# Lab 1: Insert a data in 3rd position of existing array.

## Algorithm:

Step 1: Start

Step 2: Initialize the array and define it with fixed size

Step 3: initialize the size of the array.

Step 4: Define a variable for the data to be inserted

Step 5: Shift elements to the right to make space for the new element

For i from size (which is 4) to 3 (inclusive), decrementing i in each step:

Set array[i] = array[i - 1].

Step 6: Insert the new element at the 3rd position (index 2):

Set array[2] = new\_data.

Step 7: Update the size of the array to reflect addition of new element.

Step 8: Print the updated array.

Step 9:End

## Implementation of algorithm:

#include <stdio.h>

int main() {

int array[5] = {1, 2, 4, 5};

int size = 4;

int new\_data = 3;

for (int i = size; i > 2; i--) {

array[i] = array[i - 1];

}

array[2] = new\_data;

size++;

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

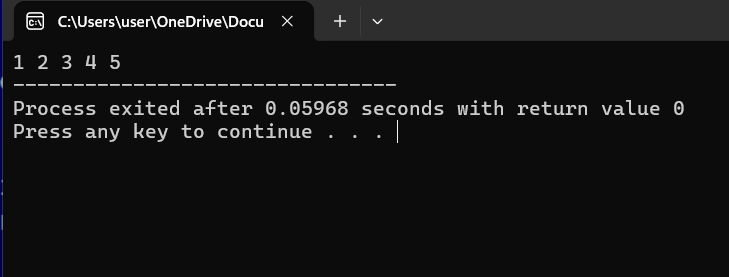
printf("%d ", array[i]);

}

return 0;

}

## Output:



# Lab 2: Find the Product of odd position and sum of even positions of an array

## Algorithm:

Step 1:Start

Step 2: Initialize an array and variables.

Declare an array ‘a’ with 5 elements:{1,2,3,4,5}

Declare integer ‘s ’for sum and initialize it to 0

Declare integer ‘p’ for product and initialize it to 1

Step 3: Loop through the array

For i from 0 to 4 (inclusive):

Check if i is even

If i % 2 == 0 (i.e., i is even):

Add a[i] to s

Check if i is odd

If i % 2 != 0 (i.e., i is odd):

Multiply p by a[i]

Step 4:Print results

Print the value of s with the message "sum="

Print the value of p with the message "Product="

Step 5:End

## Implementation of algorithm:

#include<stdio.h>

int main(){

int a[5]={1,2,3,4,5};

int s=0,p=1;

for(int i=0;i<5;i++){

if(i%2==0){

s=s+a[i];

}

else{

p=p\*a[i];

}

}

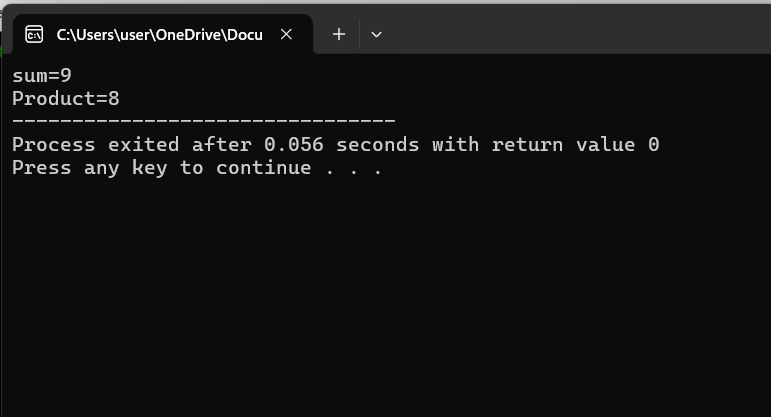
printf("sum=%d",s);

printf("\nProduct=%d",p);

return 0;

}

## Output:



## Lab 3: Show Sequential/Linear Search using iterative approach.

## Algorithm:

Step 1:Start

Step 2: Initialize and array , key and length variable.

Declare an array a with 7 elements:{5,1,9,7,6,2,0}

Take key/target value ‘n’

Take ‘length’

Step 3: Start a conditional branch.

Step 4: Compare the elements of an array with Key till it is found or end of array.

If a[i]==n //found

If a[i]!=n//not found

Step 5: Display the result by printing the index of that key element.

If not found print Not Found

Step 6: End

## Implementation of algorithm:

#include<stdio.h>

int main(){

int a[7]={5,1,9,7,6,2,0};

printf("Elments of array:");

for(int i=0;i<7;i++){

printf("%d\t",a[i]);

}

int n,t=0;

int length=sizeof(a)/sizeof(int);

printf("\nEnter element to search:");

scanf("%d",&n);

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

if(a[i]==n){

printf("Found at index:%d",i);

t=1;

}

}

if(t==0){

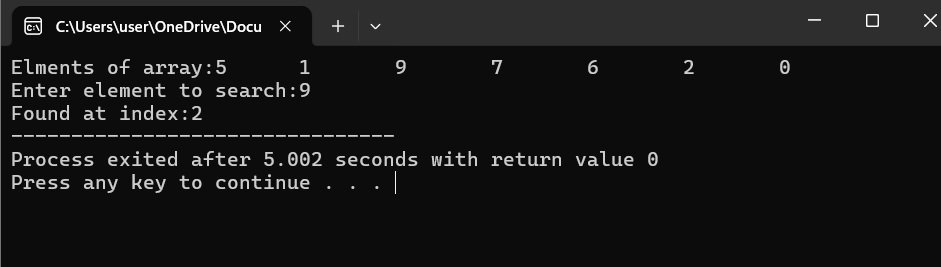
printf("Not found");

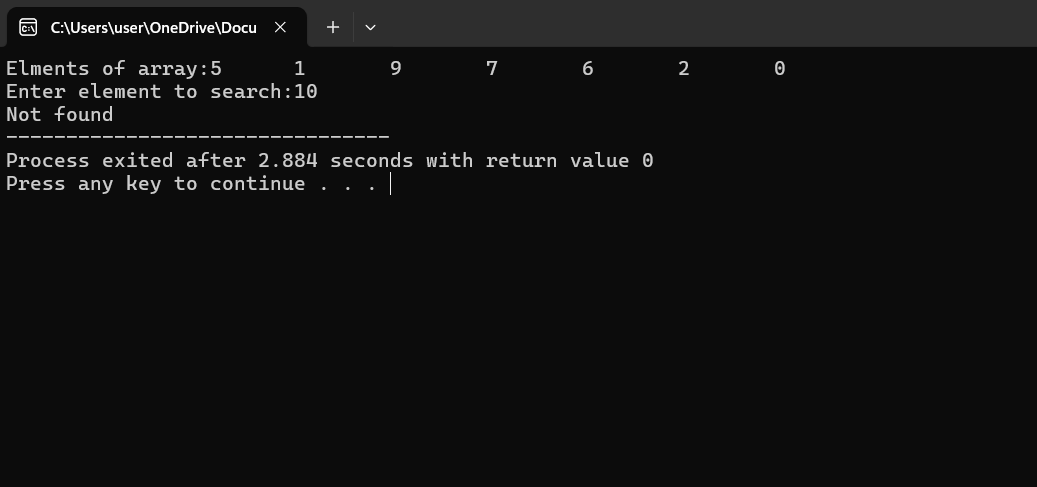
}

return 0;

}

## Output





## Lab 4: Show Binary Search

## Algorithm:

Step 1: Start

Step 2: Define the binary search function ‘binsearch’

Input: Integer array arr, integer lowerbound, integer upperbound, integer target

Output: Integer index of target if found, otherwise -1

Step 3: Initialize the binary search loop in binsearch function

* Loop: While lowerbound is less than or equal to upperbound

Calculate mid as (lowerbound + upperbound) / 2

Condition: If arr[mid] equals target

Return mid (target found at index mid)

Condition: If arr[mid] is less than target

Set lowerbound to mid + 1 (ignore left half)

Condition: If arr[mid] is greater than target

Set upperbound to mid - 1 (ignore right half)

End Loop

Return -1 (target not found)

Step 4: Define the main function

Step 6: Initialize the array ‘arr’ with values {5, 7, 8, 11, 17, 18, 19, 30}

Calculate the size of the array n as sizeof(arr) / sizeof(arr[0])

Print "Elements of array are:" followed by the elements of the array in a loop

Step 7: Ask the user for target element and read the input into variable ‘target’.

Step 8: Call the binsearch function

Pass the array arr, lowerbound as 0, upperbound as n - 1, and the target element

Store the result in variable result

Step 9: Output the result of the binary search

Condition: If result is not -1

Print "Element found at index" followed by result

Output the result of the binary search

Condition: If result is not -1

Print "Element found at index" followed by result

Step 10: End

## Implementation of algorithm:

#include <stdio.h>

int binsearch(int arr[], int lowerbound, int upperbound, int target) {

while (lowerbound <= upperbound) {

int mid = (lowerbound + upperbound) / 2;

if (arr[mid] == target)

return mid;

else if (arr[mid] < target)

lowerbound = mid + 1;

else

upperbound = mid - 1;

}

return -1;

}

int main() {

int arr[] = {5, 7, 8, 11, 17, 18, 19, 30};

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

printf("Elements of array are:");

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

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

}

int target;

printf("\nEnter element to search: ");

scanf("%d", &target);

int result = binsearch(arr, 0, n - 1, target);

if (result != -1) {

printf("Element found at index %d\n", result);

} else {

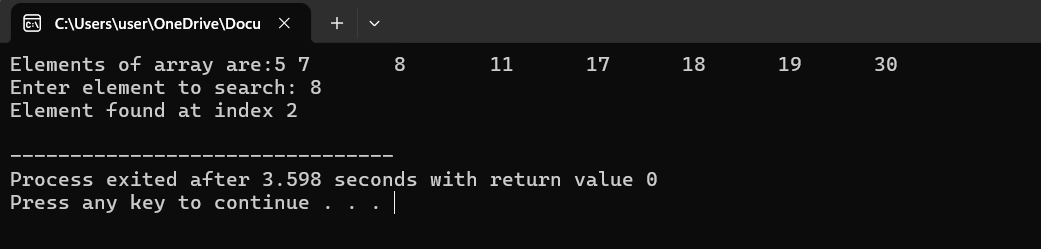
printf("Element not found in the array\n");

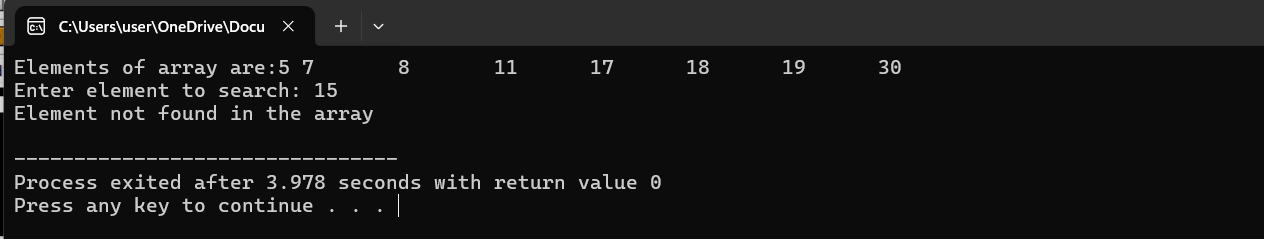
}

return 0;

}

## Output





## Lab 5: Operation of a linked list(create, insert, search, delete)

## Algorithm:

Srep 1: Start

Step2:Initialize head as NULL.

Step 3:While choice is not 9:

1. Display the main menu.
2. Prompt the user to enter a choice.
3. Read the choice.
4. Switch based on the choice:

Case 1: Call beginsert().

Case 2: Call lastinsert().

Case 3: Call randominsert().

Case 4: Call begin\_delete().

Case 5: Call last\_delete().

Case 6: Call random\_delete().

Case 7: Call search().

Case 8: Call display().

Case 9: Exit the program.

Default: Print "Please enter a valid choice".

Step 4: End

## beginsert Algorithm

1. Start
2. Allocate memory for a new node.
3. If memory allocation fails, print "OVERFLOW" and return.
4. Prompt the user to enter a value.
5. Read the value and assign it to data of the new node.
6. Set the next of the new node to head.
7. Set head to the new node.
8. Print "Node inserted".
9. End

## lastinsert Algorithm

1. Start
2. Allocate memory for a new node.
3. If memory allocation fails, print "OVERFLOW" and return.
4. Prompt the user to enter a value.
5. Read the value and assign it to data of the new node.
6. If head is NULL:
7. Set head to the new node.
8. Set the next of the new node to NULL.
9. Print "Node inserted".
10. Else:
11. Initialize temp as head.
12. While temp->next is not NULL, move to the next node.
13. Set the next of temp to the new node.
14. Set the next of the new node to NULL.
15. Print "Node inserted".
16. End

## randominsert Algorithm

1. Start
2. Allocate memory for a new node.
3. If memory allocation fails, print "OVERFLOW" and return.
4. Prompt the user to enter a value.
5. Read the value and assign it to data of the new node.
6. Prompt the user to enter the location after which to insert.
7. Read the location.
8. Initialize temp as head.
9. For i from 0 to loc - 1:
10. Move to the next node.
11. If temp is NULL, print "can't insert" and return.
12. Set the next of the new node to the next of temp.
13. Set the next of temp to the new node.
14. Print "Node inserted".
15. End

## begin\_delete Algorithm

1. Start
2. If head is NULL, print "List is empty" and return.
3. Initialize ptr as head.
4. Set head to the next of ptr.
5. Free ptr.
6. Print "Node deleted from the beginning".
7. End

## last\_delete Algorithm

1. Start
2. If head is NULL, print "List is empty" and return.
3. If head->next is NULL:
4. Free head.
5. Set head to NULL.
6. Print "Only node of the list deleted".
7. Else:
8. Initialize ptr as head.
9. While ptr->next is not NULL:
10. Set ptr1 as ptr.
11. Move to the next node.
12. Set ptr1->next to NULL.
13. Free ptr.
14. Print "Deleted node from the last".
15. End

## random\_delete Algorithm

1. Start
2. Prompt the user to enter the location after which to delete.
3. Read the location.
4. Initialize ptr as head.
5. For i from 0 to loc - 1:
6. Set ptr1 as ptr.
7. Move to the next node.
8. If ptr is NULL, print "Can't delete" and return.
9. Set the next of ptr1 to the next of ptr.
10. Free ptr.
11. Print "Deleted node loc + 1".
12. End

## search Algorithm

1. Start
2. Initialize ptr as head.
3. If ptr is NULL, print "Empty List" and return.
4. Prompt the user to enter the item to search.
5. Read the item.
6. Initialize i as 0 and flag as 1.
7. While ptr is not NULL:
8. If ptr->data equals item, print "Item found at location i + 1", set flag to 0, and break.
9. Increment i.
10. Move to the next node.
11. If flag is 1, print "Item not found".
12. End

## display Algorithm

1. Start
2. Initialize ptr as head.
3. If ptr is NULL, print "Nothing to print" and return.
4. Print "Printing values...".
5. While ptr is not NULL:
6. Print the data of ptr.
7. Move to the next node.
8. End

## Implementation of algorithm:

#include <stdio.h>

#include <stdlib.h>

struct node {

int data;

struct node \*next;

};

struct node \*head = NULL;

void beginsert();

void lastinsert();

void randominsert();

void begin\_delete();

void last\_delete();

void random\_delete();

void display();

void search();

int main() {

int choice = 0;

while (choice != 9) {

printf("\n\n\*\*\*\*\*\*\*\*\*Main Menu\*\*\*\*\*\*\*\*\*\n");

printf("\nChoose one option from the following list ...\n");

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

printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at any random location\n4.Delete from Beginning\n5.Delete from last\n6.Delete node after specified location\n7.Search for an element\n8.Show\n9.Exit\n");

printf("\nEnter your choice?\n");

scanf("%d", &choice);

switch (choice) {

case 1:

beginsert();

break;

case 2:

lastinsert();

break;

case 3:

randominsert();

break;

case 4:

begin\_delete();

break;

case 5:

last\_delete();

break;

case 6:

random\_delete();

break;

case 7:

search();

break;

case 8:

display();

break;

case 9:

exit(0);

break;

default:

printf("Please enter a valid choice..");

}

}

}

void beginsert() {

struct node \*ptr;

int item;

ptr = (struct node \*) malloc(sizeof(struct node));

if (ptr == NULL) {

printf("\nOVERFLOW");

} else {

printf("\nEnter value\n");

scanf("%d", &item);

ptr->data = item;

ptr->next = head;

head = ptr;

printf("\nNode inserted");

}

}

void lastinsert() {

struct node \*ptr, \*temp;

int item;

ptr = (struct node \*) malloc(sizeof(struct node));

if (ptr == NULL) {

printf("\nOVERFLOW");

} else {

printf("\nEnter value?\n");

scanf("%d", &item);

ptr->data = item;

ptr->next = NULL;

if (head == NULL) {

head = ptr;

printf("\nNode inserted");

} else {

temp = head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = ptr;

printf("\nNode inserted");

}

}

}

void randominsert() {

int i, loc, item;

struct node \*ptr, \*temp;

ptr = (struct node \*) malloc(sizeof(struct node));

if (ptr == NULL) {

printf("\nOVERFLOW");

} else {

printf("\nEnter element value\n");

scanf("%d", &item);

ptr->data = item;

printf("\nEnter the location after which you want to insert\n");

scanf("%d", &loc);

temp = head;

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

temp = temp->next;

if (temp == NULL) {

printf("\ncan't insert\n");

return;

}

}

ptr->next = temp->next;

temp->next = ptr;

printf("\nNode inserted");

}

}

void begin\_delete() {

struct node \*ptr;

if (head == NULL) {

printf("\nList is empty\n");

} else {

ptr = head;

head = ptr->next;

free(ptr);

printf("\nNode deleted from the beginning\n");

}

}

void last\_delete() {

struct node \*ptr, \*ptr1;

if (head == NULL) {

printf("\nList is empty\n");

} else if (head->next == NULL) {

head = NULL;

free(head);

printf("\nOnly node of the list deleted\n");

} else {

ptr = head;

while (ptr->next != NULL) {

ptr1 = ptr;

ptr = ptr->next;

}

ptr1->next = NULL;

free(ptr);

printf("\nDeleted node from the last\n");

}

}

void random\_delete() {

struct node \*ptr, \*ptr1;

int loc, i;

printf("\nEnter the location of the node after which you want to perform deletion\n");

scanf("%d", &loc);

ptr = head;

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

ptr1 = ptr;

ptr = ptr->next;

if (ptr == NULL) {

printf("\nCan't delete\n");

return;

}

}

ptr1->next = ptr->next;

free(ptr);

printf("\nDeleted node %d\n", loc + 1);

}

void search() {

struct node \*ptr;

int item, i = 0, flag = 1;

ptr = head;

if (ptr == NULL) {

printf("\nEmpty List\n");

} else {

printf("\nEnter item which you want to search\n");

scanf("%d", &item);

while (ptr != NULL) {

if (ptr->data == item) {

printf("Item found at location %d\n", i + 1);

flag = 0;

break;

}

i++;

ptr = ptr->next;

}

if (flag) {

printf("Item not found\n");

}

}

}

void display() {

struct node \*ptr;

ptr = head;

if (ptr == NULL) {

printf("Nothing to print\n");

} else {

printf("\nPrinting values...\n");

while (ptr != NULL) {

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

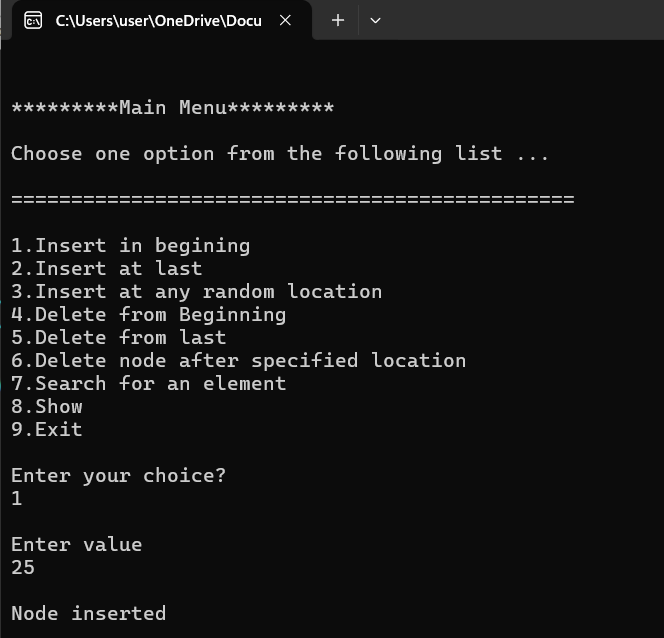
ptr = ptr->next;

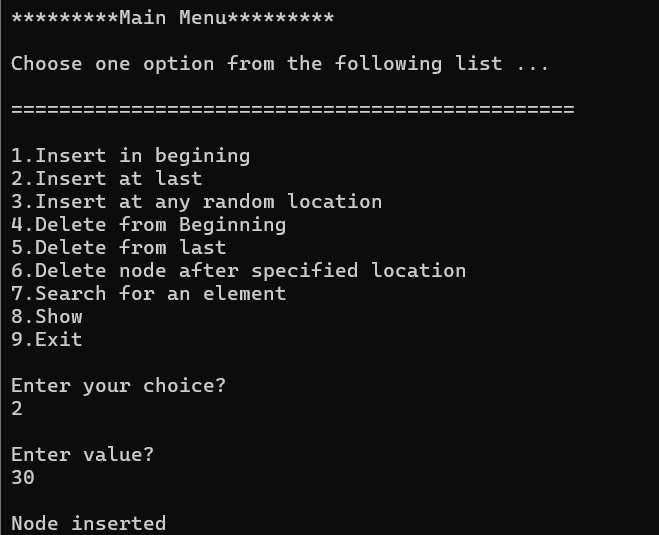
}

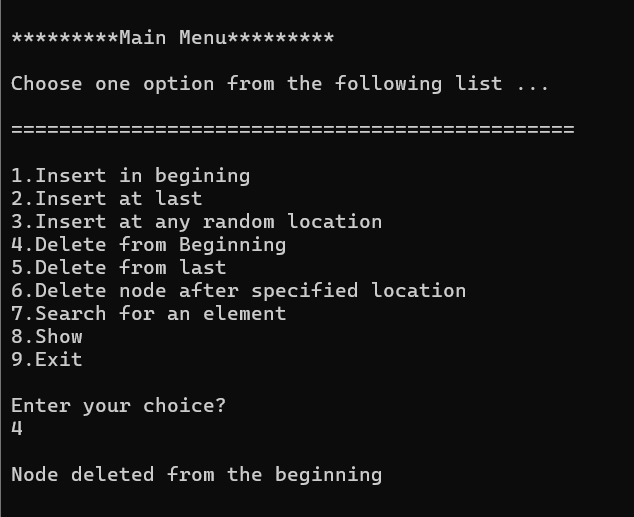
}

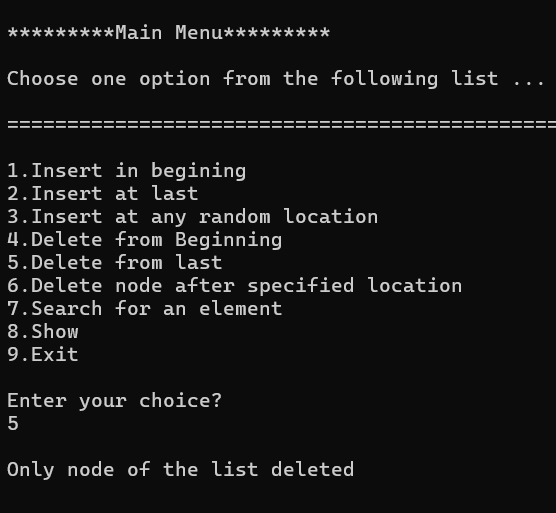
}

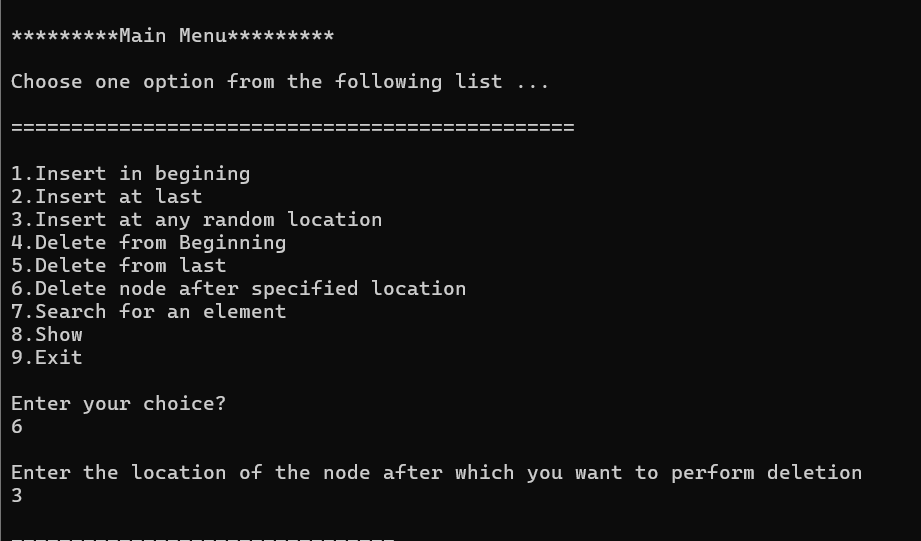
## Output

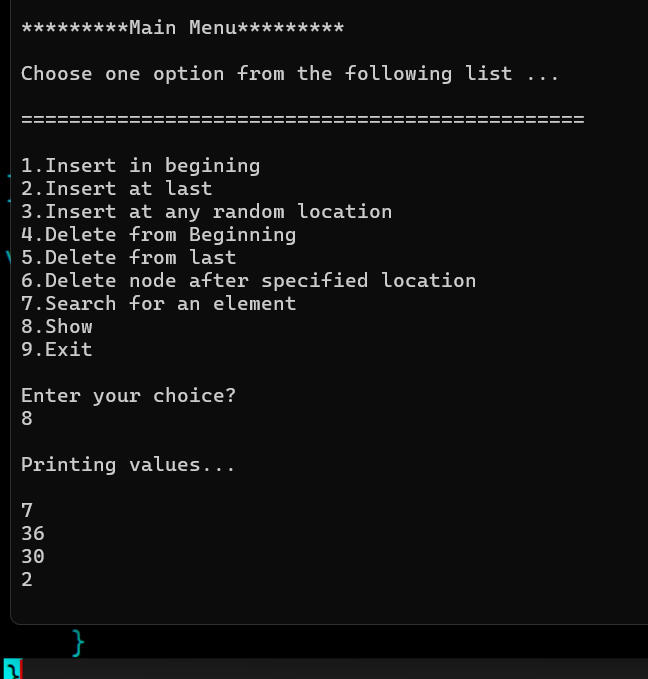
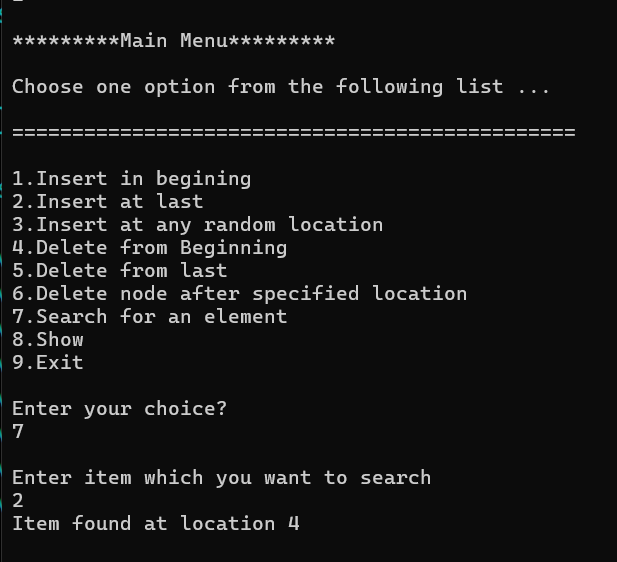


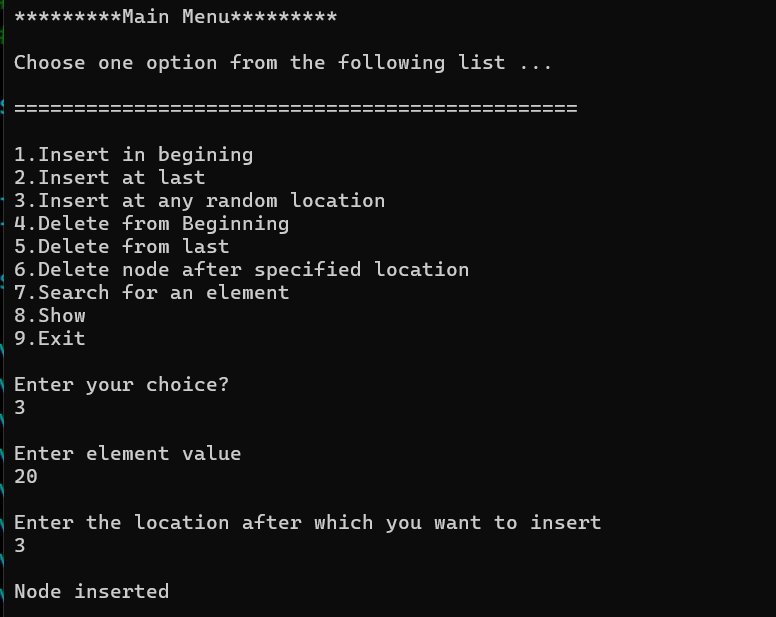












# Lab 6: Stack Functions

## Algorithm:

Step 1:Start

Step 2: Initialize Variables:

Define a constant MAX with value 100 to represent the maximum size of the stack.

Declare an integer array stack[MAX] to store stack elements.

Declare an integer variable top and initialize it to -1 to indicate the stack is empty.

Step 3: Display Menu and Get User Choice:

Continuously display a menu and prompt the user for a choice until the user chooses to exit.

Step 4: Print the menu options(Push,Display,Exit)

Read the user's choice and store it in the variable choice.

Step 5: Implement a switch-case structure to handle the user's choice:

Case 1: Push:

1. Prompt the user to enter the value to push.
2. Read the value and store it in the variable value.
3. Call the push(value) function.

Case 2: Display:

1. Call the display() function to print the stack elements.

Case 3: Exit:

1. Terminate the program using exit(0).

Default Case:

Print an error message indicating an invalid choice.

Step 6: In Push Function:

Check if the stack is full (top == MAX - 1):

If true, print "Stack Overflow! Cannot push value".

If the stack is not full:

Increment top by 1.

Assign value to stack[top].

Print "value pushed onto the stack".

Step 7: In Display Function:

Check if the stack is empty (top == -1):

If true, print "Stack is empty".

If the stack is not empty:

Print "Stack elements are:".

Iterate from top to 0 and print each element in the stack.

Step 8:For Exit simply press 3.

Step 9:End.

## Implementation of algorithm:

#include <stdio.h>

#include <stdlib.h>

#define MAX 100 // Define the maximum size of the stack

int stack[MAX]; // Array to store stack elements

int top = -1; // Initialize top to -1 indicating the stack is empty

void push(int value);

void display();

int main() {

int choice, value;

while (1) {

printf("\n\n\*\*\*\*\*\*\*\*\* Stack Menu \*\*\*\*\*\*\*\*\*\n");

printf("1. Push\n");

printf("2. Display\n");

printf("3. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the value to push: ");

scanf("%d", &value);

push(value);

break;

case 2:

display();

break;

case 3:

exit(0);

default:

printf("Invalid choice! Please enter a valid choice.\n");

}

}

return 0;

}

void push(int value) {

if (top == MAX - 1) {

printf("Stack Overflow! Cannot push %d\n", value);

} else {

top++;

stack[top] = value;

printf("%d pushed onto the stack\n", value);

}

}

void display() {

if (top == -1) {

printf("Stack is empty\n");

} else {

printf("Stack elements are:\n");

for (int i = top; i >= 0; i--) {

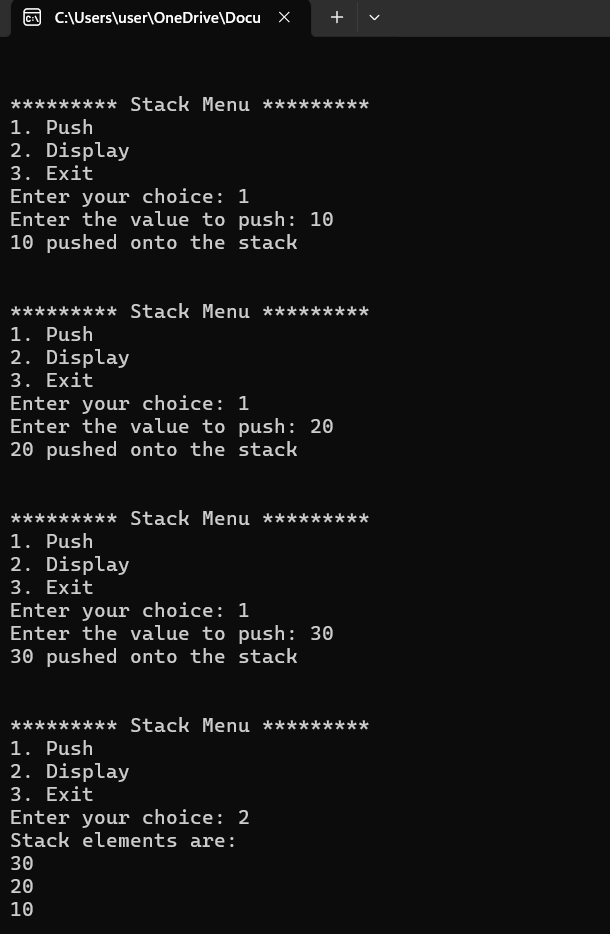
printf("%d\n", stack[i]);

}

}

}

## Output



# Lab 7: Queue Functions

# Variables:

* queue: an array to hold the queue elements.
* front: an integer to indicate the front of the queue.
* rear: an integer to indicate the rear of the queue.
* max\_size: an integer to specify the maximum size of the queue.
* item: the element to be enqueued or dequeued.

## Algorithm:

1. **Start**
2. Initialize queue as an array of size max\_size.
3. Set front to -1.
4. Set rear to -1.

**(Enqueue Operation)**

1. Check if the queue is full:

* If rear is equal to max\_size – 1(rear == max\_size – 1), the queue is full and print "Queue is full".

1. Otherwise:

* If front is -1(front == -1), set front to 0 (this is the first element being added).
* Increment rear by 1(rear = rear + 1).
* Add item to queue[rear]( queue[rear] = item).
* Print "Item enqueued".

**(Dequeue Operation)**

5. Check if the queue is empty:

* If front is -1(front == -1) or front is greater than rear(front > rear), the queue is empty and print "Queue is empty".

6. Otherwise:

* Retrieve item from queue[front]( item = queue[front]).
* Increment front by 1(front = front + 1).
* If front becomes greater than rear(front > rear), reset front(front = -1) and rear (rear = -1),(this is the last element being removed).
* Print "Item dequeued".

1. End

## Implementation of algorithm:

#include <stdio.h>

#define MAX\_SIZE 100

// Global variables

int queue[MAX\_SIZE];

int front = -1;

int rear = -1;

// Function to check if the queue is full

int isFull() {

return rear == MAX\_SIZE - 1;

}

// Function to check if the queue is empty

int isEmpty() {

return front == -1 || front > rear;

}

// Function to enqueue an item

void enqueue(int item) {

if (isFull()) {

printf("Queue is full\n");

} else {

if (front == -1) {

front = 0;

}

rear = rear + 1;

queue[rear] = item;

printf("Item enqueued: %d\n", item);

}

}

// Function to dequeue an item

void dequeue() {

if (isEmpty()) {

printf("Queue is empty\n");

} else {

int item = queue[front];

front = front + 1;

if (front > rear) {

front = -1;

rear = -1;

}

printf("Item dequeued: %d\n", item);

}

}

// Main function to demonstrate queue operations

int main() {

int choice, item;

while (1) {

printf("\nQueue Operations:\n");

printf("1. Enqueue\n");

printf("2. Dequeue\n");

printf("3. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the item to enqueue: ");

scanf("%d", &item);

enqueue(item);

break;

case 2:

dequeue();

break;

case 3:

return 0;

default:

printf("Invalid choice, please try again.\n");

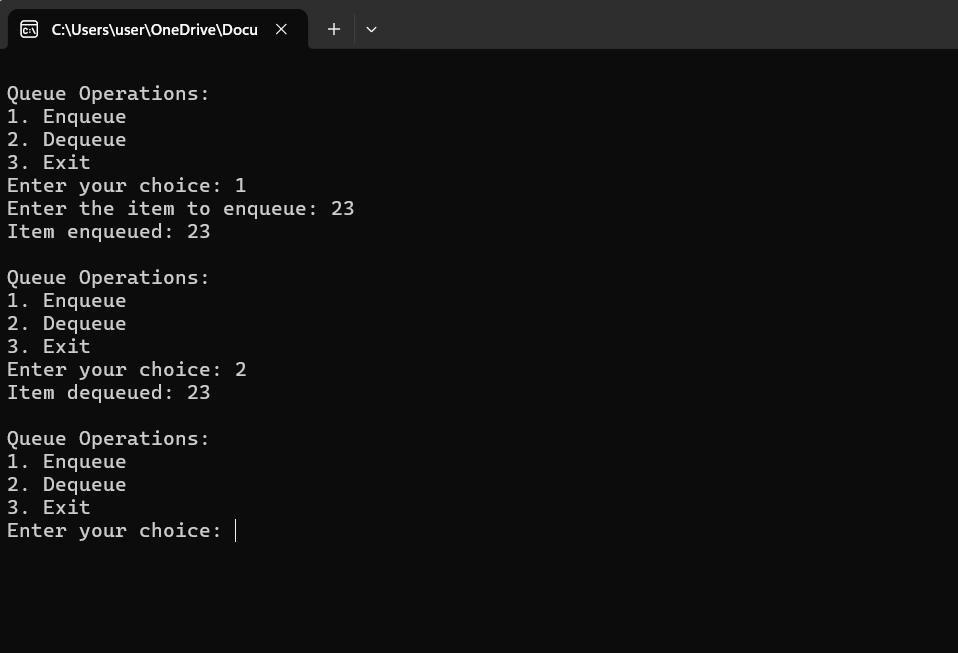
}

}

return 0;

}

## Output



# Lab 8: Find the factorial of a number using recursion.

## Algorithm:

1. **Start**
2. **Initialize**:

* n: the number for which factorial is to be calculated.

1. **Define a Recursive Function** ‘factorial(n)’:

* Input: n
* If n is 0 or 1(n == 0 or n == 1):
  + Return 1 (Base case: factorial of 0 or 1 is 1)
* Otherwise:
  + Return n \* factorial(n - 1) (Recursive case)

1. **Input**:

* Read the value of n from the user.

1. **Calculate**:

* Call the “factorial(n) “function with n and store the result in” result”.

1. **Output**:

* Print the value of “result.”

1. **End**

## Implementation of algorithm:

#include <stdio.h>

// Define a Recursive Function 'factorial(n)'

int factorial(int n) {

// If n is 0 or 1 (Base case)

if (n == 0 || n == 1) {

return 1;

} else {

// Otherwise (Recursive case)

return n \* factorial(n - 1);

}

}

int main() {

int n;

long long result;

// Input: Read the value of n from the user

printf("Enter a number: ");

scanf("%d", &n);

// Calculate: Call the 'factorial(n)' function with n and store the result in 'result'

result = factorial(n);

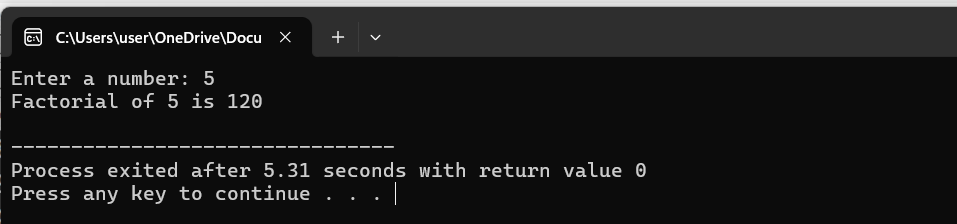
// Output: Print the value of 'result'

printf("Factorial of %d is %lld\n", n, result);

return 0;

}

## Output:



# Lab 9: Find the Fibonacci Series using recursion.

## Algorithm:

1. **Start**
2. **Initialize**:

* n: the number of terms in the Fibonacci series to be calculated.

1. **Define a Recursive Function** fibonacci(term):

* Input: term
* If term is 0 (term == 0):
  + Return 0 (Base case: Fibonacci of 0 is 0)
* If term is 1(term == 1):
  + Return 1 (Base case: Fibonacci of 1 is 1)
* Otherwise:
  + Return ‘fibonacci(term - 1) + fibonacci(term - 2) ‘(Recursive case)

1. **Input**:

* Read the value of n from the user.

1. **Calculate and Output**:

* For each term from 0 to n-1, call the fibonacci function and print the result.

1. **End**

## Implementation of algorithm:

#include <stdio.h>

// Define a Recursive Function 'fibonacci'

int fibonacci(int term) {

// Base case: Fibonacci of 0 is 0

if (term == 0) {

return 0;

}

// Base case: Fibonacci of 1 is 1

if (term == 1) {

return 1;

}

// Recursive case

return fibonacci(term - 1) + fibonacci(term - 2);

}

int main() {

int n;

// Input: Read the value of n from the user

printf("Enter the number of terms in the Fibonacci series: ");

scanf("%d", &n);

// Output the Fibonacci series up to n terms

printf("Fibonacci series up to %d terms:\n", n);

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

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

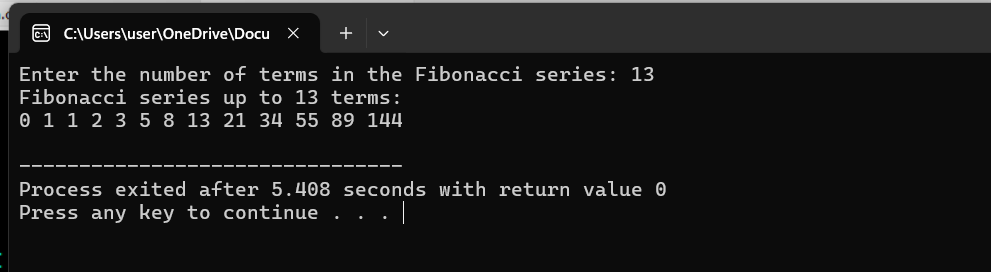
}

printf("\n");

return 0;

}

## Output:



# Lab 10: Find the GCD (Greatest Common Divisor) using recursion.

## Algorithm:

1. **Start**
2. **Initialize** a and b: the two numbers for which GCD is to be calculated.

1. **Define a Recursive Function** gcd(a, b):

* Input: a, b
* If b is 0(b == 0):
  + Return a (Base case: GCD of a and 0 is a)
* Otherwise:
  + Return gcd(b, a % b) (Recursive case: GCD of b and the remainder of a divided by b)

1. **Input**:

* Read the values of a and b from the user.

1. **Calculate**:

* Call the gcd(a, b) function with a and b and store the result in result.

1. **Output**:

* Print the value of result.

1. **End**

## Implementation of algorithm:

#include <stdio.h>

// Define a Recursive Function 'gcd'

int gcd(int a, int b) {

// Base case: if b is 0, return a

if (b == 0) {

return a;

} else {

// Recursive case

return gcd(b, a % b);

}

}

int main() {

int a, b, result;

// Input: Read the values of a and b from the user

printf("Enter the first number: ");

scanf("%d", &a);

printf("Enter the second number: ");

scanf("%d", &b);

// Calculate: Call the 'gcd(a, b)' function with a and b and store the result in 'result'

result = gcd(a, b);

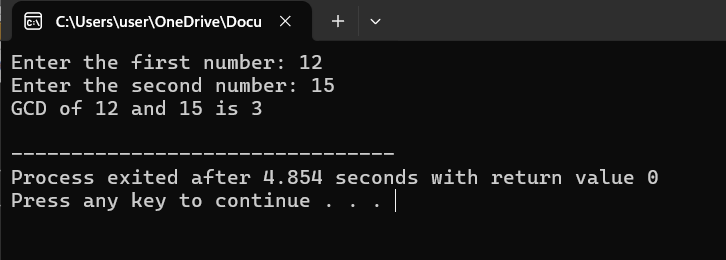
// Output: Print the value of 'result'

printf("GCD of %d and %d is %d\n", a, b, result);

return 0;

}

## Output:



# Lab 11: Tower of Hanoi using recursion.

## Algorithm:

1. **Start**
2. **Initialize**:

* n: the number of disks.
* source: the source rod.
* target: the target rod.
* auxiliary: the auxiliary rod.

1. **Define a Recursive Function** tower\_of\_hanoi(n, source, target, auxiliary):

* Input: n, source, target, auxiliary
* If n is 1(n == 1):
  + Move the disk from source to target and print the move.
* Otherwise:
  + Call tower\_of\_hanoi(n-1, source, auxiliary, target) to move n-1 disks from source to auxiliary using target as auxiliary.
  + Move the nth disk from source to target and print the move.
  + Call tower\_of\_hanoi(n-1, auxiliary, target, source) to move n-1 disks from auxiliary to target using source as auxiliary.

1. **Input**:

* Read the value of n from the user.

1. **Execute**:

* Call the tower\_of\_hanoi(n, source, target, auxiliary) function.

1. **End**

**Implementation of algorithm**:

#include <stdio.h>

// Define the Recursive Function 'tower\_of\_hanoi'

void tower\_of\_hanoi(int n, char source, char target, char auxiliary) {

// Base case: If there is only one disk, move it from source to target

if (n == 1) {

printf("Move disk 1 from rod %c to rod %c\n", source, target);

return;

}

// Recursive case

// Move n-1 disks from source to auxiliary using target as auxiliary

tower\_of\_hanoi(n - 1, source, auxiliary, target);

// Move the nth disk from source to target

printf("Move disk %d from rod %c to rod %c\n", n, source, target);

// Move n-1 disks from auxiliary to target using source as auxiliary

tower\_of\_hanoi(n - 1, auxiliary, target, source);

}

int main() {

int n;

// Input: Read the number of disks from the user

printf("Enter the number of disks: ");

scanf("%d", &n);

// Execute: Call 'tower\_of\_hanoi' function with source as 'A', target as 'C', and auxiliary as 'B'

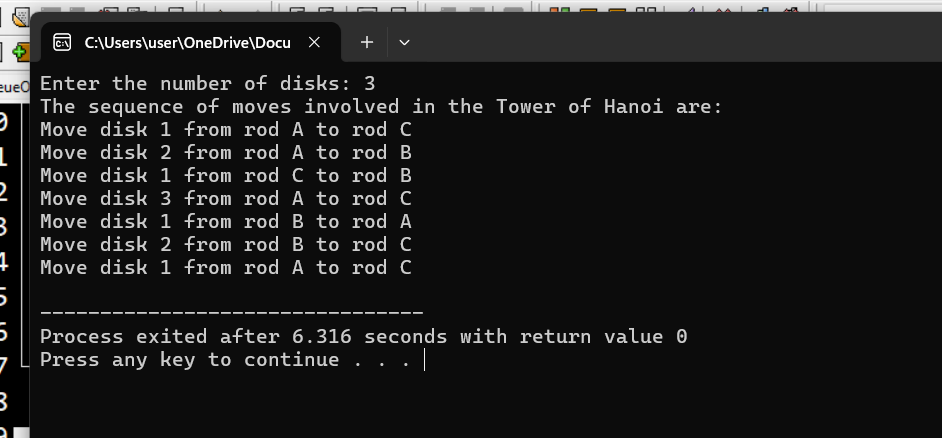
printf("The sequence of moves involved in the Tower of Hanoi are:\n");

tower\_of\_hanoi(n, 'A', 'C', 'B');

return 0;

}

## Output:



# Lab 12: Bubble Sort (Iterative Approach)

## Algorithm:

1. **Start**
2. **Initialize**:

* array: the list of elements to be sorted.
* n: the number of elements in array.

1. **Input**:

* Read the elements of array.

1. **Bubble Sort**:

* For i from 0 to n-1:
  + For j from 0 to n-i-1:
    - If array[j] is greater than array[j+1]:
      * Swap array[j] and array[j+1].

1. **Output**:

* Print the sorted array.

1. End

**Implementation of algorithm**:

#include <stdio.h>

// Function to perform Bubble Sort

void bubbleSort(int array[], int n) {

int i, j, temp;

// Iterate over each element in the array

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

// Compare adjacent elements

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

// Swap if the element is greater than the next element

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

temp = array[j];

array[j] = array[j+1];

array[j+1] = temp;

}

}

}

}

int main() {

int n, i;

// Input: Read the number of elements

printf("Enter number of elements: ");

scanf("%d", &n);

int array[n];

// Input: Read the elements of the array

printf("Enter the elements of the array: ");

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

scanf("%d", &array[i]);

}

// Perform Bubble Sort

bubbleSort(array, n);

// Output: Print the sorted array

printf("Sorted array: ");

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

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

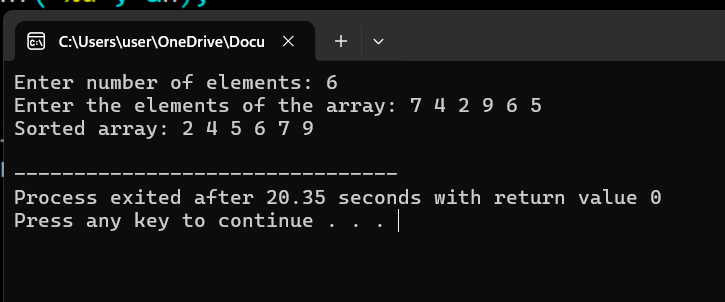
}

printf("\n");

return 0;

}

## Output:



# Lab 13: Selection Sort (Iterative Approach)

## Algorithm:

1. **Start**
2. **Initialize**:

* array: the list of elements to be sorted.
* n: the number of elements in array.

1. **Input**:

* Read the elements of array.

1. **Selection Sort**:

* For i from 0 to n-1:
  + Set min\_index to i.
  + For j from i+1 to n-1:
    - If array[j] is less than array[min\_index]:
      * Set min\_index to j.
  + If min\_index is not i:
    - Swap array[i] and array[min\_index].

1. **Output**:

* Print the sorted array.

1. **End**

## Implementation of Algorithm:

#include <stdio.h>

// Function to perform Selection Sort

void selectionSort(int array[], int n) {

int i, j, min\_index, temp;

// Iterate over each element in the array

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

// Set min\_index to the current index

min\_index = i;

// Find the minimum element in the unsorted part of the array

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

if (array[j] < array[min\_index]) {

min\_index = j;

}

}

// Swap the found minimum element with the first element

if (min\_index != i) {

temp = array[i];

array[i] = array[min\_index];

array[min\_index] = temp;

}

}

}

int main() {

int n, i;

// Input: Read the number of elements

printf("Enter number of elements: ");

scanf("%d", &n);

int array[n];

// Input: Read the elements of the array

printf("Enter the elements of the array: ");

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

scanf("%d", &array[i]);

}

// Perform Selection Sort

selectionSort(array, n);

// Output: Print the sorted array

printf("Sorted array: ");

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

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

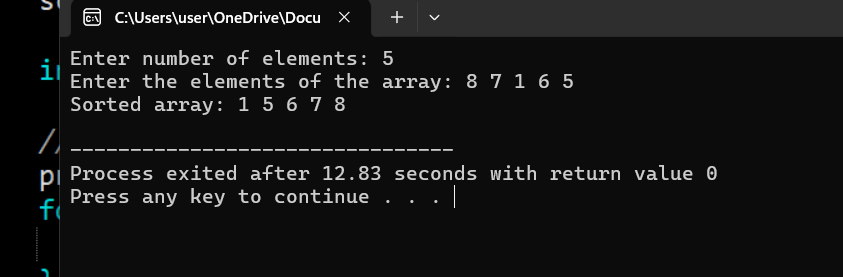
}

printf("\n");

return 0;

}

## Output



# Lab 14: Insertion Sort (Iterative Approach)

## Algorithm:

1. Start
2. Initialize:

* array: the list of elements to be sorted.
* n: the number of elements in array.

1. Input:

* Read the elements of array from the user.

1. Insertion Sort:

* For i from 1 to n-1:
  + Set key to array[i].
  + Set j to i - 1.
  + While j >= 0 and array[j] > key:
    - Set array[j + 1] to array[j].
    - Decrement j by 1.
  + Set array[j + 1] to key.

1. Output:

* Print the sorted array.

1. End

## Implementation of Algorithm:

#include <stdio.h>

// Function to perform Insertion Sort

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

int i, j, key;

// Iterate over each element in the array starting from the second element

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

// Set key to the current element

key = array[i];

// Set j to the index before the current element

j = i - 1;

// Shift elements of the sorted part of the array that are greater than the key

// to one position ahead of their current position

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

array[j + 1] = array[j];

j = j - 1;

}

// Place the key in its correct position

array[j + 1] = key;

}

}

int main() {

int n, i;

// Input: Read the number of elements

printf("Enter number of elements: ");

scanf("%d", &n);

int array[n];

// Input: Read the elements of the array

printf("Enter the elements of the array: ");

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

scanf("%d", &array[i]);

}

// Perform Insertion Sort

insertionSort(array, n);

// Output: Print the sorted array

printf("Sorted array: ");

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

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

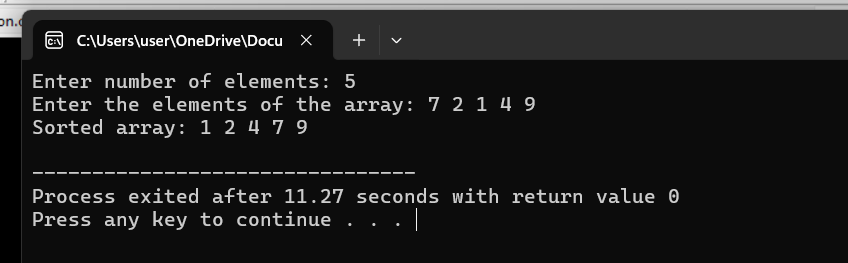
}

printf("\n");

return 0;

}

## Output



# Lab 15: Quick Sort (Divide and Conquer Approach)

## Algorithm:

1. Start
2. Initialize:

* array: the list of elements to be sorted.
* low: the starting index of the array or subarray to be sorted.
* high: the ending index of the array or subarray to be sorted.

1. **Partitioning Function** partition(array, low, high):

* Input: array, low, high
* Set pivot to array[high].
* Initialize i to low - 1.
* For j from low to high-1:
  + If array[j] is less than or equal to pivot:
    - Increment i.
    - Swap array[i] and array[j].
* Swap array[i + 1] and array[high] (place the pivot element in its correct position).
* Return i + 1 (index of the pivot element).

1. **Quick Sort Function** quickSort(array, low, high):

* Input: array, low, high
* If low is less than high:
  + Call partition(array, low, high) and store the returned index as pivotIndex.
  + Recursively call quickSort(array, low, pivotIndex - 1) (sort the left subarray).
  + Recursively call quickSort(array, pivotIndex + 1, high) (sort the right subarray).

1. **Input**:

* Read the elements of array.

1. **Execute Quick Sort**:

* Call quickSort(array, 0, n-1) where n is the number of elements in array.

1. **Output**:

* Print the sorted array.

1. End

## Implementation of Algorithm:

#include <stdio.h>

// Function to swap two elements in an array

void swap(int\* a, int\* b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// Function to partition the array and return the pivot index

int partition(int array[], int low, int high) {

int pivot = array[high]; // Pivot element (last element in the array)

int i = low - 1; // Index of smaller element

for (int j = low; j < high; j++) {

// If current element is smaller than or equal to pivot

if (array[j] <= pivot) {

i++; // Increment index of smaller element

swap(&array[i], &array[j]); // Swap array[i] and array[j]

}

}

swap(&array[i + 1], &array[high]); // Swap array[i + 1] and array[high] (pivot)

return i + 1; // Return the partition index

}

// Function to implement Quick Sort

void quickSort(int array[], int low, int high) {

if (low < high) {

// Partition the array and get the pivot index

int pivotIndex = partition(array, low, high);

// Recursively sort elements before partition and after partition

quickSort(array, low, pivotIndex - 1);

quickSort(array, pivotIndex + 1, high);

}

}

int main() {

int n;

// Input: Read the number of elements

printf("Enter number of elements: ");

scanf("%d", &n);

int array[n];

// Input: Read the elements of the array

printf("Enter the elements of the array: ");

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

scanf("%d", &array[i]);

}

// Perform Quick Sort

quickSort(array, 0, n - 1);

// Output: Print the sorted array

printf("Sorted array: ");

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

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

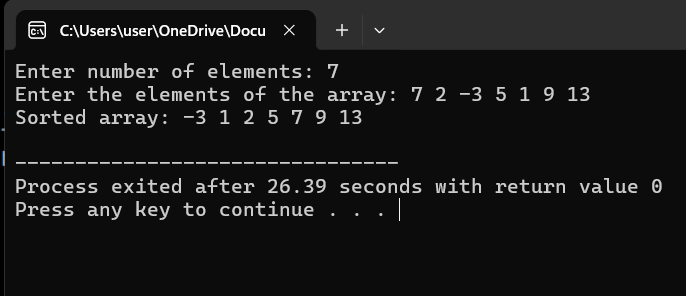
}

printf("\n");

return 0;

}

## Output



# Lab 16: Merge Sort (Divide and Conquer Approach)

## Algorithm:

1. Start
2. Initialize:

* array: the list of elements to be sorted.
* n: the number of elements in array.

1. Merge Sort Function mergeSort(array, left, right):

* Input: array, left, right
* If left < right:
  + Set mid to (left + right) / 2.
  + Recursively call mergeSort(array, left, mid) (sort the left subarray).
  + Recursively call mergeSort(array, mid + 1, right) (sort the right subarray).
  + Merge the two sorted subarrays using merge(array, left, mid, right).

1. Merge Function merge(array, left, mid, right):

* Input: array, left, mid, right
* Initialize temporary arrays L and R.
* Copy data from array to L and R based on the split mid.
* Initialize indices i, j, and k for L, R, and array respectively.
* Merge L and R back into array:
  + While i < size of L and j < size of R:
    - If L[i] <= R[j], set array[k] to L[i] and increment i and k.
    - Else, set array[k] to R[j] and increment j and k.
  + Copy the remaining elements of L and R if any.

1. **Input**:

* Read the elements of array.

1. **Execute Merge Sort**:

* Call mergeSort(array, 0, n-1) where n is the number of elements in array.

1. **Output**:

* Print the sorted array.

1. **End**

## Implementation of Algorithm:

#include <stdio.h>

// Function to merge two subarrays of array[]

// First subarray is array[left..mid]

// Second subarray is array[mid+1..right]

void merge(int array[], int left, int mid, int right) {

int i, j, k;

int n1 = mid - left + 1; // Size of left subarray

int n2 = right - mid; // Size of right subarray

// Create temporary arrays

int L[n1], R[n2];

// Copy data to temporary arrays L[] and R[]

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

L[i] = array[left + i];

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

R[j] = array[mid + 1 + j];

// Merge the temporary arrays back into array[left..right]

i = 0; // Initial index of first subarray

j = 0; // Initial index of second subarray

k = left; // Initial index of merged subarray

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

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

array[k] = L[i];

i++;

} else {

array[k] = R[j];

j++;

}

k++;

}

// Copy the remaining elements of L[], if any

while (i < n1) {

array[k] = L[i];

i++;

k++;

}

// Copy the remaining elements of R[], if any

while (j < n2) {

array[k] = R[j];

j++;

k++;

}

}

// Function to implement Merge Sort

void mergeSort(int array[], int left, int right) {

if (left < right) {

// Find the middle point

int mid = left + (right - left) / 2;

// Recursively sort first and second halves

mergeSort(array, left, mid);

mergeSort(array, mid + 1, right);

// Merge the sorted halves

merge(array, left, mid, right);

}

}

int main() {

int n;

// Input: Read the number of elements

printf("Enter number of elements: ");

scanf("%d", &n);

int array[n];

// Input: Read the elements of the array

printf("Enter the elements of the array: ");

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

scanf("%d", &array[i]);

}

// Perform Merge Sort

mergeSort(array, 0, n - 1);

// Output: Print the sorted array

printf("Sorted array: ");

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

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

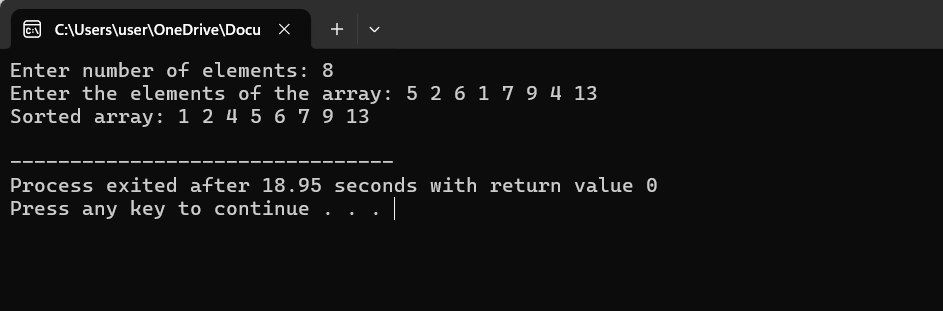
}

printf("\n");

return 0;

}

## Output



# Lab 17: Radix Sort (upto thousands)

## Algorithm:

1. **Start**
2. **Initialize**:

* array: the list of elements to be sorted.
* n: the number of elements in array.

1. **Find Maximum Element**:

* Iterate through array to find the maximum element max\_element.

1. **Perform Radix Sort**:

* For digit from 1 to 1000 (since elements are up to 4 digits):
  + Create 10 buckets (queues) for digits 0 to 9.
  + Distribute elements from array into the buckets based on the current digit:
    - Extract the digit from each element (array[i] / digit % 10).
    - Place each element into the respective bucket.
  + Collect elements back from the buckets into array.

1. **Output**:

* Print the sorted array.

1. **End**

## Implementation of Algorithm:

#include <stdio.h>

// Function to get the maximum element from the array

int getMax(int array[], int n) {

int max = array[0];

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

if (array[i] > max) {

max = array[i];

}

}

return max;

}

// Function to perform counting sort based on the digit represented by exp

void countSort(int array[], int n, int exp) {

int output[n]; // Output array

int count[10] = {0}; // Initialize count array for digits 0-9

// Store count of occurrences in count[]

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

count[(array[i] / exp) % 10]++;

}

// Change count[i] to contain the actual position of this digit in output[]

for (int i = 1; i < 10; i++) {

count[i] += count[i - 1];

}

// Build the output array

for (int i = n - 1; i >= 0; i--) {

output[count[(array[i] / exp) % 10] - 1] = array[i];

count[(array[i] / exp) % 10]--;

}

// Copy the output array to array[], so that array[] now contains sorted numbers according to the current digit

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

array[i] = output[i];

}

}

// Radix Sort function to sort array of size n

void radixSort(int array[], int n) {

// Find the maximum number to know number of digits

int max = getMax(array, n);

// Do counting sort for every digit. exp is 1, 10, 100, 1000, ...

for (int exp = 1; max / exp > 0; exp \*= 10) {

countSort(array, n, exp);

}

}

int main() {

int n;

// Input: Read the number of elements

printf("Enter number of elements: ");

scanf("%d", &n);

int array[n];

// Input: Read the elements of the array

printf("Enter the elements of the array (up to 4-digit numbers): ");

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

scanf("%d", &array[i]);

}

// Perform Radix Sort

radixSort(array, n);

// Output: Print the sorted array

printf("Sorted array: ");

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

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

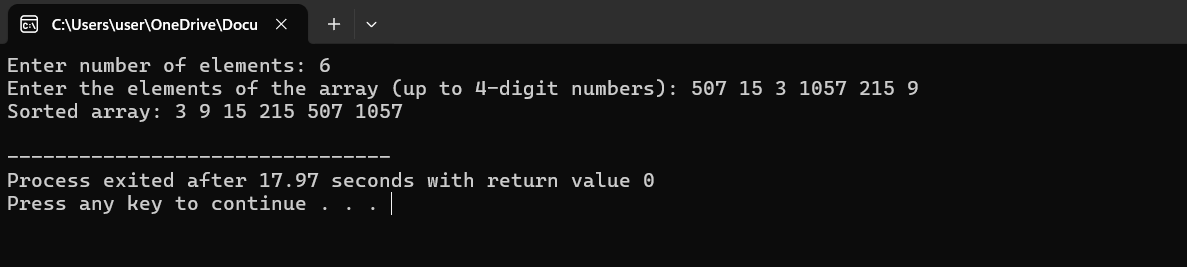
}

printf("\n");

return 0;

}

## Output:



## Lab 18: List Operations

## Algorithm:

1. **Start**
2. Display the menu with options.
3. Initialize variables: n = 0, m = 0, a[100], b[100].
4. Loop until the user chooses to exit:

* Prompt the user to enter their choice.
* Based on the user's choice, call the corresponding function:
  + **Case 1**: Call create(a, &n) to create the first list.
  + **Case 2**: Call insert(a, &n) to insert an element into the first list.
  + **Case 3**: Call delete(a, &n) to delete an element from the first list.
  + **Case 4**: Call update(a, &n) to update an element in the first list.
  + **Case 5**: Call traverse(a, &n) to traverse and display the first list.
  + **Case 6**: Call searching(a, &n) to search for an element in the first list.
  + **Case 7**: Call create(b, &m) to create the second list, then call merging(a, b, &n, &m) to merge the lists.
  + **Case 8**: Exit the loop and terminate the program

## create Function Algorithm

1. Prompt the user to enter the elements of the list.
2. Loop through the range of n to input elements into array a.

**insert Function Algorithm**

1. Prompt the user to enter the position and the element to insert.
2. Check if the position is valid (0 to n). If not, print "Invalid position" and return.
3. Shift elements from the end to the position to make space for the new element.
4. Insert the new element at the specified position.
5. Increment the size of the array n.

**delete Function Algorithm**

1. Prompt the user to enter the position of the element to delete.
2. Check if the position is valid (0 to n-1). If not, print "Invalid position" and return.
3. Shift elements from the position to the end to remove the element.
4. Decrement the size of the array n.

**update Function Algorithm**

1. Prompt the user to enter the position and the new element.
2. Check if the position is valid (0 to n-1). If not, print "Invalid position" and return.
3. Update the element at the specified position with the new element.

**traverse Function Algorithm**

1. Print the elements of the array a from 0 to n-1.

**searching Function Algorithm**

1. Prompt the user to enter the element to search for.
2. Loop through the array a from 0 to n-1 to find the target element.
3. If the element is found, print the position and return.
4. If the element is not found, print "Element not found"

**merging Function Algorithm**

1. Prompt the user to enter the elements of the second list b.
2. Loop through the range of m to input elements into array b.
3. Append elements of b to the end of array a.
4. Update the size of array a to include elements from b.
5. Print the merged array.
6. **End**

## Implementation of Algorithm:

#include <stdio.h>

#include <stdlib.h>

void create(int a[], int \*n);

void insert(int a[], int \*n);

void delete (int a[], int \*n);

void update(int a[], int \*n);

void traverse(int a[], int \*n);

void searching(int a[], int \*n);

void merging(int a[], int b[], int \*n, int \*m);

int main() {

int choice, n = 0, m, a[100], b[100];

printf("\n======= Menu for Program =======\n");

printf("\n1: Create\n2: Insert\n3: Delete\n4: Update\n5: Traverse\n6: Searching\n7: Merging\n8: Exit\n");

do {

printf("\nEnter your choice:\t");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter size of first list:\t");

scanf("%d", &n);

create(a, &n);

break;

case 2:

insert(a, &n);

break;

case 3:

delete(a, &n);

break;

case 4:

update(a, &n);

break;

case 5:

traverse(a, &n);

break;

case 6:

searching(a, &n);

break;

case 7:

printf("Enter size of second list:\t");

scanf("%d", &m);

create(b, &m);

merging(a, b, &n, &m);

break;

case 8:

exit(0);

break;

default:

printf("Invalid choice! Please choose between 1 and 8.\n");

}

} while (choice != 8);

return 0;

}

void create(int a[], int \*n) {

printf("Enter elements of the list:\n");

for (int i = 0; i < \*n; i++) {

scanf("%d", &a[i]);

}

}

void insert(int a[], int \*n) {

int pos, elem;

printf("Enter a position (0 to %d):\t", \*n);

scanf("%d", &pos);

if (pos < 0 || pos > \*n) {

printf("Invalid position.\n");

return;

}

printf("Enter the element:\t");

scanf("%d", &elem);

for (int i = \*n; i > pos; i--) {

a[i] = a[i - 1];

}

a[pos] = elem;

(\*n)++;

}

void delete(int a[], int \*n) {

int pos;

printf("Enter position (0 to %d):\t", \*n - 1);

scanf("%d", &pos);

if (pos < 0 || pos >= \*n) {

printf("Invalid position.\n");

return;

}

for (int i = pos; i < \*n - 1; i++) {

a[i] = a[i + 1];

}

(\*n)--;

}

void update(int a[], int \*n) {

int pos, elem;

printf("Enter position (0 to %d):\t", \*n - 1);

scanf("%d", &pos);

if (pos < 0 || pos >= \*n) {

printf("Invalid position.\n");

return;

}

printf("Enter the new element:\t");

scanf("%d", &elem);

a[pos] = elem;

}

void traverse(int a[], int \*n) {

printf("\nElements are:\n");

for (int i = 0; i < \*n; i++) {

printf("%d\t", a[i]);

}

printf("\n");

}

void searching(int a[], int \*n) {

int target;

printf("Enter element to search:\t");

scanf("%d", &target);

for (int i = 0; i < \*n; i++) {

if (a[i] == target) {

printf("Element found at position %d\n", i);

return;

}

}

printf("Element not found.\n");

}

void merging(int a[], int b[], int \*n, int \*m) {

for (int i = 0; i < \*m; i++) {

a[\*n + i] = b[i];

}

\*n += \*m;

printf("Merged Array:\n");

for (int i = 0; i < \*n; i++) {

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

}

printf("\n");

}

## Output:

