| Hands-on Activity 6.1 | |
|---|---|
| **Searching Techniques** | |
| **Course Code:** CPE010 | **Program:** Computer Engineering |
| **Course Title:** Data Structures and Algorithms | **Date Performed:** 09 - 16 - 25 |
| **Section:** CPE21S4 | **Date Submitted:** 09 - 16 - 25 |
| **Name(s):** Santiago, David Owen A. | **Instructor:** Engr. Jimlord Quejado |

**6. Output**

Table 6.1. Data Generated and Observations.

**Code**:

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

const int max_size = 50;

int main() {

   //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] << " ";
   } std::cout << std::endl;


   return 0;
}
```

**Output**:

```
"D:\College\Year 2 CPE\Y2 Sem 1\CPE 010\Clion Files\Searching Techniques\cmake-build-debug\Clion_Files.exe"
15450 32534 17402 15798 26682 3763 8623 11321 32271 16477 28019 30689 6773 18515 11357 27647 3019 27267 17777 31507 1737
2 13729 8294 29320 3262 29815 8621 5791 1275 18812 10158 15018 9967 10407 27989 10618 24433 9093 30157 29210 30646 15242
 25995 27838 22129 11941 27791 3062 23615 29206

Process finished with exit code 0
```

This is a simple random number generator where each iteration gets stored inside an array with the max size. I also noticed that each digit has a random number of digits. The library cstdlib is used to include rand and time.h for the random seed.

Table 6-2a. Linear Search for Arrays

Code:

**Header File:**

```
//
// Created by David Owen Santiago on 16/09/2025.
//
#ifndef SEARCHING_H
#define SEARCHING_H

#include <iostream>

// Linear search function template
template <typename T>
bool linearSearch(const T data[], int size, T item) {
   for (int i = 0; i < size; ++i) {
       if (data[i] == item) {
           std::cout << "Searching is successful.\nFound at index: " << i << "\n";
           return true;
       }
   }
   std::cout << "Searching is not succesfull.\n";
   return false;
}

#endif // SEARCHING_H
```

**Main**:

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

const int max_size = 50;

int main() {

   //generate random values
   int dataset[max_size];
   srand(time(0));
   for(int i = 0; i < max_size; i++){
       dataset[i] = rand() % 90 + 10;
   }
   //show your datasets content
   for(int i = 0; i < max_size; i++){
       std::cout << dataset[i] << " ";
   } std::cout << std::endl;

   // Testing the function
   linearSearch(dataset, max_size, 45);

   return 0;
}
```

In this program, to ensure that the random numbers are smaller, I added the dataset[i] = rand() % 90 + 10; line to ensure that only random numbers with two digits will be inserted in the array. Then, in my header library, I included a bool function that does a linear search within an array. It takes three parameters: the array, its size, and the target value. It will then print out a corresponding message and index location if the target value is found or not.

Output:

```
"D:\College\Year 2 CPE\Y2 Sem 1\CPE 010\Clion Files\Searching Techniques\cmake-build-debug\Clion_Files.exe"
69 85 22 12 26 65 62 66 59 32 48 37 37 52 46 83 40 35 30 86 55 27 61 37 27 85 10 76 98 90 87 88 26 24 22 63 89 60 68 24
71 68 39 21 89 35 92 64 48 80
Searching is not succesfull.

Process finished with exit code 0
```

```
"D:\College\Year 2 CPE\Y2 Sem 1\CPE 010\Clion Files\Searching Techniques\cmake-build-debug\Clion_Files.exe"
75 41 71 19 98 96 88 71 31 26 51 21 23 45 60 87 69 87 85 89 99 77 15 49 40 55 83 77 68 78 53 97 46 40 83 88 49 89 60 13
90 44 90 78 95 50 70 70 31 19
Searching is successful.
Found at index: 13

Process finished with exit code 0
```

Table 6.2b. Linear Search for Linked List

Code:

Header File:

```
//
// Created by David Owen A. Santiago on 16/09/2025.
//
#ifndef SEARCHING_H
#define SEARCHING_H

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

// Linear search function template
template <typename T>
bool arrayLinearSearch(const T data[], int size, T item) {
   for (int i = 0; i < size; ++i) {
       if (data[i] == item) {
           std::cout << "Searching is successful.\nFound at index: " << i << "\n";
           return true;
       }
   }
   std::cout << "Searching is not successful.\n";
   return false;
}


// Added for linked list
template <typename T>
```

```
bool linkedLinearSearch(Node<T> *head, T item ) {
   int index = 0;
   Node<T> *temp = head;
   while (temp != nullptr) {
       if (temp->data == item) {
           std::cout << "Searching is successful.\nFound at node: " << index << "\n";
           return true;
       }
       temp = temp->next;
       ++index;
   }

   std::cout << "Searching is not successful.\n";
   return false;
}

#endif // SEARCHING_H
```

Main File:

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

const int max_size = 50;

void line() {
   std::cout << "-------------------------------\n";
}
int main() {

   Node<char> *name1 = new Node<char>('O');
   Node<char> *name2 = new Node<char>('W');
   Node<char> *name3 = new Node<char>('E');
   Node<char> *name4 = new Node<char>('N');

   name1->next = name2;
   name2->next = name3;
   name3->next = name4;
   name4->next = nullptr;

   std::cout << "Searching using linked list... \n";
   linkedLinearSearch(name1, 'O');
   linkedLinearSearch(name1, 'W');
   linkedLinearSearch(name1, 'E');
   linkedLinearSearch(name1, 'N');
   linkedLinearSearch(name1, 'D');

   return 0;
}
```

Output:

```
"D:\College\Year 2 CPE\Y2 Sem 1\CPE 010\Clion Files\Searching Techniques\cmake-build-debug\Clion_Files.exe"
Searching using linked list...
Searching is successful.
Found at node: 0
Searching is successful.
Found at node: 1
Searching is successful.
Found at node: 2
Searching is successful.
Found at node: 3
Searching is not successful.

Process finished with exit code 0
```

In this program, I implemented linear searching with linked lists this time. I used a separate header file for the node for consistency and modularity, which I also included in the searching.h header. Then, I created another bool function that takes the parameters: head and the target value. In the function, I created a temporary node pointer that holds the address of the head so that I can iterate through each node using a while loop. If the temp->data matches with the target value, then I return true, else I return false; all of which contain corresponding messages and index location.

Table 6.3a. Binary Search for Arrays

Code:

**Header File:**

Added code to the header file **searching.h**:

```
template <typename T>
bool arrayBinarySearch(const T arr[], int size, T item) {
   int low = 0;
   int up = size - 1;

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

       if (arr[mid] == item) {
           std::cout << "Search element is found at index: " << mid << "\n";
           return true;
       } else if (item < arr[mid]) {
           up = mid - 1;
       } else {
           low = mid + 1;
       }
   }

   std::cout << "Search element is not found.\n";
   return false;
}
```

**Main File:**

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

```cpp
#include "nodes.h"
#include "searching.h"

const int max_size = 50;

void line() {
   std::cout << "------------------------------\n";
}
int main() {

   int dataset[] = {1, 2, 3, 4, 5};

   arrayBinarySearch(dataset, sizeof(dataset) / sizeof(dataset[0]), 3);

   line();

   return 0;
}
```

Output:

```
"D:\College\Year 2 CPE\Y2 Sem 1\CPE 01
Search element is found at index: 2
------------------------------

Process finished with exit code 0
```

In this function, the arrayBinarySearch takes three parameters: the array, its size, and the target data. Then, in the function, we set the low to 0 (indicating the first element's index) and the up to the index of the last element. Then we calculate for the middle index then test its value if it matches with the target. If it doesn't we then compare it to two statements: if the target < middle value, then we update the top value to be the middle value and cycle again, only this time we're comparing all of the values to the left. However, if the target > middle value, then we update the low value to be the middle value then start the cycle again, except thai time we're comparing all of the values to the right. If the middle value never matches with the target, then it returns false and prints a corresponding error message.

Table 6.3b. Binary Search for Linked List

Code:

**Header File:**

**In searching.h**

```cpp
template <typename T>
Node<T>* getMiddle(Node<T>* start, Node<T>* end) {
   if (start == nullptr) return nullptr;
   Node<T>* slow = start;
```

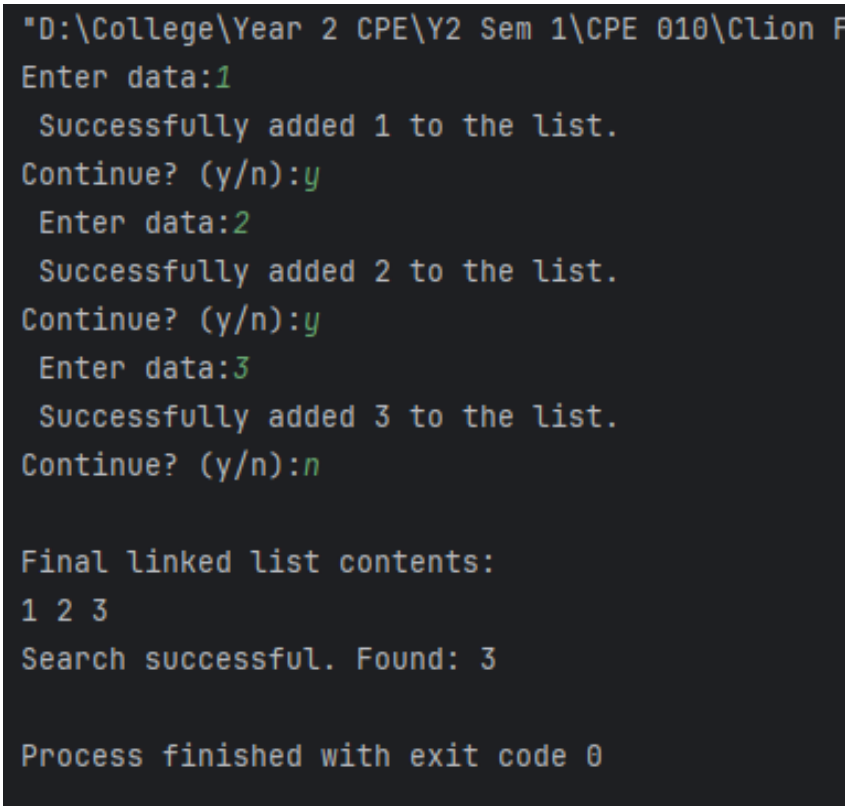
```
    Node<T>* fast = start;
    while (fast != end && fast->next != end) {
        fast = fast->next->next;
        slow = slow->next;
    }
    return slow;
}

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

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

        if (mid == nullptr) return nullptr;

        if (mid->data == key) {
            std::cout << "Search successful. Found: " << mid->data << "\n";
            return mid;
        } else if (key < mid->data) {
            end = mid;
        } else {
            start = mid->next;
        }
    }

    std::cout << "Search unsuccessful. Element not found.\n";
    return nullptr;
}
```

**Main File:**

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

const int max_size = 50;

void line() {
    std::cout << "--------------------------------\n";
}
int main() {
    char choice = 'y';
    int count = 1;
    int newData;
    Node<int>* temp, *head = nullptr, *node;
    Node<int> dummy;

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

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

```
            temp = head;
            while (temp->next != nullptr) {
                temp = temp->next;
            }
            node = dummy.new_node(newData);
            temp->next = node;
            std::cout << "Successfully added " << node->data << " to the list.\n";
        }
        count++;
        std::cout << "Continue? (y/n): ";
        std::cin >> choice;
        choice = std::tolower(choice);

        if (choice != 'y' && choice != 'n') {
            std::cout << "Invalid choice.\n";
            break;
        }
    }

    // Display the final linked list
    std::cout << "\nFinal linked list contents:\n";
    Node<int>* currNode = head;
    while (currNode != nullptr) {
        std::cout << currNode->data << " ";
        currNode = currNode->next;
    }
    std::cout << '\n';

    linkedBinarySearch(head, 3);

    // destructor
    currNode = head;
    while (currNode != nullptr) {
        Node<int>* nextNode = currNode->next;
        delete currNode;
        currNode = nextNode;
    }

    return 0;
}
```

Output:




In this program, the getMiddle function takes two parameters: start and end, which are pointers to nodes in the linked list. Inside this function, we first check if the starting node is nullptr. If it is, we simply return nullptr. We then declare two pointers, slow and fast, both initially pointing at start. Using the slow and fast pointer technique, slow moves one node at a time while fast moves two nodes at a time. This continues until fast reaches the end or the node before it. At that point, slow will be pointing to the middle node between start and end, which we then return.

In the linkedBinarySearch function, we begin with two pointers, start (pointing to the head of the list) and end (initially nullptr to represent the end of the list). We run a loop while start is not equal to end. Inside the loop, getMiddle is called to find the middle node between start and end. If the returned middle node is nullptr, we immediately return nullptr.

Next, we compare the key we are searching for with mid->data. If it matches, we print a success message and return the node. If the key is less than mid->data, we shift our search range to the left side and discard all right values by setting end to mid. However, if the key is greater than mid->data, we shift our search range to the right by setting start to mid->next. This continues until either we find the element or the loop finishes. If the loop exits without finding the key, we print an error message and return nullptr.

Lastly, in the main function, the user enters data to build the linked list. We do this by first checking if it is the very first node (creating the head) and then, for every next input, traversing to the end and adding the new node there. We also display the final linked list contents. We then call linkedBinarySearch to search for the target value. After finishing, I created a destructor code block to free the memory by deleting all nodes in the linked list to prevent memory leaks.

**7. Supplementary Activity**

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:**

**Header**:
Added functions in **searching.h**:

For Arrays

```
template <typename T>
int arrayLinearSearchComparisons(const T data[], int size, T item) {
   int comparisons = 0;
   for (int i = 0; i < size; ++i) {
       comparisons++;
       if (data[i] == item) break;
   }
   return comparisons;
}
```

For Linked List

```
template <typename T>
int linkedLinearSearchComparisons(Node<T>* head, T item) {
   int comparisons = 0;
   Node<T>* temp = head;
   while (temp != nullptr) {
       comparisons++;
       if (temp->data == item) break;
       temp = temp->next;
   }
   return comparisons;
}
```

Testing in **main:**

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

const int max_size = 50;

void line() {
   std::cout << "------------------------------\n";
}
int main() {
   int arr1[] = {15, 18, 2, 19, 18, 0, 8, 14, 19, 14, 18};
   int size1 = sizeof(arr1) / sizeof(arr1[0]);
```

```
    int key = 18;

    std::cout << "Array Linear Search for 18:\n";
    arrayLinearSearch(arr1, size1, key);

    int comparisons = arrayLinearSearchComparisons(arr1, size1, key);
    std::cout << "Comparisons done to find " << key <<": "<< comparisons << '\n';

    //std::cout << "Array occurrences of 18: " << countOccurrencesArray(arr1, size1, 18)
<< "\n";

    line();

    // Build linked list
    Node<int> dummy;
    Node<int>* head1 = nullptr;
    for (int i = 0; i < size1; ++i) {
        Node<int>* newNode = dummy.new_node(arr1[i]);
        if (!head1) head1 = newNode;
        else {
            Node<int>* temp = head1;
            while (temp->next) temp = temp->next;
            temp->next = newNode;
        }
    }
    std::cout << "Linked List Linear Search for 18:\n";
    linkedLinearSearch(head1, key);

    comparisons = linkedLinearSearchComparisons(head1, key);
    std::cout << "Comparisons done to find " << key <<": "<< comparisons << '\n';
    //std::cout << "Linked list occurrences of 18: " << countOccurrencesLinkedList(head1,
key) << "\n";


    // destructor
    Node<int>* temp = head1;
    while (temp!=nullptr) {
        Node<int>* newNode = temp->next;
        delete temp;
        temp = newNode;
    }

    return 0;
}
```

**Output:**

```
"D:\College\Year 2 CPE\Y2 Sem 1\CPE 01
Array Linear Search for 18:
Searching is successful.
Found at index: 1
Comparisons done to find 18: 2
------------------------------
Linked List Linear Search for 18:
Searching is successful.
Found at node: 1
Comparisons done to find 18: 2

Process finished with exit code 0
```

In this program, I added two new functions inside searching.h.

For arrays, the arrayLinearSearchComparisons function takes three parameters: the array, its size, and the item we want to find. Inside the function, we declare a variable comparisons and set it to 0. We then loop through each element of the array using a for loop, increasing comparisons by 1 every time we check an element. If we find the element we are looking for, we break. Else, the function returns the total number of comparisons done during the search.

For linked lists, the linkedLinearSearchComparisons function takes two parameters: the head of the list and the target value. We declare a variable comparisons and set it to 0. We then create a temporary pointer temp that starts at the head of the linked list. While temp is not nullptr, we increase comparisons by 1 each time we check a node. If the current node's data matches the item, we break. Otherwise, we move temp to the next node. After the loop ends, the function returns the total number of comparisons.

To test the functions, in main, we first create an array arr1 and a variable size1 to store its size. I then declared the key to search which is 18. Then, the function calls arrayLinearSearch to search for 18 in the array. After that, we call arrayLinearSearchComparisons to count how many comparisons were needed to find the key, and we print the result.

Next, the program builds a linked list with the same data as the array. This is done by looping through each element of the array, creating a new node with dummy.new_node, and then linking the new node to the end of the list. Once the list is built, we print a message and call linkedLinearSearch to search for the same key in the linked list. We then call linkedLinearSearchComparisons to count the number of comparisons done and print the result.

After finishing, I created a destructor code block to free the memory by deleting all nodes in the linked list to prevent memory leaks.

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:**

**Header:**
Added to **searching.h**

```
template <typename T>
int countOccurrencesArray(const T data[], int size, T item) {
    int count = 0;
    for (int i = 0; i < size; ++i)
        if (data[i] == item) count++;
    return count;
}
```

```
template <typename T>
int countOccurrencesLinkedList(Node<T>* head, T item) {
    int count = 0;
    Node<T>* temp = head;
    while (temp) {
        if (temp->data == item) count++;
        temp = temp->next;
    }
    return count;
}
```

Testing in **main:**

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

const int max_size = 50;

void line() {
    std::cout << "-------------------------------\n";
}
int main() {
    int arr1[] = {15, 18, 2, 19, 18, 0, 8, 14, 19, 14, 18};
    int size1 = sizeof(arr1) / sizeof(arr1[0]);
    int key = 18;

    std::cout << "Array Linear Search for 18:\n";
    arrayLinearSearch(arr1, size1, key);
    std::cout << "Array occurrences of 18: " << countOccurrencesArray(arr1, size1, 18) <<
"\n";

    line();

    // Build linked list
    Node<int> dummy;
    Node<int>* head1 = nullptr;
    for (int i = 0; i < size1; ++i) {
        Node<int>* newNode = dummy.new_node(arr1[i]);
        if (!head1) head1 = newNode;
        else {
            Node<int>* temp = head1;
```

```
            while (temp->next) temp = temp->next;
            temp->next = newNode;
        }
    }
    std::cout << "Linked List Linear Search for 18:\n";
    linkedLinearSearch(head1, key);
    std::cout << "Linked list occurrences of 18: " << countOccurrencesLinkedList(head1,
key) << "\n";


    // destructor
    Node<int>* temp = head1;
    while (temp!=nullptr) {
        Node<int>* newNode = temp->next;
        delete temp;
        temp = newNode;
    }

    return 0;
}
```

**Output:**

```
"D:\College\Year 2 CPE\Y2 Sem 1\CPE
Array Linear Search for 18:
Searching is successful.
Found at index: 1
Array occurrences of 18: 3
------------------------------
Linked List Linear Search for 18:
Searching is successful.
Found at node: 1
Linked list occurrences of 18: 3

Process finished with exit code 0
```

In this program, I implemented counting the occurrences of a specific value in both arrays and linked lists. I added two new template functions in the searching.h header to handle the counting separately for arrays and for linked lists.

For arrays, I created the countOccurrencesArray function that takes three parameters: the array, its size, and the target item to count. Inside the function, I declared an integer count and set it to 0. Then, I used a for loop to go through every element of the array. If the current element matches the target item, I increase count by one. After finishing the loop, I return the total count.

For linked lists, I created the countOccurrencesLinkedList function that takes two parameters: the head of the list and the target item to count. Inside the function, I declared an integer count and set it to 0. Then, I created a temporary node pointer starting at the head. Using a while loop, I moved through every node. If temp->data matches the target item, I increase count by one. After finishing the loop, I return the total count.

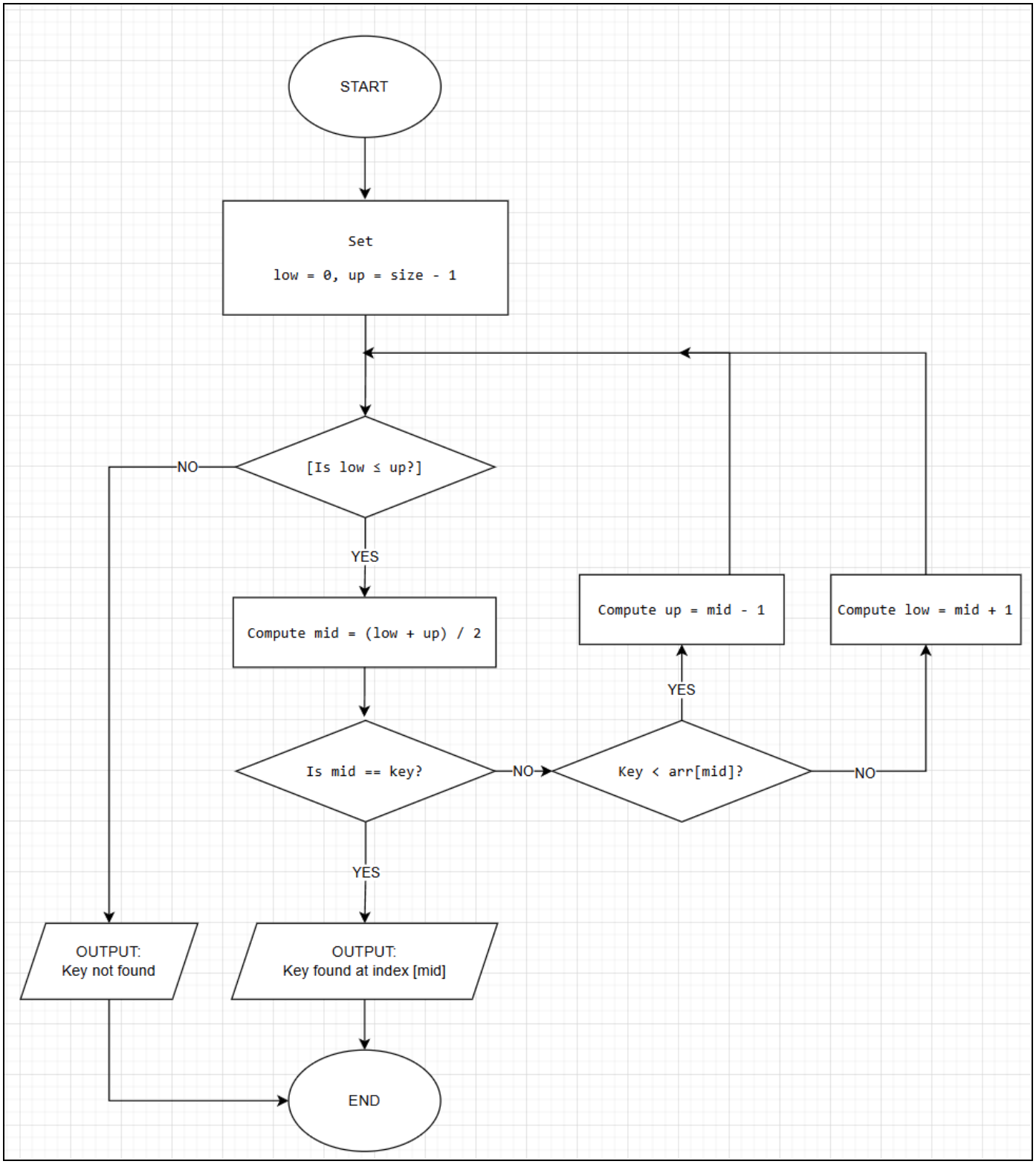
In main, I first created an integer array and stored its size. I picked the key value 18 to search for. Then, I called arrayLinearSearch to do a linear search on the array and printed its result. After that, I called countOccurrencesArray to print how many times 18 occurred in the array.

Next, I built a linked list using the same data as the array. I did this by looping through every element of the array, creating a new node with dummy.new_node, and linking it at the end of the list. Then, I called linkedLinearSearch to perform a linear search in the linked list and printed its result. After that, I called countOccurrencesLinkedList to print how many times 18 occurred in the linked list.

After finishing, I created a destructor code block to free the memory by deleting all nodes in the linked list to prevent memory leaks.

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 want 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.

**Diagram:**




I created a flowchart to clearly show how binary searching works. The values up and low are set first then the first condition is tested: if low < up, then it means the program has searched through each element and didn't find the target value so the program ends. Otherwise, it proceeds by computing the middle value. If it matches the target, then we print the target is found and end the program. Else, we move on to the next decision block that tests if the target or key is less than the value of arr[mid]. If yes, then the new up will be the middle value, indicating that we're discarding all values higher than the key at the right side. Otherwise, the new low will be the middle value indicating that we're discarding all values less than the key or at the left side. After either decision is satisfied, we return to the original

decision that tests if we've tested all values, if not, compute mid again, then repeat the loop until we either go through all the elements or find the target data.

**Code:**

**Header:**

```
template <typename T>
bool arrayBinarySearch(const T arr[], int size, T item) {
   int low = 0;
   int up = size - 1;

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

       if (arr[mid] == item) {
           std::cout << "Search element is found at index: " << mid << "\n";
           return true;
       } else if (item < arr[mid]) {
           up = mid - 1;
       } else {
           low = mid + 1;
       }
   }
```

In **main**

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

const int max_size = 50;

void line() {
   std::cout << "-------------------------------\n";
}
int main() {
   int arr1[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   int size1 = sizeof(arr1) / sizeof(arr1[0]);
   int key = 8;

   arrayBinarySearch(arr1, size1, key);

   return 0;
}
```

**Iterations:**

| Iteration | low | up | mid | arr[mid] | Action |
|---|---|---|---|---|---|
| 1 | 0 | 9 | 4 | 5 | key > arr[mid] → low = 5 |
| 2 | 5 | 9 | 7 | 8 | key > arr[mid] → low = 8 |
| 3 | 8 | 9 | 8 | 9 | key > arr[mid] → low = 9 |
| 4 | 9 | 9 | 9 | 10 | key == arr[mid] → found |

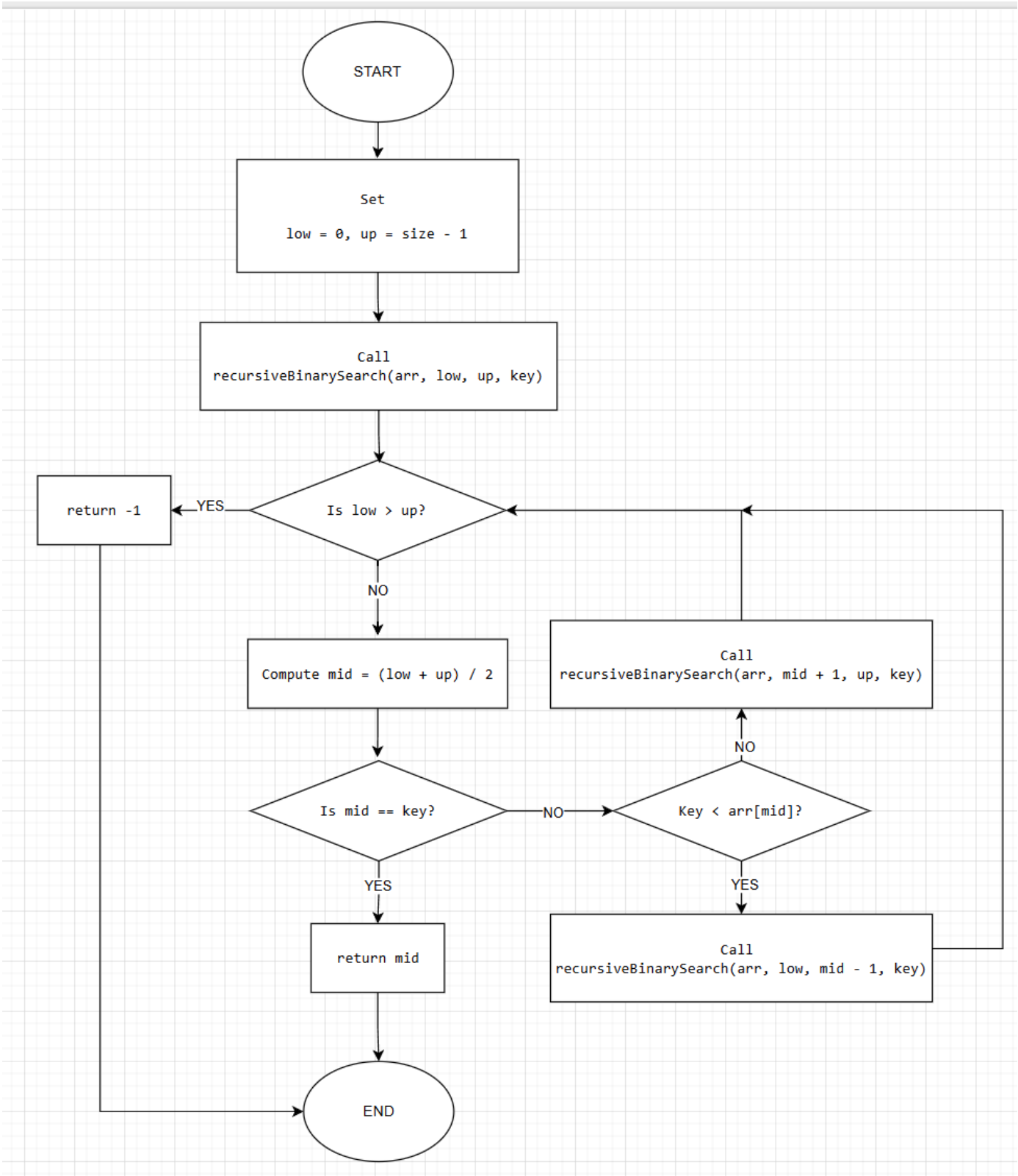
**Output:**

```
"D:\College\Year 2 CPE\Y2 Sem 1\CPE 010\Cli
Search element is found at index: 7

Process finished with exit code 0
```

Using the binary search algorithm logic, I applied it by using my existing array linear search function in the header file which confirms that it follows the diagram. I also included a table indicating the values per iteration in the low, up, mid, and arr[mid] as well as each direction the function goes to.

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.

**Diagram:**




This diagram is slightly different from the standard looping algorithm of the binary search array. In this case, there is only one loop toward the right side. The only way for the loop to return false is through the base case which is low > up which checks if we've gone through each element already. It mostly follows the same logic, only the loops are replaced by recursive calls which makes it faster and simpler than the standard iterating binary search algorithm.

**Code:**

**Header**

Added recursive binary search function in searching.h

```
template <typename T>
int recursiveBinarySearch(const T arr[], T low, T up, T key) {
   if (low > up) return -1;

   int mid = (low + up) / 2;

   if (arr[mid] == key)
       return mid;
   else if (key < arr[mid])
       return recursiveBinarySearch(arr, low, mid - 1, key);

   else
       return recursiveBinarySearch(arr, mid + 1, up, key);
}
```

In **main**

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

const int max_size = 50;

void line() {
   std::cout << "-------------------------------\n";
}
int main() {
   int arr1[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   int size1 = sizeof(arr1) / sizeof(arr1[0]);
   int key = 8;

   int result = recursiveBinarySearch(arr1, 0, size1 - 1 , key);

   if (result == -1) {
       std::cout << key << " not found\n";
   }
   else {
       std::cout << key << " found at index: " << result << '\n';
   }

   return 0;
}
```

| Call # | low | up | mid | arr[mid] | Comparison Result | Next Call Range |
|---|---|---|---|---|---|---|
| 1 | 0 | 9 | 4 | 5 | 8 > 5 ⟶ search right | low = 5, up = 9 |
| 2 | 5 | 9 | 7 | 8 | 8 == 8 ⟶ found | return 7 |

**Output:**

```
"D:\College\Year 2 CPE\Y2 Sem 1\CPE
8 found at index: 7

Process finished with exit code 0
```

As seen in the result as well as the table, only **two** recursive calls were made, much less than the **four** iterations that the standard looping algorithm does. This proves that recursive calls can be generally more efficient than standard looping and iteration.

## 8. Conclusion

In conclusion, this activity demonstrated and helped me learn how searching algorithms work. I was able to implement both array and linked list implementations of linear and binary search algorithms. Additionally, I was able to visualize how the function behaves when I created the two flowcharts in the supplementary activity that demonstrate their behaviour clearly. I learned the pros and cons of linear and binary searching; linear searching is best used in unsorted arrays since we have to check each element, however it is much slower and requires more iterations; binary searching is much faster and requires less iterations or recursions and is best used in sorted arrays because, as its con, it has the possibility of skipping the value we're looking for if we search through an unsorted array. Overall, this was a deep dive into the logic of searching algorithms that I was able to appreciate and apply.

## 9. Assessment Rubric