

Seatwork 6.1	
Linear and Binary Search	
<b>Course Code:</b> CPE010	<b>Program:</b> Computer Engineering
<b>Course Title:</b> Data Structures and Algorithms	<b>Date Performed:</b> 09 - 11 - 25
<b>Section:</b> CPE21S4	<b>Date Submitted:</b> 09 - 11 - 25
<b>Name(s):</b> Santiago, David Owen A.	<b>Instructor:</b> Engr. Jimlord Quejado
<b>6. Output</b>	
<p><b>1. What is a search tree in data structures?</b></p> <p>A search tree is a data structure in which each node contains a unique key and satisfies a specific ordering property wherein all nodes in the left subtree of a node contain less than the node's value and all the nodes in the right subtree contain values greater than the node's value. To visualize, a single node would branch toward two sub nodes; the left one will contain a value less than the original node's value while the right one would contain a value greater.</p>	
<p><b>2. What are the different types of search algorithms in data structures? Differentiate each type of search.</b></p> <p>There are two types of search algorithms in data structures—linear and binary. Linear searching involves traversing through each element in a data set until it matches with the specific element you're looking for. These are best used for unsorted arrays to ensure that we don't overlook the element that we're finding. On the other hand, binary searching involves retrieving the middle value of a data set then checking if it matches the data we're finding. If it doesn't match, then it compares which is larger. If the middle value is less than our target, then we discard all of the values lower than it or at the left and focus on the right. Otherwise, we discard all values higher than it or at the right then focus on the left. This process continues until the middle value matches with our target which means that we've found the target value in the data set. Binary searching is best used in sorted arrays since elements are arranged in ascending or descending order.</p>	
<p><b>3. What operations / implementations can be performed using binary and linear search operations?</b></p> <p>We can implement binary and linear searching using arrays and linked lists. Linear searching is best used when the list or array is unsorted which ensures that we can verify whether or not our target is inside. Meanwhile, binary searching is much faster but requires the list or array to be sorted. This is because binary searching involves comparing values and discarding values lower or higher than the</p>	

middle value. Both of them can be used for searching the value or, in the case of arrays, the index where it can be found.

**4. What are the advantages in using binary search trees as data structure?**

The advantage of using binary search trees is efficiency and a faster time complexity than linear searching algorithms. To be specific, linear searching has a time complexity of  $O(n)$  because it has to iterate through each element, however, binary searching has a time complexity of  $O(\log n)$  which significantly reduces the searching time. Therefore, binary searching is generally faster and more efficient unless we're working with unsorted arrays where the elements aren't arranged in order.

5. Give an example program using binary search and linear search.

```
1 #include <iostream>
2
3 void linearSearchArr();
4 void binarySearchArr();
5 int binSearch(int arr[], int n, int target);
6
7 int main(){
8
9     std::cout << "Linear Search \n";
10    linearSearchArr();
11    std::cout << '\n';
12
13    std::cout << "Binary Search \n";
14    binarySearchArr();
15    std::cout << '\n';
16
17    return 0;
18}
19
20 void linearSearchArr(){
21     int arrSize, dataFind, x;
22
23     std::cout << "Enter array size: ";
24     std::cin >> arrSize; std::cout << '\n';
25
26     int arr[arrSize];
27     for (int j = 0; j < arrSize; j++){
28         std::cout << "Enter element " << j + 1 << ": ";
29         std::cin >> x;
30         arr[j] = x;
31         std::cout << '\n';
32     }
33
34     std::cout << "Enter data to be searched: ";
35     std::cin >> dataFind;
```

```
36    for (int i = 0; i < arrSize; i++){
37        if(arr[i] == dataFind){
38            std::cout << "Data found at index: " << i << '\n';
39            return;
40        }
41    }
42    std::cout << "Data: (" << dataFind << ") wasn't found in the array. \n";
43}
44
45 void binarySearchArr(){
46     int arrSize, dataFind, arr[arrSize], x, f;
47
48     std::cout << "Enter array size: ";
49     std::cin >> arrSize; std::cout << '\n';
50
51 for (int j = 0; j < arrSize; j++){
52     std::cout << "Enter element " << j + 1 << ": ";
53     std::cin >> x;
54     arr[j] = x;
55     std::cout << '\n';
56 }
57
58 std::cout << "Enter data to be searched: ";
59 std::cin >> dataFind;
60
61 f = binSearch(arr, arrSize, dataFind);
62
63 if (f == -1){
64     std::cout << "Element is not found. \n";
65 }
66 else{
67     std::cout << "Element found at index: " << f;
68 }
69 }
70
71 int binSearch(int arr[], int n, int target) {
72     int left = 0, right = n - 1;
73     while (left <= right) {
74         int mid = left + (right - left) / 2;
75         if (arr[mid] == target)
76             return mid;
77         else if (arr[mid] < target)
78             left = mid + 1;
79         else
80             right = mid - 1;
81     }
82     return -1;
83 }
```

**Output:**

```
Linear Search
Enter array size: 4

Enter element 1: 1

Enter element 2: 2

Enter element 3: 3

Enter element 4: 4

Enter data to be searched: 3
Data found at index: 2

Binary Search
Enter array size: 4

Enter element 1: 1

Enter element 2: 2

Enter element 3: 3

Enter element 4: 4

Enter data to be searched: 3
Element found at index: 2

-----
Process exited after 11.59 seconds with return value 0
Press any key to continue . . . |
```

```
Linear Search
Enter array size: 4

Enter element 1: 1

Enter element 2: 2

Enter element 3: 3

Enter element 4: 4

Enter data to be searched: 9
Data: (9) wasn't found in the array.
```

```
Binary Search
Enter array size: 4

Enter element 1: 1

Enter element 2: 2

Enter element 3: 3

Enter element 4: 4

Enter data to be searched: 9
Element is not found.
```

---

```
Process exited after 11.01 seconds with return value 0
Press any key to continue . . . |
```

## Analysis:

### Linear Search

```
20 void linearSearchArr(){
21     int arrSize, dataFind, x;
22
23     std::cout << "Enter array size: ";
24     std::cin >> arrSize; std::cout << '\n';
25
26     int arr[arrSize];
27     for (int j = 0; j < arrSize; j++){
28         std::cout << "Enter element " << j + 1 << ": ";
29         std::cin >> x;
30         arr[j] = x;
31         std::cout << '\n';
32     }
33
34     std::cout << "Enter data to be searched: ";
35     std::cin >> dataFind;
36     for (int i = 0; i < arrSize; i++){
37         if(arr[i] == dataFind){
38             std::cout << "Data found at index: " << i << '\n';
39             return;
40         }
41     }
42     std::cout << "Data: (" << dataFind << ") wasn't found in the array. \n";
43 }
44 }
```

The function iterates through each element of the array to check if it matches with the target element. Once it matches, the program prompts a message confirming the element is found, otherwise it prompts a “not found” message.

### Binary Search

```
int binSearch(int arr[], int n, int target) {
    int left = 0, right = n - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target)
            return mid;
        else if (arr[mid] < target)
            left = mid + 1;
        else
            right = mid - 1;
    }
    return -1;
}
```

This function takes three parameters—the array, the size of the array, and the target element. It then finds the middle value by dividing the array into two and getting the index of the middle. If it matches, then the function returns the index of the target element, otherwise it keeps looping. If the middle value is less than the target element, then all elements toward the left are discarded and a new middle value is found through the loop.

Otherwise, all elements toward the right are discarded and the new middle value will be searched in the left half of the original array. If the middle never matches the target element, then the function will return -1 indicating that the target element isn't in the array.

```
45 □ void binarySearchArr(){
46     int arrSize, dataFind, arr[arrSize], x, f;
47
48     std::cout << "Enter array size: ";
49     std::cin >> arrSize; std::cout << '\n';
50
51 □     for (int j = 0; j < arrSize; j++){
52         std::cout << "Enter element " << j + 1 << ": ";
53         std::cin >> x;
54         arr[j] = x;
55         std::cout << '\n';
56     }
57
58     std::cout << "Enter data to be searched: ";
59     std::cin >> dataFind;
60
61     f = binSearch(arr, arrSize, dataFind);
62
63 □     if (f == -1){
64         std::cout << "Element is not found. \n";
65     }
66 □     else{
67         std::cout << "Element found at index: " << f;
68     }
69 }
```

I created another function that initializes the array creation. Within the function, I created the f variable that would store the returned value of the binSearch function. If it is equal to -1, then it prints out a message saying the element isn't found in the array. Otherwise, it will print a message that shows the element is found at the index f.

## **References:**

*Binary Search (With Code).* (n.d.). <https://www.programiz.com/dsa/binary-search>

*Binary Search in C++.* (n.d.). <https://www.tutorialspoint.com/binary-search-in-cplusplus>

GeeksforGeeks. (2024, September 4). *Binary search on singly linked list.* GeeksforGeeks.

<https://www.geeksforgeeks.org/dsa/binary-search-on-singly-linked-list/>

GeeksforGeeks. (2025, July 23). *Searching algorithms.* GeeksforGeeks.

<https://www.geeksforgeeks.org/dsa/searching-algorithms/>

GeeksforGeeks. (2025b, September 4). *Binary search tree.* GeeksforGeeks.

<https://www.geeksforgeeks.org/dsa/binary-search-tree-data-structure/>

## **7. Supplementary Activity**

## **8. Conclusion**

This activity introduced the different searching algorithms used in programming. These two are linear and binary searching. The former involves iterating through each element of the data set; which is effective in searching but has a high time complexity of  $O(n)$ . Hence, it is best used for unsorted data sets where we cannot use the alternative binary searching since we risk skipping our target element. Meanwhile, the latter involves taking the middle value of the data set and comparing it to the target element. If it matches, then it means we found the target at that index. Otherwise, it will compare the middle value if it is less than or greater than the target value. If it is less, then all of the elements toward the right are discarded and a new middle value is found at the left side or all elements less than the first middle value. However, if the target value is greater than the middle value, then all elements toward the left are discarded and a new middle value is found at the right side or where all the elements are greater than the first middle value. This process is repeated until the target is found. Since this searching algorithm involves comparing values, it is best used in sorted data sets to ensure we don't accidentally skip the value we're looking for. In conclusion, both algorithms are useful and effective at analyzing data sets with their own advantages and disadvantages. Knowing how to implement and use both depending on the data at hand is crucial in creating an effective and efficient searching system.

## **9. Assessment Rubric**