VIETNAM NATIONAL UNIVERSITY OF HO CHI MINH CIY

UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

Data Structures and Algorithms

Exercise 1 Report

Implementing Stack and Queue from scratch

Student Name	Student ID
Nguyen Thi Khanh Linh	23127082

Lecturers:

Bui Duy Dang

Truong Tan Khoa

Nguyen Thanh Tinh

Submission Date : June 16 2024

Exercise 1 Report
Implementing Stack and Queue from scratch

Nguyen Thi Khanh Linh

Contents

1 Introduction

Purpose of the Exercise: This exercise requires students to research and implement Stack and Queue using both arrays and linked lists.
Importance of Stack and Queue: Understanding stack and queue is essential for developing effective algorithms. These data structures are the basic core of many applications and algorithms, making them indispensable in computer science.

2 Student Information

Student ID: 23127082
Full Name: Nguyen Thi Khanh Linh
Class: 23CLC09

3 Stack Implementation

3.1 Stack Using Array

A Stack structure consists of:

- Variables:
  - *items:* Data array.
  - *top:* Index of the top item.
  - *maxSize:* Maximum size of the stack.
- Functions:
  - *init:* Initialize an empty stack with the provided size.
  - *copyStack:* Initialize a stack from another stack.
  - *release:* Clean up the stack.
  - *isEmpty:* Check if the stack is empty.
  - *isFull:* Check if the stack is full.
  - *push:* Push a new item into the stack. Print to the console if the stack is full.
  - *pop:* Pop the top item.
  - *topValue:* Get the value of the top item.
  - *print:* Print the stack to the console.

Code Structure:

- `stack.h`: Containing the declaration of Stack structure.
- `stack.cpp`: Containing the implementation for the Stack.
- `main.cpp`: Using the Stack as the library.

Experiments:

A stack is initialized with maxSize = 5

- Menu

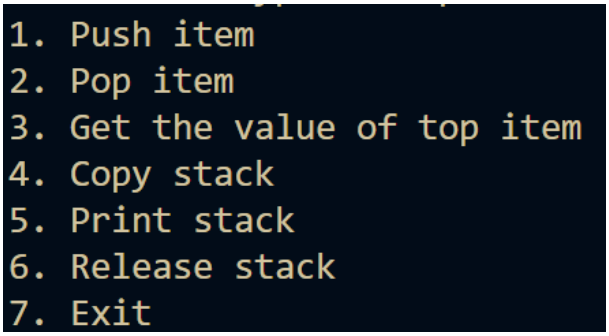

Figure 1: The menu of implementations

- ***push()*** operator

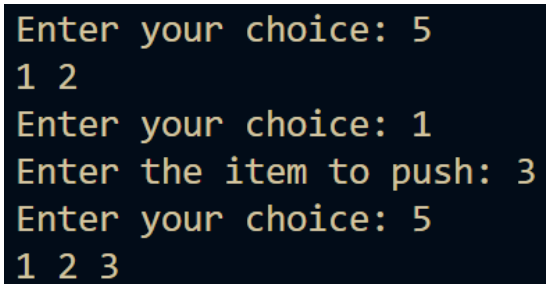

Figure 2: *push* in normal case

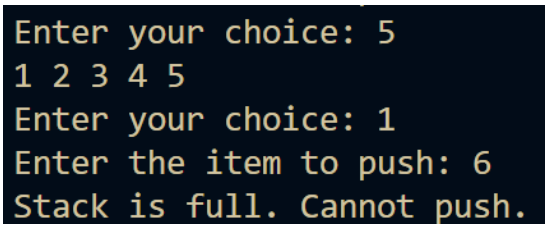

Figure 3: *push* when the array is full

- ***pop()*** operator

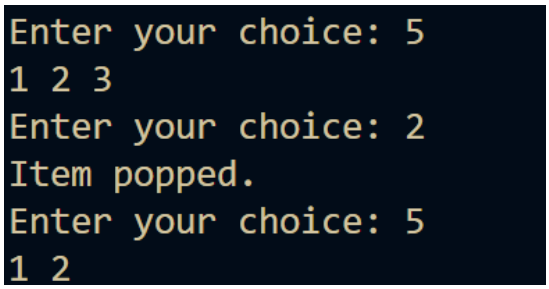

Figure 4: *pop* in normal case

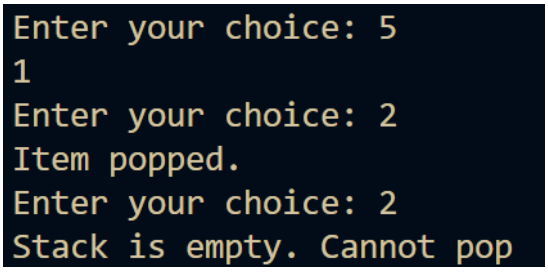

Figure 5: *pop* when the stack is empty

3.2 Stack Using Linked List

A Stack structure consists of:

- Variables:

  - *top:* Pointer to the top node.
  - Node structure containing data and a pointer to the next node.

- Functions: Similar to the array version, but adapted for a linked list. The linked list version does not include the isFull function.

Code Structure: Similar to the array version, but adapted for a linked list.

Experiments:

- Menu: Similar to the array version.
- ***push()*** operator
  Implementing Stack using linked list does not require a fixed size of memory. Therefore, it does not need to check if the Stack is full or not.

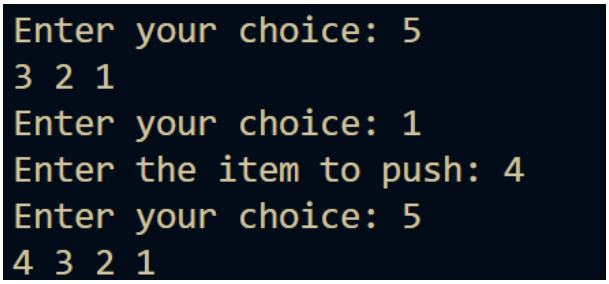

Figure 6: *push* in normal case

- ***pop()*** operator

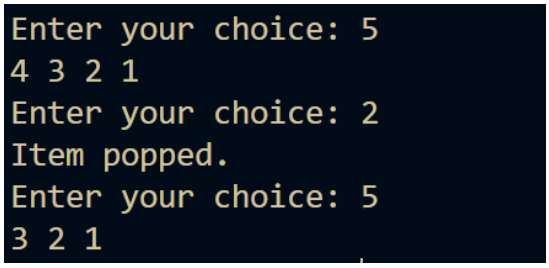

Figure 7: *pop* in normal case

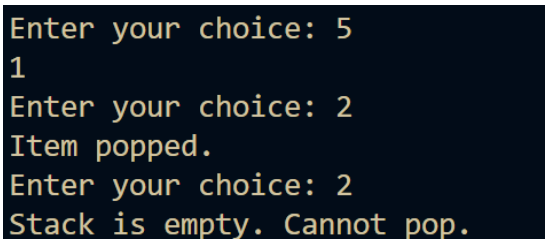

Figure 8: *pop* when the stack is empty

4 Queue Implementation

4.1 Queue Using Array

A Queue structure consists of:

- Variables:
  - *items:* Data array.
  - *front:* Index of the front item.

  - *rear:* Index of the rear item.
  - *count:* The number of items stored in the queue.
  - *maxSize:* Maximum size of the queue.

- Functions:
  - *init:* Initialize an empty queue with the provided size.
  - *copyQueue:* Initialize a queue from another queue.
  - *release:* Clean up the queue.
  - *isEmpty:* Check if the queue is empty.
  - *isFull:* Check if the queue is full.
  - *enqueue:* Add a new item to the rear of the queue.
  - *dequeue:* Remove the front item.
  - *frontValue:* Get the value of the front item.
  - *print:* Print the queue.

Code Structure:

- `queue.h`: Containing the declaration of Queue structure.
- `queue.cpp`: Containing the implementation for the Queue.
- `main.cpp`: Using the queue as the library.

Experiments:

A queue is initialized with maxSize = 5

- Menu

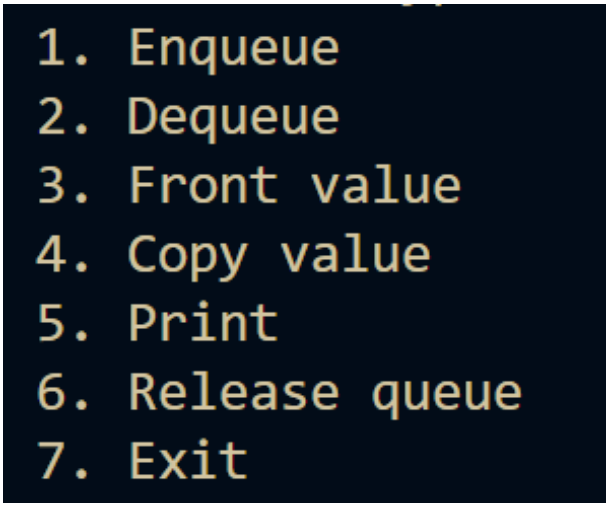

Figure 9: The menu of implementations

- ***enqueue()*** operator

```
Enter your choice: 5
1 2 3
Enter your choice: 1
Enter the item to enqueue: 4
Enter your choice: 5
1 2 3 4
```

Figure 10: *enqueue* in normal case

```
Enter your choice: 5
1 2 3 4 5
Enter your choice: 1
Enter the item to enqueue: 6
Queue is full. Cannot enqueue.
```

Figure 11: *enqueue* when the array is full

- ***dequeue()*** operator

```
Enter your choice: 5
3 4 5
Enter your choice: 2
Item dequeued
Enter your choice: 5
4 5
```

Figure 12: *dequeue* in normal case

```
Enter your choice: 5
5
Enter your choice: 2
Item dequeued
Enter your choice: 2
Queue is empty. Cannot dequeue.
```

Figure 13: *dequeue* when the queue is empty

4.2 Queue Using Linked List

A Queue structure consists of:

- Variables:
  - *front:* Pointer to the front node.
  - *rear:* Pointer to the rear node.
  - Node structure containing data and a pointer to the next node.
- Functions: Similar to the array version, but adapted for a linked list. The linked list version does not include the isFull function.

Code Structure: Similar to the array version, but adapted for a linked list.

Experiments:

- Menu: Similar to the array version.
- ***enqueue()*** operator
  Implementing Queue using linked list does not require a fixed size of memory. Therefore, it does not need to check if the Queue is full or not.

```
Enter your choice: 5
1 2 3
Enter your choice: 1
Enter the item to enqueue: 4
Enter your choice: 5
1 2 3 4
```

Figure 14: *enqueue* in normal case

- ***dequeue()*** operator

```
Enter your choice: 5
1 2 3 4
Enter your choice: 2
Item dequeued
Enter your choice: 5
2 3 4
```

Figure 15: *dequeue* in normal case

```
Enter your choice: 5
4
Enter your choice: 2
Item dequeued
Enter your choice: 2
Queue is empty. Cannot dequeue.
Item dequeued
```

Figure 16: *dequeue* when the queue is empty

5 Recursive Versions

5.1 Theoretical Time Complexity

- Loop-Based Implementations: Generally efficient and straightforward, with $O(n)$ complexity for *copyStack*. For *release*, it is potentially $O(1)$ if the structure is implemented using array and $O(n)$ for using linked list.
- Recursive Implementations: Similarly efficient in terms of time complexity but may have extra overhead from recursive calls and higher stack usage.

Function	Implementation Type	Time Complexity
`copyStack`	Loop	$O(n)$
`copyStack`	Recursive	$O(n)$
`release`	Loop	$O(1)$ or $O(n)$
`release`	Recursive	$O(n)$
`copyQueue`	Loop	$O(n)$
`copyQueue`	Recursive	$O(n)$
`release`	Loop	$O(1)$ or $O(n)$
`release`	Recursive	$O(n)$

Table 1: Summary of Time Complexities

5.2 Actual Execution Time Compare

Functions that are converted into recursive form: *copyStack, release* (linked list versions only), *print.*

Code Structure: I used the files that I have implemented above. Besides, I also added a few more files.

- For the Stack, the new files are:
  - `recursive_stack.h`: Containing the declaration of Stack structure for the recursive implementation.
  - `recursive_stack.cpp`: Containing the recursive implementation for the Stack.
  - `time.cpp`: Using Stack as the library and comparing the actual execution time between loop-based and recursive methods.
- For the Queue, the new files are:
  - `recursive_queue.h`: Containing the declaration of Queue structure for the recursive implementation.
  - `recursive_queue.cpp`: Containing the recursive implementation for the Queue.
  - `time.cpp`: Using Queue as the library and comparing the actual execution time between loop-based and recursive methods.

Execution time comparison:

- To compare the execution time of these functions in two methods (using for loop and using recursion), I utilized `std::chrono` library introduced in C++11 [1].
- There are four figures below. Each figure shows the results of 5 tries of comparison. In each try, the functions work with the same 10,000 elements in both methods of implementation. Note that the elements for each try are different.

Stack Using Array

```
Try 1:
Time to copy stack using for loop: 60 microseconds
Time to copy stack using recursion: 467 microseconds
Try 2:
Time to copy stack using for loop: 60 microseconds
Time to copy stack using recursion: 117 microseconds
Try 3:
Time to copy stack using for loop: 42 microseconds
Time to copy stack using recursion: 88 microseconds
Try 4:
Time to copy stack using for loop: 34 microseconds
Time to copy stack using recursion: 63 microseconds
Try 5:
Time to copy stack using for loop: 35 microseconds
Time to copy stack using recursion: 177 microseconds
```

Figure 17: Execution time compare of Stack using Array

Stack Using Linked List

```
Try 1:
Time to copy stack using for loop: 279 microseconds
Time to copy stack using recursion: 527 microseconds
Time to release stack using for loop: 136 microseconds
Time to release stack using recursion: 258 microseconds
Try 2:
Time to copy stack using for loop: 298 microseconds
Time to copy stack using recursion: 361 microseconds
Time to release stack using for loop: 146 microseconds
Time to release stack using recursion: 191 microseconds
Try 3:
Time to copy stack using for loop: 309 microseconds
Time to copy stack using recursion: 474 microseconds
Time to release stack using for loop: 127 microseconds
Time to release stack using recursion: 247 microseconds
Try 4:
Time to copy stack using for loop: 292 microseconds
Time to copy stack using recursion: 469 microseconds
Time to release stack using for loop: 133 microseconds
Time to release stack using recursion: 212 microseconds
Try 5:
Time to copy stack using for loop: 286 microseconds
Time to copy stack using recursion: 382 microseconds
Time to release stack using for loop: 162 microseconds
Time to release stack using recursion: 224 microseconds
```

Figure 18: Execution time compare of Stack using Linked list

Queue Using Array

```
Try 1:
Time to copy queue using for loop: 35 microseconds
Time to copy queue using recursion: 259 microseconds
Try 2:
Time to copy queue using for loop: 41 microseconds
Time to copy queue using recursion: 79 microseconds
Try 3:
Time to copy queue using for loop: 41 microseconds
Time to copy queue using recursion: 56 microseconds
Try 4:
Time to copy queue using for loop: 39 microseconds
Time to copy queue using recursion: 56 microseconds
Try 5:
Time to copy queue using for loop: 38 microseconds
Time to copy queue using recursion: 65 microseconds
```

Figure 19: Execution time compare of Queue using Array

Queue Using Linked List

```
Try 1:
Time to copy queue using for loop: 481 microseconds
Time to copy queue using recursion: 605 microseconds
Time to release queue using for loop: 134 microseconds
Time to release queue using recursion: 278 microseconds
Try 2:
Time to copy queue using for loop: 275 microseconds
Time to copy queue using recursion: 345 microseconds
Time to release queue using for loop: 139 microseconds
Time to release queue using recursion: 236 microseconds
Try 3:
Time to copy queue using for loop: 253 microseconds
Time to copy queue using recursion: 380 microseconds
Time to release queue using for loop: 157 microseconds
Time to release queue using recursion: 268 microseconds
Try 4:
Time to copy queue using for loop: 264 microseconds
Time to copy queue using recursion: 361 microseconds
Time to release queue using for loop: 196 microseconds
Time to release queue using recursion: 440 microseconds
Try 5:
Time to copy queue using for loop: 263 microseconds
Time to copy queue using recursion: 430 microseconds
Time to release queue using for loop: 134 microseconds
Time to release queue using recursion: 243 microseconds
```

Figure 20: Exucution time compare of Queue using Linked list

- Overall, the actual execution time of using for loop is faster than using recursion. As explained above, this is because in recursion, each function call adds a new frame to the call stack, which takes additional time and memory. This can lead to increased execution time and higher memory usage.

6 Self-Evaluation

6.1 Learning Outcomes

From working on this exercise, I have learned a lot from various aspects:

- I deepened my understanding of the fundamental data structures, Stack and Queue, and how they can be implemented using both arrays and linked lists.
- I learned how to use templates and some parts of object-oriented programming in C++.
- I learned how to measure execution time using `std::chrono` library in C++ [1].
- I learned how to write a report using LaTeX.

6.2 Challenges

- There is new knowledge I have never approached and need to spend time researching it on many websites.
- During the implementation, managing the different edge cases for each operation, particularly for the linked list versions is also challenging.
- When I implemented the copy functions, the missing of initializing the copied Stack/Queue took me plenty of time to discover and fix them.

6.3 Improvements

To improve the implementation, I would:

- Start to work on the project earlier so I would have more time to research knowledge.
- Optimizing the recursive implementations to minimize overhead and improve performance.
- Investigating other potential implementations or variations of stack and queue to broaden understanding.

7 Exercise Feedback

7.1 Experience

Overall, the exercise was both challenging and rewarding. It provided a solid foundation in working with stacks and queues and allowed me to explore different methods of implementation. The hands-on approach of writing the code and conducting experiments helped reinforce theoretical concepts and enhanced my problem-solving skills.

7.2 Recommendations

For future exercises or improvements to the current structure, I would recommend:

- Providing more detailed guidelines or examples on measuring execution time and performance analysis.
- Adding an advanced section on implementing priority queues or double-ended queues (deques) for further learning.

8 Conclusion

In this report, I have detailed the process of implementing Stack and Queue data structures using arrays and linked lists. I explored both loop-based and recursive methods for common operations and analyzed their theoretical and actual performance. The exercise emphasized the importance of understanding these data structures and gave valuable insights into the pros and cons of different implementation methods. Overall, this exercise has strengthened my knowledge and skills in data structures, which are crucial for developing efficient algorithms and software applications.

9 References

[1] **Measure execution time of a function in C++ library documentation** — Last Updated : 03 May, 2023 by sayan mahapatra.