| Hands-on Activity 6.1 | |
|---|---|
| **Searching Techniques** | |
| **Course Code:** CPE010 | **Program:** Computer Engineering |
| **Course Title:** Data Structures and Algorithms | **Date Performed:** 10/15/24 |
| **Section:** CPE21S4 | **Date Submitted:** 10/15/24 |
| **Name(s): CALVIN EARL PLANTA** | **Instructor:** Prof. Sayo |

**6. Output**

**Code**

```cpp
#include <iostream>
#include <cstdlib>
#include <time.h>
using namespace std;

const int max_size = 50;

int main() {
    int dataset[max_size];
    srand(time(0));

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

    for(int i = 0; i < max_size; i++) {
        cout << dataset[i] << " ";
    }
    cout << endl;

    return 0;
}
```

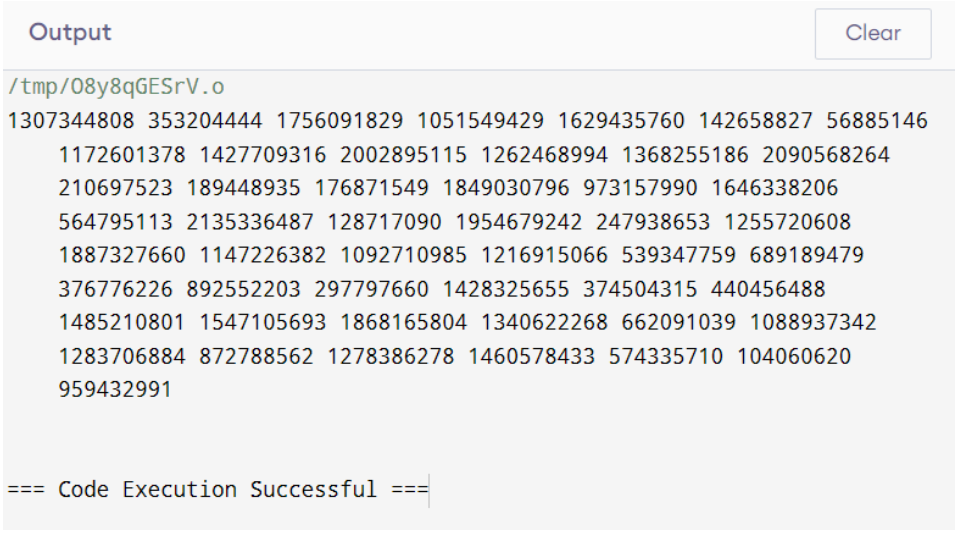
| | |
|---|---|
| **Output** |  |
| **Observation** | The code illustrates binary search for sorted arrays and sequential search for arrays and linked lists. It produces a linked list, generates random data, and outputs search results and comparison counts in an understandable manner. |

**Table 6-1. Data Generated and Observations**

**Code**

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

int dataset(int arr[], int size, int key) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == key) {
            return i;
        }
    }
    return -1;
}

int main() {
    int size;
    cout << "Enter the size of the array: ";
    cin >> size;

    int arr[size];

    srand(time(0));

    cout << "Randomly generated array: ";
    for (int i = 0; i < size; i++) {
        arr[i] = rand() % 100;
        cout << arr[i] << " ";
    }
    cout << endl;

    int key;
    cout << "Enter the number to search for: ";
    cin >> key;

    int result = dataset(arr, size, key);

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

    return 0;
}
```

| | |
|---|---|
| **Output** | Output<br>/tmp/906Ate2H9E.o<br>Enter the size of the array: 10<br>Randomly generated array: 15 64 89 6 45 82 22 58 38 22<br>Enter the number to search for: 82<br>Element found at index: 5<br><br>=== Code Execution Successful === |
| **Observation** | Linear search has a time complexity of O(n), meaning that in the worst case, the program needs to check each element of the array until the target value is found (or not found). This means for large datasets, the program may take more time compared to more efficient search algorithms like binary search, but binary search requires a sorted array. |

**Table 6-2a. Linear Search for Arrays**

**Code**

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

struct Node {
    int data;
    Node* next;
};

Node* createNode(int value) {
    Node* newNode = new Node;
    newNode->data = value;
    newNode->next = nullptr;
    return newNode;
}

void appendNode(Node*& head, int value) {
    Node* newNode = createNode(value);
    if (head == nullptr) {
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

int linearSearch(Node* head, int target) {
    Node* temp = head;
    int index = 0;

    while (temp != nullptr) {
        if (temp->data == target) {
            return index;
        }
        temp = temp->next;
        index++;
    }

    return -1;
}

void printList(Node* head) {
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " -> ";
        temp = temp->next;
```

```
    }
    cout << "nullptr" << endl;
}

int main() {
    const int size = 10;
    Node* head = nullptr;

    srand(time(0));

    cout << "Linked list elements: ";
    for (int i = 0; i < size; i++) {
        int randomValue = rand() % 100 + 1;
        appendNode(head, randomValue);
    }

    printList(head);

    int target;
    cout << "Enter the number to search for: ";
    cin >> target;

    int result = linearSearch(head, target);

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

    return 0;
}
```

**Output**

```
Output                                                    Clear

/tmp/D7qV3KyER6.o
Linked list elements: 93 -> 94 -> 21 -> 28 -> 4 -> 86 -> 83 -> 82 -> 25 -> 65 -> nullptr
Enter the number to search for: 4
Element found at index 4


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

| **Observation** | Similar to the array-based search, the program will only return the index of the first occurrence of the target element. If duplicates exist, the search stops at the first match. |
|---|---|

**Table 6-2b. Linear Search for Linked List**

**Code**

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <algorithm>
using namespace std;

int binarySearch(int arr[], int size, int target) {
    int left = 0;
    int right = size - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target)
            return mid;

        if (arr[mid] < target)
            left = mid + 1;

        else
            right = mid - 1;
    }

    return -1;
}

int main() {
    const int size = 20;
    int arr[size];

    srand(time(0));

    cout << "Unsorted array elements: ";
    for (int i = 0; i < size; i++) {
```

```
    for (int i = 0; i < size; i++) {
        arr[i] = rand() % 100 + 1;
        cout << arr[i] << " ";
    }
    cout << endl;

    sort(arr, arr + size);

    cout << "Sorted array elements: ";
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    int target;
    cout << "Enter the number to search for: ";
    cin >> target;

    int result = binarySearch(arr, size, target);

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

    return 0;
```

**Output**

```
Output                                                    Clear

/tmp/4GQPzlrlk6.o
Unsorted array elements: 85 43 60 19 44 55 86 16 86 70 79 21 65 81 25 45 66 7 86 19
Sorted array elements: 7 16 19 19 21 25 43 44 45 55 60 65 66 70 79 81 85 86 86 86
Enter the number to search for: 21
Element found at index 4


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

**Observation**

Binary search can only be applied to sorted data. The array is sorted using sort() in O(n log n) time. If the dataset is not sorted beforehand, binary search would give incorrect results.

**Table 6-3a. Binary Search for Arrays**

**Code**

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>

struct Node {
    int data;
    Node* next;
};

Node* createNode(int value) {
    Node* newNode = new Node;
    newNode->data = value;
    newNode->next = nullptr;
    return newNode;
}

void appendNode(Node*& head, int value) {
    Node* newNode = createNode(value);
    if (head == nullptr) {
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

void printList(Node* head) {
    Node* temp = head;
    while (temp != nullptr) {
        std::cout << temp->data << " -> ";
        temp = temp->next;
```

```
    }
    std::cout << "nullptr" << std::endl;
}

int countNodes(Node* head) {
    int count = 0;
    Node* temp = head;
    while (temp != nullptr) {
        count++;
        temp = temp->next;
    }
    return count;
}

Node* getNodeAtIndex(Node* head, int index) {
    Node* temp = head;
    for (int i = 0; i < index && temp != nullptr; i++) {
        temp = temp->next;
    }
    return temp;
}

int binarySearch(Node* head, int target) {
    int left = 0;
    int right = countNodes(head) - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        Node* midNode = getNodeAtIndex(head, mid);

        if (midNode == nullptr) {
            return -1;
        }

```

```
        if (midNode->data == target) {
            return mid;
        } else if (midNode->data < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return -1;
}

int main() {
    const int size = 10;
    Node* head = nullptr;

    srand(time(0));

    for (int i = 0; i < size; i++) {
        int randomValue = rand() % 100 + 1;
        appendNode(head, randomValue);
    }

    Node* sortedHead = nullptr;
    Node* current = head;
    while (current != nullptr) {
        Node* nextNode = current->next;
        if (sortedHead == nullptr || sortedHead->data >= current->data) {
            current->next = sortedHead;
            sortedHead = current;
        } else {
            Node* temp = sortedHead;
            while (temp->next != nullptr && temp->next->data < current->data) {
                temp = temp->next;
            }
            current->next = temp->next;
            temp->next = current;
        }
        current = nextNode;
    }

    std::cout << "Sorted linked list: ";
    printList(sortedHead);

    int target;
    std::cout << "Enter the number to search for: ";
    std::cin >> target;

    int result = binarySearch(sortedHead, target);

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

    return 0;
```

| | |
|---|---|
| **Output** | Output Clear<br>/tmp/lyaXAOh3qv.o<br>Sorted linked list: 10 -> 20 -> 23 -> 67 -> 78 -> 79 -> 80 -> 87 -> 89 -> 96 -> nullptr<br>Enter the number to search for: 96<br>Element found at index 9<br><br>=== Code Execution Successful === |
| **Observation** | Binary Search requires a sorted dataset, and the program includes a sorting step to ensure the linked list is sorted. However, this adds additional complexity and time cost ($O(n^2)$ using insertion sort). Therefore, sorting a linked list just for Binary Search is impractical. |

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

## 7. Supplementary Activity

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 it be necessary to find the key '18'?

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

struct Node {
    int data;
    Node* next;
};

void appendNode(Node*& head, int value) {
    Node* newNode = new Node;
    newNode->data = value;
    newNode->next = nullptr;

    if (head == nullptr) {
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

int linearSearchArray(int arr[], int size, int target, int& comparisons) {
    for (int i = 0; i < size; i++) {
        comparisons++;
        if (arr[i] == target)
            return i;
    }
    return -1;
}

int linearSearchLinkedList(Node* head, int target, int& comparisons) {
```

```
    Node* temp = head;
    int index = 0;

    while (temp != nullptr) {
        comparisons++;
        if (temp->data == target)
            return index;
        temp = temp->next;
        index++;
    }

    return -1;
}

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

    int comparisonsArray = 0;
    int resultArray = linearSearchArray(arr, size, target, comparisonsArray);

    if (resultArray != -1)
        cout << "Element found in array at index " << resultArray << " after " <<
            comparisonsArray << " comparisons." << endl;
    else
        cout << "Element not found in array after " << comparisonsArray << " comparisons
            ." << std::endl;

    Node* head = nullptr;
    for (int i = 0; i < size; i++) {
        appendNode(head, arr[i]);
    }

    int comparisonsLinkedList = 0;
    int resultLinkedList = linearSearchLinkedList(head, target, comparisonsLinkedList);

    if (resultLinkedList != -1)
        cout << "Element found in linked list at index " << resultLinkedList << " after
            " << comparisonsLinkedList << " comparisons." << endl;
    else
        cout << "Element not found in linked list after " << comparisonsLinkedList << "
            comparisons." << endl;

    return 0;
}
```

Output

```
/tmp/uq2GAUN4xr.o
Element found in array at index 1 after 2 comparisons.
Element found in linked list at index 1 after 2 comparisons.


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

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

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

struct Node {
    int data;
    Node* next;
};

void appendNode(Node*& head, int value) {
    Node* newNode = new Node;
    newNode->data = value;
    newNode->next = nullptr;

    if (head == nullptr) {
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

int linearSearchArray(int arr[], int size, int target) {
    int count = 0;
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {
            count++;
        }
    }
    return count;
}

```

```cpp
int linearSearchLinkedList(Node* head, int target) {
    int count = 0;
    Node* temp = head;

    while (temp != nullptr) {
        if (temp->data == target) {
            count++;
        }
        temp = temp->next;
    }

    return count;
}

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

    int countArray = linearSearchArray(arr, size, target);
    cout << "Element " << target << " found " << countArray << " times in the array." <<
        endl;

    Node* head = nullptr;
    for (int i = 0; i < size; i++) {
        appendNode(head, arr[i]);
    }

    int countLinkedList = linearSearchLinkedList(head, target);
    cout << "Element " << target << " found " << countLinkedList << " times in the
        linked list." << endl;

    return 0;
```

Output

```
/tmp/YQz1d1QRBB.o
Element 18 found 2 times in the array.
Element 18 found 2 times in the linked list.


=== 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.

Initial list: [3, 5, 6, 8, 11, 12, 14, 15, 17, 18]
Low Index: 0
High Index: 9
Mid Index: (0 + 9) / 2 = 4 (integer division)
Mid Value: 11

Iteration 1: New Range: [3, 5, 6, 8]
Low Index: 0
High Index: 3
Mid Index: (0 + 3) / 2 = 1
Mid Value: 5

Iteration 2: New Range: [6, 8]

Low Index: 2

High Index: 3

Mid Index: (2 + 3) / 2 = 2

Mid Value: 6

Iteration 3: New Range: [8]

Low Index: 3

High Index: 3

Mid Index: (3 + 3) / 2 = 3

Mid Value: 8

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

int binarySearch(int arr[], int size, int target) {
    int left = 0;
    int right = size - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        cout << "Current Range: [" << left << ", " << right << "], Mid Index: " << mid
            << ", Mid Value: " << arr[mid] << endl;

        if (arr[mid] == target) {
            return mid;
        }

        if (arr[mid] < target) {
            left = mid + 1;
        }
        else {
            right = mid - 1;
        }
    }

    return -1;
}

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

    int result = binarySearch(arr, size, target);

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

    return 0;
}
```

Output

```
/tmp/OoCFVrlmTP.o
Current Range: [0, 9], Mid Index: 4, Mid Value: 11
Current Range: [0, 3], Mid Index: 1, Mid Value: 5
Current Range: [2, 3], Mid Index: 2, Mid Value: 6
Current Range: [3, 3], Mid Index: 3, Mid Value: 8
Element found at index 3.


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

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;

int recursiveBinarySearch(int arr[], int left, int right, int target) {
    if (left > right) {
        return -1;
    }

    int mid = left + (right - left) / 2;

    if (arr[mid] == target) {
        return mid;
    }

    if (arr[mid] < target) {
        return recursiveBinarySearch(arr, mid + 1, right, target);
    }
    else {
        return recursiveBinarySearch(arr, left, mid - 1, target);
    }
}

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

    int result = recursiveBinarySearch(arr, 0, size - 1, target);

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

Output

```
/tmp/swnzPIDIuw.o
Element found at index 3.


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

**8. Conclusion**

Both linear search and binary search have their specific use cases in C++. Linear search is best for small or unsorted datasets, while binary search is highly efficient for large, sorted datasets. Understanding the data structure and whether or not sorting is possible will help determine the most appropriate search technique to apply in a given scenario.