XIAMEN UNIVERSITY MALAYSIA

Course Code	:	CST204
Course Name	:	Data Structures
Lecturer	:	Dr.Raja Majid Mehmood
Academic Session	:	2023/04
Assessment Title	:	Assignment- Sorting, Search, and Graph Algorithm
Submission Due Date	:	28/12/2023

Prepared by :

Student ID	Student Name
CST2209134	Bai Rui

Date Received :

Feedback from Lecturer:	
	Mark:

XIAMEN UNIVERSITY MALAYSIA

Own Work Declaration

I/We hereby understand my/our work would be checked for plagiarism or other misconduct, and the softcopy would be saved for future comparison(s).

I/We hereby confirm that all the references or sources of citations have been correctly listed or presented and I/we clearly understand the serious consequence caused by any intentional or unintentional misconduct.

This work is not made on any work of other students (past or present), and it has not been submitted to any other courses or institutions before.

Signature:

Date:20/12/2023

Task 1

Functionality Demonstration

a. Stack-Based Sorting Program

The program implemented a stack-based sorting algorithm. The primary stack st contains the elements to be sorted, and the temporary stack tmpst is used to facilitate the sorting process.

```
cout << "(2) The process of sorting elements of stack st is: " << endl;
// Sorting algorithm using two stacks
while (!st.is_empty()) {
    int tmp = st.get_top();  // Get the top element from the main stack
    st.pop();                // Pop it from the main stack
    cout << "st: pop out " << tmp << "=>" << endl;

    // Move elements from the temporary stack back to the main stack if they are greater than the current element
    while (!tmpst.is_empty() && tmpst.get_top() > tmp) {
        cout << "tmpst: get the top element " << tmpst.get_top() << "." << endl;
        cout << "Since " << tmpst.get_top() << ">" << tmp << " tmpst: pop out " << tmpst.get_top() << "." << endl;
        st.push(tmpst.get_top());
        tmpst.pop();
        cout << "st: push " << tmp << "." << endl;
    }
    tmpst.push(tmp);  // Push the current element onto the temporary stack
    cout << "tmpst: push " << tmp << "." << endl;
}

// Move all elements back to the main stack
while (!tmpst.is_empty()) {
    st.push(tmpst.get_top());
    tmpst.pop();
}
```

b. Input Handling

The user is prompted to enter the number of elements and the actual elements. For this discussion, the input provided was 85, 75, 95, 80.

```
cout << "Please Input The Number of Elements: ";
cin >> num;
stack tmpst(num);  // Temporary stack used for sorting
stack st(num);     // Main stack

cout << "Please Input " << num << " Elements" << endl;
cout << "(1)push the elements ";
while (num--) {
    cin >> temp;
    st.push(temp);  // Push elements onto the main stack
}
cout << "onto the stack list" << endl;
```

c. Push and Pop Operations

The program output demonstrates a series of push and pop operations:

The elements are pushed onto st in the order they are entered.

To sort, the program pops elements from st, compares them with the top of tmpst, and selectively pushes elements between the two stacks to ensure they are sorted in tmpst.

Every push and pop operation is printed in a format that clearly shows the action and the value involved.

```
// Move elements from the temporary stack back to the main stack if they are greater than the current element
while (!tmpst.is_empty() && tmpst.get_top() > tmp) {
    cout << "tmpst: get the top element " << tmpst.get_top() << "." << endl;
    cout << "Since " << tmpst.get_top() << ">" << tmp << " tmpst: pop out " << tmpst.get_top() << "." << endl;
    st.push(tmpst.get_top());
    tmpst.pop();
    cout << "st: push " << tmp << "." << endl;
}
tmpst.push(tmp);  // Push the current element onto the temporary stack
cout << "tmpst: push " << tmp << "." << endl;
```

d. Final Sorted Output

Once the sorting is complete, the elements in tmpst are in descending order. These elements are then popped from tmpst and pushed back onto st, resulting in st having the elements in ascending order with the smallest element on top.

```
    cout << "(3)the sorting of stack st ends" << endl;
    cout << "(4)the popping sequence of st is:";

    // Print the elements in sorted order
    while (!st.is_empty()) {
        cout << st.get_top() << " ";
        st.pop();
    }
    cout << endl;
```

```
Please Input 4 Elements
(1)push the elements 85 75 95 80
onto the stack list
```

The final sorted stack is printed to the screen, confirming the successful sorting of elements. In the case of the input provided, the final output from bottom to top is 95, 85, 80, 75, which satisfies the requirement of having the smallest element at the top.

Result

The program's output, as demonstrated, accurately reflects the sequence of operations and the final sorted state of the stack. The algorithm is effective for the given task, using only the allowed data structures and producing the correct output. This result highlights the importance of algorithm design when working with constraints and the utility of simple data structures in achieving complex tasks.

```
Please Input 4 Elements
(1)push the elements 85 75 95 80
onto the stack list
(2) The process of sorting elements of stack st is:
st: pop out 80=>
tmpst: push 80.
st: pop out 95=>
tmpst: push 95.
st: pop out 75=>
tmpst: get the top element 95.
Since 95>75 tmpst: pop out 95.
st: push 75.
tmpst: get the top element 80.
Since 80>75 tmpst: pop out 80.
st: push 75.
tmpst: push 75.
st: pop out 80=>
tmpst: push 80.
st: pop out 95=>
tmpst: push 95.
st: pop out 85=>
tmpst: get the top element 95.
Since 95>85 tmpst: pop out 95.
st: push 85.
tmpst: push 85.
st: pop out 95=>
tmpst: push 95.
(3)the sorting of stack st ends
(4)the popping sequence of st is:75 80 85 95
```

Discussion

The program's approach to stack-based sorting showcases a straightforward but effective method of sorting with limited data structures. The logic behind using two stacks involves comparing the top elements and arranging them such that the smallest

element bubbles up to the top of st. The algorithm is comparable to a selection sort, where the smallest elements are selected and moved to their correct position iteratively.

```
// Definition of the stack class
class stack {
private:
    int* arr;       // Pointer to dynamically allocated array for stack elements
    int arrsize;        // Current size (number of elements) of the stack
    int capacity;    // Maximum capacity of the stack
    int top;         // Index of the top element in the stack

public:
    // Constructor for stack with a specified capacity
    stack(int new_capacity) {
        arr = new int[new_capacity];  // Allocate memory for stack elements
        capacity = new_capacity;       // Set the capacity
        arrsize = 0;                     // Initialize size
        top = -1;                      // Initialize top index (empty stack)
    }

    // Destructor for stack
    ~stack() {
        delete[] arr;  // Release dynamically allocated memory
    }

    // Method to push a value onto the stack
    void push(int val) {
        if (arrsize == capacity) {
            cout << "The Stack is Full" << endl;  // Check for stack overflow
            return;
        }
```

```
    // Method to push a value onto the stack
    void push(int val) {
        if (arrsize == capacity) {
            cout << "The Stack is Full" << endl;  // Check for stack overflow
            return;
        }
        else {
            top = (top + 1) % capacity;  // Update the top index
            arr[top] = val;              // Add the new value
            arrsize++;                      // Increment size
        }
    }

    // Method to get the top element of the stack
    int get_top() {
        if (is_empty()) {
            cout << "The stack is empty" << endl;
        }
        else {
            return arr[top];  // Return the top element
        }
    }
```

The implementation is robust in handling user input and provides a clear and detailed account of the sorting process through its output statements. One limitation, however, is that the algorithm's time complexity is O(n^2) due to the nested operations when

sorting elements between the two stacks. For larger inputs, this could lead to performance issues.

Task 2

Functionality Demonstration

a.Sparse Graph (Case-A)

The sparse graph is created with 1000 vertices and 10,000 edges. This represents a scenario where each vertex is connected to, on average, 10 other vertices. Sparse graphs are common in situations like road networks or social networks where not every node is connected to every other node.

b.Dense Graph (Case-B)

The dense graph also contains 1000 vertices but has 360,000 edges. This means that each vertex is connected to many other vertices, making the graph dense. Dense graphs are typical in telecommunications networks or in representing relationships in datasets where many nodes are interconnected.

c. Complete Graph (Case-C)

The complete graph is the densest of all, with 1000 vertices and 1,000,000 edges, where each vertex is connected to every other vertex exactly once. This is a special case of a dense graph and is used in scenarios where every node must interact with every other node, such as in fully connected neural networks or complete tournament schedules.

d. how to count total number of edges in graph

The countEdges method that iterates through each vertex adjacency list to tally the total number of edges in the graph.

```
// Count the total number of edges in the graph
int countEdges() {
    int count = 0;
    for (int i = 0; i < adjList.arrsize(); i++) {
        count += adjList[i].arrsize();
    }
    return count;
}
```

Results

Upon running the program, it initializes three separate graphs with the specifications outlined above. The rand() function is utilized to assign random weights and establish connections between vertices to simulate the edges.

```
int main() {
    srand(time(NULL));  // Seed the random number generator
    Graph A(1000, 100 * 100);   // Sparse Graph with 100x100 edges
    Graph B(1000, 600 * 600);   // Dense Graph with 600x600 edges
    Graph C(1000, 1000 * 1000); // Complete Graph with 1000x1000 edges

    // Print the number of edges in each graph
    cout << "Graph\tA\tB\tC\n";
    cout << "Edges\t" << A.countEdges() << "\t" << B.countEdges() << "\t" << C.countEdges() << "\n";
```

```
Graph   A       B       C
Edges   10000   360000  1000000
```

Discussion

The program successfully demonstrates the versatility of adjacency lists in representing graphs of varying densities. The use of a template class for the list and the internal node structure is a classic approach in C++ to create a linked list that can store any data type.

```cpp
// Template class List for creating a linked list
template <class T>
class List {
private:
    // Internal node structure
    struct Node {
        T arr;       // Data stored in the node
        Node* next;   // Pointer to the next node
        Node(T d) : arr(d), next(NULL) {}  // Node constructor
    };
    Node* head;  // Head node of the list

public:
    List() : head(NULL) {}  // Constructor initializes an empty list

    // Add a new node at the beginning of the list
    void push(T arr) {
        Node* newNode = new Node(arr); // Create a new node
        newNode->next = head;          // New node points to the original head node
        head = newNode;                // Update head to the new node
    }
```

```cpp
    // Return the size of the list
    int arrsize() {
        Node* temp = head;
        int num = 0;
        while (temp) {
            temp = temp->next;
            num++;
        }
        return num;
    }

    // Destructor to free memory used by the list
    ~List() {
        Node* current = head;
        while (current != NULL) {
            Node* next = current->next;
            delete current;
            current = next;
        }
    }

    // Overload the [] operator to access elements in the list
    T& operator [] (int index) {
        Node* temp = head;
        for (int i = 0; i < index; i++) {
            temp = temp->next;
        }
        return temp->arr; // Return the data at the given index
    }
};
```

The Graph class methods such as addVertex() and addEdge() are straightforward and perform their intended tasks efficiently. The countEdges() function iterates over each vertex's adjacency list to tally the total number of edges, providing a clear and accurate count for each graph case.

```
// Graph class representing a directed, weighted graph using an adjacency list
class Graph {
private:
    // Internal structure to represent an edge
    struct Edge {
        int weight;  // Weight of the edge
        int dest;    // Destination vertex
        Edge(int w, int d) : weight(w), dest(d) {} // Edge constructor
    };
    List<List<Edge>> adjList; // Adjacency list to store the edges

public:
    // Add a vertex to the graph
    void addVertex() {
        List<Edge> newVertex;
        adjList.push(newVertex);
    }

    // Add an edge to the graph
    void addEdge(int u, int v, int w) {
        Edge e(w, v);       // Create an edge
        adjList[u].push(e); // Add the edge to the adjacency list
    }
```

```
    // Count the total number of edges in the graph
    int countEdges() {
        int count = 0;
        for (int i = 0; i < adjList.arrsize(); i++) {
            count += adjList[i].arrsize();
        }
        return count;
    }

    // Constructor to create a graph with specified number of vertices and edges
    Graph(int num_vertices, int num_edges) {
        for (int i = 0; i < num_vertices; i++) {
            addVertex();
        }
        for (int i = 0; i < num_edges; i++) {
            int u = rand() % num_vertices; // Randomly choose a source vertex
            int v = rand() % num_vertices; // Randomly choose a destination vertex
            int w = rand() % 100;          // Randomly assign a weight
            addEdge(u, v, w);              // Add the edge to the graph
        }
    }
};
```

One point to note is that the program does not ensure the uniqueness of edges, which may lead to multiple edges between two vertices. This might not be an issue for a directed graph, but if the graph were undirected, it would be necessary to check for existing edges before adding a new one.

Task 3

Functionality Demonstration

The program is designed to perform the following tasks:

a. Generate Random Data

The generateRandom() function initializes the global array arr with random numbers. It attempts to ensure that there are no duplicate entries, although the current implementation may not correctly prevent duplicates because the fallback strategy (when a duplicate is found) is deterministic and could lead to a predictable pattern of values.

```
// Initialize array with random data while avoiding duplicates
void generateRandom() {
    for (long i = 0; i < arrsize; i++) {
        long x = rand() % arrsize;
        // Check if the number is already in the array to avoid duplicates
        arr[i] = findByBinSearch(x) != -1 ? x : arrsize + i;
    }
}
```

b. Sort the Array

The doSortedList() function sorts the array using the C++ Standard Library's sort function, preparing it for binary search.

```
// Sort the array using the built-in sort algorithm
void doSortedList() {
    sort(arr, arr + arrsize);
}
```

c. Perform Searches

Both findBySeqSearch() and findByBinSearch() functions perform searches on the sorted array and count the number of comparisons (or array accesses in the case of sequential search) made until the key is found or determined to be absent.

```
// Implement binary search and count the number of comparisons at each step
int findByBinSearch(int key) {
    int count = 0; // Count of comparisons
    long left = 0, right = arrsize - 1, mid;
    while (left <= right) {
        count++; // Increment count for each comparison
        mid = (left + right) / 2;
        if (arr[mid] == key) {
            return count; // Return count if key is found
        }
        else if (arr[mid] < key) {
            left = mid + 1;
        }
        else {
            right = mid - 1;
        }
    }
    return -1; // Return -1 if key is not found
}

// Implement sequential search and count the number of accesses at each step
int findBySeqSearch(int key) {
    int count = 0; // Count of accesses
    for (long i = 0; i < arrsize; i++) {
        count++; // Increment count for each access
        if (arr[i] == key) {
            return count; // Return count if key is found
        }
    }
    // If not found, return -1
    return -1;
}
```

d. Print Results and Analyze Costs

The main() function runs the entire process for arrays of increasing sizes and prints out the search costs for both algorithms. The search cost is classified into the best, average, or worst case based on the number of comparisons or accesses required to find the key.

```
int main() {
    srand(time(NULL)); // Seed the random number generator
    cout << "Size\t\tindex\tComplexity-Sequential\tComplexity-Binary\tSeq.Cost\tBinary Cost" << endl;
    // Loop over different sizes of the array
    for (arrsize = 10; arrsize < 10000000; arrsize *= 10) {
        arr = new long[arrsize]; // Dynamically allocate memory for the array
        generateRandom(); // Fill the array with random numbers
        doSortedList(); // Sort the array
        long index = rand() % arrsize; // Choose a random index for the key
        long key = arr[index]; // The key to be searched
        int seqCost = findBySeqSearch(key); // Perform sequential search and get the cost
        int binCost = findByBinSearch(key); // Perform binary search and get the cost
        // Print the results
        cout << arrsize << "\t\t" << index << "\t\t" << seqCost << "\t\t" << binCost << "\t\t\t";
```

Results

When executed, the program demonstrates the fundamental differences in performance between sequential and binary search algorithms.

Size	index	Complexity-Sequential	Complexity-Binary	Seq.Cost	Binary Cost
10	0	1	3	best	average
100	78	79	6	average	average
1000	31	32	9	average	average
10000	5266	5267	14	average	worst
100000	29937	29938	17	average	worst
1000000	2658	2659	19	average	average

Discussion

Sequential Search

Best Case: Occurs when the key is at the beginning of the array. The best-case cost is always 1.

Worst Case: Occurs when the key is at the end of the array or not present. The worst-case cost is equal to the array size.

Average Case: On average, the cost is approximately half of the array size, as it's equally likely for the key to be anywhere in the array.

```
// Determine the case for sequential search (best, worst, average)
if (seqCost == 1)
{
    cout << "best\t\t";
}
else if (seqCost == arrsize)
{
    cout << "worst\t\t";
}
else
{
    cout << "average\t\t";
}
```

Binary Search

Best Case: Occurs when the key is at the middle of the array, where the first comparison finds the key. The best-case cost is 1.

Worst Case: Occurs when the algorithm reaches the maximum number of divisions possible for the array size, which is log2(n)+1, rounded down to the nearest whole number.

Average Case: Varies, but it will always be significantly less than the worst case and is logarithmic in relation to the array size.

```
// Determine the case for binary search (best, worst, average)
if (binCost == 1)
{
    cout << "best" << endl;
}
else if (binCost == floor(log2(arrsize) + 1))
{
    cout << "worst" << endl;
}
else
{
    cout << "average" << endl;
}
```

The program's output would reflect these complexities, showing the stark contrast in efficiency between binary and sequential search, especially as the array size grows. Binary search's logarithmic complexity means it scales well with larger data sets, while sequential search's linear complexity becomes impractical as data size increases.

APPENDIX 1

MARKING RUBRICS

Component Title	Assignment: Tasks 1 to 3		**Percentage (%)**	**15**	
Criteria	**Score and Descriptors**			**Weight (%)**	**Marks**
	(5)	**(3-4)**	**(0-2)**		
Task 1) Stack-based Sorting	Completed 100% of task requirements & Delivered on time, and in correct format & Student explained perfectly in results section	Completed 100% of task requirements & Delivered on time, and in correct format & Student has minor problem in explanation in result section	Completed less than 50% of the requirements. or Runtime issues or Does not comply with requirements (does something other than requirements). or Student does not explain enough in result section	**5**	
Task 2) Graph Data structure	Completed 100% of task requirements & Delivered on time, and in correct format & Student explained perfectly in results section	Completed 100% of task requirements & Delivered on time, and in correct format & Student has minor problem in explanation in result section	Completed less than 50% of the requirements. or Runtime issues or Does not comply with requirements (does something other than requirements). or Student does not explain enough in result section	**5**	
Task 3) Search Algorithms	Completed 100% of task requirements & Delivered on time, and in correct format & Student explained perfectly in results section	Completed 100% of task requirements & Delivered on time, and in correct format & Student has minor problem in explanation in result section	Completed less than 50% of the requirements. or Runtime issues or Does not comply with requirements (does something other than requirements). or Student does not explain enough in result section	**5**	
			TOTAL	**15**	

Note to students: Please print out and attach this appendix together with the submission of coursework.