemin(&pq));
    display(&pq);
    printf("Minimum element: %d\n", deletemin(&pq));
    display(&pq);
    printf("Minimum element: %d\n", deletemin(&pq));
    display(&pq);

    return 0;
}
```

```
(base) shivaaneesk@Shivaanees-MacBook-Ai
5
 10
 30
 40
 15
 Minimum element: 5
10
 15
 30
 40
 Minimum element: 10
15
 40
 30
 Minimum element: 15
30
 40
```

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

struct Employee {
    char id;
    int salary;
};

struct PriorityQueue {
    struct Employee arr[100];
    int size;
};

void configure(struct PriorityQueue *PQ) {
    PQ->size = 0;
}

void insert(struct PriorityQueue *PQ, char id, int salary) {
    struct Employee newEmployee;
    newEmployee.id = id;
    newEmployee.salary = salary;

    int i = ++PQ->size;
    while (i > 1 && PQ->arr[i / 2].salary < salary) {
        PQ->arr[i] = PQ->arr[i / 2];
        i /= 2;
    }
    PQ->arr[i] = newEmployee;
}

struct Employee deletemax(struct PriorityQueue *PQ) {
    int i, child;
    struct Employee maxElement = PQ->arr[1];
    struct Employee lastElement = PQ->arr[PQ->size--];
```

```
    for (i = 1; i * 2 <= PQ->size; i = child) {
        child = i * 2;
        if (child != PQ->size && PQ->arr[child + 1].salary > PQ->arr[child].salary) {
            child++;
        }
        if (lastElement.salary < PQ->arr[child].salary) {
            PQ->arr[i] = PQ->arr[child];
        } else {
            break;
        }
    }
    PQ->arr[i] = lastElement;
    return maxElement;
}

void display(struct PriorityQueue *PQ) {
    for (int i = 1; i <= PQ->size; i++) {
        printf("('%c', %d)\n", PQ->arr[i].id, PQ->arr[i].salary);
    }
}

int main() {
    struct PriorityQueue pq;
    configure(&pq);

    insert(&pq, 'A', 15000);
    insert(&pq, 'K', 12000);
    insert(&pq, 'R', 4000);
    insert(&pq, 'T', 3500);
    insert(&pq, 'L', 4600);
    insert(&pq, 'P', 6000);
    insert(&pq, 'Y', 8600);

    printf("Priority Queue after insertions:\n");
    display(&pq);

    printf("\nEmployees removed in the following order:\n");
    while (pq.size > 0) {
        struct Employee maxEmp = deletemax(&pq);
        printf("('%c', %d)\n", maxEmp.id, maxEmp.salary);
    }

    return 0;
}
```

```
Priority Queue after insertions:
('A', 15000)
('K', 12000)
('Y', 8600)
('T', 3500)
('L', 4600)
('R', 4000)
('P', 6000)

Employees removed in the following order:
('A', 15000)
('K', 12000)
('Y', 8600)
('P', 6000)
('L', 4600)
('R', 4000)
('T', 3500)
```

Technical Outcomes:

Design	3	
Understanding DS	3	
Usage of DS	2	Needs improvement
Debugging	3	

Best Practices:

Design before coding	3	
Usage of algo	3	
Multifile	2	Needs improvement
versioning	3	

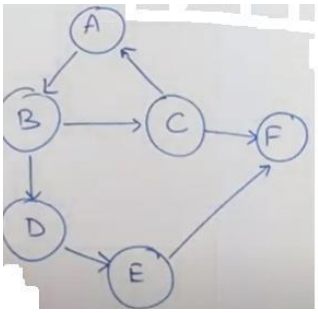
UCS 2312 Data Structures Lab

Assignment 9: Graph Traversal and its Applications

The cityADT consists of adjacency matrix that represents the connection between the cities. Adjacency matrix has an entry 1, if there is a connection between the cities.
Implement the following methods. [CO2, K3]

- void create(cityADT *C) – will create the graph using adjacency matrix
- void disp(cityADT *C) – display the adjacency matrix
- void BFS(cityADT *C) – provides the output of visiting the cities by following breadth first
- void DFS(cityADT *C) – provides the output of visiting the cities by following depth first

1. Demonstrate the ADT with the following Graph



Enter the no. of vertices: 6
Enter the no. of edges: 7
AB, BC, BD, CA, CF, DE, EF

Adjacency Matrix

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	1	1	0	0
C	1	0	0	0	0	1
D	0	0	0	0	1	0
E	0	0	0	0	0	1
F	0	0	0	0	0	0

BFS Output: ABCDFE for Start vertex A

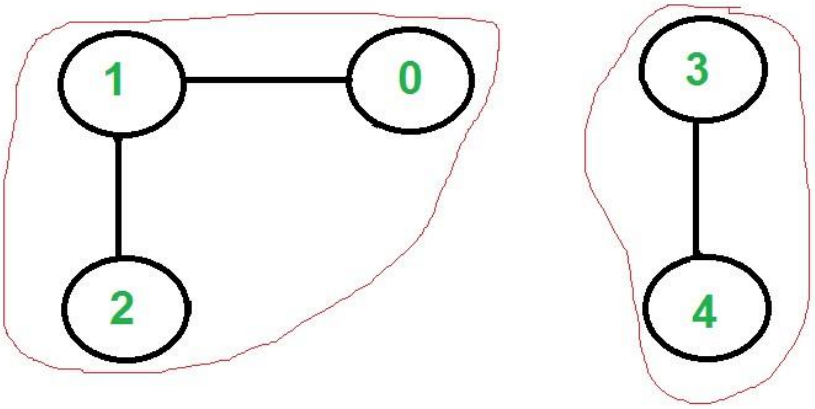
DFS Output: ABCFDE for Start vertex A

2. Write an application to utilize traversals to do the following:

a. Given the source and destination cities, find whether there is a path from source to destination
b. Find the connected components in a given graph

Test the application with the following Input:
Source: D Destination: F
Output:
Path exists Input:
Source: F Destination: B
Output:
Path not exists

Input:

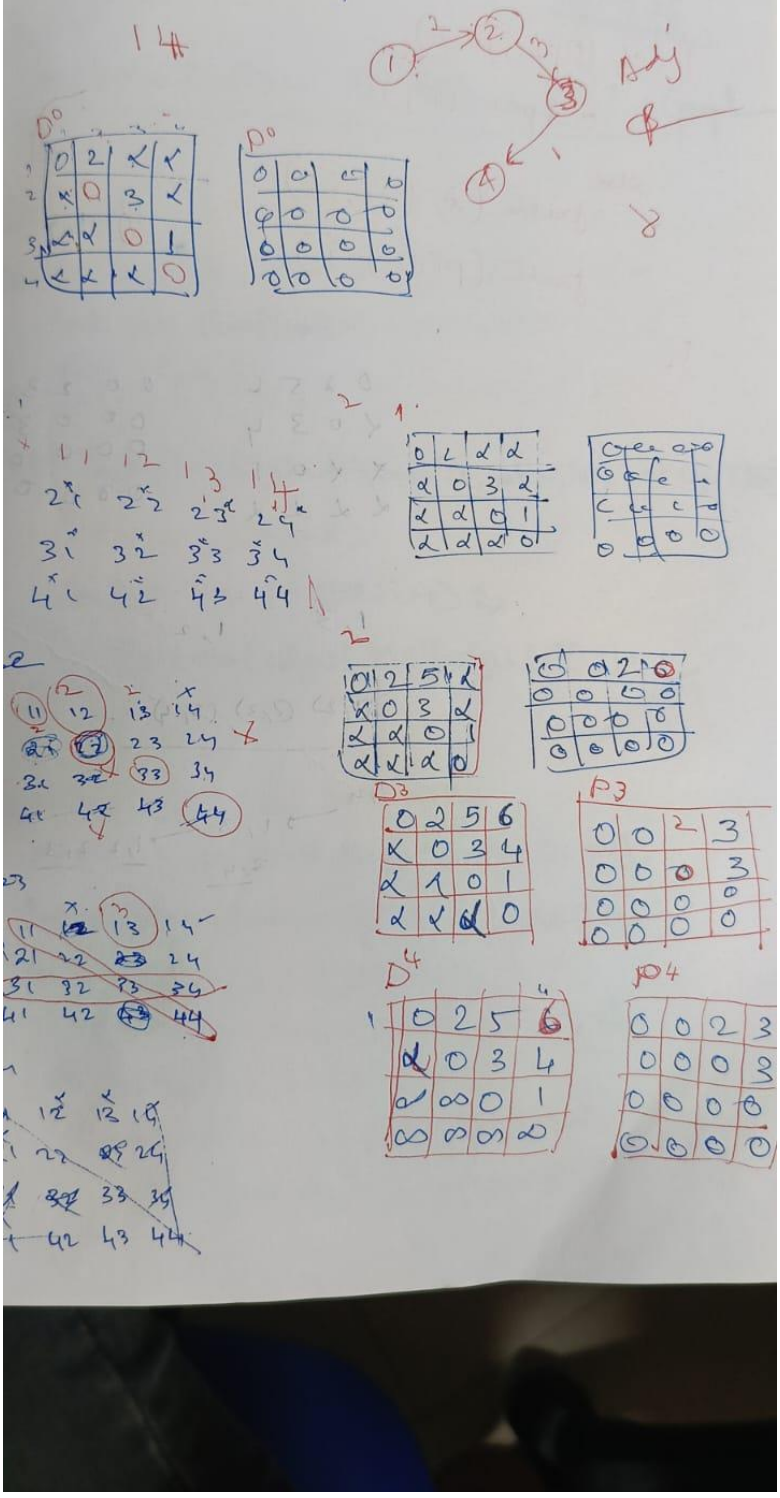


There are two connected components in above undirected graph
0 1 2
3 4

Ex 9

8/10/24

Implementation of BFS & DFS



```
path(i,j)
{ if (P[i,j]==0)
    output (i,j)
  else
    path (i, P[i,j])
    path (P[i,j], j)
}.
```

path (1,

i,j) 1,4

0	2	5	6
x	0	3	4
x	x	0	1
x	x	x	x

0	0	2	3
0	0	0	3
0	0	0	1
0	0	0	0

1,3 1,2

(1,2) (2,3) (3,4)

1,4 → 1,3 → 1,2 2,3

3,4

Path

Ex 9 ~~A~~ BFS 8/10/24

Implementation of BFS & DFS in Graph

ADT.h

```
struct Graph {
    int arr[100][100];
    int v, e;
};
void CreateGraph(struct Graph *G, int v, int e, int edges
    G->v = v; G->e = e;
    for(int i=0; i<G->e; i++) {
        G->arr[edges[i][0]][edges[i][1]] = 1;
    }
}

BFS(struct Graph * G, int x) {
    struct queue *Q; struct Array * visited[G->v];
    visited[x] = visited->arr[x] = 2; enqueue(Q, x);
    while(Q->f != -1 && Q->r != -1) {
        int z = dequeue(Q); printf("%d ", z);
        for(int i=0; i<G->v; i++) {
            if(G->arr[z][i] == 1 && find(visited, i) == -1) {
                insert(visited, i); enqueue(Q, i);
            }
        }
    }
}
```

```
DFS(struct Graph *G, int s){
    struct stack *S;
    struct Array *visited[G->V], *output[G->V];
    visited->arr[0] = s;
    push(S, s);
    while(S->top != -1){
        int t = peek(S);
        for(int ele=0; ele < G->V; ele++){
            if(find(visited, ele) == -1 && G->arr[t][ele] == 1)
                insert(visited, ele); push(S, ele);
            else
                insert(output, pop(S));
        }
    }
    printReverse(output);
}

Recursive(struct Graph *G, int s){
    struct Array *visited[G->V];
    visited->arr[0] = s;
    printf("%d", s);
    for(int i=0; i < G->V; i++){
        if(G->arr[s][i] == 1 && find(visited, i) == -1
            DFSRecursive(i);
    }
}
```

```c
#include <stdio.h>
#include "Q.h"
#include "Stack1.h"

#define MAX_NODES 100

void createGraph(int adjMatrix[MAX_NODES][MAX_NODES], int edges[][2], int numEdges, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            adjMatrix[i][j] = 0;
        }
    }
    for (int i = 0; i < numEdges; i++) {
        int from = edges[i][0] - 1;
        int to = edges[i][1] - 1;
        adjMatrix[from][to] = 1;
    }
}

void printMatrix(int adjMatrix[MAX_NODES][MAX_NODES], int n) {
    printf("[");
    for (int i = 0; i < n; i++) {
        printf("[");
        for (int j = 0; j < n; j++) {
            printf("%d", adjMatrix[i][j]);
            if (j < n - 1) {
                printf(",");
            }
        }
        printf("]");
        if (i < n - 1) {
            printf(",\n ");
        }
    }
    printf("]\n");
```

```c
int* bfs(int adjMatrix[MAX_NODES][MAX_NODES], int startNode, int n, int* result) {
    struct Queue Q;
    createQ(&Q, n);

    int visited[MAX_NODES] = {0};
    int idx = 0;

    visited[startNode] = 1;
    Enqueue(&Q, startNode);

    while (!isEmpty(&Q)) {
        int z = Dequeue(&Q);
        result[idx++] = z + 1;

        for (int y = 0; y < n; y++) {
            if (adjMatrix[z][y] == 1 && !visited[y]) {
                visited[y] = 1;
                Enqueue(&Q, y);
            }
        }
    }
    return result;
}

int* dfs(int adjMatrix[MAX_NODES][MAX_NODES], int startNode, int n, int* result) {
    struct Stack S;
    createStack(&S, n);

    int visited[MAX_NODES] = {0};
    int idx = 0;

    visited[startNode] = 1;
    result[idx++] = startNode + 1;
    Push(&S, startNode);

    while (!isStackEmpty(&S)) {
        int t = top(&S);
        int unvisitedNeighbor = -1;

        for (int y = 0; y < n; y++) {
            if (adjMatrix[t][y] == 1 && !visited[y]) {
                unvisitedNeighbor = y;
                break;
            }
        }

        if (unvisitedNeighbor != -1) {
            visited[unvisitedNeighbor] = 1;
            result[idx++] = unvisitedNeighbor + 1;
            Push(&S, unvisitedNeighbor);
        } else {
            pop(&S);
        }
    }
    return result;
}

int* bfs_disconnected(int adjMatrix[MAX_NODES][MAX_NODES], int n, int* result) {
    struct Queue Q;
    createQ(&Q, n);

    int visited[MAX_NODES] = {0};
    int idx = 0;

    for (int startNode = 0; startNode < n; startNode++) {
        if (!visited[startNode]) {
            visited[startNode] = 1;
```

```
            Enqueue(&Q, startNode);

            while (!isEmpty(&Q)) {
                int z = Dequeue(&Q);
                result[idx++] = z + 1;

                for (int y = 0; y < n; y++) {
                    if (adjMatrix[z][y] == 1 && !visited[y]) {
                        visited[y] = 1;
                        Enqueue(&Q, y);
                    }
                }
            }
        }
    }

    return result;
}

int* dfs_disconnected(int adjMatrix[MAX_NODES][MAX_NODES], int n, int* result) {
    struct Stack S;
    createStack(&S, n);

    int visited[MAX_NODES] = {0};
    int idx = 0;

    for (int startNode = 0; startNode < n; startNode++) {
        if (!visited[startNode]) {
            visited[startNode] = 1;
            result[idx++] = startNode + 1;
            Push(&S, startNode);
            while (!isStackEmpty(&S)) {
                int t = top(&S);
                int unvisitedNeighbor = -1;

                for (int y = 0; y < n; y++) {
                    if (adjMatrix[t][y] == 1 && !visited[y]) {
                        unvisitedNeighbor = y;
                        break;
                    }
                }

                if (unvisitedNeighbor != -1) {
                    visited[unvisitedNeighbor] = 1;
                    result[idx++] = unvisitedNeighbor + 1;
                    Push(&S, unvisitedNeighbor);
                } else {
                    pop(&S);
                }
            }
        }
    }

    return result;
}
```

```c
#include "adj_matrix.h"

int main() {
    int adjMatrix[MAX_NODES][MAX_NODES];
    int edges[][2] = {{1,2}, {2,3}, {2,1}, {3,2}, {4,5}, {5,4}};
    int numEdges = sizeof(edges) / sizeof(edges[0]);
    int n = 5;

    createGraph(adjMatrix, edges, numEdges, n);
    printMatrix(adjMatrix, n);

    int result[MAX_NODES];
    int startNode = 0;

    int* bfsResult = bfs_disconnected(adjMatrix, n, result);
    printf("BFS Traversal: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", bfsResult[i]);
    }
    printf("\n");

    int* dfsResult = dfs_disconnected(adjMatrix, n, result);
    printf("DFS Traversal: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", dfsResult[i]);
    }
    printf("\n");

    return 0;
}
```

```
[[0,1,0,0,0],
 [1,0,1,0,0],
 [0,1,0,0,0],
 [0,0,0,0,1],
 [0,0,0,1,0]]
BFS Traversal: 1 2 3 4 5
DFS Traversal: 1 2 3 4 5
```

Technical Outcomes:

Design	3	
Understanding DS	3	
Usage of DS	2	Needs improvement
Debugging	3	

Best Practices:

Design before coding	3	
Usage of algo	3	
Multifile	3	
versioning	3	

UCS 2312 Data Structures Lab

Assignment 10: Implementation of Shortest Path Finding algorithm

The cityADT contains the number of cities and the connectivity information between the cities (adjacency matrix). Write the following methods. [CO2, K3]

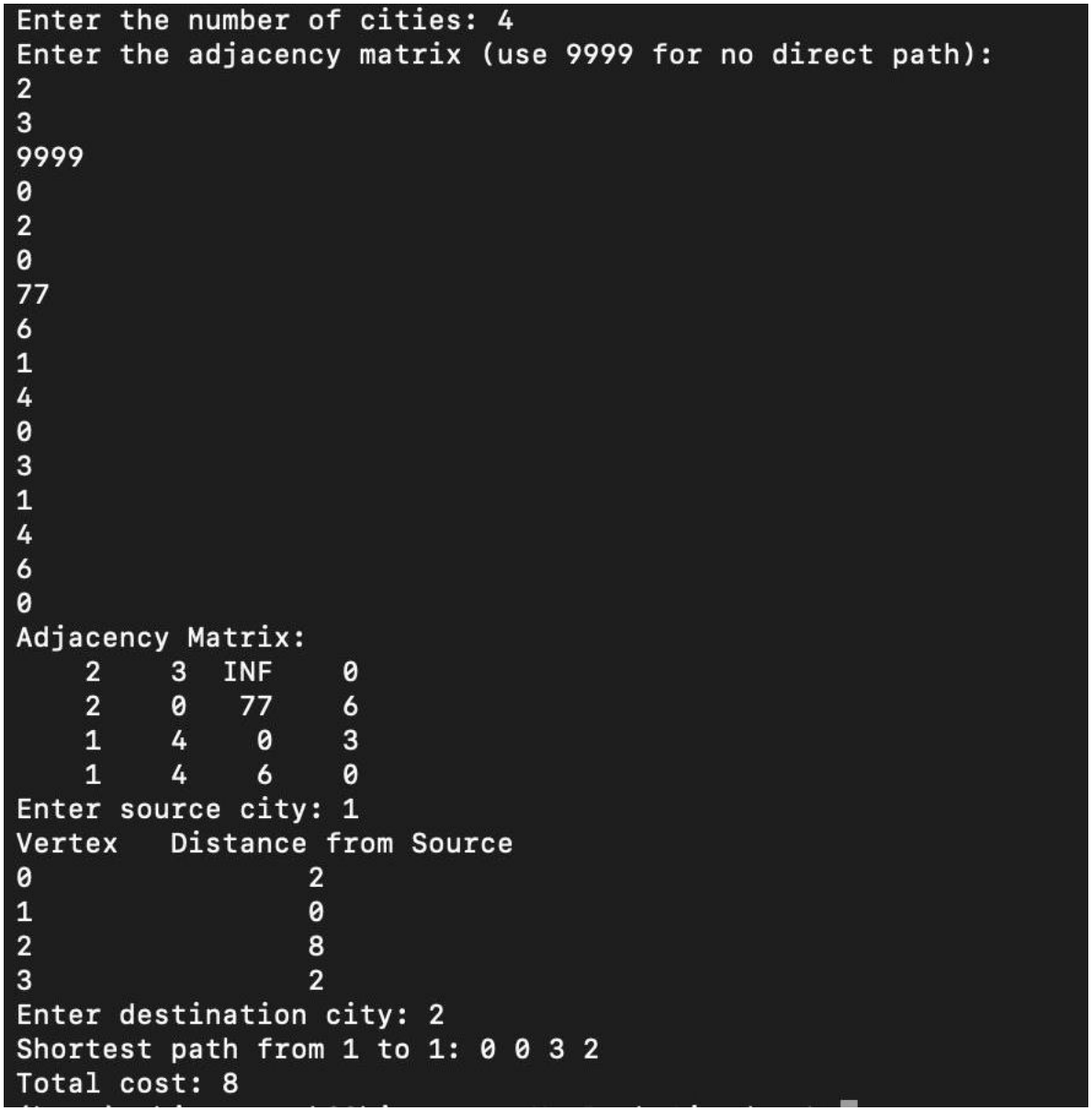
- void create(cityADT *C) – will represent the graph using adjacency matrix
- void disp(cityADT *C) – Display the graph
- void Dijkstra(cityADT *C)
  - Displays the intermediate and final tables
- char * displayPath(cityADT *C, source, destination)
  - Find the path of the intermediate cities between the source and destination cities along with the cost

(1) Implementation of searching & sorting

```
void insertelement(struct numberADT *N, int x[10])
    for(i=0; i<10; i++)
        N->arr[i] = x[i]
    N->size = 10
void insert(struct numberADT *N) {
    for(i=0; i<N->size; i++)
        min = i
        for(j=i+1; j<N->size; j++)
            if(N->arr[j] < N->arr[min])
                min = j
        temp = N->arr[min]
        N->arr[min] = N->arr[i]
        N->arr[i] = temp ;
}
void shellsort(struct NumberADT *N) {
    for(gap = N->size/2; gap>0; gap /= 2)
        for(i=gap; i<N->size; i++)
            keep = N->arr[i]
            for(j=i; j>=gap && N->arr[
                -> temp
                j -=
            N->arr[j] = N->arr[j-
```

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

#define MAX_CITIES 100
#define INF 9999 // Representing no direct path with a large number

// Structure for cityADT
typedef struct {
    int numCities;
    int adjMatrix[MAX_CITIES][MAX_CITIES];
} cityADT;

// Function to create the adjacency matrix (graph representation)
void create(cityADT *C) {
    printf("Enter the number of cities: ");
    scanf("%d", &C->numCities);

    printf("Enter the adjacency matrix (use %d for no direct path):\n", INF);
    for (int i = 0; i < C->numCities; i++) {
        for (int j = 0; j < C->numCities; j++) {
            scanf("%d", &C->adjMatrix[i][j]);
        }
    }
}

// Function to display the adjacency matrix
void disp(cityADT *C) {
    printf("Adjacency Matrix:\n");
    for (int i = 0; i < C->numCities; i++) {
        for (int j = 0; j < C->numCities; j++) {
            if (C->adjMatrix[i][j] == INF)
                printf("%5s", "INF");
            else
                printf("%5d", C->adjMatrix[i][j]);
        }
        printf("\n");
    }
}

// Utility function to find the minimum distance vertex
int minDistance(int dist[], bool sptSet[], int numCities) {
    int min = INF, minIndex;

    for (int v = 0; v < numCities; v++) {
        if (!sptSet[v] && dist[v] <= min) {
            min = dist[v], minIndex = v;
        }
    }

    return minIndex;
}

// Dijkstra's algorithm to find the shortest path
void Dijkstra(cityADT *C, int src) {
    int dist[MAX_CITIES]; // Output array. dist[i] will hold the shortest distance from src to i
    bool sptSet[MAX_CITIES]; // sptSet[i] will be true if vertex i is included in shortest path tree
    int parent[MAX_CITIES]; // To store the path information

    // Initialize distances and parent nodes
    for (int i = 0; i < C->numCities; i++) {
        dist[i] = INF;
        sptSet[i] = false;
        parent[i] = -1; // No parent initially
    }
```

```
    dist[src] = 0; // Distance of source city to itself is 0

    for (int count = 0; count < C->numCities - 1; count++) {
        int u = minDistance(dist, sptSet, C->numCities);

        sptSet[u] = true;

        for (int v = 0; v < C->numCities; v++) {
            if (!sptSet[v] && C->adjMatrix[u][v] != INF && dist[u] != INF
                && dist[u] + C->adjMatrix[u][v] < dist[v]) {
                dist[v] = dist[u] + C->adjMatrix[u][v];
                parent[v] = u; // Track the parent for path reconstruction
            }
        }
    }

    printf("Vertex \t Distance from Source\n");
    for (int i = 0; i < C->numCities; i++) {
        printf("%d \t\t %d\n", i, dist[i]);
    }
}

// Function to print the path from source to destination using parent array
void printPath(int parent[], int j) {
    if (parent[j] == -1)
        return;

    printPath(parent, parent[j]);
    printf("%d ", j);
}
// Function to display the path (intermediate cities) between source and destination
void displayPath(cityADT *C, int src, int dest) {
    int dist[MAX_CITIES], parent[MAX_CITIES];
    bool sptSet[MAX_CITIES];

    for (int i = 0; i < C->numCities; i++) {
        dist[i] = INF;
        sptSet[i] = false;
        parent[i] = -1;
    }

    dist[src] = 0;

    for (int count = 0; count < C->numCities - 1; count++) {
        int u = minDistance(dist, sptSet, C->numCities);

        sptSet[u] = true;

        for (int v = 0; v < C->numCities; v++) {
            if (!sptSet[v] && C->adjMatrix[u][v] != INF && dist[u] != INF
                && dist[u] + C->adjMatrix[u][v] < dist[v]) {
                dist[v] = dist[u] + C->adjMatrix[u][v];
                parent[v] = u;
            }
        }
    }

    printf("Shortest path from %d to %d: %d ", src, src);
    printPath(parent, dest);
    printf("\nTotal cost: %d\n", dist[dest]);
}
int main() {
    cityADT C;
    int source, destination;

    create(&C);
    disp(&C);

    printf("Enter source city: ");
    scanf("%d", &source);

    Dijkstra(&C, source);

    printf("Enter destination city: ");
    scanf("%d", &destination);

    displayPath(&C, source, destination);

    return 0;
}
```



Technical Outcomes:

Design	2	Needs improvement
Understanding DS	2	Needs improvement
Usage of DS	2	
Debugging	3	

Best Practices:

Design before coding	2	
Usage of algo	2	
Multifile	3	
versioning	3	

UCS 2312 Data Structures Lab

Exercise 11: Implementation of Hash Table using Closed and Open addressing methods

[CO2, K3]

The HashTableADT contains hash table and its size. Hash function to be used for the insertion of elements is ***x mod tableSize***. Use Separate chaining method to resolve the collision.

- void init(HashTableADT *H) – To initialize the size of Hash Table
- void insertElementL (HashTableADT *H, int x)– To insert the input key into the hash table
- int searchElement(HashTableADT *H, int key) – Searching an element in the hash table, if found return 1, otherwise return -1
- void displayHT(HashTableADT *H) – Display the elements in the hash table

Note:

1. Implement HashTableADT with the specified operations in HashTableADTImpl.h
2. Write a menu driven application to utilize the HashTableADT.

1. Demonstrate ADT with the following test case
insert 23, 45, 69, 87, 48, 67, 54, 66, 53

Contents of Hash Table
(Separate Chaining)
0 :
1 :
2 : 3 : 23 ❼ 53
4 : 54
5 : 45
6 : 66
7 : 87 ❼ 67
8 : 48
9 : 69

2. Create another hash table ADT with following functions for open addressing methods, namely, Quadratic probing and Double Hashing.

- void insertElementL (HashTableADT *H, int x)– To insert the input key into the hash table
- void displayHT(HashTableADT *H) – Display the elements in the hash table

Note: For Double hashing, the second hash function is 7-(x%7)

Demonstrate the ADT with the following testcase
(i) Contents of Hash Table

(Quadratic Probing)
0 →
1 →67
2 →53
3 →23
4 →54
5 →45
6 →66
7 →87
8 →48
9 →69

(ii) Contents of Hash Table
(Double Hashing)
0 →67
1 →
2 →53
3 →23
4 →54
5 →45
6 →66
7 →87
8 →48
9 →69

Ex 1 Hash Table Implementation

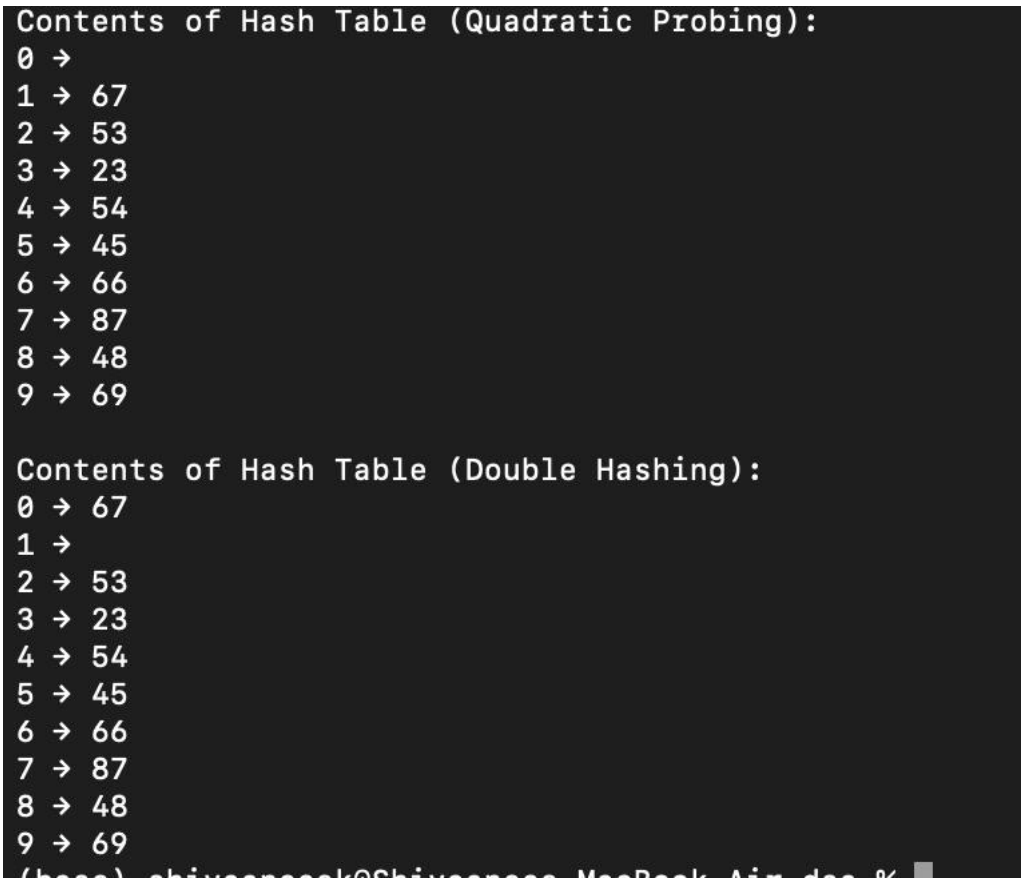
```
void InsertElement(struct HashTable ADT *h, int n) {
    Index = n % HTSize
    struct Node *newNode = (Node *)malloc(...)
    newNode->data = n
    newNode->next = NULL
    if (H->table[index] == NULL)
        H->table[index] = newNode
    else
        newNode->next = H->table[index]
        H->table[index] = newNode
}

void display(struct HashTable ADT *H) {
    for (int i=0; i<H->size; i++)
        struct Node *temp = H->table[i]
        while (temp != NULL)
            printf("%d -> ", temp->data)
            temp = temp->next
        printf(NULL)
}

int searchele(struct HashTable ADT *H, int key) {
    int index = key % H->size;
    struct Node *temp = H->table[index]
    while (temp != NULL)
        if (temp->data == key)
            return 1
        temp = temp->next
    return -1
}
```

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

#define TABLE_SIZE 10

// Structure for node in the linked list (used for separate chaining)
typedef struct node {
    int data;
    struct node* next;
} Node;

// Structure for Hash Table using separate chaining
typedef struct {
    Node* table[TABLE_SIZE];
} HashTableADT;

// Function to initialize the hash table
void init(HashTableADT* H) {
    for (int i = 0; i < TABLE_SIZE; i++) {
        H->table[i] = NULL;
    }
}

// Hash function (x mod tableSize)
int hashFunction(int x) {
    return x % TABLE_SIZE;
}

// Function to insert an element using separate chaining
void insertElementL(HashTableADT* H, int x) {
    int index = hashFunction(x);
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = x;
    newNode->next = NULL;
    if (H->table[index] == NULL) {
        H->table[index] = newNode;
    } else {
        // Collision occurred, add to the linked list at that index
        Node* current = H->table[index];
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = newNode;
    }
}

// Function to search for an element in the hash table
int searchElement(HashTableADT* H, int key) {
    int index = hashFunction(key);
    Node* current = H->table[index];
    while (current != NULL) {
        if (current->data == key) {
            return 1;  // Element found
        }
        current = current->next;
    }
    return -1;  // Element not found
}

// Function to display the hash table (separate chaining)
void displayHT(HashTableADT* H) {
    for (int i = 0; i < TABLE_SIZE; i++) {
        printf("%d : ", i);
        Node* current = H->table[i];
        while (current != NULL) {
            printf("%d -> ", current->data);
            current = current->next;
        }
```

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

// Main function to test separate chaining
int main() {
    HashTableADT H;
    init(&H);

    int elements[] = {23, 45, 69, 87, 48, 67, 54, 66, 53};
    int numElements = sizeof(elements) / sizeof(elements[0]);

    for (int i = 0; i < numElements; i++) {
        insertElementL(&H, elements[i]);
    }

    printf("Contents of Hash Table (Separate Chaining):\n");
    displayHT(&H);

    int key;
    printf("\nEnter element to search: ");
    scanf("%d", &key);
    int result = searchElement(&H, key);
    if (result == 1) {
        printf("Element %d found in the hash table.\n", key);
    } else {
        printf("Element %d not found in the hash table.\n", key);
    }

    return 0;
}
```

```
Contents of Hash Table (Separate Chaining):
0 :
1 :
2 :
3 : 23 -> 53 ->
4 : 54 ->
5 : 45 ->
6 : 66 ->
7 : 87 -> 67 ->
8 : 48 ->
9 : 69 ->

Enter element to search: 48
Element 48 found in the hash table.
(base) shivaaneesk@Shivaanees-MacBook-Air dsa % gcc SeparateChaining.c
(base) shivaaneesk@Shivaanees-MacBook-Air dsa % ./a.out
Contents of Hash Table (Separate Chaining):
0 :
1 :
2 :
3 : 23 -> 53 ->
4 : 54 ->
5 : 45 ->
6 : 66 ->
7 : 87 -> 67 ->
8 : 48 ->
9 : 69 ->

Enter element to search: 1
Element 1 not found in the hash table.
```

```c
#include <stdio.h>

#define TABLE_SIZE 10
#define EMPTY -1

// Structure for Hash Table using open addressing
typedef struct {
    int table[TABLE_SIZE];
} HashTableADT;

// Function to initialize the hash table
void init(HashTableADT* H) {
    for (int i = 0; i < TABLE_SIZE; i++) {
        H->table[i] = EMPTY;
    }
}

// Hash function (x mod tableSize)
int hashFunction(int x) {
    return x % TABLE_SIZE;
}

// Second hash function for double hashing
int doubleHashFunction(int x) {
    return 7 - (x % 7);
}

// Function to insert an element using quadratic probing
void insertQuadraticProbing(HashTableADT* H, int x) {
    int index = hashFunction(x);
    int i = 0;

    while (H->table[(index + i * i) % TABLE_SIZE] != EMPTY) {
        i++;
    }
    H->table[(index + i * i) % TABLE_SIZE] = x;
}

// Function to insert an element using double hashing
void insertDoubleHashing(HashTableADT* H, int x) {
    int index = hashFunction(x);
    int step = doubleHashFunction(x);
    int i = 0;

    while (H->table[(index + i * step) % TABLE_SIZE] != EMPTY) {
        i++;
    }

    H->table[(index + i * step) % TABLE_SIZE] = x;
}

// Function to display the hash table
void displayHT(HashTableADT* H) {
    for (int i = 0; i < TABLE_SIZE; i++) {
        printf("%d → ", i);
        if (H->table[i] != EMPTY) {
            printf("%d", H->table[i]);
        }
        printf("\n");
    }
}

// Main function to test quadratic probing and double hashing
int main() {
    HashTableADT H;
    init(&H);

    int elements[] = {23, 45, 69, 87, 48, 67, 54, 66, 53};
    int numElements = sizeof(elements) / sizeof(elements[0]);
    printf("Contents of Hash Table (Quadratic Probing):\n");
    for (int i = 0; i < numElements; i++) {
        insertQuadraticProbing(&H, elements[i]);
    }
    displayHT(&H);

    init(&H);  // Reinitialize the hash table

    printf("\nContents of Hash Table (Double Hashing):\n");
    for (int i = 0; i < numElements; i++) {
        insertDoubleHashing(&H, elements[i]);
    }
    displayHT(&H);

    return 0;
}
```



Technical Outcomes:

Design	3	
Understanding DS	3	
Usage of DS	2	Needs improvement
Debugging	3	

Best Practices:

Design before coding	3	
Usage of algo	3	
Multifile	1	Needs improvement
versioning	3	

UCS 2312 Data Structures Lab

Assignment 12: Implementation of Searching and Sorting algorithms

The numberADT contains the size of integer array and an array of integers. [CO2, K3]

- void init(numbertADT *N) – To initialize the size of the array
- void insertElements (numberADT *N, int x[10])– To copy the elements from x to the array in numberADT
- void insSort(numberADT *N) – Sorting of elements in the array using selection sort
- void shellSort(numberADT *N) – Sorting of elements in the array using shell sort
- void display(numberADT *N) – Display the elements in the array

Additional Routines
Write a routine in ADT to find the element that appears once in sorted array with O(log n) time complexity.
Write a routine in ADT to count the number of 1's in sorted array with O(log n) time complexity

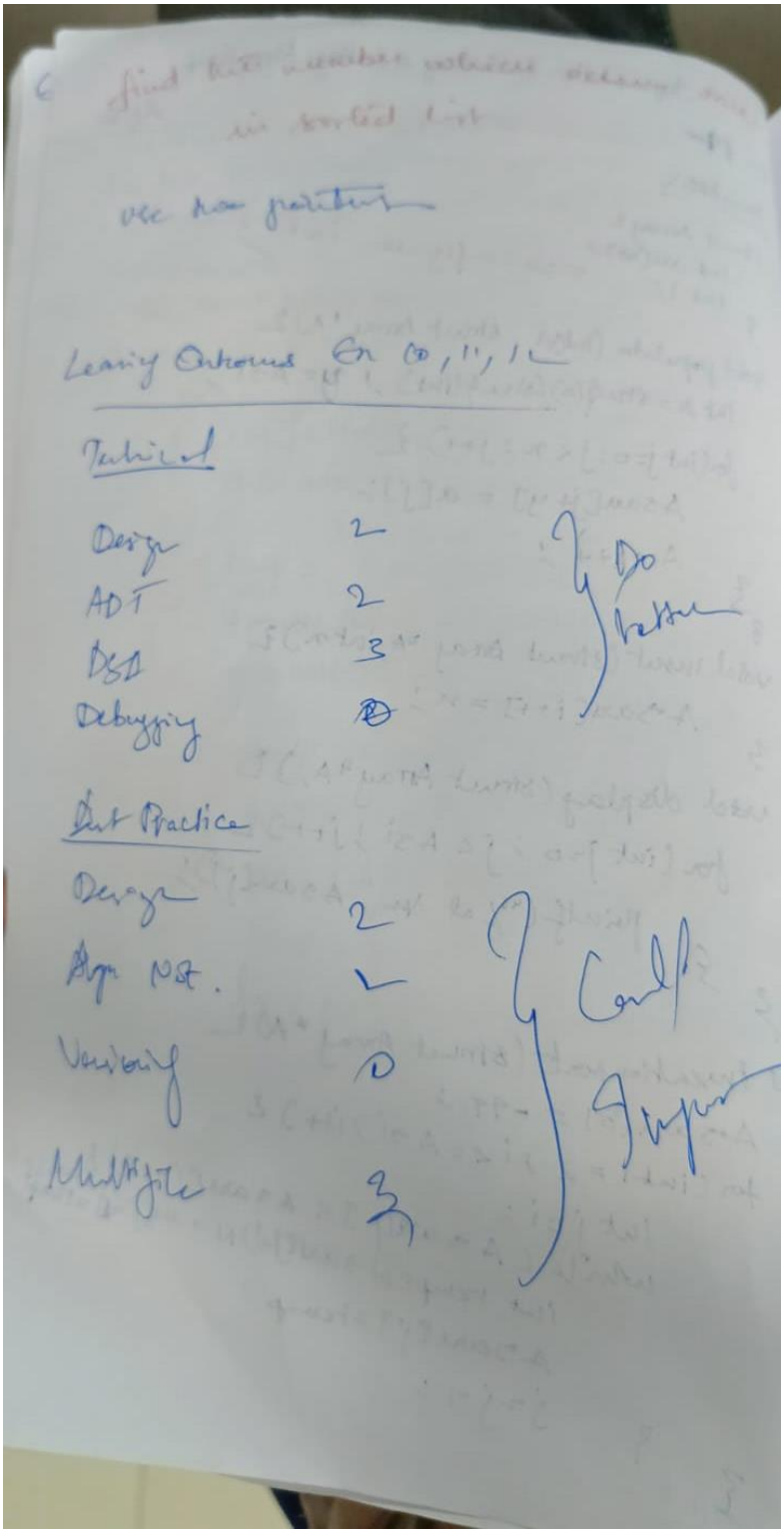
In your notebook, design atleast 3 test cases to realize best/worst/average cases. For each test case, trace the additional routines and write the state of the array in each iteration. Verify your expected output with the actual output of your function.

Ex 12

ArrayADT.h

```
struct Array {
    int arr[100];
    int i;
};
```
⟹ Configure i = 1;

```
void populate(int a, struct Array *A) {
    int x = sizeof(a)/sizeof(int); int y = A->i;
    for(int j=0; j<x; j++) {
        A->arr[j+y] = a[j];
        A->i++;
    }
}
void insert(struct Array *A, int x) {
    A->arr[i++] = x;
}
void display(struct Array *A,) {
    for(int j=0; j<A->i; j++) {
        printf("%d\n", A->arr[j]);
    }
}
insertionsort(struct Array *A) {
A->arr[0] = -99;
for(int i = 2; i<=A->i; i++) {
    int j = i;
    while(A->arr[j] < A->arr[j-1]) {
        int temp = N->arr[j-1]; N->arr[j-1] = temp
        A->arr[j] = temp;
    }   j = j-1;
```



Technical Outcomes:

Design	3	
Understanding DS	3	
Usage of DS	2	Needs improvement
Debugging	3	

Best Practices:

Design before coding	3	
Usage of algo	3	
Multifile	1	Needs improvement
versioning	3	