Activity No. 5	
SEARCHING TECHNIQUES	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 10/14/2024**
Section: CpE21S4	**Date Submitted: 10/15/2024**
Name(s): Alexander B. San Jose	**Instructor: Prof. Maria. Rizette Sayo**
6. Output	

Screenshot	Output Clear /tmp/mEgBOOauDV.o 1087187464 697151493 851055334 1256050044 489685481 1440942254 202178091 928148279 50224962 1403092919 78757585 707994558 1318404173 16665397 1951983618 554451651 1209737712 1908049598 2012588657 684999539 2043735519 1140817738 493110941 590676306 1569361202 306536427 1492971212 1297325369 1294782288 1179227036 1774381673 234486104 1876378529 477953359 1490536148 218580362 1918895614 1692714239 1146728641 1969120576 948323511 1225486226 529631486 119244036 1242151623 334131456 673695687 304405687 94697406 538800696 === Code Execution Successful ===
Observations	The output shows 50 random numbers, which change every time you run the code because srand(time(0)) resets the seed to the current time. This makes sure the numbers aren’t the same every time. The loop fills the array, and another loop prints it out in one line. It's simple but works to generate random data.

Table 6-1. Data Generated and Observations.

Code

```cpp
#ifndef SEARCHING_H
#define SEARCHING_H

#include <iostream>

void linearSearch(int data[], int N, int item) {
    int i = 0;  // Step 1: Initialize index to 0

    // Step 2: Repeat while i <= N
    while (i < N) {
        if (data[i] == item) {
            std::cout << "Searching is successful. Item found at index: " << i << std::endl;
            return;
        }
        i++;  // Move to the next element
    }

    std::cout << "Searching is unsuccessful. Item not found." << std::endl;
}

#endif

#include <iostream>
#include "searching.h" // Include the linear search header file

const int max_size = 50;

int main() {
    // Generate a simple dataset for demonstration
    int dataset[max_size];
    for (int i = 0; i < max_size; i++) {
        dataset[i] = i + 1; // Filling the array with numbers 1 to 50
    }

    int item = 25;  // Example item to search for
    linearSearch(dataset, max_size, item); // Call the linearSearch function

    return 0;
}
```

Output

```
g++ main.cpp -o search_program
./search_program
```

```
Searching is successful. Item found at index: 24
```

```
Searching is unsuccessful. Item not found.
```

Observations

The code searches for an item in the array and prints if it's found or not. The output shows the index of the item if successful, or says it's not found if the item isn't in the array. Simple linear search logic.

Table 6-2a. Linear Search for Arrays

Code

```cpp
#include <iostream>

// Define the Node structure
template <typename T>
struct Node {
    T data;
    Node* next;
};

// Function to create a new node
template <typename T>
Node<T>* new_node(T data) {
    Node<T>* newNode = new Node<T>;
    newNode->data = data;
    newNode->next = nullptr;
    return newNode;
}

// Sequential search function for linked list
template <typename T>
void linearLS(Node<T>* head, T dataFind) {
    Node<T>* current = head; // Start from the head node
    while (current != nullptr) {
        if (current->data == dataFind) {
            std::cout << "Searching is successful. Item '" << dataFind << "' found." << std::endl;
            return; // Exit if the item is found
        }
        current = current->next; // Move to the next node
    }
    std::cout << "Searching is unsuccessful. Item '" << dataFind << "' not found." << std::endl;
}

int main() {
    // Create linked list for the name "Roman"
    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');

    // Link the nodes
    name1->next = name2;
    name2->next = name3;
    name3->next = name4;
    name4->next = name5;
    name5->next = nullptr;

    // Perform a linear search in the linked list
    linearLS(name1, 'n'); // Searching for 'n'

    return 0;
}

```

Output

```
Output
/tmp/4I2hWEdfvB.o
Searching is successful. Item 'n' found.
```

Observations

The code builds a linked list for the name "Roman," where each letter is in its own node. It has a search function that goes through the list to find a specific letter. If it finds the letter, it says it's successful; if not, it says it's not found. It shows how linked lists work and how to search through them.

Table 6-2b. Linear Search for Linked List

Code

```cpp
#include <iostream>
#include "searching.h"

using namespace std;

int main() {
    int sortedArray[] = {2, 8, 14, 15, 18, 19}; // Sorted array for binary search
    int size = sizeof(sortedArray) / sizeof(sortedArray[0]);
    int searchKey = 18; // Key to search for

    // Perform binary search
    int result = binarySearch(sortedArray, size, searchKey);

    if (result != -1) {
        cout << "Element " << searchKey << " found at index: " << result << endl;
    } else {
        cout << "Element " << searchKey << " not found in the array." << endl;
    }

    // Test with an element that is not in the array
    searchKey = 10; // Key not in the array
    result = binarySearch(sortedArray, size, searchKey);

    if (result != -1) {
        cout << "Element " << searchKey << " found at index: " << result << endl;
    } else {
        cout << "Element " << searchKey << " not found in the array." << endl;
    }

    return 0;
}
```

Output

```
Search element is found!
Element 18 found at index: 4
Search element is not found
Element 10 not found in the array.
```

Observations

In the main program, the binary search function needs a sorted array to work properly, which is pretty important. It figures out how many elements are in the array on the fly, so it's flexible and doesn't need any hardcoded numbers. The code tests for both cases - when the number is found and when it isn't - showing how the algorithm handles different situations. The output messages are clear, so you know if your search was successful. Plus, the way the search function is set up makes it easy to use with different arrays, which is a smart way to keep things organized and simple.

Table 6-3a. Binary Search for Arrays

Code

```cpp
#include <iostream>

// Node structure for the linked list
template <typename T>
struct Node {
    T data;
    Node* next;
};

// Function to create a new node
template <typename T>
Node<T>* new_node(T data) {
    Node<T>* node = new Node<T>;
    node->data = data;
    node->next = nullptr;
    return node;
}

// Function to display the linked list
template <typename T>
void displayList(Node<T>* head) {
    Node<T>* currNode = head;
    while (currNode != nullptr) {
        std::cout << currNode->data << " ";
        currNode = currNode->next;
    }
    std::cout << std::endl;
}

// Function to get the middle node of the linked list
Node<int>* getMiddle(Node<int>* start, Node<int>* end) {
    if (start == nullptr) return nullptr;

    Node<int>* slow = start;
    Node<int>* fast = start->next;

    while (fast != end) {
        fast = fast->next;
        if (fast != end) {
            slow = slow->next;
            fast = fast->next;
        }
    }
    return slow;
}
```

\

```cpp
// Function for binary search in the linked list
Node<int>* binarySearch(Node<int>* head, int key) {
    Node<int>* start = head;
    Node<int>* end = nullptr;

    while (start != end) {
        Node<int>* mid = getMiddle(start, end);

        if (mid == nullptr) return nullptr;

        if (mid->data == key) {
            return mid; // Found the key
        }
        else if (mid->data > key) {
            end = mid; // Search in the left half
        }
        else {
            start = mid->next; // Search in the right half
        }
    }
    return nullptr; // Key not found
}
```

```cpp
// Main function to create the linked list and perform binary search
int main() {
    char choice = 'y';
    int count = 1;
    int newData;

    Node<int>* head = nullptr;
    Node<int>* temp, *node;

    // Create a linked list for binary search
    while (choice == 'y') {
        std::cout << "Enter data (must be ordered): ";
        std::cin >> newData;

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

        // Ask if the user wants to continue
        std::cout << "Continue? (y/n): ";
        std::cin >> choice;
    }

    // Display the linked list
    std::cout << "Linked List: ";
    displayList(head);
```

```cpp
    // Test cases for binary search
    int searchKey1 = 5; // Case 1: Element exists
    Node<int>* foundNode1 = binarySearch(head, searchKey1);
    if (foundNode1) {
        std::cout << "Element " << searchKey1 << " found in the linked list." << std::endl;
    } else {
        std::cout << "Element " << searchKey1 << " not found in the linked list." << std::endl;
    }

    int searchKey2 = 10; // Case 2: Element does not exist
    Node<int>* foundNode2 = binarySearch(head, searchKey2);
    if (foundNode2) {
        std::cout << "Element " << searchKey2 << " found in the linked list." << std::endl;
    } else {
        std::cout << "Element " << searchKey2 << " not found in the linked list." << std::endl;
    }

    int searchKey3 = 1; // Case 3: Element exists (at the start)
    Node<int>* foundNode3 = binarySearch(head, searchKey3);
    if (foundNode3) {
        std::cout << "Element " << searchKey3 << " found in the linked list." << std::endl;
    } else {
        std::cout << "Element " << searchKey3 << " not found in the linked list." << std::endl;
    }

    return 0;
}
```

Output

```
Output
/tmp/sJb9T1S6rU.o
Enter data (must be ordered): 1
Successfully added 1 to the list.
Continue? (y/n): y
Enter data (must be ordered): 3
Successfully added 3 to the list.
Continue? (y/n): y
Enter data (must be ordered): 5
Successfully added 5 to the list.
Continue? (y/n): y
Enter data (must be ordered): 7
Successfully added 7 to the list.
Continue? (y/n): y
Enter data (must be ordered): 9
Successfully added 9 to the list.
Continue? (y/n): n
Linked List: 1 3 5 7 9
Element 5 found in the linked list.
Element 10 not found in the linked list.
Element 1 found in the linked list.


=== Code Execution Successful ===
```

Observations

This code creates a linked list for binary search, which is awesome for finding values quickly It ensures the list is sorted as you add values, and it confirms each addition, which is nice. The middle-finding function uses two pointers, making the search more efficient. The output clearly

	shows whether a value is found or not. Overall, it's a smart way to combine linked lists with binary search.

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'?

```cpp
#include <iostream>
#include <list>

using namespace std;

// Function to perform sequential search on an array
int sequentialSearchInArray(int arr[], int length, int target) {
    int comparisonCount = 0; // Counter for comparisons

    for (int index = 0; index < length; index++) {
        comparisonCount++; // Increment comparisons
        if (arr[index] == target) {
            return comparisonCount; // Return number of comparisons if found
        }
    }
    return -1; // Return -1 if not found
}

// Function to perform sequential search on a linked list
int sequentialSearchInLinkedList(list<int>& myList, int target) {
    int comparisonCount = 0; // Counter for comparisons

    for (auto iterator = myList.begin(); iterator != myList.end(); iterator++) {
        comparisonCount++; // Increment comparisons
        if (*iterator == target) {
            return comparisonCount; // Return number of comparisons if found
        }
    }
    return -1; // Return -1 if not found
}

int main() {
    int numbersArray[] = {15, 18, 2, 19, 18, 0, 8, 14, 19, 14};
    int arrayLength = sizeof(numbersArray) / sizeof(numbersArray[0]);
    list<int> numbersList = {15, 18, 2, 19, 18, 0, 8, 14, 19, 14};
    int searchKey = 18;

    int arrayComparisons = sequentialSearchInArray(numbersArray, arrayLength, searchKey);
    int listComparisons = sequentialSearchInLinkedList(numbersList, searchKey);

    if (arrayComparisons != -1) {
        cout << "Array: Found key '18' after " << arrayComparisons << " comparisons." << endl;
    } else {
        cout << "Array: Key '18' not found." << endl;
    }

    if (listComparisons != -1) {
        cout << "Linked List: Found key '18' after " << listComparisons << " comparisons." << endl;
    } else {
        cout << "Linked List: Key '18' not found." << endl;
    }

    return 0;
}
```

Output

```
/tmp/vgpaPGd9Hp.o
Array: Found key '18' after 2 comparisons.
Linked List: Found key '18' after 2 comparisons.


=== Code Execution Successful ===
```

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.

```cpp
#include <iostream>
#include <list>

using namespace std;

// Function to count occurrences of a key in an array
int countOccurrencesInArray(int arr[], int length, int target) {
    int occurrenceCount = 0; // Counter for occurrences
    for (int index = 0; index < length; index++) {
        if (arr[index] == target) {
            occurrenceCount++; // Increment counter if target is found
        }
    }
    return occurrenceCount; // Return the total count
}

// Function to count occurrences of a key in a linked list
int countOccurrencesInLinkedList(list<int>& myList, int target) {
    int occurrenceCount = 0; // Counter for occurrences
    for (auto iterator = myList.begin(); iterator != myList.end(); iterator++) {
        if (*iterator == target) {
            occurrenceCount++; // Increment counter if target is found
        }
    }
    return occurrenceCount; // Return the total count
}

int main() {
    int numbersArray[] = {15, 18, 2, 19, 18, 0, 8, 14, 19, 14};
    int arrayLength = sizeof(numbersArray) / sizeof(numbersArray[0]);
    list<int> numbersList = {15, 18, 2, 19, 18, 0, 8, 14, 19, 14};
    int searchKey = 18;

    // Count occurrences in the array
    int arrayCount = countOccurrencesInArray(numbersArray, arrayLength, searchKey);
    // Count occurrences in the linked list
    int listCount = countOccurrencesInLinkedList(numbersList, searchKey);

    cout << "In the array, key '18' appears " << arrayCount << " times." << endl;
    cout << "In the linked list, key '18' appears " << listCount << " times." << endl;

    return 0;
}

```

```
Output
/tmp/PG93LfjwiR.o
In the array, key '18' appears 2 times.
In the linked list, key '18' appears 2 times.


=== Code Execution Successful ===
```

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.

```cpp
#include <iostream>

using namespace std;

// Function to perform binary search
int performBinarySearch(int sortedArray[], int start, int end, int target) {
    if (end >= start) {
        int midpoint = start + (end - start) / 2; // Calculate the middle index

        // Check if the target is at the midpoint
        if (sortedArray[midpoint] == target) {
            return midpoint; // Target found
        }

        // If target is smaller than the midpoint value, search the left subarray
        if (sortedArray[midpoint] > target) {
            return performBinarySearch(sortedArray, start, midpoint - 1, target);
        }

        // If target is larger than the midpoint value, search the right subarray
        return performBinarySearch(sortedArray, midpoint + 1, end, target);
    }

    return -1; // Target not found
}

int main() {
    int sortedList[] = {3, 5, 6, 8, 11, 12, 14, 15, 17, 18};
    int totalElements = sizeof(sortedList) / sizeof(sortedList[0]);
    int searchKey = 8; // Key to search for

    // Perform binary search
    int searchResult = performBinarySearch(sortedList, 0, totalElements - 1, searchKey);

    // Output the result
    if (searchResult == -1) {
        cout << "The element was not found in the list." << endl;
    } else {
        cout << "The element is located at index: " << searchResult << endl;
    }

    return 0;
}
```

Output

```
/tmp/7KSGXmGcPo.o
The element is located at index: 3


=== Code Execution Successful ===
```

```
graph LR
A(Iteration 1) --> B(Iteration 2) --> C(Iteration 3)
A[3, 5, 6, 8, 11, 12, 14, 15, 17, 18]
B[3, 5, 6, 8]
C[8]

subgraph A {
    label "Iteration 1"
    mid(mid)
}
subgraph B {
    label "Iteration 2"
    mid(mid)
}
subgraph C {
    label "Iteration 3"
    mid(mid)
}
```

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.

```cpp
#include <iostream>

using namespace std;

// Recursive function to perform binary search
int recursiveBinarySearch(int sortedArray[], int startIndex, int endIndex, int target) {
    // Base condition: if the range is valid
    if (endIndex >= startIndex) {
        int midIndex = startIndex + (endIndex - startIndex) / 2; // Calculate the middle index

        // Check if the target is at the midpoint
        if (sortedArray[midIndex] == target) {
            return midIndex; // Target found
        }

        // If the target is smaller, search the left subarray
        if (sortedArray[midIndex] > target) {
            return recursiveBinarySearch(sortedArray, startIndex, midIndex - 1, target);
        }

        // If the target is larger, search the right subarray
        return recursiveBinarySearch(sortedArray, midIndex + 1, endIndex, target);
    }

    return -1; // Target not found
}

int main() {
    int sortedList[] = {3, 5, 6, 8, 11, 12, 14, 15, 17, 18};
    int totalElements = sizeof(sortedList) / sizeof(sortedList[0]);
    int searchKey = 8; // Key to search for

    // Call the recursive binary search function
    int searchResult = recursiveBinarySearch(sortedList, 0, totalElements - 1, searchKey);

    // Output the result
    if (searchResult == -1) {
        cout << "The key was not found in the list." << endl;
    } else {
        cout << "The key is located at index: " << searchResult << endl;
    }

    return 0;
}
```

Output

```
/tmp/AAhVJN7fjU.o
The key is located at index: 3


=== Code Execution Successful ===
```

8. Conclusion

In conclusion, this activity helped me understand different searching techniques in C++. I learned how linear search works with both arrays and linked lists and how to count repeated values. I also explored binary search, first with an iterative approach and then by making it recursive. The hands-on coding and testing showed me the importance of data structures and how they affect search efficiency. The supplementary activity effectively highlighted the practical applications of searching techniques in C++. Problem 1 emphasized the efficiency differences between sequential search in arrays and linked lists. Problem 2 enhanced understanding by adapting the search to count repeated instances. Problem 3 illustrated the importance of sorted data for binary search and reinforced theoretical concepts through implementation. Finally, Problem 4 introduced recursion, making the binary search implementation more elegant and deepening comprehension of recursive algorithms. Overall, it was a fun way to dive deeper into algorithms and improve my programming skills.
9. Assessment Rubric