**INTRODUCTION**

In computer science, efficient management and manipulation of data are essential for developing robust software applications. Two fundamental data structures that facilitate such management are Stack and Queue. These structures enable specific ways of accessing and processing data, which can be particularly useful in various algorithms and real-world applications. Additionally, sorting algorithms are vital for organizing data, making it easier to search and analyze. **Stack** is a linear data structure that follows the Last-In, First-Out (LIFO) principle. **Queue** is another linear data structure that follows the First-In, First-Out (FIFO) principle. **Sorting algorithms** are fundamental in computer science for organizing data. **Selection Sort** is a simple comparison-based algorithm where the list is divided into a sorted and an unsorted region.

This program provides an interactive console menu system that allows users to perform various operations on stacks and queues and sort a list of integers. The menu-driven approach ensures that users can choose and execute operations easily, making it user-friendly and efficient for educational and practical purposes.

**FEATURES**

* **Stack Operations:** Users can push, pop, peek, check if the stack is full or empty, and display the elements.
* **Queue Operations:** Users can enqueue, dequeue, check if the queue is full or empty, and display the elements.
* **Sorting Operations:** Users can input a list of integers and sort them using the selection sort algorithm.

**DATA ANALYSIS**

**Stack**

A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. The element that is added last is the one that is removed first. It has two main operations:

Push: Add an element to the top of the stack.

Pop: Remove the element from the top of the stack.

**Queue**

A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle. The element that is added first is the one that is removed first. It has two main operations: Enqueue: Add an element to the end of the queue.

Dequeue: Remove the element from the front of the queue.

**Sorting Algorithm (Selection Sort)**

Selection Sort is a simple comparison-based sorting algorithm. It works by repeatedly finding the minimum element (considering ascending order) from the unsorted part and putting it at the beginning.

Consider the array [64, 25, 12, 22, 11].

**First Pass:**

* + Find the minimum element in the unsorted array [64, 25, 12, 22, 11].
  + Swap it with the first element.
  + Array after first pass: [11, 25, 12, 22, 64].

**Second Pass:**

* + Find the minimum element in the unsorted array [25, 12, 22, 64].
  + Swap it with the second element.
  + Array after second pass: [11, 12, 25, 22, 64].

**FUNCTIONS**

**Main Function** It handles the main menu to choose between stack operations, queue operations, and sorting a list.

**Stack Class**

* **isEmpty()**: Checks if the stack is empty.
* **isFull()**: Checks if the stack is full.
* **push(int val)**: Adds an element to the top of the stack.
* **pop()**: Removes an element from the top of the stack and returns it.
* **count()**: Returns the number of elements in the stack.
* **peek(int pos)**: Returns the element at the specified position.
* **change(int pos, int val)**: Changes the element at the specified position to a new value.
* **display()**: Displays all the elements in the stack from top to bottom.

**Queue Class**

* **isEmpty()**: Checks if the queue is empty.
* **isFull()**: Checks if the queue is full.
* **enqueue(int val)**: Adds an element to the end of the queue.
* **dequeue()**: Removes an element from the front of the queue and returns it.
* **count()**: Returns the number of elements in the queue.
* **display()**: Displays all the elements in the queue from front to rear.

**sortList()**

* **sortList()**: Prompts the user to input a list of integers, sorts the list using the Selection Sort algorithm, and then displays the sorted list.

**Detailed Explanation of Each Function**

**STACK FUNCTIONS**

* **isEmpty()**: Check if the stack is empty by comparing top to size -1. Returns true if empty, false otherwise.

bool isEmpty() {

return top == -1;

}

* **isFull()**: Checks if the stack is full by comparing top to size - 1. Returns true if full, false otherwise.

bool isFull() {

return top == size - 1;

}

* **push(int val)**: Adds an element to the top of the stack. Checks if the stack is full before adding.

void push(int val) {

if (isFull()) {

cout << "Stack Overflow" << endl;

} else {

top++;

arr[top] = val;

}

}

* **pop()**: Removes and returns the top element of the stack. Checks if the stack is empty before removing.

int pop() {

if (isEmpty()) {

cout << "Stack Underflow" << endl;

return 0;

} else {

int popValue = arr[top];

arr[top] = 0;

top--;

return popValue;

}

}

* **count()**: It returns the number of elements in the stack.

int count() {

return (top + 1);

}

* **peek(int pos)**: It returns the element at the specified position in the stack.

int peek(int pos) {

if (isEmpty()) {

cout << "Stack Underflow" << endl;

return 0;

} else {

return arr[pos];

}

}

* **change(int pos, int val)**: It changes the element at the specified position to a new value.

void change(int pos, int val) {

arr[pos] = val;

cout << "Value successfully changed at index " << pos << endl; }

* **display()**: It displays all the elements in the stack from top to bottom.

void display() {

for (int i = size - 1; i >= 0; i--) {

cout << arr[i] << endl;

}

}

**QUEUE FUNCTIONS**

* **isEmpty()**: It checks if the queue is empty by comparing front to -1. Returns true if empty, false otherwise.

bool isEmpty() {

return front == -1;

}

* **isFull()**: It checks if the queue is full by comparing rear to size - 1. Returns true if full, false otherwise.

bool isFull() {

return rear == size - 1;

}

* **enqueue(int val)**: It adds an element to the end of the queue. Checks if the queue is full before adding.

void enqueue(int val) {

if (isFull()) {

cout << "Queue Overflow" << endl;

} else if (isEmpty()) {

front = rear = 0;

arr[rear] = val;

} else {

rear++;

arr[rear] = val;

}

}

* **dequeue()**: It removes and returns the front element of the queue and checks if the queue is empty before removing the element.

int dequeue() {

int x = 0;

if (isEmpty()) {

cout << "Queue Underflow" << endl;

return 0;

} else if (front == rear) {

x = arr[front];

arr[front] = 0;

front = rear = -1;

} else {

x = arr[front];

arr[front] = 0;

front++;

}

return x;

}

* **count()**: It returns the number of elements in the queue.

int count() {

return (rear - front + 1);

}

**display()**:It displays all the elements in the queue from front to rear.

void display() {

for (int i = 0; i < size; i++) {

cout << arr[i] << " ";

}

cout << endl;

}

**sortList() Function**

1. Prompts the user to input the number of elements in the list.
2. Accepts the elements and stores them in a vector.
3. Uses the sort() function from the <algorithm> library to sort the vector.
4. Displays the sorted list.

void sortList() {

vector<int> list;

int n, value;

cout << "Enter the number of elements: ";

cin >> n;

cout << "Enter the elements:\n";

for (int i = 0; i < n; ++i) {

cin >> value;

list.push\_back(value);

}

sort(list.begin(), list.end());

cout << "Sorted list:\n";

for (const auto& elem : list) {

cout << elem << " ";

}

cout << endl;

}

**CONCLUSION**

This documentation explains the functionality and implementation of stack and queue data structures along with a sorting algorithm. Each function is detailed with its purpose, and diagrams are provided for better understanding. The main menu system allows users to interact with these data structures and sorting algorithms through a console-based interface.

**REFERENCES**

* GeeksforGeeks (2020) 'Stack Data Structure (Introduction and Program)', GeeksforGeeks.
* GeeksforGeeks (2020) 'Queue Data Structure (Introduction and Program)', GeeksforGeeks.
* GeeksforGeeks (2020) 'Selection Sort', GeeksforGeeks.