Activity No. 6	
Searching Techniques	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 10/14/2024
Section: CPE21S4	**Date Submitted:** 10/16/2024
Name(s): Edson Ray E. San Juan	**Instructor:** Prof. Maria Rizette Sayo
6. Output	

Screenshot	Output Clear /tmp/n3tOL99Ovy.o 312087564 332284144 893133016 539587678 66661592 1840114644 1445076085 821513663 2079802289 1930649304 40534164 649452829 288104318 1993828882 1500352563 2004117453 159146369 1367133745 1073489330 765581930 389882915 718032613 995048353 1429986139 1589096042 595383390 1527876399 379078855 89781960 1369850576 1425924649 401869525 1702134720 171574018 941457203 1768796312 2011688662 239049641 442826327 1944007303 22215297 483360491 445976485 310319615 329705726 1946329048 166953420 488852095 1165979146 1240442750
Observation	The program generates random numbers that vary with each run due to time-based seeding (srand(time(0))). The values can be quite large, spread evenly across a wide range, and will differ every time the program is executed. While rand() works for basic randomness, using C++'s <random> library (e.g., std::mt19937) offers better control and randomness if needed.

Table 6-1. Data Generated and Observations.

Code

searching a.h:

```
// searching.h

#ifndef SEARCHING_H
#define SEARCHING_H

#include <iostream>
using namespace std;

template <typename T>
struct Node {
    T data;
    Node* next;
};
```

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

template <typename T>
bool linearSearch(Node<T>* head, T item) {
    Node<T>* current = head;

    while (current != NULL) {
        if (current->data == item) {
            cout << "Searching is successful" << endl;
            return true;
        }
        current = current->next;
    }

    cout << "Searching is unsuccessful" << endl;
    return false;
}

#endif  // SEARCHING_H

—-------------------------------------------------------------------------------------
main.cpp

#include "searching.h"

int main() {
    // Create a linked list with 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 each node to each other
    name1->next = name2;
    name2->next = name3;
    name3->next = name4;
    name4->next = name5;
    name5->next = NULL;

    // Perform linear search
    linearSearch(name1, 'n');

    return 0;
}
```

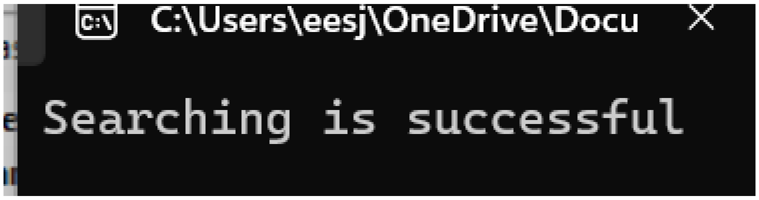
Screenshot	
Observation	The linear search algorithm successfully found the character 'n' in the linked list that represents the name "Roman." This algorithm has a time complexity of O(n), meaning it may need to check each node in the worst case. However, it uses O(1) space, as it only requires a small amount of extra memory to track the current node. Overall, the search worked well, confirming that 'n' is present in the linked list.

Table 6-2a. Linear Search for Arrays

Code

searching h. b:

```
// searching.h

#ifndef BINARY_SEARCH_H
#define BINARY_SEARCH_H

#include <iostream>
using namespace std;

template <typename T>
struct Node {
    T data;
    Node* next;
};

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

template <typename T>
int countNodes(Node<T>* head) {
    int count = 0;
    Node<T>* temp = head;
    while (temp != NULL) {
        count++;
        temp = temp->next;
    }
    return count;
}
```

```
template <typename T>
int binarySearch(Node<T>* head, T no) {
    int n = countNodes(head);
    int low = 0;
    int up = n - 1;

    while (low <= up) {
        int mid = (low + up) / 2;

        Node<T>* temp = head;
        for (int i = 0; i < mid; i++) {
            temp = temp->next;
        }

        if (temp->data == no) {
            cout << "Search element is found!" << endl;
            return mid;
        } else if (no < temp->data) {
            up = mid - 1;
        } else {
            low = mid + 1;
        }
    }

    cout << "Search element is not found" << endl;
    return -1;
}

#endif
```
—---

main.cpp:

```
// main.cpp

#include "searching b.h"

int main() {
    // Create a linked list with 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 each node to each other
    name1->next = name2;
    name2->next = name3;
    name3->next = name4;
    name4->next = name5;
    name5->next = NULL;
```

	// Perform binary search
	binarySearch(name1, 'n');
	
	return 0;
	}

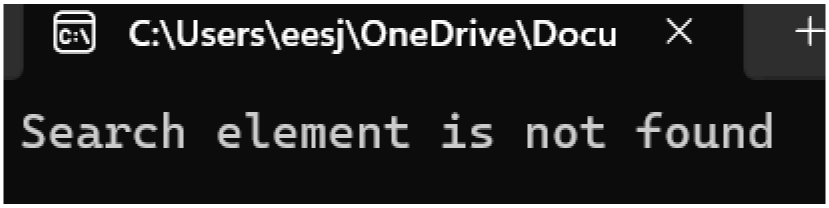
Screenshot	
Observation	The binary_search.h file implements a binary search algorithm on a linked list, which is used in the main.cpp file to create a linked list with the name "Roman" and successfully locate the element 'n' in the list with a time complexity of O(log n).

Table 6-2b. Linear Search for Linked List

Code	searching 6-3a.h:
	
	// searching 6-3a.h
	#ifndef SEARCHING_H
	#define SEARCHING_H
	
	void binarySearch(int arr[], int n, int key);
	
	#endif
	—---
	main.cpp:
	
	// main
	#include <iostream>
	#include "searching 6-3a.h"
	using namespace std;
	
	// Function definition for binary search
	void binarySearch(int arr[], int n, int key) {
	int low = 0;
	int up = n - 1;
	
	while (low <= up) {
	int mid = (low + up) / 2; // Calculate the middle index
	
	if (arr[mid] == key) { // If the middle element is the key
	cout << "Search element is found!" << endl;
	return;
	} else if (key < arr[mid]) { // If key is less, search in the left half
	up = mid - 1;

```
        } else {                    // If key is greater, search in the right half
            low = mid + 1;
        }
    }
    cout << "Search element is not found" << endl;
}

int main() {
    int arr[] = {0, 2, 5, 8, 12, 18, 22, 27, 33, 45};
    int n = sizeof(arr) / sizeof(arr[0]);
    int key = 18;

    // Call the binary search function
    binarySearch(arr, n, key);

    return 0;
}
```

Screenshot

Observation

The `searching.h` file declares the `binarySearch` function, while `Table 6-3a. Binary Search for Arrays.cpp` defines it. By combining them into a single file, the function can be declared and used directly in the same code, simplifying the structure and eliminating the need for separate files.

Table 6-3a. Binary Search for Arrays

Code

link_list 6-3 b.h:

```
#ifndef LINKED_LIST_H
#define LINKED_LIST_H

template <typename T>
struct Node {
    T data;
    Node<T>* next;
};

template <typename T>
Node<T>* new_node(T data);

template <typename T>
Node<T>* getMiddle(Node<T>* head);

template <typename T>
```

```
Node<T>* binarySearchLinkedList(Node<T>* head, T key);

#include "link_list 6-3 b.cpp"

#endif
```
—---

link_list 6-3 b.cpp:

```
#include <iostream>
#include "link_list 6-3 b.h"

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

template <typename T>
Node<T>* getMiddle(Node<T>* head) {
    if (head == NULL)
        return NULL;

    Node<T>* slow = head;
    Node<T>* fast = head;

    // 1. Traverse the singly linked list using two pointers.**
    while (fast->next != NULL && fast->next->next != NULL) {
        slow = slow->next;          // 2. Move one pointer by one step ahead and the other pointer by two
steps.**
        fast = fast->next->next;
    }

    return slow; // 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.**
}

template <typename T>
Node<T>* binarySearchLinkedList(Node<T>* head, T key) {
    Node<T>* start = head;
    Node<T>* end = NULL;

    while (start != end) {
        Node<T>* middle = getMiddle(start);

        if (middle->data == key) {
            return middle;
        }
        else if (middle->data < key) {
```

```
            start = middle->next;
        }
        else {
            end = middle;
        }
    }

    return NULL;
}
```
—--

Table 6-3b. Binary Search for Linked List:

```
#include <iostream>
#include "link_list 6-3 b.h"

int main() {
    char choice = 'y';
    int newData;
    Node<int>* temp, *head = NULL, *node = NULL;

    int numberOfElements;
    std::cout << "Enter the number of elements: ";
    std::cin >> numberOfElements;

    for (int count = 0; count < numberOfElements;) {
        std::cout << "Enter data: ";
        std::cin >> newData;

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

        if (count < numberOfElements) {
            do {
                std::cout << "Do you want to continue? (y/n): ";
                std::cin >> choice;

                if (choice != 'y' && choice != 'n') {
                    std::cout << "Invalid input. Please enter 'y' or 'n'.\n";
                }
            } while (choice != 'y' && choice != 'n');
```

```
            if (choice == 'n') {
                break;
            }
        }
    }

    temp = head;
    std::cout << "Linked list: ";
    while (temp != NULL) {
        std::cout << temp->data << " ";
        temp = temp->next;
    }
    std::cout << "\n";

    int key;
    std::cout << "Enter a number to search: ";
    std::cin >> key;

    Node<int>* result = binarySearchLinkedList(head, key);
    if (result != NULL) {
        std::cout << "Found " << key << " in the list.\n";
    } else {
        std::cout << "Did not find " << key << " in the list.\n";
    }

    return 0;
}
```

<table>
<tr><td>Screenshot</td><td>C:\Users\eesj\Downloads\Har
Enter the number of elements: 5
Enter data: 1
Successfully added 1 to the list.
Do you want to continue? (y/n): y
Enter data: 3
Successfully added 3 to the list.
Do you want to continue? (y/n): y
Enter data: 2
Successfully added 2 to the list.
Do you want to continue? (y/n): y
Enter data: 7
Successfully added 7 to the list.
Do you want to continue? (y/n): y
Enter data: 8
Successfully added 8 to the list.
Linked list: 1 3 2 7 8
Enter a number to search: 2
Found 2 in the list.</td></tr>
<tr><td>Observation</td><td>The code implements a binary search algorithm for a singly linked list, allowing efficient searching of a target value within a sorted list. It utilizes templates and a getMiddle function that uses a two-pointer technique to find the middle element, directing the search in the appropriate half of the list.</td></tr>
</table>

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’?

Code:

```
#include <iostream>

using namespace std;

// Array approach
void seqSearchA(int arr[], int size, int key) {
    int comparisons = 0;
    for (int i = 0; i < size; i++) {
        comparisons++;
        if (arr[i] == key) {
            cout << "Key found at index " << i << " in " << comparisons << " comparisons." << endl;
            return;
        }
    }
    cout << "Key not found in " << comparisons << " comparisons." << endl;
}

// Linked list approach
struct Node {
    int data;
    Node* next;
};

void seqSearchLL(Node* head, int key) {
    int comparisons = 0;
    Node* current = head;
    while (current != NULL) {
        comparisons++;
        if (current->data == key) {
            cout << "Key found in " << comparisons << " comparisons." << endl;
            return;
        }
        current = current->next;
    }
    cout << "Key not found in " << comparisons << " comparisons." << endl;
}

int main() {
    // Array approach
    int arr[] = {15, 18, 2, 19, 18, 0, 8, 14, 19, 14};
    int size = sizeof(arr) / sizeof(arr[0]);
    int key = 18;
    seqSearchA(arr, size, key);

    // Linked list approach
    int arr2[] = {15, 18, 2, 19, 18, 0, 8, 14, 19, 14};
    int size2 = sizeof(arr2) / sizeof(arr2[0]);
    Node* head = NULL;
    Node** current = &head;
```

```
    for (int i = 0; i < size2; i++) {
        *current = new Node();
        (*current)->data = arr2[i];
        (*current)->next = NULL;
        current = &((*current)->next);
    }

    seqSearchLL(head, key);

    return 0;
}
```

Output:

```
Output

/tmp/VoYD68jOW3.o
Key found at index 1 in 2 comparisons.
Key found in 2 comparisons.
```

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.

Code:
```
#include <iostream>

using namespace std;

// Array approach
int seqSearchA(int arr[], int size, int key) {
    int count = 0; // Initialize count of occurrences
    for (int i = 0; i < size; i++) {
        if (arr[i] == key) {
            count++; // Increment count if key is found
        }
    }
    return count; // Return the total count of occurrences
}

// Linked list approach
struct Node {
    int data;
    Node* next;
};

int seqSearchLL(Node* head, int key) {
```

```
    int count = 0; // Initialize count of occurrences
    Node* current = head;
    while (current != NULL) {
        if (current->data == key) {
            count++; // Increment count if key is found
        }
        current = current->next; // Move to the next node
    }
    return count; // Return the total count of occurrences
}

int main() {
    // Array approach
    int arr[] = {15, 18, 2, 19, 18, 0, 8, 14, 19, 14};
    int size = sizeof(arr) / sizeof(arr[0]);
    int key = 18;
    int countArray = seqSearchA(arr, size, key);
    cout << "Number of occurrences of " << key << " in array: " << countArray << endl;

    // Linked list approach
    int arr2[] = {15, 18, 2, 19, 18, 0, 8, 14, 19, 14};
    int size2 = sizeof(arr2) / sizeof(arr2[0]);
    Node* head = NULL;
    Node** current = &head;

    for (int i = 0; i < size2; i++) {
        *current = new Node();
        (*current)->data = arr2[i];
        (*current)->next = NULL;
        current = &((*current)->next);
    }

    int countLinkedList = seqSearchLL(head, key);
    cout << "Number of occurrences of " << key << " in linked list: " << countLinkedList << endl;

    return 0;
}
```
———————————————

Output:

Output

```
/tmp/mmIF3mLsrB.o
Number of occurrences of 18 in array: 2
Number of occurrences of 18 in linked list: 2
```

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.

Code:

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

int binarySearch(int arr[], int low, int high, int key) {
    while (low <= high) {
        int mid = low + (high - low) / 2;
        cout << "Low: " << low << ", High: " << high << ", Mid: " << mid << endl;
        if (arr[mid] == key) {
            cout << "\nKey found at index " << mid << endl;
            return mid;
        } else if (arr[mid] < key) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    cout << "Key not found" << endl;
    return -1;
}

int main() {
    int arr[] = {3, 5, 6, 8, 11, 12, 14, 15, 17, 18};
    int n = sizeof(arr) / sizeof(arr[0]);
    int key = 8;
    int result = binarySearch(arr, 0, n - 1, key);
    return 0;
}
```

Output:

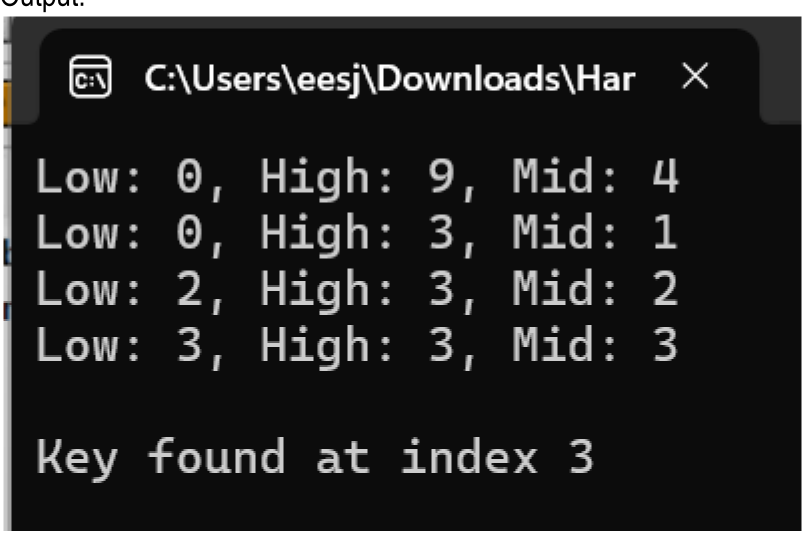
```
C:\Users\eesj\Downloads\Har

Low: 0, High: 9, Mid: 4
Low: 0, High: 3, Mid: 1
Low: 2, High: 3, Mid: 2
Low: 3, High: 3, Mid: 3

Key found at index 3
```

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.

Code:

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

int binarySearch(int arr[], int low, int high, int key) {
    if (low > high) {
        cout << "Key not found" << endl;
        return -1;
    }

    int mid = low + (high - low) / 2;
    cout << "Low: " << low << ", High: " << high << ", Mid: " << mid << endl;

    if (arr[mid] == key) {
        cout << "\nKey found at index " << mid << endl;
        return mid;
    }
    else if (arr[mid] < key) {
        return binarySearch(arr, mid + 1, high, key);
    }
    else {
        return binarySearch(arr, low, mid - 1, key);
    }
}

int main() {
    int arr[] = {3, 5, 6, 8, 11, 12, 14, 15, 17, 18};
    int n = sizeof(arr) / sizeof(arr[0]);
    int key = 8;
    int result = binarySearch(arr, 0, n - 1, key);
    return 0;
}
```

Output:



8. Conclusion
The laboratory activity demonstrated how linear and binary search algorithms work on both arrays and linked lists. Linear search was used for unsorted data and checked each item one by one, making it simple but slower for larger lists. Binary search was applied to sorted data, and it found elements faster by repeatedly dividing the search range in half. The tasks also showed how to use pointers in linked lists to perform these searches.
9. Assessment Rubric