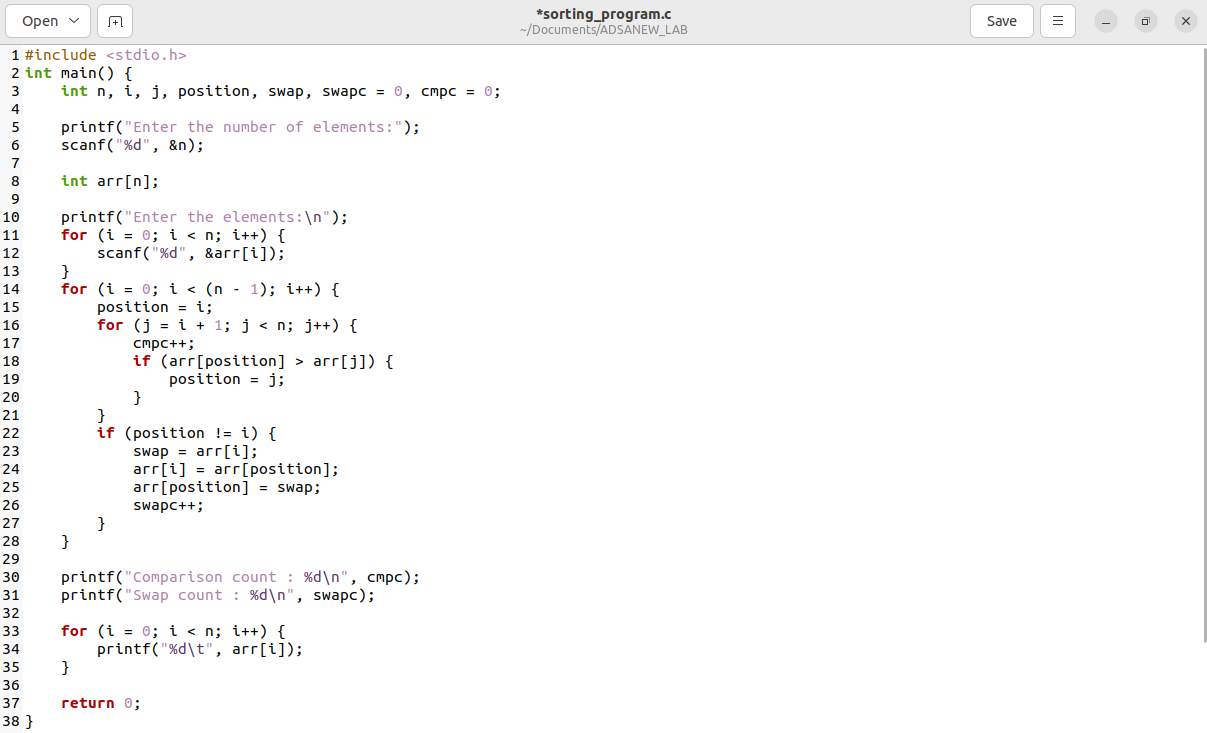
**Lab 1 :**

Write and Execute the following also analyze the time complexity

1. Sort a list of N integers using

(a) Selection sort

A screenshot of a computer screen

Description automatically generated

Analysis:

* For each position, find the smallest number in the remaining part.
* If a smaller number is found, swap it with the current position.
* Display comparison and swap counts.
* Show the sorted numbers.

Time Complexity

1. Best Case: O(n^2) - Although no swaps are needed, comparisons still occur.
2. Average Case: O(n^2) - Roughly (n^2)/2 comparisons and n swaps on average.
3. Worst Case: O(n^2) - Maximum comparisons and swaps when the array is reverse sorted.

(b) Bubble sort

A screenshot of a computer program

Description automatically generated

A computer screen with blue and white text

Description automatically generated

Analysis

* The program uses Bubble Sort to arrange input numbers.
* Bubble Sort has a slow time complexity of O(n^2) for sorting.
* It counts comparisons and swaps during sorting.
* For already sorted input, Bubble Sort is best with O(n) time.
* For reverse order input, it's worst with O(n^2) time.
* On average, it's still O(n^2) due to multiple comparisons and swaps.

Time Complexity:

Bubble Sort has an average and worst-case time complexity of O(n^2), making it inefficient for large datasets.

1. Best Case:

Best-case time complexity is O(n) when the array is already sorted. Minimal comparisons and no swaps are needed.

1. Worst Case:

Worst-case time complexity is O(n^2) when the array is in reverse order, leading to maximum comparisons and swaps.

1. Average Case:

Average-case time complexity is also O(n^2), as Bubble Sort typically performs a similar number of comparisons and swaps as the worst case.

2. Binary search technique over a list of integers.

A screenshot of a computer program

Description automatically generated

A computer screen shot of a computer screen

Description automatically generated

Analysis

* Binary search algorithm is applied to locate a target value in an array.
* Recursion is used to repeatedly narrow down the search range.
* Comparison occurs between the target and the middle element of the range.
* Once found or exhausted, it reports if the target is present.
* The program displays the number of comparisons made during the search.
* Overall, the code efficiently finds a target in an array and reports comparisons.

Time Complexity:

Binary search takes a time proportional to the logarithm of the number of elements in the array.

Written as O(log n), where 'n' is the number of elements.

1. Best Case:

When the target is right in the middle, you find it in just one step.

The best case time is constant and is O(1).

1. Worst Case:

If the target is at an end or absent, you keep splitting the array until there's only one element left.

The worst-case time is still pretty good, it's O(log n).

1. Average Case:

On average, binary search takes about log₂(n) steps to find the target.

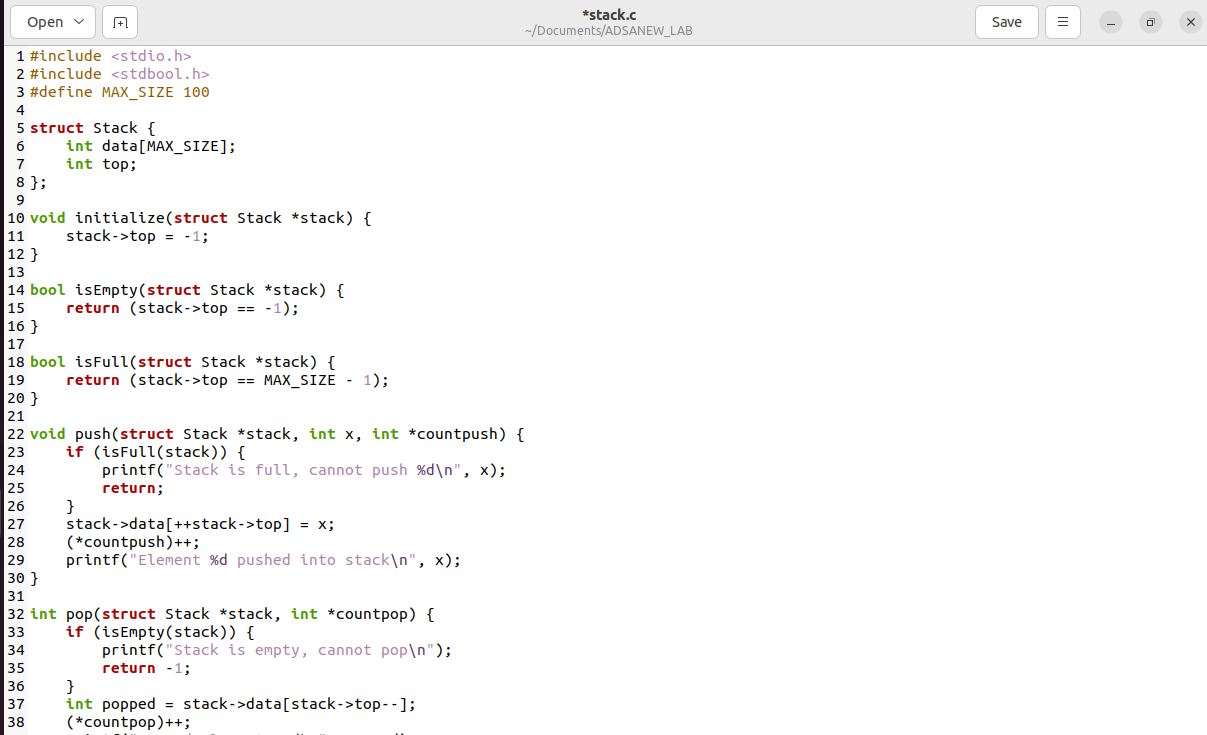
The average case time is also O(log n).

3. Stack operations

(i) Stack-empty(S)

(ii) Push(S, x)

(iii) Pop(S)



A screenshot of a computer

Description automatically generated

A close-up of a white background

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

Analysis

* Stack is implemented using an array.
* Struct named Stack holds array and top index.
* Initialization sets top to -1.
* isEmpty checks if top is -1.
* push adds element if not full.
* pop removes element if not empty.
* User interface in main function.
* Tracks push and pop counts.
* Push and pop counters passed as pointers.

Time complexity

1. Push Operation:

* Best Case: O(1)

When the stack is not full, the push operation involves a constant time array assignment and incrementing the top pointer.

* Worst Case: O(1)

Even in the worst case (stack being just one element away from full), the push operation is still a constant time operation.

* Average Case: O(1)

The average time complexity remains constant since each push operation takes a constant amount of time.

1. Pop Operation:

* Best Case: O(1)

When the stack is not empty, the pop operation involves a constant time array access and decrementing the top pointer.

* Worst Case: O(1)

Similar to the push operation, the worst case time complexity for popping is also constant.

* Average Case: O(1)

The average time complexity remains constant because each pop operation takes constant time regardless of the stack's size.

1. isEmpty Operation:

* Best Case: O(1)

Checking if the stack is empty involves a single comparison (constant time) of the top pointer.

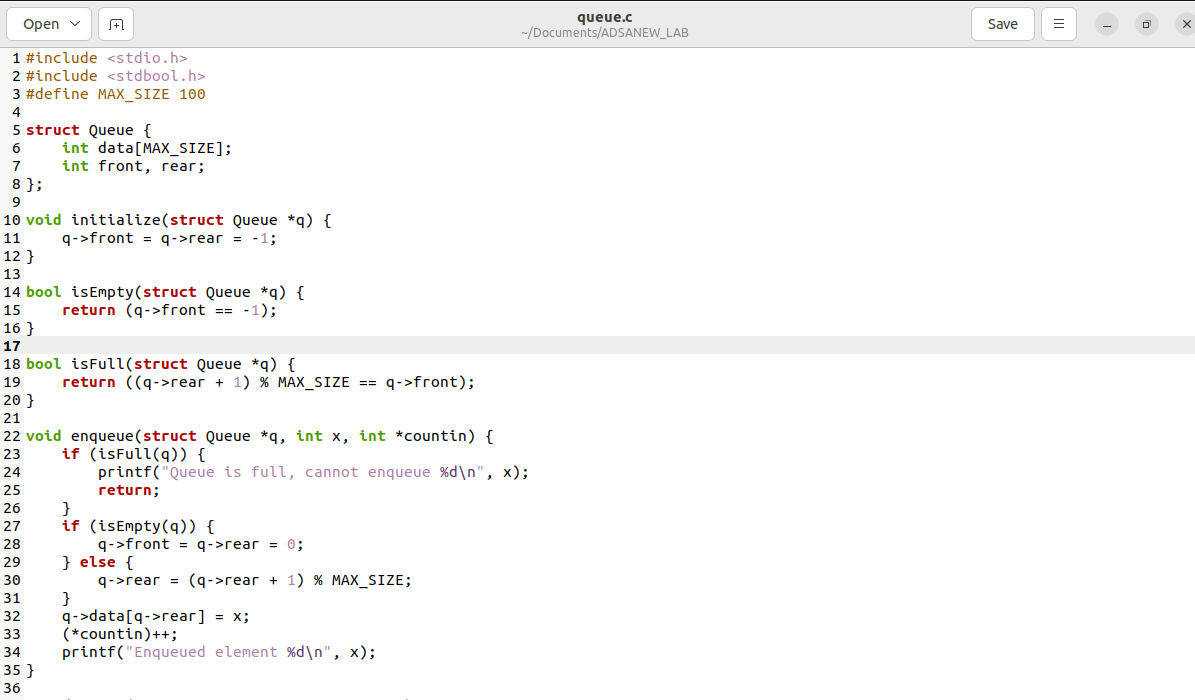
* Worst Case: O(1)

Similarly, the worst and average cases involve a single constant time comparison.

4. Queue Q also analyze the time complexity.

(i) Enqueue(Q, x)

(ii) Dequeue(Q)



A computer screen shot of a code

Description automatically generatedA white background with black text

Description automatically generated

A screenshot of a computer

Description automatically generated

Analysis

(i) Enqueue operation:

The enqueue operation involves inserting an element at the rear of the queue. Here's how the time complexity breaks down:

* Checking if the queue is full (isFull): O(1)
* Inserting the element into the queue: O(1)
* Updating the rear pointer: O(1)
* Incrementing the countin variable: O(1)
* Printing a message: O(1)
* Overall, the enqueue operation has a constant time complexity of O(1).

1. Enqueue operation:

Best/Average/Worst-case: Always O(1)

Enqueueing is fast and constant-time because it involves simple array updates and checks.

(ii) Dequeue operation:

The dequeue operation involves removing an element from the front of the queue. Here's how the time complexity breaks down:

* Checking if the queue is empty (isEmpty): O(1)
* Retrieving the element from the queue: O(1)
* Updating the front pointer: O(1)
* Incrementing the count out variable: O(1)
* Printing a message: O(1)
* Overall, Dequeue operation also has a constant time complexity of O(1).

1. Dequeue operation:

Best/Average/Worst-case: Always O(1)

Dequeueing is also fast and constant-time due to straightforward array updates and checks.