Lecture Notes

Stacks and Queues

Welcome to session on Stacks and Queues.

In the OOPS course, you learnt about two data structures: array lists and linked lists. You saw how both are implementations of the same abstract data type list. You learnt about the various capabilities and properties of lists, and how they could be used in solving practical software development problems (recall Institute Management System). You learnt how despite being functionally nearly identical to each other, array lists and linked lists were quite different from each other. In particular, when you take into account, their performance aspects, you observe that one of them fairs better than the other in certain scenarios, and worse in certain others. For example, getting a value by its index is typically much faster in an array list. However, adding/removing an element at an arbitrary location may be faster in a linked list.

Data Structures

Algorithms are computational procedures which deal with data. They create them, modify them and refer to them time to time. In order to deliver good results, your algorithms need the program data to be arranged in a way suitable for the purpose. A well arranged data will make it easy to implement algorithms (and software systems in general) that not only are fast and efficient, but are also flexible while being robust and secure. Hence, how the data is arranged within a program is a crucial question that must be answered at a very early stage of design of a program.

The subject of data structures is the study of some of these 'arrangements' of data which have been found useful by software developers in a variety of scenarios. While studying a data structure, our aim should be to understand:

1. What are their capabilities?

2. In what scenarios can they be used and how?

3. What are various implementations and how do they compare in terms of performance (space/time)?

You have till now seen the use of terms - *abstract data types* and *data structures* apparently synonymously, but there is a difference between the two. Abstract data type is the definition of the interface (i.e. the properties and capabilities) of a data arrangement. Data structure is a specific implementation. For example, list is an ADT (where the elements are linearly sequence, each associated with an index or position represented by a nonnegative integer); linked list is an implementation. In other words, there may be multiple data structures which implement the same ADT. In most places, you'll find these two terms used as near synonyms.

Stacks

Consider a stack of books - or plates or some such things. If you were to pick one book from the stack, from which portion of it would you pick it? From the top? From the bottom? Or from somewhere in between? Of course, any sane person would pick it up from the top. Trying to pull out a book from anywhere else in the stack would be possible, but would lead to difficulties, and untoward results. In the same manner, if you were to add a book to this stack, where would you prefer to add it? Again, at the top, of course.

This structure of books where you add things at the top and remove things again from the top is analogous to a data structure called "**Stacks**". In a stack, what goes in last is the first to come out. This property of the above structure is called last in first out (or LIFO) order.

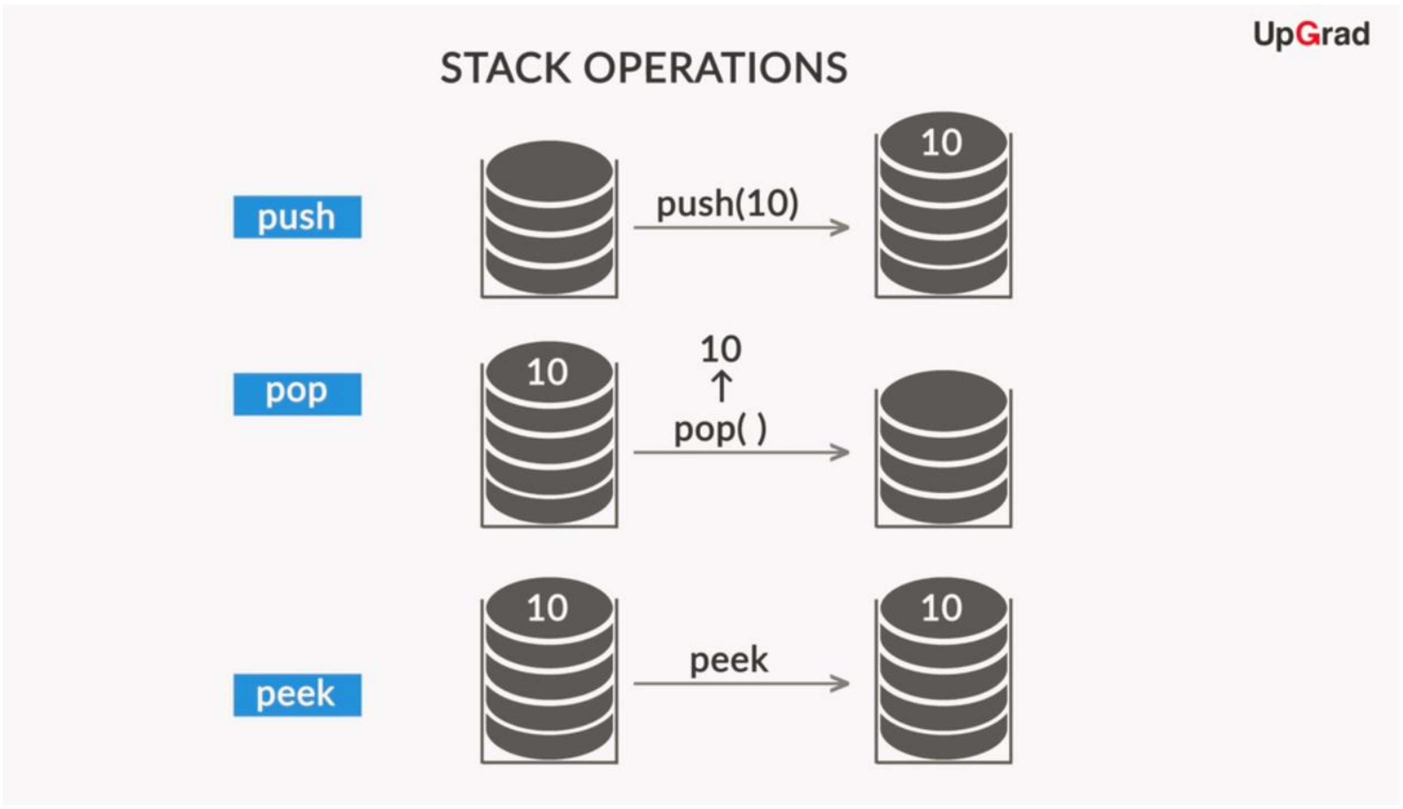
In real life, and in computing, there happen umpteen instances wherein things have to inserted into something and removed from it in a LIFO order. These somethings at least in computer science are called stacks. A stack is therefore anything:

1. into which you insert (push) and remove (pop) things from one end of it.

2. that follows LIFO rule.

Stack Operations

There are four general operations you do on a stack –



Apart from the 3 operations in the image above, there is another operation called – isEmpty(), which returns true if the stack is empty and false otherwise.

There are two kinds of exceptions you could encounter in stacks:

- Underflow: Trying to pop/peek an element from an empty stack
- Overflow: Trying to push an element into a stack that is already at its max capacity

You try to avoid these exceptions by writing checks in your code or by handling exceptions. For the underflow, you saw that a condition was framed, where a pop is not allowed from an empty stack. On the other hand, you learnt that a stack overflow is caused when you've used up more memory for a stack than your program was supposed to use.

Implementing a Stack

You could either implement a stack using a custom class made using lists (array lists or linked lists) such as the one below, where you used a linked list to implement a stack.

In the following program, you added the courses you have done to a stack and then popped them one by one.

```
import java.util.List;
import java.util.LinkedList;
import java.util.EmptyStackException;

public class MyStack<T> {

    public static void main(String[] arg) {
        MyStack<String> stack = new MyStack<String>();

        stack.push("OOP");
        stack.push("Algorithms");
        stack.push("Data Structures");

        try {
            while (true) {
                System.out.println("Popped " + stack.pop());
            }
        } catch (EmptyStackException e) {
            System.out.println("Done!");
        }
    }

    private LinkedList<T> list = new LinkedList<T>();

    public void push(T e) {
        this.list.add(e);
    }
    public T pop() {
        if (this.list.size() > 0) {
            T e = list.get(list.size() - 1);
            list.remove(list.size() - 1);
            return e;
        }
        throw new EmptyStackException();
    }

    public Boolean isEmpty() {
        return this.list.size() == 0;
    }
}
```

The output for this code was –

```
Popped Data Structures
Popped Algorithms
Popped OOP
```

You can see that the element which was pushed in first was the last one to be popped out, thus demonstrating the Last in, First out order followed in a stack.

You could also use the internal Stack library provided by JAVA, which you could call using -

```
import java.util.Stack;
```

The features push, pop, isEmpty and peek can all be called using this library.

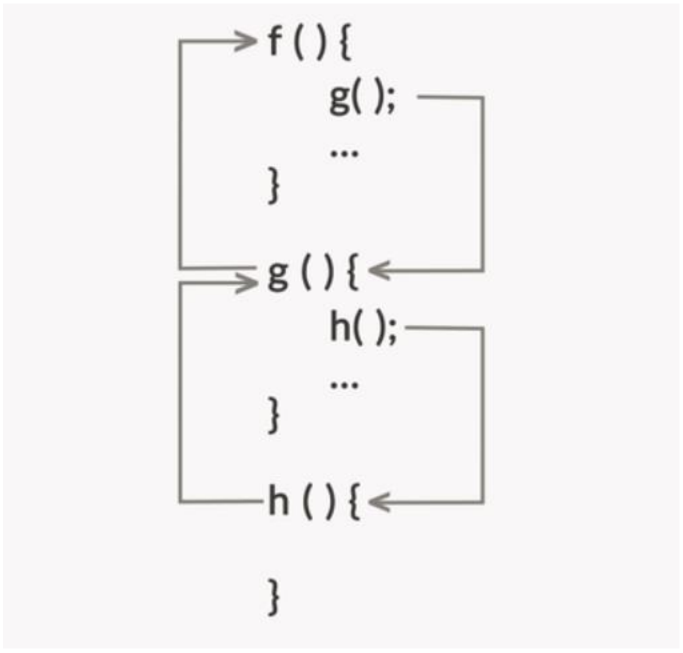
So the above code implemented using this library would have looked like the following:

```
import java.util.Stack;
import java.util.EmptyStackException;

public class MyStack<T> {

    public static void main(String[] arg) {
        Stack<String> stack = new Stack<String>();

        stack.push("OOP");
        stack.push("Algorithms");
        stack.push("Data Structures");

        try {
            while (true) {
                System.out.println("Popped " + stack.pop());
            }
        } catch (EmptyStackException e) {
            System.out.println("Done!");
        }
    }
}
```

Industry Applications

You saw the example of how an internet browser implements this stack functionality. Whenever you visit a website, it is added on top of a stack. When you push the back button, the website at the top of the stack is popped off, and you come at the site which was the previous visited website.

You also saw the example of a program stack. If you are given a function which makes call to different functions, it is done with the help of a stack data structure. Lets say you have a function – f() which calls a function g() which in turn calls a function h(). You could visualise using the diagram below:



```
f ( ) {
    g( );
    ...
}
g ( ) {
    h( );
    ...
}
h ( ) {

}
```

So, the program stack first finds a reference to the function f(), pushes it into the stack. Then finds the reference to function g(), then pushes it into the stack. Then, it finds a reference to function h(), again pushes it into the stack. When, h() is solved, it is popped from stack. Then g() is solved and popped and finally f() is popped.

Compilers (programs which take your program and turns it into a program in some other language, like machine language or byte code) do a very important thing called parsing. Parsing, in simple language, means reading. A compiler really uses parsing to figure out if a program it's compiling is really well-formed or not. For a program to successfully compile, it's necessary but not sufficient that it should be well-formed. If a program is not well-formed, i.e. if it's ill-formed, the compiler will typically detect that, and will flag a syntax error. Parsing is a complex process. Although, it was beyond the scope of discussion for this session, it is important to note that it uses Stacks for some of its functions. One of the functions is matching Parentheses.

Matching Parentheses Problem

One of the things that must happen for a program to be well-formed is that all its parentheses should match, i.e. if there's an open parenthesis anywhere in the program, there must also be a corresponding closing parenthesis. Obviously, the closing parenthesis must follow, and not precede, its corresponding open paren. For example, "(())" and "()()" are well-formed strings, but ")(" is not. So, having an equal number of open and close parentheses is necessary but not sufficient for a string to well-formed. You saw the following approaches to solve this problem.

Algorithm - Matching Parentheses using a *count* variable
1. Initialise variable *count* to 0.
2. Scan the string from left to right.
3. As you go symbol by symbol, whenever the algorithm scans an opening parenthesis, increase *count* by 1.
4. If the algorithm scans a closing parenthesis, decrease *count* by 1. However, if *count* is already 0, it means error; so return false.
5. If the algorithm reaches the end of string, and *count* = 0, it means that the string is well-formed; so return true. Otherwise, return false.

Algorithm - Matching Parentheses using Stack

1. Initialise stack *S* to be empty.
2. Scan the string from left to right.
3. As you go symbol by symbol, whenever you meet an open parenthesis, push '(' into *S*.
4. If you meet a closing parenthesis, pop a '(' from *S*. However, if *S* is already empty, it means error; so return false.
5. If you reach the end of string, and *S* is empty now, it means that the string is well-formed; so return true. Otherwise, return false.

The counter based approach fails when we consider multiple parenthesis. So, you modify the counter based approach to have two counters, one tracking parenthesis and the other tracking braces. The problem with this algorithm is that it will correctly accept all well-formed strings and will correctly reject some of the ill-formed strings, however, it will fail to detect strings where both the parentheses and braces are individually matched, but their relative positions are messed up, e.g. "({)}".

The stack algorithm when you consider multiple types of parentheses would be -

1. Initialise stack *S* to be empty.
2. Scan the string from left to right.
3. As you go symbol by symbol, whenever you meet an open parenthesis, push '(' into *S*. If see an open brace push '*{* ' into *S*.
4. If you meet a close parenthesis, pop a symbol from *S*. If it's not an open parenthesis (e.g. if it's an open brace), return false. Also, if *S* is already empty, it means error; so return false.
5. If you meet a close brace, pop a symbol from *S*. If it's not an open brace (e.g. if it's an open parenthesis), return false. Also, if *S* is already empty, it means error; so return false.
6. If you reach the end of string, and *S is empty*, it means that the string is well-formed; so return true. Otherwise, return false.

File Versioning System

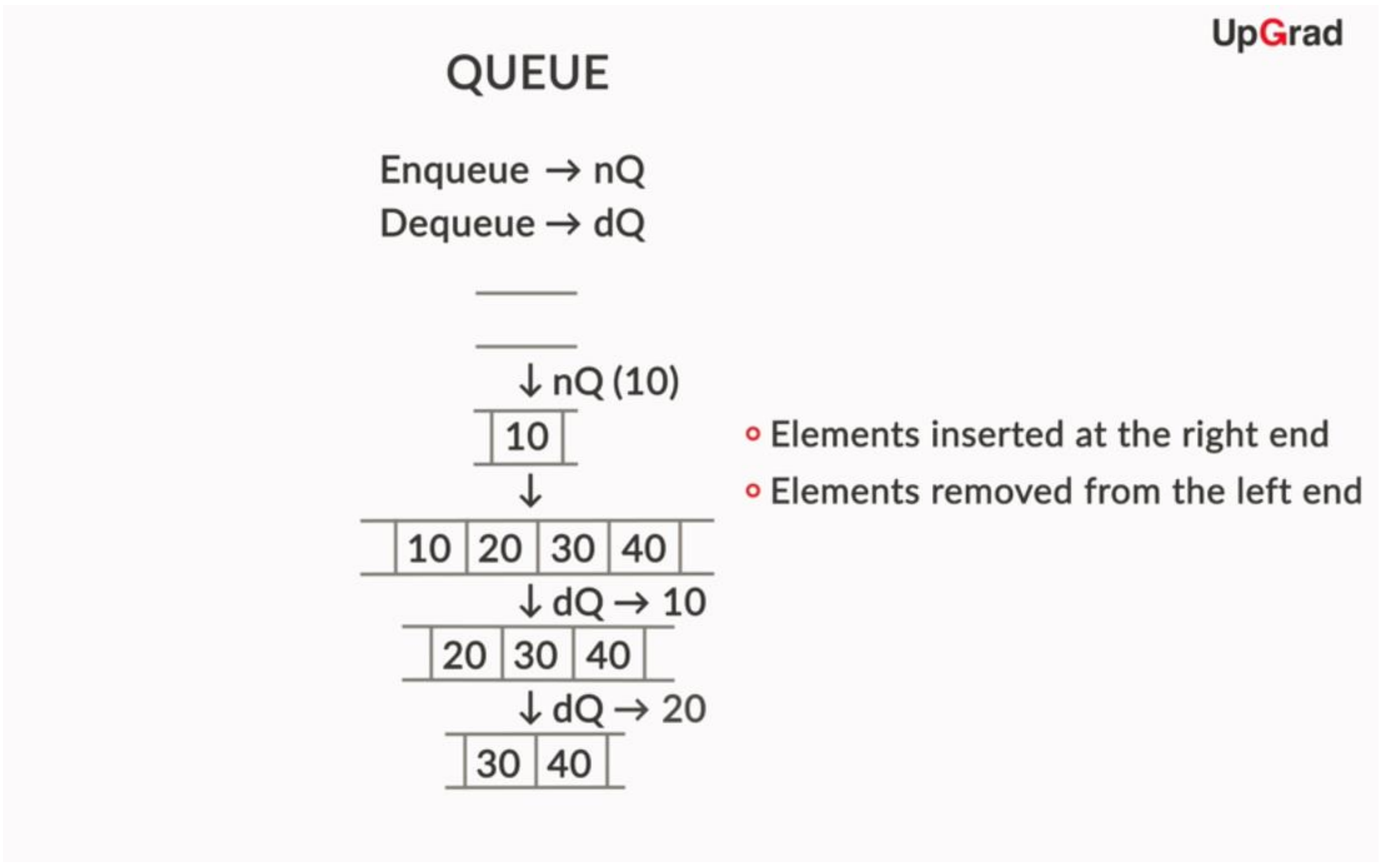
You then saw how to implement a file versioning system, like the one you see in MS Word or Google Docs. Stacks are best suited for this purpose. Whenever you make changes to a file lets say – documentVersion1, and you save the file, the new file gets stored as documentVersion2 and is pushed on top of the stack. Now when you have to undo your changes, you call, pop() which reverts to the previous version. You could go back multiple versions to restore an older version, or could see the no. of versions etc..

Queues

Stacks are data structures where elements follow **Last in, first out** order. You could add and remove objects from the same end, called the top of the stack.

Objects in Queues follow **First in, first out** principle. You could relate these with queues you see in movie theatres, where, the person who gets in first in the line is the first to get the ticket. Similarly, you see examples of queues in other booking systems as well.

Analogous to a push in stack, there is an enqueue in Queue. Similarly, analogous to pop in stack, there is a dequeue in Queue. You could visualise these operations through the image below.



Queues find use in various computing processes, such as scheduling printing jobs, scheduling system processes such as maintenance etc..

Implementing a Queue

Unlike Stack, which is a class in Java, Queue is an interface that often needs to be implemented as a linked list. You learnt about interfaces in polymorphism. They are classes that contain only abstract methods and cannot be instantiated. In Java, unlike Stack, Queue is an interface, which needs to be implemented by a class (usually the Linked List class) before you can instantiate and use it.

You can see the following implementation of the Queue. Where you initially add some tasks to the queue and then remove them.

```
import java.util.Queue;
import java.util.LinkedList;

public class ToDoList {
    public static void main(String[] args) {
        Queue<String> todolist = new LinkedList();
        makeToDoList(todolist);
        doAllTasks(todolist);
    }

    public static void makeToDoList(Queue<String> todolist) {
        todolist.add("task 1");
        todolist.add("task 2");
        todolist.add("task 3");
        todolist.add("task 4");
    }

    public static void doAllTasks(Queue<String> todolist) {
        while (todolist.size() != 0) {
            System.out.println(todolist.remove());
        }
    }
}
```

The output for the code above was –

```
task 1
task 2
task 3
task 4
```

You can see that the tasks were printed(removed) from the queue in the same order in which they were added.

Booking System using Queues

Ticket booking systems heavily use queues for their processes. You saw the implementation of such a system using Queues. When an individual would make a booking request, it would be stored in queues, so that the customer who makes the request first, gets tickets first. Another, important feature in the system was that a request could be accepted only if there were sufficient tickets in the system. So, if a customer is making a request for 4 tickets, but there are only three tickets in the system, this request would have to be denied. Thus, each time a request is made, the tickets available are checked. If the tickets are available, then the request is dequeued and the next request in the queue is processed.

More Applications

You would see the implementations of stacks and queues in Graphs as well, where you would be performing the breadth-first and the depth-first searches.