

**Lab report of Data Structure and Algorithm**

**Submitted by: Submitted to:**

Name: Rashik Shrestha Chandra Prakash Bhatta

Roll no.: ‘11’

Section: ‘B’

Faculty: BSc.CSIT

**Table of Contents**

|  |  |  |
| --- | --- | --- |
| **Lab no.** | **Lab name** | **Page no.** |
| **01** | **WAP for static and dynamic memory allocation.** | **1** |
| **02** | **WAP in C to implement stack operation.** | **5** |
| **03** | **Write the program to convert infix expression to postfix expression.** | **8** |
| **04** | **Write the program to implement (enqueue and dequeue) queue in C.** | **11** |
|  |  |  |

**LAB-01: WAP for static and dynamic memory allocation.**

**THEORY:**

**Static memory allocation:**

The memory allocation where the memory is allocated to the variable at the beginning is called static memory allocation. In static memory allocation, the memory to be allocated is fixed and is determined by the compiler at the compile-time itself.

For e.g.:

int array[10];

**Dynamic memory allocation:**

The mechanism by which memory can be allocated to variables during the run time is called dynamic memory allocation. The problem like memory overflow, memory wastage, etc in static memory allocation is solved by dynamic memory allocation.

C provides the following dynamic memory allocation and de-allocation function:

1. malloc()
2. calloc()
3. realloc()
4. free()

**PROGRAMS:**

**Program for static memory allocation:**

#include <stdio.h>

#include <stdlib.h>

int main()

{

int size;

printf("Enter limit of the text: \n");

scanf("%d", &size);

char str[size];

printf("Enter some text: \n");

fflush(stdin);

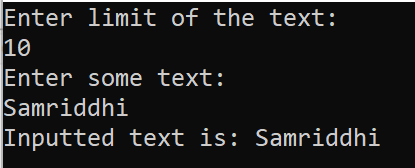
gets(str);

printf("Inputted text is: %s\n", str);

return 0;

}

**OUTPUT:**

****

**Program for dynamic memory allocation:**

#include <stdio.h>

#include <stdlib.h>

int main()

{

int size, resize;

char \*str = NULL;

printf("Enter limit of the text: \n");

scanf("%d", &size);

str = (char\*)malloc(size \* sizeof(char));

if (str != NULL)

{

printf("Enter some text: \n");

fflush(stdin);

gets(str);

printf("Inputted text by allocating memory using malloc() is: %s\n", str);

}

free(str);

str = (char\*)calloc(50, sizeof(char));

if (str != NULL)

{

printf("Enter some text: \n");

scanf(" ");

gets(str);

printf("Inputted text by allocating memory using calloc() is: %s\n", str);

}

printf("Enter the new size: \n");

scanf("%d", &resize);

str = (char\*)realloc(str, resize \* sizeof(char));

printf("Memory is successfully reallocated by using realloc() \n");

if (str != NULL)

{

printf("Enter some text: \n");

scanf(" ");

gets(str);

printf("Inputted text by reallocating memory using realloc()is: %s\n", str);

}

free(str);

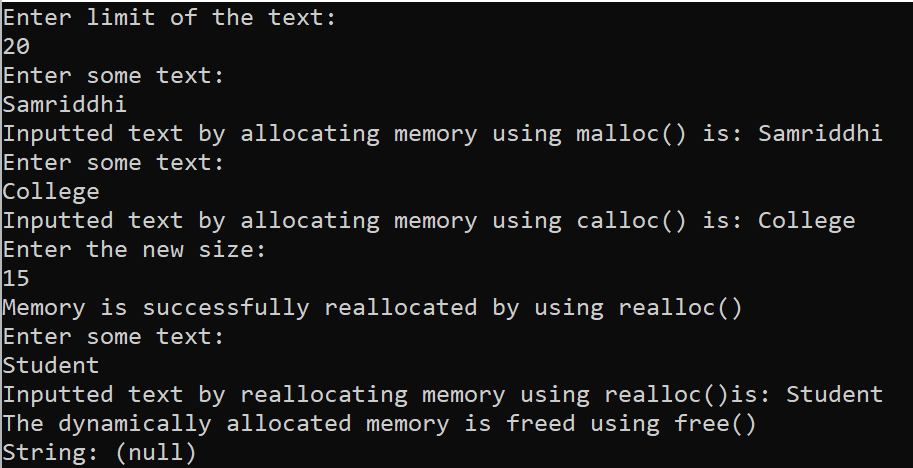
str = NULL;

printf("The dynamically allocated memory is freed using free()\nString: %s",str);

return 0;

}

**OUTPUT:**

****

**LAB-02: WAP in C to implement stack operation.**

**THEORY:**

**Stack:**

Stack is an ordered collection of item into which new data item may be inserted and from which item may be deleted at one end called top of the stack. The last added element will be removed first from the stack, this manner is called LIFO. There are two ways of stack implementation and they are:

1. Static implementation
2. Array implementation

Some of the stack operations are:

1. PUSH (X,S):

The process of adding new element to the top of the stack is called PUSH operation.

1. POP (S):

The process of deleting an element from top of the stack is called POP operation.

1. MAKENULL (S):

It makes stack (S) as an empty stack.

1. TOP (S):

It returns the element at the top of the stack (S).

1. EMPTY (S):

It returns true if stack is empty and returns false otherwise.

**PROGRAM:**

#include <iostream>

using namespace std;

int top = -1;

void push(int \* arr,int size,int data){

if (size - 1 == top ){ //check if the stack is full

cout << "stack overflow" << endl;

exit(0);

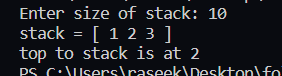
}else{

arr[++top] = data;

}

}

void print(int \* arr){

 cout << "stack = [ ";

for (int i= 0;i<= top;i++){

cout << arr[i] << " ";

}

cout << "]";

}

void pop(int \*arr){

// check if the stack is empty

if (top < 0){

cout << "Error, stack is already empty"<<endl;

}

else{

top -= 1;

}

}

int main(){

int size;

cout << "Enter size of stack: "; cin >> size;

int arr[size];

push(arr,size,1);

push(arr,size,2);

push(arr,size,3);

push(arr,size,55);

pop(arr);

print(arr);

cout << "\ntop to stack is at "<< top << endl;

}

**LAB-03: Write the program to convert infix expression to postfix expression.**

**THEORY:**

**Infix notation:**

The typical mathematical form of expression that we encounter generally is known as infix notation. In infix form, an operator is written in between two operands.

**Postfix notation:**

The expression where an operator is written after its operands is called postfix expression. This notation is also known as “Polished notation”.

**Algorithm to convert infix expression to postfix expression:**

Let P be the infix expression and Q be the postfix expression.

Step-I: Push ‘(’ onto the STACK and add ‘)’ to end of P.

Step-II: Scan P from left to right and repeat steps III to VI for each element of P until the stack is empty.

Step-III: If an operand is encountered, add it to Q.

Step-IV: If a left parenthesis is occurred, push it onto stack.

Step-V: If an operator ⨂ is encountered then,

1. Repeatedly pop from stack and add Q each operator (on the top of stack), which has the same precedence as, or higher precedence than operator ⨂.
2. Add operator ⨂ to stack.

Step-VI: If a right parenthesis is encountered then,

1. Repeatedly pop from stack and add to Q (on the top of stack until a left parenthesis is encountered).
2. Remove the left parenthesis. (Do not add the left parenthesis to Q)

Step-VII: Exit

**PROGRAM:**

#include <stdio.h>

#include<ctype.h>

char stack[50];

int top=-1;

void push(char elem)

{

stack[++top]=elem;

}

char pop()

{

return(stack[top--]);

}

int pr(char symbol)

{

if(symbol == '^')

{

return(3);

}

else if(symbol == '\*' || symbol == '/')

{

return(2);

}

else if(symbol == '+' || symbol == '-')

{

return(1);

}

else

{

return(0);

}

}

int main()

{

char infix[50],postfix[50],ch,elem;

int i=0,k=0;

printf("Enter Infix Expression : ");

scanf("%s",infix);

push('#');

while( (ch=infix[i++]) != '\0')

{

if( ch == '(')

{

push(ch);

}

else if(isalnum(ch))

{

postfix[k++]=ch;

}

else if( ch == ')')

{

while( stack[top] != '(')

{

postfix[k++]=pop();

}

elem=pop();

}

else

{

while( pr(stack[top]) >= pr(ch) )

{

postfix[k++]=pop();

}

push(ch);

}

}

while( stack[top] != '#')

{

postfix[k++]=pop();

}

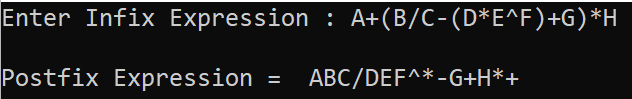
postfix[k]='\0';

printf("\nPostfix Expression = %s\n",postfix);

return 0;

}

**OUTPUT:**

****

**LAB-04: Write the program to implement (enqueue and dequeue) queue in C.**

**THEORY:**

**Queue:**

A queue is logically a first come first serve linear data structure. It is a homogeneous collection of elements in which new elements are added at one end called rear and the existing elements are deleted from other end called front.

The basic operations in queue are:

1. Enqueue ():

Insert an element to the queue from the rear end.

1. Dequeue ():

Delete an element from the queue from the front end.

1. Front:

Returns the object that is at the front of the queue without removing it.

1. Empty:

Returns true if the queue is empty otherwise returns false.

1. Size:

Returns the number of items in the queue.

**PROGRAM:**

#include <stdio.h>

#include<stdlib.h>

void insert();

void remov();

void display();

int queue[6], rear=-1, front=-1, item;

int main()

{

int ch;

do

{

printf("\n1. Insert\n2. Delete\n3. Display\n4. Exit\n");

printf("\nEnter your choice:");

scanf("%d", &ch);

switch(ch)

{

case 1:

insert();

break;

case 2:

remov();

break;

case 3:

display();

break;

case 4:

exit(0);

default:

printf("\n\nInvalid entry. Please try again...\n");

}

}while(ch<=4);

return 0;

}

void insert()

{

if(rear == 5)

{

printf("\nQueue is full.\n");

}

else

{

printf("\nEnter integer ITEM:");

scanf("%d", &item);

if (rear == -1 && front == -1)

{

rear = 0;

front = 0;

}

else

{

rear++;

}

queue[rear] = item;

printf("\nItem inserted: %d\n", item);

}

}

void remov()

{

if(front == -1)

{

printf("\nQueue is empty.\n");

}

else

{

item = queue[front];

if (front == rear)

{

front = -1;

rear = -1;

}

else

{

front++;

}

printf("\nItem deleted: %d\n", item);

}

}

void display()

{

int i;

if(front == -1)

{

printf("\nQueue is empty.\n");

}

else

{

printf("\nThe item/s in queue:\n");

for(i=front; i<=rear; i++)

{

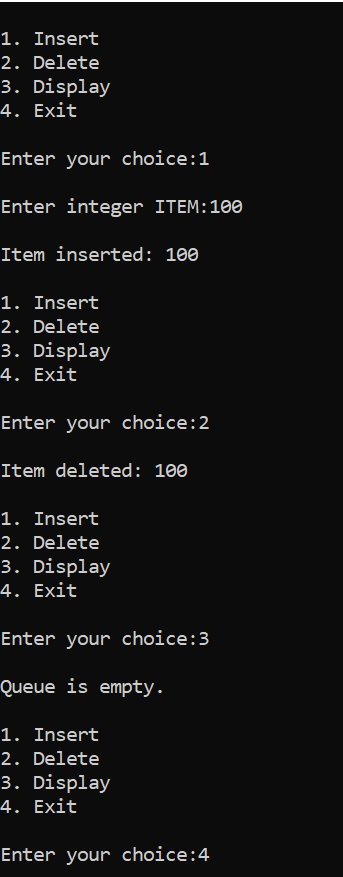
printf("%d ", queue[i]);

}

}

}

**OUTPUT:**

****

**LAB-05: Factorial**

**THEORY:**

**Factorial:**

Factorial can be understood as the product of all the integers from 1 to n, where n is the number of which we have to find the factorial of. The factorial of a negative number doesn’t exist and the factorial of 0 is 1.

Factorial of a number can be found using both iteration and recursion.

**Factorial of a number using iteration:**

#include <stdio.h>

int main(){

int n, i;

int fact = 1;

printf("Enter an integer: ");

scanf("%d", &n);

if (n < 0)

printf("Error! Factorial of a negative number doesn't exist.");

else{

for (i = 1; i <= n; i++){

fact \*= i;

}

printf("Factorial of %d = %d", n, fact);

}

return 0;

}

**Factorial of a number using recursion:**

#include<stdio.h>

int fact(int n){

if (n == 0){

return 1;

}

else{

return n \* fact(n-1);

}

}

int main(){

int n;

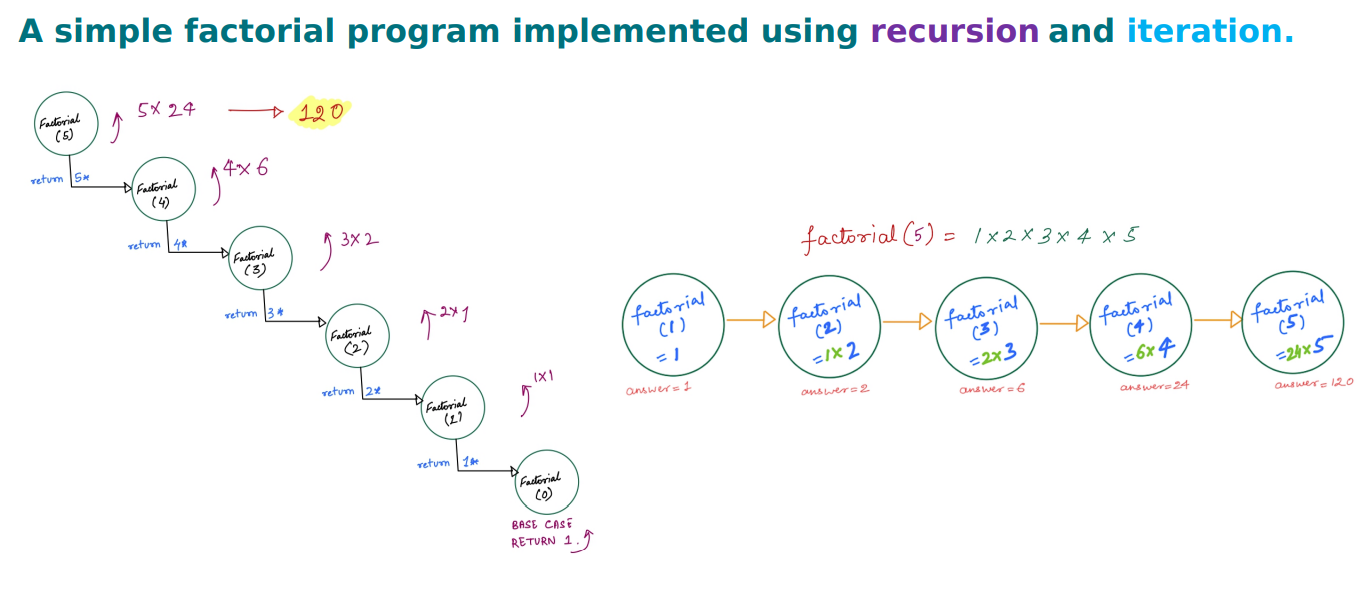
printf("Enter an integer: ");

scanf("%d", &n);

printf("Factorial of %d = %d", n, fact(n));

return 0;

}

**Flowchart:**

**LAB-06: Fibonacci Sequence**

**Theory:**

The Fibonacci sequence,also known as Fibonacci numbers, is defined as the sequence of numbers in which each number in the sequence is equal to the sum of two numbers before it.

The Fibonacci sequence is given as:

Fibonacci sequence = 0, 1, 1, 2, 3, 5, 8, 13, 21…

Here, the third term ‘1’ is obtained by adding the first and second term. (i.e., 0+1 = 1)

Similarly,

‘2’ is obtained by adding the second and third term (1+1 = 2)

‘3’ is obtained by adding the third and fourth term (1+2=3) and so on.

For example, the next term after 21 can be found by adding 13 and 21.

The Fibonacci sequence of numbers “Fn” is defined using the recursive relation with the seed values F0=0 and F1=1:

Fn = Fn-1+Fn-2

**Algorithm**

* Declare variables i, a,b , show
* Initialize the variables, a=0, b=1, and show =0
* Enter the number of terms of Fibonacci series to be printed
* Print First two terms of series
* Use loop for the following steps

**->** show=a+b

-> a=b

-> b=show

-> increase value of i each time by 1

-> print the value of show

**End**

**Fibonacci sequence using iteration:**

#include <stdio.h>

int main() {

int n, first = 0, second = 1, result, i;

printf("Give an input upto you want to print sequence : \n");

scanf("%d", & n);

printf("Fibonacci Series is:\n");

for (i = 0; i < n; i++) {

if (i <= 1)

result = i;

else {

result = first + second;

first = second;

second = result;

}

printf("%d ", result);

}

return 0;

}

**Fibonacci sequence using recursion:**

#include <stdio.h>

int fibbonacci(int n) {

if (n == 0) {

return 0;

} else if (n == 1) {

return 1;

} else {

return (fibbonacci(n - 1) + fibbonacci(n - 2));

}

}

int main() {

int n;

printf("Please give an input upto you want to print sequence : \n");

scanf("%d", & n);

int i;

printf("Fibbonacci sequence of %d: ", n);

for (i = 0; i < n; i++) {

printf("%d ", fibbonacci(i));

}

return 0;

}

**Fibonacci sequence using tail recursion:**

#include <stdio.h>

int fib(int n, int a, int b) {

if (n == 0)

return a;

else if (n == 1)

return b;

else

return fib(n - 1, b, a + b);

}

int main() {

int n = 9, a = 0, b = 1;

printf("%d ", fib(n, a, b));

return 0;

}

**LAB-07: Greatest Common Divisor (GCD)**

**GCD using iteration:**

include<stdio.h>

#include<conio.h>

int gcd(int, int);

int main() {

int a, b;

int gcd1;

printf("Enter the first number: ");

scanf("%d", & a);

printf("\nEnter second number:");

scanf("%d", & b);

gcd1 = gcd(a, b);

printf("The gcd of two numbers %d and %d is: %d", a, b, gcd1);

}

int gcd(int a, int b) {

int remainder;

if (a == 0) {

return (b);

} else if (b == 0) {

return (a);

} else {

while (b != 0) {

remainder = a % b;

a = b;

b = remainder;

}

return (a);

}

}

**GCD using recursion:**

#include <stdio.h>

int gcd(int x, int y) {

if (y == 0)

return x;

else

gcd(y, x % y);

}#int main()

{

int a, b, g;

printf("Enter a and b:\n");

scanf("%d%d", &a, &b);

g = gcd(a, b);

if(g < 0)

{

g \*= -1;

}

printf("GCD(%d, %d) = %d\n", a, b, g);

return 0;

}



**LAB-08: Tower of Hanoi (TOH)**

**Tower of Hanoi (TOH) using iteration:**

#include <stdio.h>

#include <math.h>

#include <stdlib.h>

#include <limits.h>

struct Stack {

unsigned capacity;

int top;

int \* array;

};

// function to create a stack of given capacity.

struct Stack \* createStack(unsigned capacity) {

struct Stack \* stack =

(struct Stack \* ) malloc(sizeof(struct Stack));

stack -> capacity = capacity;

stack -> top = -1;

stack -> array =

(int \* ) malloc(stack -> capacity \* sizeof(int));

return stack;

}

// Stack is full when top is equal to the last index

int isFull(struct Stack \* stack) {

return (stack -> top == stack -> capacity - 1);

}

// Stack is empty when top is equal to -1

int isEmpty(struct Stack \* stack) {

return (stack -> top == -1);

}

// Function to add an item to stack. It increases

// top by 1

void push(struct Stack \* stack, int item) {

if (isFull(stack))

return;

stack -> array[++stack -> top] = item;

}

// Function to remove an item from stack. It

// decreases top by 1

int pop(struct Stack \* stack) {

if (isEmpty(stack))

return INT\_MIN;

return stack -> array[stack -> top--];

}

//Function to show the movement of disks

void moveDisk(char fromPeg, char toPeg, int disk) {

printf("Move the disk %d from \'%c\' to \'%c\'\n", disk, fromPeg, toPeg);

}

// Function to implement legal movement between

// two poles

void moveDisksBetweenTwoPoles(struct Stack \* src, struct Stack \* dest, char s, char d) {

int pole1TopDisk = pop(src);

int pole2TopDisk = pop(dest);

// When pole 1 is empty

if (pole1TopDisk == INT\_MIN) {

push(src, pole2TopDisk);

moveDisk(d, s, pole2TopDisk);

}

// When pole2 pole is empty

else if (pole2TopDisk == INT\_MIN) {

push(dest, pole1TopDisk);

moveDisk(s, d, pole1TopDisk);

}

// When top disk of pole1 > top disk of pole2

else if (pole1TopDisk > pole2TopDisk) {

push(src, pole1TopDisk);

push(src, pole2TopDisk);

moveDisk(d, s, pole2TopDisk);

}

// When top disk of pole1 < top disk of pole2

else {

push(dest, pole2TopDisk);

push(dest, pole1TopDisk);

moveDisk(s, d, pole1TopDisk);

}

}

//Function to implement TOH puzzle

void tohIterative(int num\_of\_disks, struct Stack \* src, struct Stack \* aux, struct Stack \* dest) {

int i, total\_num\_of\_moves;

char s = 'J', d = 'K', a = 'L';

//If number of disks is even, then interchange

//destination pole and auxiliary pole

if (num\_of\_disks % 2 == 0) {

char temp = d;

d = a;

a = temp;

}

total\_num\_of\_moves = pow(2, num\_of\_disks) - 1;

//Larger disks will be pushed first

for (i = num\_of\_disks; i >= 1; i--)

push(src, i);

for (i = 1; i <= total\_num\_of\_moves; i++) {

if (i % 3 == 1)

moveDisksBetweenTwoPoles(src, dest, s, d);

else if (i % 3 == 2)

moveDisksBetweenTwoPoles(src, aux, s, a);

else if (i % 3 == 0)

moveDisksBetweenTwoPoles(aux, dest, a, d);

}

}

// Driver Program

int main() {

// Input: number of disks

unsigned num\_of\_disks;

printf("Enter number of disks:");

scanf("%d", & num\_of\_disks);

struct Stack \* src, \* dest, \* aux;

// Create three stacks of size 'num\_of\_disks'

// to hold the disks

src = createStack(num\_of\_disks);

aux = createStack(num\_of\_disks);

dest = createStack(num\_of\_disks);

tohIterative(num\_of\_disks, src, aux, dest);

return 0;

}

**Tower of Hanoi (TOH) using recursion:**

#include <stdio.h>

int counter = 1;

void hanoi(int n,char source,char dest,char aux){

if (n == 0){

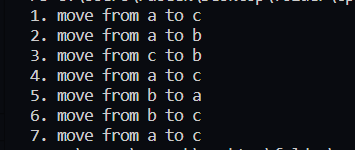
return;

}

hanoi(n-1,source,aux,dest);

printf("%d. move from %c to %c\n",counter++,source,dest);

hanoi(n-1,aux,dest,source);

****

}

int main(){

hanoi(3,'a','c','b');

}

**LAB-09: Linked List**

**Theory:**

A linked list is a linear grouping of data pieces in computer science whose order is not determined by their actual location in memory. Instead, every part relates to the one before it. It is a data structure made up of a number of nodes that collectively stand for a sequence

**Singly Linked List:**

**Algorithm:**

**1. Insert at the End**

1. Allocate memory for new node
2. Store data
3. Traverse to last node
4. Change next of last node to recently created node

**2. Insert at the beginning**

1. Allocate memory for new node
2. Store data
3. Change next of new node to point to head
4. Change head to point to recently created node

**3. Insert at the Middle**

1. Allocate memory and store data for new node
2. Traverse to node just before the required position of new node
3. Change next pointers to include new node in between

**4. Delete from beginning**

1. Point head to the second node
2. Free the first node

**5. Delete from end**

1. Traverse to second last element
2. Change its next pointer to null and free the last node

**6. Delete from middle**

1. Traverse to element before the element to be deleted
2. Change next pointers to exclude the node from the chain and free the deleted node

**Singly Linked List**

#include <stdio.h>

#include <stdlib.h>

struct Node{

int data;

struct Node \*next;

};

typedef struct Node NodeType;

NodeType \*head = NULL;

NodeType \*getnode(){

NodeType \*p;

p=(NodeType\*)malloc(sizeof(NodeType));

return(p);

}

void display(){

// printf("linked list elements:\n");

if (head == NULL){

printf("Linked list is empty\n");

}

else{

NodeType \*ptr;

ptr = head;

printf("[ ");

while (ptr != NULL){

if (ptr->next == NULL){

printf("%d",ptr->data);

}else{

printf("%d -> ",ptr->data);

}

ptr = ptr->next;

}

printf(" ]\n");

}

}

void insertEnd(int x){

NodeType \*p;

p = (NodeType\*)malloc(sizeof(NodeType));

p->data = x;

p->next = NULL;

if (head == NULL){

head = p;

}else{

NodeType \*ptr = head;

while (ptr->next!=NULL){

ptr = ptr->next;

}

ptr->next = p;

}

}

void insertBeg(int x){

NodeType \*ptr = getnode();

ptr->data = x;

ptr->next = head;

head = ptr;

}

void insert(int x,int index){

if (head == NULL){

printf("Empty List: index out of range");

return;

}

if (index ==0){

insertBeg(x);

return;

}

NodeType \*prev = getnode();

NodeType \*ptr = head;

// ptr = head;

for (int i=0;i<index;i++){

prev = ptr;

ptr = ptr->next;

}

NodeType \*p =getnode();

p->data = x;

p->next = ptr;

prev->next = p;

}

void deleteEnd(){

if (head==NULL){

printf("Cannot delete. List is already empty\n");

return;

}

NodeType \*prev = getnode();

NodeType \*ptr = head;

while (ptr->next!=NULL){

prev = ptr;

ptr = ptr->next;

}

prev->next = NULL;

free(ptr);

}

void deleteBeg(){

if (head==NULL){

printf("Cannot delete. List is already empty\n");

return;

}

NodeType \*ptr = head;

head = ptr->next;

free(ptr);

}

void del(int index){

if (head==NULL){

printf("Cannot delete. List is already empty\n");

return;

}

if (index == 0){

deleteBeg;

return;

}

NodeType \*ptr = head;

NodeType \*prev;

for (int i=0;i<index;i++){

prev = ptr;

ptr = ptr->next;

}

prev->next = ptr->next;

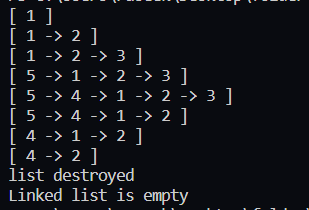
free(ptr);

}

void destroy(){

while (head!=NULL){

NodeType \*toDel = head;

**** head = head->next;

free(toDel);

}

printf("list destroyed\n");

}

int main(){

insertEnd(1); // add 1

display();

insertEnd(2); // add 2

display();

insertEnd(3); // add 3

display();

insertBeg(5); // add 5 at beginning

display();

insert(4,1); // add 4 at 1st index

display();

deleteEnd(); // delete end elemnt

display();

deleteBeg(); // delete 1st element

display();

del(1); //delete elem at index 1

display();

destroy();

display();

}

**LAB-10: Sorting Algorithm**

**Theory:**

A sorting algorithm is a technique for arranging a lot of things in a particular order, like alphabetically, from highest to lowest value, or from shortest to longest distance. Input lists of items are used as the basis for sorting algorithms, which then apply certain operations to those lists to produce sorted arrays as output.

**Bubble Sort**

Step 1: Start

Step 2: Read the array of given items from the user.

Step 3: Take the first element(index = 0), compare the current element with the next element.

Step 4: If the current element is greater than the next element, swap them.

Step 5: Else, If the current element is less than the next element, then move to the next element.

Step 6: Repeat Step 3 to Step 5 until all elements are sorted.

Step 7: Stop

#include <stdio.h>

void bubbleSort(int \*arr,int size){

int temp;

for (int i=0;i<size-1;i++){

for (int j=0;j<size -1 -i;j++){

if (arr[j] > arr[j+1]){

temp = arr[j+1];

arr[j+1] = arr[j];

arr[j] = temp;

}

}

}

}

void printarr(int \*arr,int size){

for (int i=0;i<size;i++){

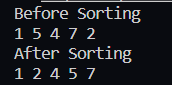
printf("%d ",arr[i]);

}

}

int main(){

int size = 5;

 int arr[size] = {1,5,4,7,2};

printf("Before Sorting\n");

printarr(&arr[0], size);

bubbleSort(&arr[0],size);

printf("\nAfter Sorting\n");

printarr(&arr[0],size);

}

**Selection Sort**

Step 1 : Set MIN to location 0

Step 2 : Search the minimum element in the list

Step 3 : Swap with value at location MIN

Step 4 : Increment MIN to point to next element

Step 5 : Repeat until list is sorted

#include<stdio.h>

void printarr(int \*arr,int size){

for (int i=0;i<size;i++){

printf("%d ",arr[i]);

}

}

int main(){

int size=10,min=0,temp;

int arr[size] = {23,10,20,3,45,76,67,24,2,11};

printf("Bofore sorting:\n");

printarr(&arr[0],size);

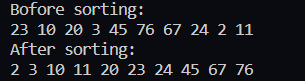
for (int i=0;i<size-1;i++){

min = i;

for (int j=i+1;j<size;j++){

if (arr[j] < arr[min]){

min = j;

 }

}

if (min!=i){

temp = arr[min];

arr[min] = arr[i];

arr[i] = temp;

}

}

printf("\nAfter sorting:\n");

printarr(&arr[0],size);

}

**Insertion Sort**

**Algorithm**

1. Iterate from arr[1] to arr[N] over the array.
2. Compare the current element (key) to its predecessor.
3. If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

#include <stdio.h>

void insertionSort(int arr[], int n){

int i, key, j;

for (i = 1; i < n; i++) {

key = arr[i];

j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j = j - 1;

}

 arr[j + 1] = key;

}

}

void printArray(int arr[], int n){

int i;

for (i = 0; i < n; i++)

printf("%d ", arr[i]);

printf("\n");

}

int main()

{

int arr[] = { 12, 11, 13, 5, 6 };

int n = sizeof(arr) / sizeof(arr[0]);

insertionSort(arr, n);

printArray(arr, n);

}

**Merge Sort**

**Algorithm**

step 1: start

step 2: declare array and left, right, mid variable

step 3: perform merge function.

mergesort(array,left,right)

mergesort (array, left, right)

if left > right

return

mid= (left+right)/2

mergesort(array, left, mid)

mergesort(array, mid+1, right)

merge(array, left, mid, right)

step 4: Stop

#include <stdio.h>

#include <stdlib.h>

void merge(int arr[], int l, int m, int r){

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

int L[n1], R[n2];

for (i = 0; i < n1; i++)

L[i] = arr[l + i];

for (j = 0; j < n2; j++)

R[j] = arr[m + 1 + j];

i = 0; // Initial index of first subarray

j = 0; // Initial index of second subarray

k = l; // Initial index of merged subarray

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

}

else {

arr[k] = R[j];

j++;

}

k++;

}

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

void mergeSort(int arr[], int l, int r)

{

if (l < r) {

int m = l + (r - l) / 2;

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

merge(arr, l, m, r);

}

}

void printArray(int A[], int size)

{

int i;

for (i = 0; i < size; i++)

printf("%d ", A[i]);

printf("\n");

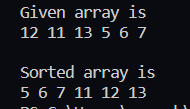
}

int main()

{

int arr[] = { 12, 11, 13, 5, 6, 7 };

int arr\_size = sizeof(arr) / sizeof(arr[0]);

 printf("Given array is \n");

printArray(arr, arr\_size);

mergeSort(arr, 0, arr\_size - 1);

printf("\nSorted array is \n");

printArray(arr, arr\_size);

return 0;

}