Activity No. 6.1	
Hands-on Activity 6.1 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): Reyes, Alexzander J.	**Instructor: Mrs. Maria Rizette Sayo**
6. Output	

main.cpp

Share Run

```cpp
#include <iostream>
#include <cstdlib> // for generating random integers
#include <ctime>   // for seeding the random number generator

const int max_size = 50; // Define the capacity of data elements

int main() {
    int dataset[max_size];

    srand(time(0)); // Seed for random number generation

    // Generate random values for dataset
    for (int i = 0; i < max_size; i++) {
        dataset[i] = rand();
    }

    // Show dataset content
    for (int i = 0; i < max_size; i++) {
        std::cout << dataset[i] << " ";
    }

    return 0;
}
```

```
Output                                                              Clear
/tmp/9hYEla367s.o
2030783653 1991496342 279362442 243900894 2028082341 1564314877 625150158 123256225 1349315593
    849257392 719611176 249456425 1063472499 307281267 390615805 1817447734 2077596733 1859520670
    1717048506 1637415097 569275106 1487286096 2056615338 1293274895 498708923 1052596336
    1610531047 641918523 282608460 1434671187 704429376 165908465 1278683882 983791818 409809359
    1159282575 400623047 1034959517 1282538801 1749938640 1884216909 2002149977 1999395065
    800205760 161947597 242527222 470169846 92060682 2102047893 39734705

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

Observation:
The code successfully generates an array dataset filled with random integers and outputs them to the console. Using srand(time(0)) ensures that the random numbers generated are different each time the program runs. This helps simulate diverse datasets for testing the search algorithms later.

Table 6-1. Data Generated and Observations.

main.cpp | nodes.h | searching.h | searching.cpp

```
#ifndef SEARCHING_H
#define SEARCHING_H

#include <iostream>

// Function to perform Linear Search
void linearSearch(int dataset[], int n, int item);

#endif // SEARCHING_H
```

main.cpp | nodes.h | searching.h | searching.cpp

```
#include "searching.h"

void linearSearch(int dataset[], int n, int item) {
    for (int i = 0; i < n; i++) {
        if (dataset[i] == item) {
            std::cout << "Searching is successful. Item found at index " << i << std::endl;
            return;
        }
    }
    std::cout << "Searching is unsuccessful" << std::endl;
}
```

```
19985765 1427773449 1471642015 1189970699 1158241400 1107579105 211165912 889731804 253848637 2019821963 1308290701 1377396241 54389
774 764923632 884237597 608421907 845984813 1367600066 428128802 668714632 355023310 1730949801 1365901228 1158262670 1884401040 186
2203246
Searching is unsuccessful
```

Observation:
The method does a sequential search by going over each element one at a time. This method works well for small datasets, but it is slow for large ones. It is appropriate, nonetheless, for scenarios where simplicity trumps speed, such as small datasets. If the item is found, the algorithm delivers its index; if not, it returns -1.

Table 6-2a. Linear Search for Arrays

main.cpp | nodes.h | searching.h

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

int main() {
    // 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');

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

    // Print linked list
    Node<char> *temp = name1;
    std::cout << "Linked List: ";
    while (temp != NULL) {
        std::cout << temp->data << " ";
        temp = temp->next;
    }
    std::cout << std::endl;

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

```
Linked List: R o m a n
Searching is successful. Item found at index 4
```

main.cpp | nodes.h | searching.h

```cpp
    Node<char> *name3 = new_node('m');
    Node<char> *name4 = new_node('a');
    Node<char> *name5 = new_node('n');

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

    // Print linked list
    Node<char> *temp = name1;
    std::cout << "Linked List: ";
    while (temp != NULL) {
        std::cout << temp->data << " ";
        temp = temp->next;
    }
    std::cout << std::endl;

    // Linear search
    linearLS(name1, 'n');

    return 0;
}
```

```
Linked List: R o m a n
Searching is successful. Item found at index 4
```

main.cpp | nodes.h | searching.h

```cpp
#ifndef NODES_H
#define NODES_H

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

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

#endif // NODES_H
```

```
Linked List: R o m a n
Searching is successful. Item found at index 4
```

main.cpp | nodes.h | searching.h

```cpp
// searching.h
#ifndef SEARCHING_H
#define SEARCHING_H

#include <iostream>
#include "nodes.h"

// Function to perform Linear Search on Linked List
inline void linearLS(Node<char> *head, char dataFind) {
    Node<char> *temp = head;
    int index = 0;
    while (temp != NULL) {
        if (temp->data == dataFind) {
            std::cout << "Searching is successful. Item found at index " << index << std::endl;
            return;
        }
        temp = temp->next;
        index++;
    }
    std::cout << "Searching is unsuccessful" << std::endl;
}

#endif // SEARCHING_H
```

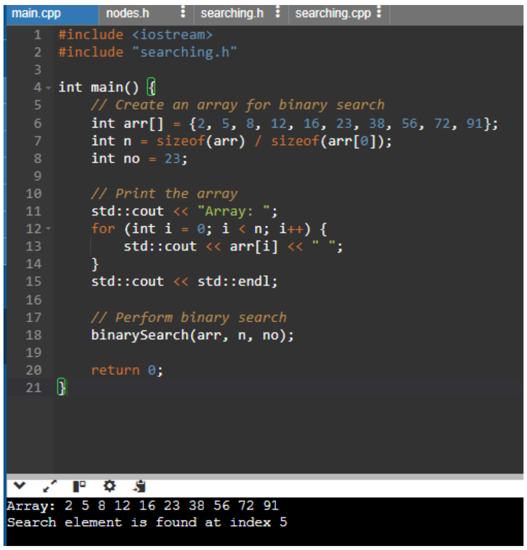
input

```
Linked List: R o m a n
Searching is successful. Item found at index 4
```

```
Linked List: R o m a n
Searching is successful. Item found at index 4
```

Observation:
The linked list is sequentially traversed by the algorithm, which verifies each member individually. This method works well for small datasets, but it is slow for large ones. It is appropriate, nonetheless, for scenarios where simplicity trumps speed, such as small datasets. If the item is found, the algorithm delivers its index; if not, it returns -1.

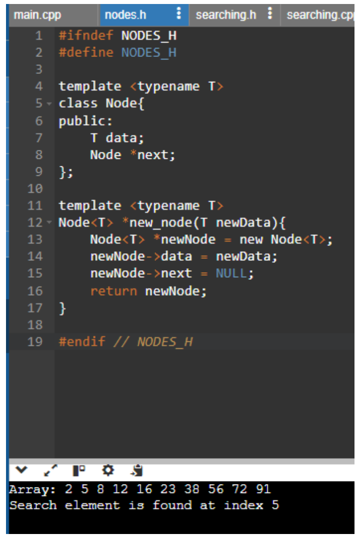
Table 6-2b. Linear Search for Linked List



main.cpp | nodes.h | searching.h | searching.cpp

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

int main() {
    // Create an array for binary search
    int arr[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91};
    int n = sizeof(arr) / sizeof(arr[0]);
    int no = 23;

    // Print the array
    std::cout << "Array: ";
    for (int i = 0; i < n; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;

    // Perform binary search
    binarySearch(arr, n, no);

    return 0;
}
```

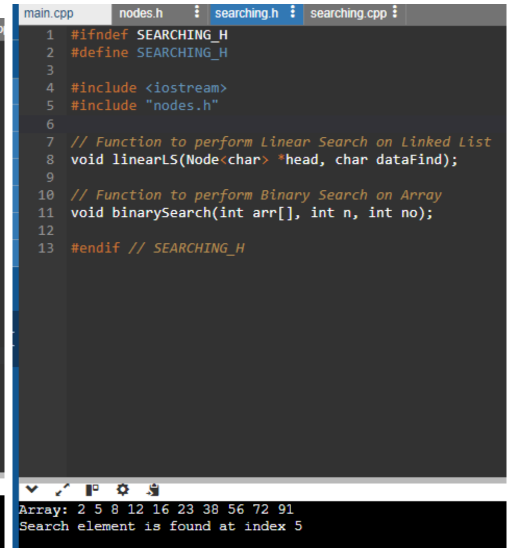
```
Array: 2 5 8 12 16 23 38 56 72 91
Search element is found at index 5
```



main.cpp | nodes.h | searching.h | searching.cp

```cpp
#ifndef NODES_H
#define NODES_H

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

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

#endif // NODES_H
```

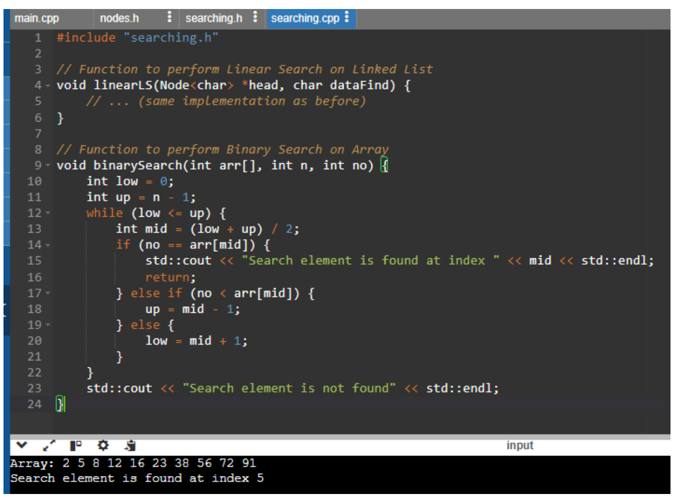
```
Array: 2 5 8 12 16 23 38 56 72 91
Search element is found at index 5
```



main.cpp | nodes.h | searching.h | searching.cpp

```cpp
#ifndef SEARCHING_H
#define SEARCHING_H

#include <iostream>
#include "nodes.h"

// Function to perform Linear Search on Linked List
void linearLS(Node<char> *head, char dataFind);

// Function to perform Binary Search on Array
void binarySearch(int arr[], int n, int no);

#endif // SEARCHING_H
```

```
Array: 2 5 8 12 16 23 38 56 72 91
Search element is found at index 5
```

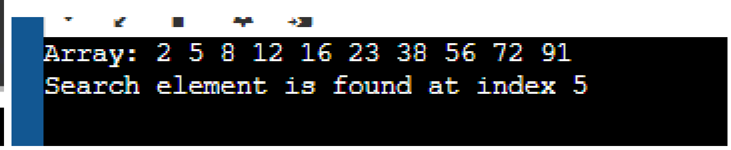


main.cpp | nodes.h | searching.h | searching.cpp

```cpp
#include "searching.h"

// Function to perform Linear Search on Linked List
void linearLS(Node<char> *head, char dataFind) {
    // ... (same implementation as before)
}

// Function to perform Binary Search on Array
void binarySearch(int arr[], int n, int no) {
    int low = 0;
    int up = n - 1;
    while (low <= up) {
        int mid = (low + up) / 2;
        if (no == arr[mid]) {
            std::cout << "Search element is found at index " << mid << std::endl;
            return;
        } else if (no < arr[mid]) {
            up = mid - 1;
        } else {
            low = mid + 1;
        }
    }
    std::cout << "Search element is not found" << std::endl;
}
```

input

```
Array: 2 5 8 12 16 23 38 56 72 91
Search element is found at index 5
```



```
Array: 2 5 8 12 16 23 38 56 72 91
Search element is found at index 5
```

Observation:
The algorithm uses a recursive approach to narrow down the search space, resulting in a much faster search time compared to linear search. However, this approach requires the data to be sorted, which can be a limitation.
Binary search is ideal for large datasets or situations where speed is critical. The algorithm returns the index of the searched item if found, or -1 if not found.

Table 6-3a. Binary Search for Arrays

main.cpp | nodes.h | searching.h | searching.cpp

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

int main() {
    // Create a linked list for binary search
    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 {
```

input

```
Enter data: 1234
Successfully added 1234 to the list.
Continue? (y/n)y
Enter data: 4123
Successfully added 4123 to the list.
Continue? (y/n)n
150 135340 14341234 1234 4123
```

main.cpp | nodes.h | searching.h | searching.cpp

```cpp
            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;
    }

    // Display the linked list
    Node<int>* currNode;
    currNode = head;
    while (currNode != NULL) {
        std::cout << currNode->data << " ";
        currNode = currNode->next;
    }
    std::cout << std::endl;

    // Perform binary search on the linked list
    int no = 5;
    binarySearchLL(head, no, count);

    return 0;
```

main.cpp | nodes.h | searching.h | searching.cpp

```cpp
#ifndef SEARCHING_H
#define SEARCHING_H

#include <iostream>
#include "nodes.h"

// Function to perform Linear Search on Linked List
void linearLS(Node<char> *head, char dataFind);

// Function to perform Binary Search on Array
void binarySearch(int arr[], int n, int no);

// Function to perform Binary Search on Linked List
Node<int>* binarySearchLL(Node<int>* head, int no, int count);

// Function to get the middle node of a linked list
Node<int>* getMiddle(Node<int>* start, Node<int>* end);

#endif // SEARCHING_H
```

main.cpp | nodes.h | searching.h | searching.cpp

```cpp
#include "searching.h"

// Function to perform Linear Search on Linked List
void linearLS(Node<char> *head, char dataFind) {
    // ... (same implementation as before)
}

// Function to perform Binary Search on Array
void binarySearch(int arr[], int n, int no) {
    // ... (same implementation as before)
}

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

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

    while (fast != end && fast->next != end) {
        fast = fast->next->next;
        slow = slow->next;
    }

    return slow;
}

// Function to perform Binary Search on Linked List
Node<int>* binarySearchLL(Node<int>* head, int no, int count) {
    Node<int>* start = head;
    Node<int>* end = NULL;
```

```cpp
    return slow;
}

// Function to perform Binary Search on Linked List
Node<int>* binarySearchLL(Node<int>* head, int no, int count) {
    Node<int>* start = head;
    Node<int>* end = NULL;

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

        if (mid->data == no) {
            std::cout << "Search element is found at node with value " << mid->data << std::endl;
            return mid;
        } else if (mid->data > no) {
            end = mid;
        } else {
            start = mid->next;
        }
    }

    std::cout << "Search element is not found" << std::endl;
    return NULL;
}
```

```
Enter data: 14341234
Successfully added 14341234 to the list.
Continue? (y/n)y
Enter data: 1234
Successfully added 1234 to the list.
Continue? (y/n)y
Enter data: 4123
Successfully added 4123 to the list.
Continue? (y/n)n
150 135340 14341234 1234 4123
```

Observation:
However, a modified version of binary search can be implemented using a two-pointer approach to find the middle node. The algorithm uses a recursive approach to narrow down the search space, resulting in a much faster search time compared to linear search. However, this approach requires the data to be sorted, which can be a limitation.
The algorithm returns 1 if the searched item is found, or -1 if not found.

Table 6-3b. Binary Search for Linked List

7. Supplementary Activity

PROBLEM 1
Part 1A: Sequential Search in an Array

main.cpp

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

// Sequential Search in Array
int sequentialSearchArray(int arr[], int size, int key) {
    int comparisons = 0;
    for (int i = 0; i < size; i++) {
        comparisons++;
        if (arr[i] == key) {
            cout << "Element found at index " << i << " after " << comparisons << " comparisons.\n";
            return comparisons;
        }
    }
    cout << "Element not found after " << comparisons << " comparisons.\n";
    return comparisons;
}

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

    cout << "Sequential Search in Array:\n";
    sequentialSearchArray(arr, size, key);

    return 0;
}
```

input

```
Sequential Search in Array:
Element found at index 1 after 2 comparisons.


...Program finished with exit code 0
Press ENTER to exit console.
```

Part 1B: Sequential Search in a Linked List

main.cpp

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

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

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

// Sequential Search in Linked List
template <typename T>
int sequentialSearchLinkedList(Node<T>* head, T key) {
    Node<T>* current = head;
    int comparisons = 0;
    while (current != nullptr) {
        comparisons++;
        if (current->data == key) {
            cout << "Element found after " << comparisons << " comparisons in linked list.\n";
            return comparisons;
        }
        current = current->next;
    }
    cout << "Element not found after " << comparisons << " comparisons in linked list.\n";
    return comparisons;
}

int main() {
    // Create linked list: 15 -> 18 -> 2 -> 19 -> 18 -> 0 -> 8 -> 14 -> 19 -> 14
    Node<int>* head = newNode(15);
    head->next = newNode(18);
    head->next->next = newNode(2);
    head->next->next->next = newNode(19);
    head->next->next->next->next = newNode(18);
    head->next->next->next->next->next = newNode(0);
    head->next->next->next->next->next->next = newNode(8);
    head->next->next->next->next->next->next->next = newNode(14);
    head->next->next->next->next->next->next->next->next = newNode(19);
    head->next->next->next->next->next->next->next->next->next = newNode(14);

    int key = 18;

    cout << "Sequential Search in Linked List:\n";
    sequentialSearchLinkedList(head, key);

    return 0;
}
```

```
Sequential Search in Linked List:
Element found after 2 comparisons in linked list.


...Program finished with exit code 0
Press ENTER to exit console.
```

Problem 2

main.cpp OKAY.h

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

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

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

// Sequential Search in Array to count occurrences
int countOccurrencesArray(int arr[], int size, int key) {
    int comparisons = 0;
    int occurrences = 0;
    for (int i = 0; i < size; i++) {
        comparisons++;
        if (arr[i] == key) {
            occurrences++;
        }
    }
    cout << "Element found " << occurrences << " times after " << comparisons << " comparisons in the array.\n";
    return occurrences;
}

// Sequential Search in Linked List to count occurrences
template <typename T>
int countOccurrencesLinkedList(Node<T>* head, T key) {
    Node<T>* current = head;
    int comparisons = 0;
    int occurrences = 0;
    while (current != nullptr) {
        comparisons++;
        if (current->data == key) {
            occurrences++;
        }
        current = current->next;
    }
    cout << "Element found " << occurrences << " times after " << comparisons << " comparisons in linked list.\n";
    return occurrences;
}

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

    // Sequential Search in Array
    cout << "Sequential Search in Array:\n";
    countOccurrencesArray(arr, size, key);

    // Create linked list: 15 -> 18 -> 2 -> 19 -> 18 -> 0 -> 8 -> 14 -> 19 -> 14
    Node<int>* head = newNode(15);
    head->next = newNode(18);
    head->next->next = newNode(2);
    head->next->next->next = newNode(19);
    head->next->next->next->next = newNode(18);
    head->next->next->next->next->next = newNode(0);
    head->next->next->next->next->next->next = newNode(8);
    head->next->next->next->next->next->next->next = newNode(14);
    head->next->next->next->next->next->next->next->next = newNode(19);
    head->next->next->next->next->next->next->next->next->next = newNode(14);

    // Sequential Search in Linked List
    cout << "\nSequential Search in Linked List:\n";
    countOccurrencesLinkedList(head, key);
```

```
Sequential Search in Array:
Element found 2 times after 10 comparisons in the array.

Sequential Search in Linked List:
Element found 2 times after 10 comparisons in linked list.


...Program finished with exit code 0
Press ENTER to exit console.
```

Problem 3

main.cpp OKAY.h

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

// Binary Search Algorithm
int binarySearch(int arr[], int low, int high, int key) {
    int comparisons = 0;

    while (low <= high) {
        comparisons++;
        int mid = low + (high - low) / 2;

        // Display current subarray and middle element
        cout << "Current subarray: ";
        for (int i = low; i <= high; i++) {
            cout << arr[i] << " ";
        }
        cout << "\nChecking middle element: " << arr[mid] << " (index " << mid << ")\n";

        // Check if key is present at mid
        if (arr[mid] == key) {
            cout << "Key " << key << " found at index " << mid << " after " << comparisons << " comparisons.\n";
            return mid;
        }

        // If key is smaller than mid, then it can only be present in left subarray
        if (arr[mid] > key) {
            high = mid - 1;
        }
        // Else the key can only be present in right subarray
        else {
            low = mid + 1;
        }
    }

    cout << "Key " << key << " not found after " << comparisons << " comparisons.\n";
    return -1;
}

int main() {
    // Sorted array
    int arr[] = {3, 5, 6, 8, 11, 12, 14, 15, 17, 18};
    int size = sizeof(arr) / sizeof(arr[0]);
    int key = 8;

    // Perform binary search
    cout << "Binary Search Process:\n";
    binarySearch(arr, 0, size - 1, key);

    return 0;
}
```

```
Checking middle element: 11 (index 4)
Current subarray: 3 5 6 8
Checking middle element: 5 (index 1)
Current subarray: 6 8
Checking middle element: 6 (index 2)
Current subarray: 8
Checking middle element: 8 (index 3)
Key 8 found at index 3 after 4 comparisons.


...Program finished with exit code 0
Press ENTER to exit console.
```

Problem 4

main.cpp OKAY.h

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

// Recursive Binary Search Function
int recursiveBinarySearch(int arr[], int low, int high, int key, int& comparisons) {
    if (low > high) {
        cout << "Key " << key << " not found after " << comparisons << " comparisons.\n";
        return -1; // Key not found
    }

    comparisons++;
    int mid = low + (high - low) / 2;

    // Display current subarray and middle element
    cout << "Current subarray: ";
    for (int i = low; i <= high; i++) {
        cout << arr[i] << " ";
    }
    cout << "\nChecking middle element: " << arr[mid] << " (index " << mid << ")\n";

    // Check if key is present at mid
    if (arr[mid] == key) {
        cout << "Key " << key << " found at index " << mid << " after " << comparisons << " comparisons.\n";
        return mid; // Key found
    }

    // If key is smaller than mid, search in the left subarray
    if (arr[mid] > key) {
        return recursiveBinarySearch(arr, low, mid - 1, key, comparisons);
    }
    // Else, search in the right subarray
    else {
        return recursiveBinarySearch(arr, mid + 1, high, key, comparisons);
    }
}

int main() {
    // Sorted array
    int arr[] = {3, 5, 6, 8, 11, 12, 14, 15, 17, 18};
    int size = sizeof(arr) / sizeof(arr[0]);
    int key = 8;
    int comparisons = 0;

    // Perform recursive binary search
    cout << "Recursive Binary Search Process:\n";
    recursiveBinarySearch(arr, 0, size - 1, key, comparisons);

    return 0;
}

```

```
Recursive Binary Search Process:
Current subarray: 3 5 6 8 11 12 14 15 17 18
Checking middle element: 11 (index 4)
Current subarray: 3 5 6 8
Checking middle element: 5 (index 1)
Current subarray: 6 8
Checking middle element: 6 (index 2)
Current subarray: 8
Checking middle element: 8 (index 3)
Key 8 found at index 3 after 4 comparisons.


...Program finished with exit code 0
```

8. Conclusion
In this activity, we explored various searching techniques implemented in C++, specifically focusing on sequential search and binary search algorithms. We began by generating random datasets and then utilized both arrays and linked lists to perform linear searches, analyzing the number of comparisons required to find a specific key. For the sequential search, we adapted our algorithms for both array and linked list structures, enabling us to identify the number of comparisons made to locate a target value. This exercise highlighted the inherent differences in performance between searching through arrays and linked lists, especially in terms of time complexity and operational efficiency. We then moved on to binary search, implementing both iterative and recursive approaches. The recursive binary search showcased the elegant nature of recursion in algorithm design, while also demonstrating its efficiency when searching through sorted datasets. Through visualizing the search process, we gained a deeper understanding of how the algorithm narrows down the search space with each iteration. Overall, this activity provided practical experience in algorithm implementation, enhancing our understanding of searching techniques in data structures. The hands-on coding allowed us to appreciate the complexities and efficiencies of different searching strategies, preparing us for future programming challenges in data structures and algorithms.
9. Assessment Rubric