© 2014 IJIRT | Volume 1 Issue 6 | ISSN : 2349-6002

A COMPARATIVE STUDY OF STACK AND QUEUES IN DATA STRUCTURE

Anoop, Sunil Rai
Department of Electronics and Computer
Dronacharya College of Engineering, Gurgaon, Haryana

Abstract- **The objective of this paper is to comprehensive study related to stack and queues in data structure. Stack and Queue are abstract data type. As we know that in our daily life we faced many problems related to collection of entities in order. A stack is an ordered list in which all insertions and deletions are made at one end, called the top and A queue is an ordered list in which all insertions take place at one end, the rear, while all deletions take place at the other end, the front. Stack is known as Last-in-first-out (LIFO) whereas Queues is related to First-in-first-out (FIFO). In this paper, an attempt has been made by us to study the whole concept related to stack and queues.**

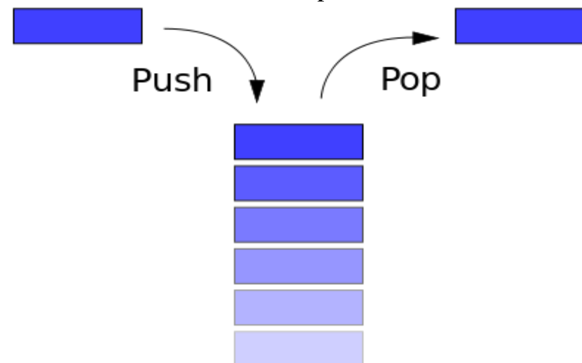
Index Terms- **LIFO; FIFO.**

I. INTRODUCTION

In our daily life we have seen that anything like books, chair, plates etc. should be placed in an order. This is the idea from where stack and queues were introduced. Basically stack and queues are the abstract data type but there is some difference in their insertion and deletion process. In stack, insertion and deletion takes place from only at one side but in case of queues insertion and deletion takes place at two ends known as rear end and front end. Elements are inserted from the rear end and deleted from front end. We will go to learn about stack as well as queues in detail.

II. STACK

A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle. In the pushdown stacks only two operations are allowed: **push** the item into the stack, and **pop** the item out of the stack. A stack is a limited access data structure - elements can be added and removed from the stack only at the top. **Push** adds an item to the top of the stack, **pop** removes the item from the top. A helpful analogy is to think of a stack of books; you can remove only the top book, also you can add a new book on the top.



A stack may be implemented to have a bounded capacity. If the stack is full and does not contain enough space to accept an entity to be pushed, the stack is then considered to be in an overflow state. The pop operation removes an item from the top of the stack. A pop either reveals previously concealed items or results in an empty stack, but, if the stack is empty, it goes into underflow state, which means no items are present in stack to be removed.

A stack is a restricted data structure, because only a small number of operations are performed on it. The nature of the pop and push operations also mean that stack elements have a natural order. Elements are removed from the stack in the reverse order to the order of their addition. Therefore, the lower elements are those that have been on the stack the longest.

Example of Stack:

A. History

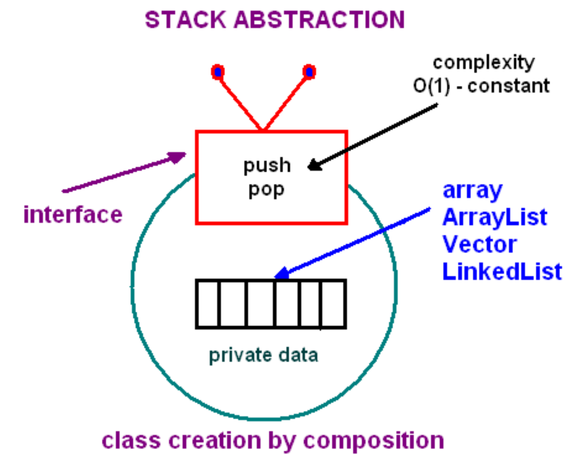
The stack was first proposed in 1946, in the computer design of Alan M. Turing (who used the terms "bury" and "unbury") as a means of calling and returning from subroutines. Subroutines had already been implemented in Konrad Zuse's Z4 in 1945. Klaus Samelson and Friedrich L. Bauer of Technical University Munich proposed the idea in 1955 and filed a patent in 1957. The same concept was developed, independently, by the Australian Charles Leonard Hamblin in the first half of 1957.

B. Implementation

In the standard library of classes, the data type stack is an adapter class, meaning that a stack is built on top of other data structures. The underlying structure for a stack could be an array, a vector, an ArrayList, a linked list, or any other collection. Regardless of the type of the underlying data structure, a Stack must implement the same functionality. This is achieved by providing a unique interface:

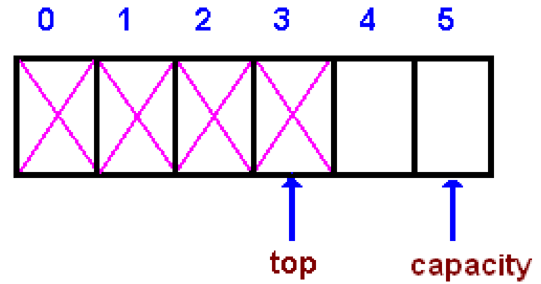
```
public interface StackInterface<AnyType>
{
  public void push(AnyType e);

  public AnyType pop();

  public AnyType peek();

  public boolean isEmpty();
}
```

The following picture demonstrates the idea of implementation by composition:



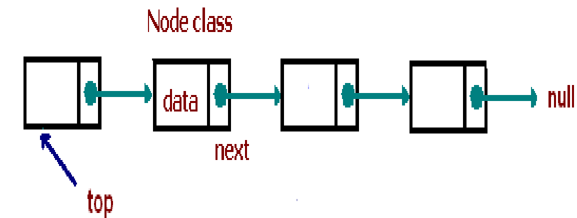
Array-based implementation

In an array-based implementation we maintain the following fields: an array A of a default size (≥ 1), the variable *top* that refers to the top element in the stack and the *capacity* that refers to the array size. The variable *top* changes from -1 to capacity - 1. We say that a stack is empty when top = -1, and the stack is full when top = capacity-1.



Linked List-based implementation

Linked List-based implementation provides the best (from the efficiency point of view) dynamic stack implementation.



C. Applications

Stacks are present everyday life, from the books in a library, to the blank sheets of paper in a printer tray. All these applications follow the *Last In First Out* (LIFO) logic, which means that (for example) a book is added on top of a pile of books, while removing a book from a pile also takes the book on top of a pile.

Below are a few applications of stacks in computing.

a. Expression evaluation and syntax parsing

Calculators employing reverse Polish notation use a stack structure to hold values. Expressions can be represented in prefix; postfix or infix notations and conversion from one form to another may be accomplished using a stack.

b. Backtracking

Another important application of stacks is backtracking. Consider a simple example of finding the correct path in a maze. There are a series of points, from the starting point to the destination. We start from one point. To reach the final destination, there are several paths. Suppose we choose a random path. After following a certain path, we realise that the path we have chosen is wrong. So we need to find a way by which we can return to the beginning of that path. This can be done with the use of stacks. With the help of stacks, we remember the point where we have reached. This is done by pushing that point into the stack. In case we end up on the wrong path, we can pop the last point from the stack and thus return to the last point and continue our quest to find the right path. This is called backtracking.

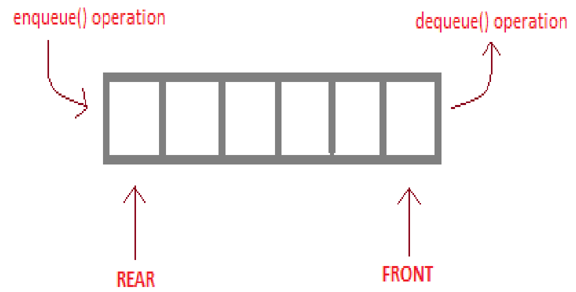
c. Runtime memory management

A number of programming languages are stack-oriented, meaning they define most basic operations (adding two numbers, printing a character) as taking their arguments from the stack, and placing any return values back on the stack. For example, PostScript has a return stack and an operand stack, and also has a graphics state stack and a dictionary stack. Many virtual machines are also stack-oriented, including the p-code machine and the Java Virtual Machine.

III. QUEUES

A queue is an ordered collection of items where the addition of new items happens at one end, called the "rear," and the removal of existing items occurs at the other end, commonly called the "front." As an element enters the queue it starts at the rear and makes its way toward the front, waiting until that time when it is the next element to be removed.

The most recently added item in the queue must wait at the end of the collection. The item that has been in the collection the longest is at the front. This ordering principle is sometimes called **FIFO**, **first-in first-out**. It is also known as "first-come first-served."



enqueue( ) is the operation for adding an element into Queue.

dequeue( ) is the operation for removing an element from Queue .

QUEUE DATA STRUCTURE

The process to add an element in the queues is known as Enqueue and the process of removal of element form the queue is known as Dequeue.

Example of Queue:

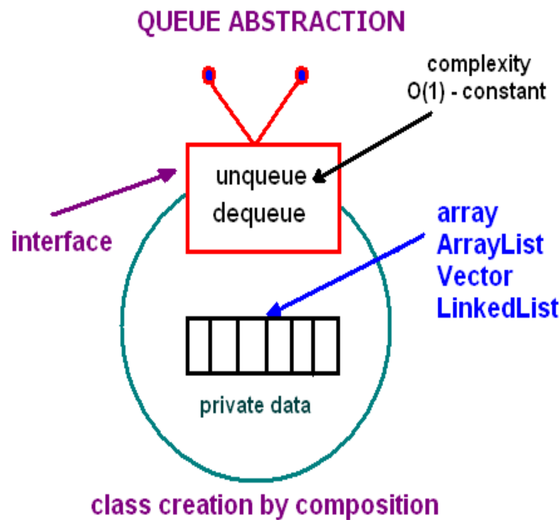
A. Implementation

In the standard library of classes, the data type queue is an adapter class, meaning that a queue is built on top of other data structures. The underlying structure for a queue could be an array, a Vector, an Array List, a Linked List, or any other collection. Regardless of the type of the underlying data structure, a queue must implement the same functionality. This is achieved by providing a unique interface.

```
interface QueueInterface‹AnyType>
{
  public boolean isEmpty();

  public AnyType getFront();
```

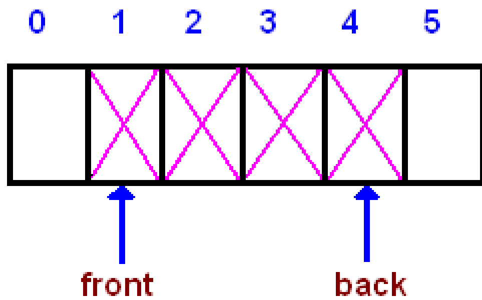
```
  public AnyType dequeue();

  public void enqueue(AnyType e);

  public void clear();
}
```

Each of the above basic operations must run at constant time O(1). The following picture demonstrates the idea of implementation by composition.

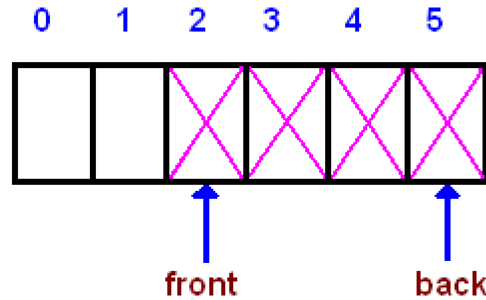


Circular Queue

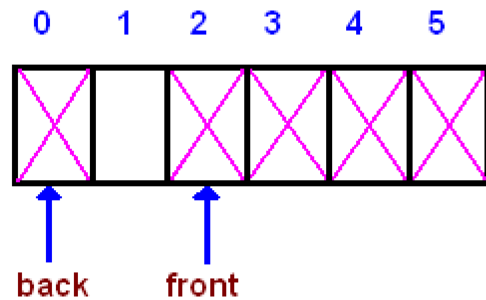
Given an array A of a default size (≥ 1) with two references *back* and *front*, originally set to -1 and 0 respectively. Each time we insert (enqueue) a new item, we increase the back index; when we remove (dequeue) an item - we increase the front index. Here is a picture that illustrates the model after a few steps:



As you see from the picture, the queue logically moves in the array from left to right. After several moves back reaches the end, leaving no space for adding new elements



However, there is a free space before the front index. We shall use that space for enqueueing new items, i.e. the next entry will be stored at index 0, then 1, until *front*. Such a model is called a **wrap around queue** or a **circular queue**



Finally, when *back* reaches *front*, the queue is full. There are two choices to handle a full queue: a) throw an exception; b) double the array size.

B. Applications

Queue, as the name suggest is used whenever we need to have any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios:

1. Serving requests on a single shared resource, like printer, CPU task scheduling etc.
2. In real life, call center phone systems will Queues, to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, first come first served.

REFERENCES

1. www.wikipedia.com
2. Victor S.Adamchik, CMU, 2009
3. www.cprogramming.com
4. Dr. Friedrich Ludwig Bauer and Dr. Klaus Samelson (30 March 1957). "Verfahren zur

automatischen Verarbeitung von kodierten Daten und Rechenmaschine zur Ausübung des Verfahrens" (in German). Germany, Munich: Deutsches Patentamt. Retrieved 2010-10-01.

5. C. L. Hamblin, "An Address less Coding Scheme based on Mathematical Notation", N.S.W University of Technology, May 1957 (typescript)
6. Pritesh tarpal, www.c4learn.com
7. Data structures, Algorithms and Applications in C++ by Sartaj Sahni
8. Gopal, Arpita. Magnifying Data Structures. PHI.