STQA Experiment No.9

Name: Abhijit Palve   **Roll No**:22101A0035   **Branch:**INFT-A

Code:

```
#include <iostream>
using namespace std;

// Function to perform binary search
int binarySearch(int arr[], int size, int target) {
    int low = 0, high = size - 1;

    while (low <= high) {
        int mid = low + (high - low) / 2; // To avoid overflow

        // Check if the target is at mid
        if (arr[mid] == target)
            return mid;

        // If target is greater, ignore the left half
        if (arr[mid] < target)
            low = mid + 1;

        // If target is smaller, ignore the right half
        else
            high = mid - 1;
    }

    // Target not found
    return -1;
}

int main() {
    int size;

    // Input the size of the array
    cout << "Enter the number of elements in the array: ";
    cin >> size;

    int arr[size];

    // Input the elements of the array
    cout << "Enter the elements of the sorted array: "<<endl;
```

```
    for (int i = 0; i < size; i++) {
        cin >> arr[i];
    }

    int target;

    // Input the target element
    cout << "Enter the target element to search: ";
    cin >> target;

    // Perform binary search
    int result = binarySearch(arr, size, target);
    if (result != -1)
        cout << "Element found at index: " << result << endl;
    else
        cout << "Element not found." << endl;

    return 0;
}
```

Node 1: Start of the `main()` function.

- The program begins by asking the user to input the size of the array and the elements of the array.

Node 2: Input the elements of the array.

- The user enters the sorted array elements.

Node 3: Input the target element to search.

- The user provides the target element that they want to search for.

Node 4: Call the `binarySearch()` function.

- The `main()` function calls the `binarySearch()` function, passing the array, its size, and the target element as arguments.

Node 5: Inside `binarySearch()`, check the condition `low <= high`.

- The binary search loop starts with the initial values of `low` (0) and `high` (size - 1).
- **True**: If `low <= high`, the algorithm continues to the next steps (since there is still a search space).

Node 6: Calculate the `mid` index using the formula `mid = low + (high - low) / 2`.

- The midpoint of the current search space is calculated to divide the array into two halves.

Node 7: Check if the target is at the `mid` index (`arr[mid] == target`).

- **False**: The target element is not at the `mid` index.

Node 8: Check if the target is greater than the middle element (`arr[mid] < target`).

- **True**: The target is larger than `arr[mid]`, so the algorithm will search the right half of the array by adjusting `low = mid + 1`.

Node 9: Update the value of `low` to `mid + 1`.

- The search now continues in the right half of the array (where `low` has been increased).

Node 5: Repeat the loop and check the condition `low <= high`.

- The algorithm repeats the binary search with the updated `low` and `high`.
- The search continues until eventually, `low > high`, meaning that the search space has been exhausted and the target element is not in the array.

Node 12: Return `-1` (target not found).

- Since the target was not found within the array, the function returns `-1`, indicating that the search was unsuccessful.

1. Independent Paths:

The independent paths in this binary search program are:

1. **Path 1**: Target is found at `mid` in the first iteration.
    - Path: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 11.
2. **Path 2**: Target is not found, and we update `high` (left half traversal).
    - Path: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 10 → 5 → 12.
3. **Path 3**: Target is not found, and we update `low` (right half traversal).
    - Path: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 5 → 12.

2. Test Cases for Basis path testing :

Test Case 1: Target found at mid in the first iteration

Input	Expected Output	Executed pah

Size: 5
Array: `{2, 3, 4, 10, 40}`
Target: 4
	`Enter the number of elements in the array: 5`
	`Enter the elements of the sorted array: 2 3 4 10`
	`Enter the target element to search: 4`
	`Element found at index: 2`
		1 → 2 → 3 → 4 → 5 → 6 → 7 → 11

Test Case 2: Target not found, left half traversal

Input	Expected Output	Executed pah

Size: 5
Array: `{2, 3, 4, 10, 40}`
Target: 3
	`Enter the number of elements in the array: 5`
	`Enter the elements of the sorted array: 2 3 4 10 40`
	`Enter the target element to search: 3`
	`Element found at index: 1`
		1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 10 → 5 → 12

Test Case 3: Target not found, right half traversal

Input	Expected Output	Executed pah

Size: 5 Array: {2, 3, 4, 10, 40} Target: 40	Enter the number of elements in the array: 5 Enter the elements of the sorted array: 2 3 4 10 40 Enter the target element to search: 40 Element found at index: 4	1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 5 → 12

Test Case 4: Element not found

Input	Expected Output	Executed pah
Size: 5 Array: {2, 3, 4, 10, 40} Target: 5	Enter the number of elements in the array: 5 Enter the elements of the sorted array: 2 3 4 10 40 Enter the target element to search: 5 Element not found.	Element not found

Test Case 5: When the array is empty (size = 0)

Input	Expected Output	Executed pah
Size: 0 Array: {} target = 5	Enter the number of elements in the array: 0 Enter the elements of the sorted array: Enter the target element to search: 5 Element not found.	Element not found