Unit 3: Stacks and Queues

Stacks

- A stack is a linear data structure that follows LIFO (Last In First Out) order to perform operations.
- Storing and retrieving operations can be performed only at one of its ends called top or tos (top of stack).
- Such a stack resembles piles of trays in a cafeteria. New trays are put on the top of the pile and top tray is the first tray removed from the stack. For this reason, a stack is called an LIFO structure: last in /first out.

The operations on stack are as follows:

1. **clear ()** : Clear the stack.
2. **isEmpty()** : Check to see if the stack is empty.
3. **Push (el)** : Put the element **el** on the top of stack.
4. **Pop()** : Take the topmost element from the stack.
5. **topEl()** : Return the topmost stack without removing it.

Stack can be implemented in two ways:

1. Array Implementation of stack

Algorithm for stack implementation using arrays

A. <u>Push Operation:</u>

- For performing push operation, check to see whether the stack is full or not.
- If we try to push an element onto a full stack, the condition is called stack overflow.
- The stack is full when **top = size-1**

<u>Algorithm that pushes an ITEM into a stack:</u>

Step 1: Is Stack full? If TOP = Size-1, then PRINT " Stack Overflow" and return.
Step 2: Set TOP = TOP+1
Step 3: Set STACK [TOP] = ITEM.
Step 4: Return

<u>Method to push an item onto a stack:</u>

```
void push (int stack[], int top, int size, int item)
{
        if(top == size - 1)
                System.out.println(" Stack Overflow");
        else
        {
                top = top + 1;
                stack[top] = item;
        }
}
```

B. Pop Operation:

- For performing pop operation, check to see whether the stack is empty or not.
- If we try to pop an element from an empty stack, the condition is called stack underflow.
- The stack is empty when **top < 0** or **top = -1**

Algorithm that pop an ITEM from a stack:

Step 1: If TOP < 0, then PRINT "Stack Underflow" and return.
Step 2: Set ITEM = STACK[TOP].
Step 3: Set TOP = TOP-1.
Step 4: Return

Method to pop an item from a stack:

```
void pop (int stack[], int top, int item)
{
        if(top == - 1)        // if(top < 0)
                System.out.println(" Stack Underflow");
        else
        {
                item = stack[top];
                top = top - 1;
        }
}
```

2. Linked List Implementation of Stack

A linked list of nodes could be used to implement a stack. Linked list representation of stack is nothing but a liked list of nodes where each node is element of the stack. To point to the top most node we have a pointer **top** that points to node recently added. Every new node is added to the beginning of the liked list and top point to it. The main advantage of this representation is that size of stack is not fixed so there is least chance of stack overflow. The structure of the node is:

```
class Node
{
        int data; Node next;
        public Node()
        {
                next = null;
        }
        public Node(int d)
        {
                data = d;
                next = null;
        }
}
```

- Generally, the stack is very useful in situations when data have to be stored and then retrieved in reverse order.

Examples of Stack Application:

1. Matching delimiters in a program
2. Adding very large numbers

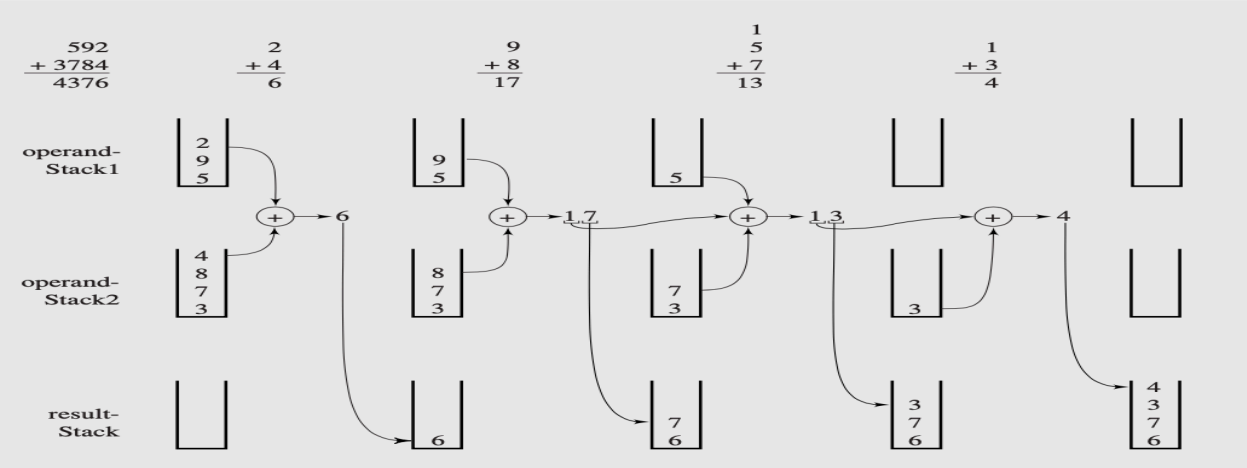
Algorithm for Matching delimiters in a program:

```
delimiterMaching(file)
    read character ch from file;
    while not end of file
        if ch is '(' ,'[', or '{'
            push(ch);
        else if ch is '/'
            read the next character;
             if this character is '*'
                 skip all characters until "*/" is found and report an error if the
                 end of file is reached before "*/" is encountered;
            else ch = the character read in;
                  continue; // go to the beginning of the loop;
        else if ch is ')', ']', or '}'
              if ch and popped off delimiter do not match
                 failure;
        // else ignore other characters;
        read next character ch from file;
    if stack is empty
        success;
    else
        failure;
```

Stack	Nonblank Character Read	Input Left
empty		s = t[5] + u / (v * (w + y));
empty	s	= t[5] + u / (v * (w + y));
empty	=	t[5] + u / (v * (w + y));
empty	t	[5] + u / (v * (w + y));
[	[	5] + u / (v * (w + y));
[	5	] + u / (v * (w + y));
empty	]	+ u / (v * (w + y));
empty	+	u / (v * (w + y));
empty	u	/ (v * (w + y));
empty	/	(v * (w + y));
(	(	v * (w + y));
(	v	* (w + y));
(	*	(w + y));
(		
(	(	w + y));
(		
(	w	+y));
(		
(	+	y));
(		
(	y	));
(	)	);
empty	)	;
empty	;	

Algorithm for Adding very large numbers:

```
addingLargeNumbers()
   read the numerals of the first number and store the numbers corresponding to
   them on one stack;
   read the numerals of the second number and store the numbers corresponding
   to them on another stack;
result = 0;
while  at least one stack is not empty
      pop a number from each nonempty stack and add them to  result;
      push the unit part on the result stack;
      store carry in  result;
   push carry on the result stack if it is not zero;
   pop numbers from the result stack and display them;
```



Queue

- The queue is a linear data structure that follows FIFO (First-In-First-Out) order to perform operations.
- In queue, enqueue (insertion) and dequeue (deletion) operations are performed at separate ends known as rear and front.

Some common operations are:

- **clear()** : Clear the queue.
- **isEmpty()** : Check to see if the queue is empty.
- **Enqueue(el)** : Put the element el at the end of the queue.
- **Dequeue()** : Take the first element from the queue.
- **firstEL()** : Return the first element in the queue without removing it.

Enqueue Operation:

- The operation of adding new element to the end of the queue is known as enqueue.
- Whenever a new item is added (enqueued) to the queue, rear pointer is used.
- During enqueue operation rear is incremented by 1 and data is stored in the queue at that location indicated by rear.

Method to insert (enqueue) an element in a queue:

```
void enqueue(int queue[], int rear, int size, int data)
{
        if(isFull())
                System.out.println("Overflow");
        else
        {
                 rear = rear + 1;
                queue[rear] = data;
                size++;
        }
}
```

Dequeue Operation:

- The operation of removing an element from the front of the queue is known as dequeue.
- Whenever an item is deleted from the queue, The front pointer is used.
- During dequeue operation, front pointer is incremented by 1 and the deleted item is returned.

Method to delete (dequeue) an element from a queue:

```
void dequeue(int queue[], int front, int size, int data)
{
        if(isEmpty())
                System.out.println("Underflow");
        else
        {
                front = front + 1;
                data = queue[front];
                size--;
        }
}
```

Queue can also be implemented in two ways.

1. Array Implementation

In the array implementation of linear queue, we take an array of fixed size and two variables front and rear. These two variables contain the current index at which insertion and deletion can be performed.
We use one dimensional array. Hence the size is static. The queue is empty if the front == rear and full if front == 0 and rear == n.
Deletion from the empty queue causes an underflow, while insertion onto a full queue causes an overflow.

2. Linked List Implementation of Queue

A linear queue can be implemented using linked list. The structure of the node remains same as that of stack but we have two pointers front and rear. The front pointer points to the stating of the queue (first item inserted) and rear points to the end of the queue (last item inserted). The structure of the node is as given below:

```
class Node
{
        protected int data; protected Node next;
        public Node()
        {
                next = null;
                data = 0;
        }
        public Node(int d)
        {
                data = d;
                next = null;
        }
}
```

Priority Queues

In many situations, simple queues are inadequate, as when first in /first out scheduling has to be overruled using some priority criteria. In a post office example, a handicapped person may have priority over others. Therefore, when a clear is available, a handicapped person is served instead of someone from the front of the queue.

In a sequence of processes, process p2 may need to be executed before process p1 for the proper functioning of the system, even though p1 was put on the queue of waiting processes before p2. In situations like these, a modified queue or priority queue is needed. In priority queues, elements are dequeued according to their priority and current position.

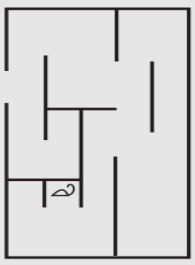
The problem with a priority queue is in finding an efficient implementation that allows relatively fast enquiring. Because elements may arrive randomly to the queue, there is no guarantee that the front element will be most likely to be dequeued and that the elements put at the end will be the most likely to be dequeued. The situation is complicated because a wide spectrum of possible priority criteria can be used in different cases such as frequency of use, birthday, salary, position, status, and others.

Priority queues can be represented by two variations of linked lists. In one type of linked list, all elements are entry ordered, and in another, order is maintained by putting a new element in its proper position according to its priority. In both cases, the total operational times are O(n) because, for an unordered list, adding an element is immediate but searching is O(n), and in a sorted list, taking an element is immediate but adding an element is O(n).

Exiting a Maze

Consider the problem of a trapped mouse that tries to find its way to an exit in a maze. The mouse hopes to escape from the maze by systematically trying all the routes. If it reaches a dead end, it retraces its steps to the last position and begins at least one more untried path. For each position, the mouse can go in one of four directions: left, right, up, down. Regardless of how close it is to the exit, it always tries the open paths in this order, which may lead to some unnecessary detours. By retaining information that allows for resuming the search after a dead end is reached, the mouse uses a method called backtracking.

(a) A mouse in a maze; (b) two-dimensional character array representing this situation.



(a)

```
11111111111
10000010001
10100010101
e0100000101
10111110101
10101000101
10001010001
11111010001
101m1010001
10000010001
11111111111
```

(b)

The maze is implemented as a two dimensional character array in which passages are marked with 0s and walls by 1s, exit position by the letter e, and the initial positions of the mouse by the letter m. In this program, the maze problem is slightly generalized by allowing the exit to be in any position of the maze and allowing passages to be on the borderline. To protect itself from falling off the array by trying to continue its path when an open cell is reached on one of the borderlines, the mouse also has to constantly check whether it is in such a borderline position or not. To avoid it, the program automatically puts a frame of 1s around the maze entered by the user.

The program uses two stacks: one to initialize the maze and another to implement backtracking.
The user enters a maze one line at a time. The maze entered by the user can have any number of rows and any number of columns. The only assumption the program makes is that all rows are of the same length and that it uses only these characters: any number of 1s, any number of 0s, one e, and one m. The rows are pushed on the stack mazeRows in the order they are entered after attaching one 1 at the beginning and one 1 at the end. After all rows are entered, the size of the array store can be determined and then the rows from the stack are transferred to the array.
A second stack, mazeStack, is used in the process of escaping the maze. To remember untried paths for subsequent tries, the positions of the untried neighbors of the current position (if any) are stored on a stack and always in the same order, first upper neighbor, then lower, then left, and finally right.

Here is a pseudo code of an algorithm for escaping a maze:

```
exitMaze()
        initialize stack, exitCell, entryCell, currentCell = entryCell;
        while currentCell is not exitCell
                mark currentCell as visited;
                push onto the stack the unvisited neighbours of currentCell;
                if stack is empty
                        failure;
                else pop off a cell from the stack and make it currentCell;
        success;
```