ACTIVITY NO. 6

SEARCHING TECHNIQUES	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:**
Section:	**Date Submitted:**
Name:	**Instructor:** Engr. Roman M. Richard

1. Objective(s)

Create C++ code for implementing sequential and binary search techniques on data.

2. Intended Learning Outcomes (ILOs)

After this activity, the student should be able to:

- Create C++ code for searching data using different techniques
- Solve different problems utilizing appropriate searching techniques in C++

3. Discussion

PART A: Introduction to Searching Algorithms

Searching is the process of finding the position an element within a given list. It is considered a success if the element is within the list, and a failure if otherwise. The main idea with searching can be summarized by the following:

Given: a collection of elements, $A = (a_1, a_2, \ldots, a_n)$ and a key element e_k.
Output: The element a_i in A that matches e_k.

There are generally many variations to searching such as:

- Find the first such element
- Find the last such element
- Find the index of the element
- Key versus "equality" based
- Find all such elements
- Find extremal elements(s)
- How to handle failed searches (such that when the object does not exist)

However, there are two types of searching techniques:

- Linear Search
- Binary Search

Part B: Linear Search

The linear search is the simplest method of searching. The element must be found as the list is sequentially searched. It can be used to search for elements in both sorted and unsorted lists.

Searching in Sorted and Unsorted lists:

- For sorted lists starts from the 0th element until the element is found or until it reaches an element that has a greater value than the one being searched for.
- For unsorted lists starts from the 0th element and continues until the element is found or when it has reached the end of the list.

Algorithm (Pseudocode)

```
N -> Boundary of the list
Item -> Searching number
Data -> Linear array
Step 1: I := 0
Step 2: Repeat while I <= n
  If (item = data[i])
  Print "Searching is successful"
  Exit
Else
  Print "Searching is Unsuccessful"
Exit
```

Analysis

Our analysis focuses on the count of times a comparison must be made before we're able to find a match to confirm the presence of the sought-after value.

- Best Case Complexity: When the key value is at the first position, therefor $T(N) = O(1)$
- Worst Case Complexity: $T(N) = O(N)$
- Average Case Complexity: $(Best + Worst)/2 \ = (1 + N)/2$ or $T(N) = O(N)$

Example:

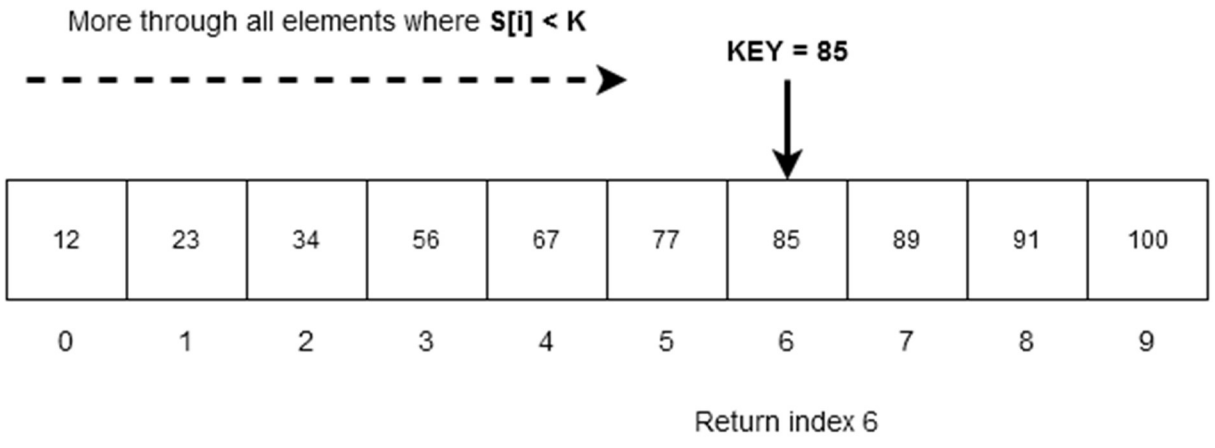

Image source: notesformsc.org

Part C: Binary Search

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise, narrow it to the upper half. Repeatedly check until the value is found, or the interval is empty. You cannot use the binary search on an unsorted list of elements.

Algorithm (Pseudocode)

```
Step 1: low :=0, up=n-1
Step 2: Repeat while low <= up
    mid = int(low+up)/2
  if(no=arr[mid])
    print "Search element is found!"
    exit
    else
  if(no < arr[mid]) then
     up =  mid-1
else
     low = mid+1
Step 3: Print "Search element is not found"
Step 4: Exit
```

Analysis

Our analysis focuses on the count of times a comparison must be made before we're able to find a match to confirm the presence of the sought-after value.

- Best Case Complexity: $T(N) = O(1)$
- Worst Case Complexity: $T(N) = O(logN)$

Example:

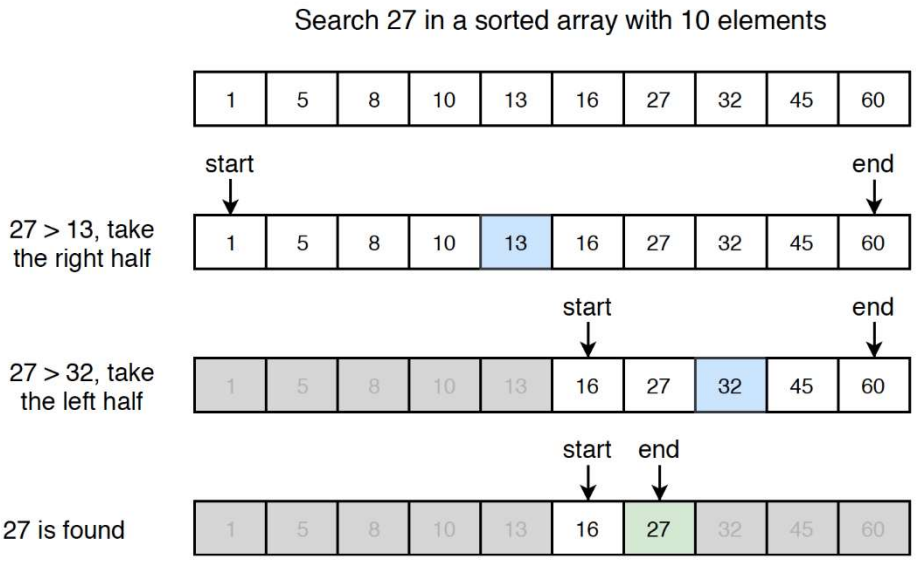

Image source: jojozhuang.github.io

4. Materials and Equipment

Personal Computer with C++ IDE
Recommended IDE:

- CLion (must use TIP email to download)
- DevC++ (use the embarcadero fork or configure to C++17)

5. Procedure

ILO A: Create C++ code for searching data using different techniques

We will implement our searching technique on a data filled with random values. To begin, we want the following directives:

```
#include <iostream>
#include <cstdlib> //for generating random integers
#include <time.h> //will be used for our seeding function
```

Then, we will define the capacity of data elements in our dataset.

```
const int max_size = 50;
```

This will be used to define your dataset. Your driver code must have the following:

```
//generate random values
int dataset[max_size];
srand(time(0));
for(int i = 0; i < max_size; i++){
    dataset[i] = rand();
}
//show your datasets content
for(int i = 0; i < max_size; i++){
    std::cout << dataset[i] << " ";
}
```

Once you are sure that you can generate random integers for your dataset, you can delete the second iterative loop and move on with the creation of the different searching techniques. **Verify this by including a screenshot of the generated data upon successful implementation of the code above in table 6-1.**

This implementation will also require the use of linked lists, so create a file “nodes.h” in your C++ project which will contain the definition of our single linked list node:

```
template <typename T>
class Node{
public:
    T data;
    Node *next;
};
```

Followed by a function that will provide us with the address of an object that points to null.

```
template <typename T>
Node<T> *new_node(T newData){
    Node<T> *newNode = new Node<T>;
    newNode->data = newData;
    newNode->next = NULL;
    return newNode;
}
```

Once this has been made, you are free to proceed to the next parts.

A.1 Sequential Search

Create a separate file implementation called “searching.h” which we will use for the implementation of the algorithms below.

```
LINEAR SEARCH ALGORITHM
N -> Boundary of the list
Item -> Searching number
Data -> Linear array
Step 1: I := 0
Step 2: Repeat while I <= n
  If (item = data[i])
  Print "Searching is successful"
  Exit
Else
  Print "Searching is Unsuccessful"
Exit
```

Create a function called linearSearch that accepts necessary parameters to execute the algorithm described above. Indicate your code, output, and observations in table 6-2a.

Create a linked list that has your first name per node’s data.

```
//create linked list for linear search
Node<char> *name1 = new_node('R');
Node<char> *name2 = new_node('o');
Node<char> *name3 = new_node('m');
Node<char> *name4 = new_node('a');
Node<char> *name5 = new_node('n');
```

Then, link each node to each other.

```
//linked list
name1->next = name2;
name2->next = name3;
name3->next = name4;
name4->next = name5;
name5->next = NULL;
```

Create a sequential search algorithm for linked list by modifying the earlier function called linearLS. This function must take the head node of your linked list and a variable dataFind that it must search for within the given list.

```
        //linear search
        linearLS(name1, 'n');
```

The sequential search algorithm does not change even if it operates on a linked list. **Show your modified code from earlier so that it will operate on a linked list. Screenshot your code and output with corresponding observations/notes in table6-2b.**

B.1 Binary Search

This algorithm must also be included in your "searching.h" file.

```
BINARY SEARCH ALGORITHM
Step 1: low :=0, up=n-1
Step 2: Repeat while low <= up
    mid = int(low+up)/2
  if(no=arr[mid])
    print "Search element is found!"
    exit
    else
  if(no < arr[mid]) then
     up =  mid-1
else
     low = mid+1
Step 3: Print "Search element is not found"
Step 4: Exit
```

Create a function called binarySearch that accepts necessary parameters to execute the algorithm described above. Indicate your code, output, and observations in table 6-3a.

Next, just like in the linked list with linear search, we will perform binary search in a linked list. However, there are pre-requisites for this to work: the linked list must have ordered values and we must know how many nodes exist in your list. For our testing, we can create our linked list with the following code:

```
//create a linked list for binary search
//make sure the values are ordered
char choice = 'y'; int count = 1; int newData;
Node<int> *temp, *head, *node;

while(choice=='y'){
    std::cout << "Enter data: ";
    std::cin >> newData;

    if(count==1){
        head = new_node(newData);
        std::cout << "Successfully added " << head->data << " to the list.\n";
        count++;
    } else if(count==2) {
        node = new_node(newData);
        head->next = node;
        node->next = NULL;
        std::cout << "Successfully added " << node->data << " to the list.\n";
```

```
            count++;
        } else {
            temp = head;
            while(true){
                if(temp->next == NULL) break;
                temp = temp->next;
            }
            node = new_node(newData);
            temp->next = node;
            std::cout << "Successfully added " << node->data << " to the list.\n";
            count++;
        }

        //when to end
        std::cout << "Continue? (y/n)";
        std::cin >> choice;
        if(choice=='n') break;
    }
```

An optional snippet of code can be added here to verify that our data is actually stored onto a linked list as intended.

```
    //display
    Node<int> *currNode;
    currNode = head;
    while(currNode!=NULL){
        std::cout << currNode->data << " ";
        currNode = currNode->next;
    }
```

Binary search still works the same, however. We need to create a function that will enable us to find the middle node. This function, getMiddle, must take the first and last node as its parameters. It follows the algorithm described below:

1. Traverse the singly linked list using two pointers.
2. Move one pointer by one step ahead and the other pointer by two steps.
3. When the fast pointer reaches the end of the singly linked list, the slow pointer will reach the middle of the singly linked list.
4. Return slow pointer address.

We will use this function to find the middle node of our linked list to properly search through the sorted list. Next, another function to implement the binary search. Step 2 of this algorithm uses the function from the algorithm above.

1. Start node is set to head of the list and the last node is set to NULL.
2. Middle element is calculated using the two pointers approach discussed above.
3. If the middle element is same as the key to be searched, we return it.
4. Else if middle element is greater than the key to be searched, we have to search is the right side of the singly linked list. So, we set start pointer to the next of middle element.
5. Else if middle element is less than the key to be searched, we have to search is the left side of the singly linked list. So, we set last pointer to the middle element.
6. If the key is found or the entire linked list gets traversed, we break the loop.

Create the C++ code for this binary search algorithm and provide the output in the three cases shown in table 6-3b. Screenshot and observation are required.

6. Output

Screenshot	
Observations	

Table 6-1. Data Generated and Observations.

Code	
Output	
Observations	

Table 6-2a. Linear Search for Arrays

Code	
Output	
Observations	

Table 6-2b. Linear Search for Linked List

Code	
Output	
Observations	

Table 6-3a. Binary Search for Arrays

Code	
Output	
Observations	

Table 6-3b. Binary Search for Linked List

7. Supplementary Activity

ILO B: Solve different problems utilizing appropriate searching techniques in C++

For each provided problem, give a screenshot of your code, the output console, and your answers to the questions.

Problem 1. Suppose you are doing a sequential search of the list [15, 18, 2, 19, 18, 0, 8, 14, 19, 14]. Utilizing both a linked list and an array approach to the list, use sequential search and identify how many comparisons would be necessary to find the key '18'?

Problem 2. Modify your sequential search algorithm so that it returns the count of repeating instances for a given search element 'k'. Test on the same list given in problem 1.

Problem 3. Suppose you have the following sorted list [3, 5, 6, 8, 11, 12, 14, 15, 17, 18] and are using the binary search algorithm. If you wanted to find the key 8, draw a diagram that shows how the searching works per iteration of the algorithm. Prove that your drawing is correct by implementing the algorithm and showing a screenshot of the code and the output console.

Problem 4. Modify the binary search algorithm so that the algorithm becomes recursive. Using this new recursive binary search, implement a solution to the same problem for problem 3.

8. Conclusion

Provide the following:

- Summary of lessons learned
- Analysis of the procedure
- Analysis of the supplementary activity
- Concluding statement / Feedback: How well did you think you did in this activity? What are your areas for improvement?

9. Assessment Rubric