18CSC201J
DATA STRUCTURES AND ALGORITHMS

UNIT 3

Syllabus

- S1- Stack ADT, Stack - Array Implementation
- S2- Stack Linked List Implementation, Applications of Stack-Infix to Postfix Conversion
- S3- Applications of Stack-Postfix Evaluation, Balancing symbols
- S6- Applications of Stack- Nested Function Calls, Recursion concept using stack
- S7- Tower of Hanoi, Queue ADT
- S8- Queue implementation using array and Linked List
- S11- Circular Queue and implementation
- S12- Applications of Queue and double ended queue
- S13- Priority Queue and its applications

SESSION 1

STACK ADT

- **Abstract Data Type (ADT)**
  - It is a mathematical model for **data types**
  - An **abstract data type** is **defined** by its behavior (semantics) from the point of view of a user, of the **data**, specifically in terms of possible values, possible operations on **data** of this **type**, and the behavior of these operations. Set of values and set of operations
- A stack is an ADT
  - It follows **Last In First Out (LIFO)** methodology perform operations push, pop, etc.

What is STACK?

Diagram Reference : http://www.firmcodes.com/write-a-c-program-to-implement-a-stack-using-an-array-and-linked-list/

- Stack works on "Last in First out" or "First in Last Out", principle.
- Plates placed one over another
- The plate at the top is removed first & the bottommost plate kept longest period of time.
- So, it follows LIFO/FILO order.

Stack Example

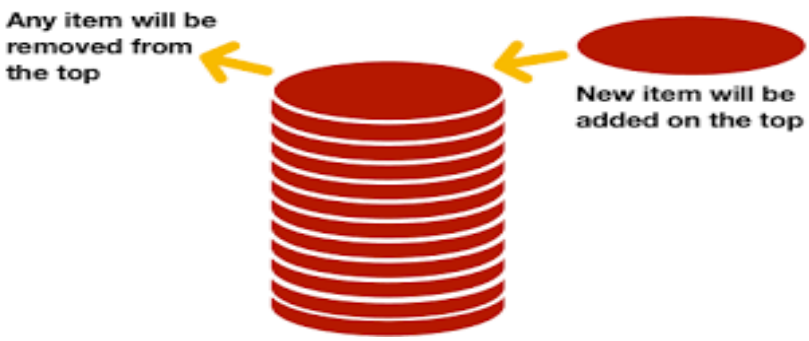


Diagram Reference:https://www.codesdope.com/course/data-structures-stacks/

- Always new items are added at top of the stack and also removed from the top of the stack only.

- Entry and Exit at same point

STACK – Data Structure

- Stack is a **Linear Data Structure**
- Follows a particular order to perform the operations.
- The order are either

LIFO(Last In First Out)

OR

FILO(First In Last Out).

STACK - Operations

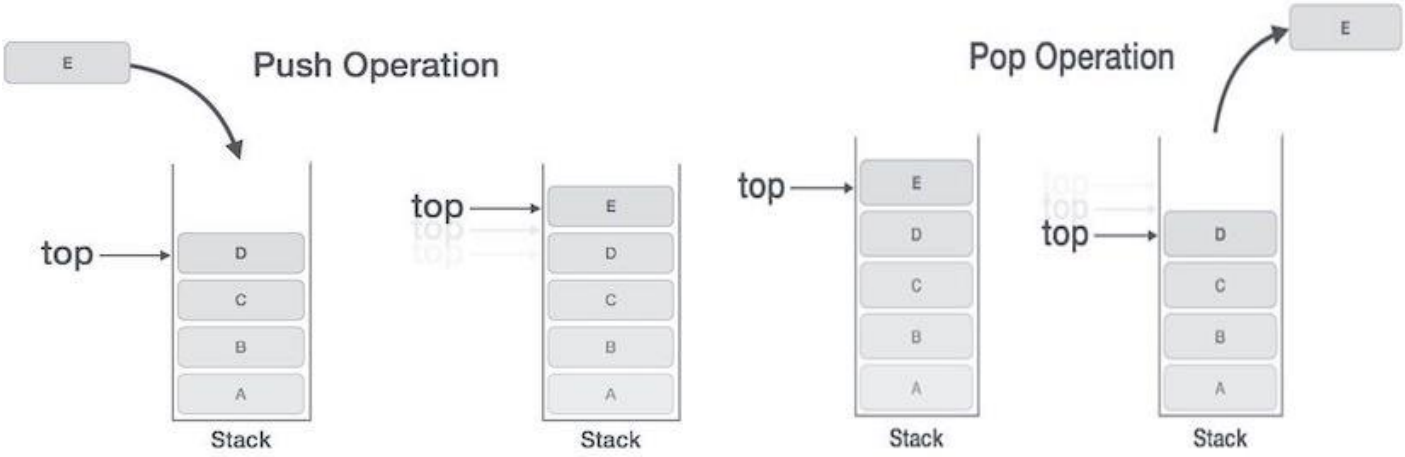
TWO PRIMARY OPERATIONS

- **push**
  - Pushing (storing) an element on the stack
  - If the stack is full, Overflow condition is enabled.
- **Pop**
  - Removing (accessing) an element from the stack.
  - The elements are popped in the decreasing order.
  - If the stack is empty, Underflow condition enabled.

STACK – Additional Functionality

- For effective utilization of the stack, Status of the stack should be checked. For that additional functionality is of the stack is given below
- **Peek or Top**
  - Accessing the top element. Without removing the top element.
- **isFull**
  - check if stack is full.
- **isEmpty**
  - check if stack is empty.

Stack Operation Example



Stack Implementation - Array

Stack - Array

- One dimensional array is enough to implement the stack
- Array size always fixed
- Easy to implement
- Create fixed size one dimensional array
  - insert or delete the elements into the array using **LIFO principle** using the variable **'top'**

Stack - ***top***

- About ***top***
  - Initial value of the top is -1
  - To insert a value into the stack, increment the top value by one and then insert
  - To delete a value from the stack, delete the top value and decrement the top value by one

Steps to create an empty stack

1. Declare the functions like push, pop, display etc. need to implement the stack.

1. Declare one dimensional array with fixed size

1. Declare a integer variable **'top'** and initialize it with **'-1'**. (**int top = -1**)

push(Value)
Inserting value into the stack

- push() is a function used to insert a new element into stack at **top** position.
- Push function takes one integer value as parameter

1. Check whether **stack** is **FULL** based on (**top == SIZE-1**)

1. If stack is **FULL**, then display **"Stack is FULL Not able to Insert element"** and terminate the function.

1. If stack is **NOT FULL**, then increment **top** value by one (**top=top+1**) and set (**stack[top] = value**)

pop() – Delete a value from the Stack

- pop() is a function used to delete an element from the stack from **top** position
- Pop function does not take any value as parameter

1. Check whether **stack** is **EMPTY** using (**top == -1**)

1. If stack is **EMPTY**, then display **"Stack is EMPTY No elements to delete"** and terminate the function

1. If stack is **NOT EMPTY**, then delete **stack[top]** and decrement **top** value by one (**top=top-1**)

display()
Displays the elements of a Stack

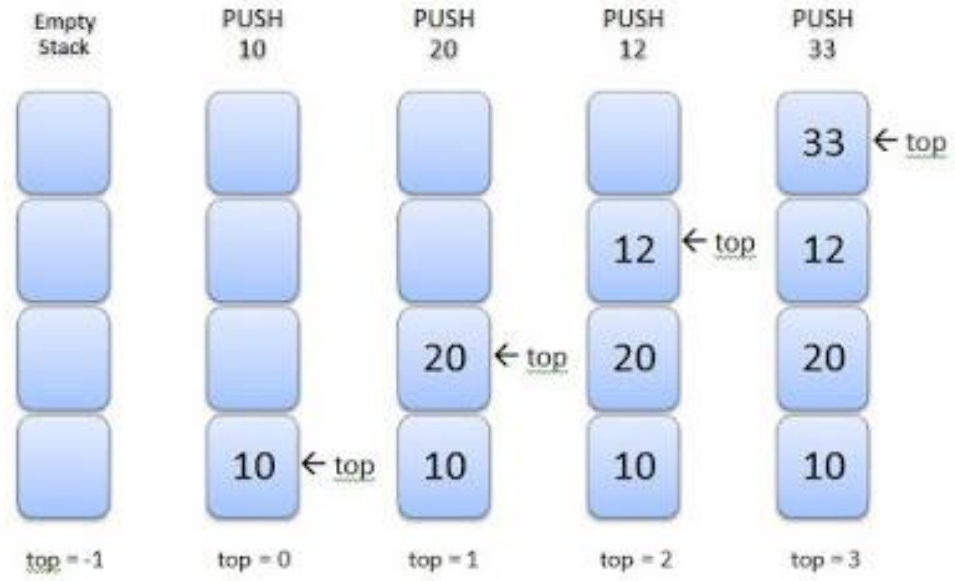
- **display() – function used to Display the elements of a Stack**

1. Check whether **stack** is **EMPTY based on** (**top == -1**)

1. If stack is **EMPTY**, then display **"Stack is EMPTY"** and terminate the function

1. If stack is **NOT EMPTY**, display the value in decreasing order of array

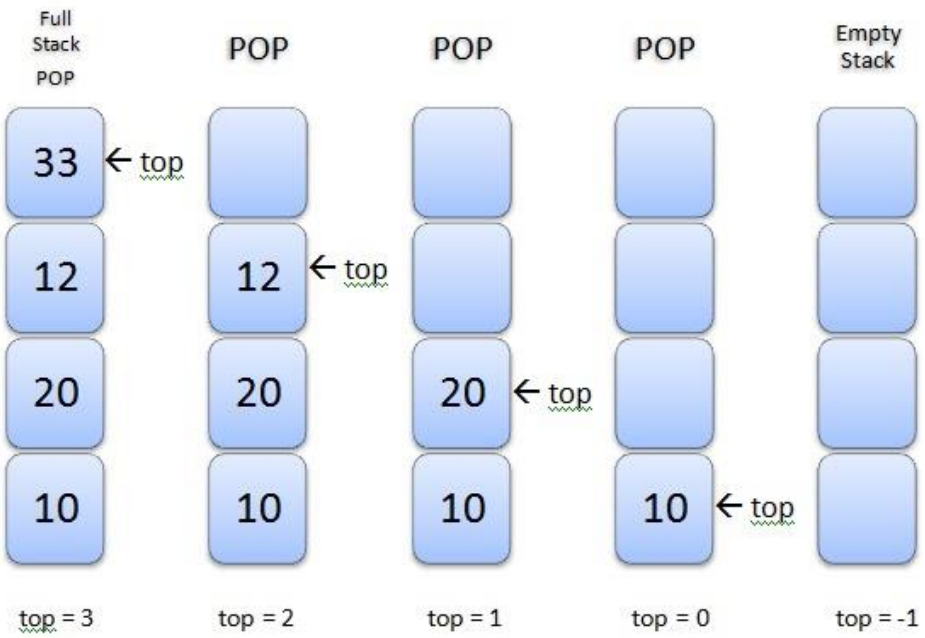
Stack Push Example



PUSH operation on Stack with the position of top pointer 'top' is initialized to -1. Each time an element is pushed, then top = top+1.

Diagram reference :http://www.exploredatabase.com/2018/01/stack-abstract-data-type-data-structure.html

Stack Pop Example



POP operation on Stack with the position of top pointer
Each time an element is popped, then top = top - 1.

Another Example of Stack Push & Pop

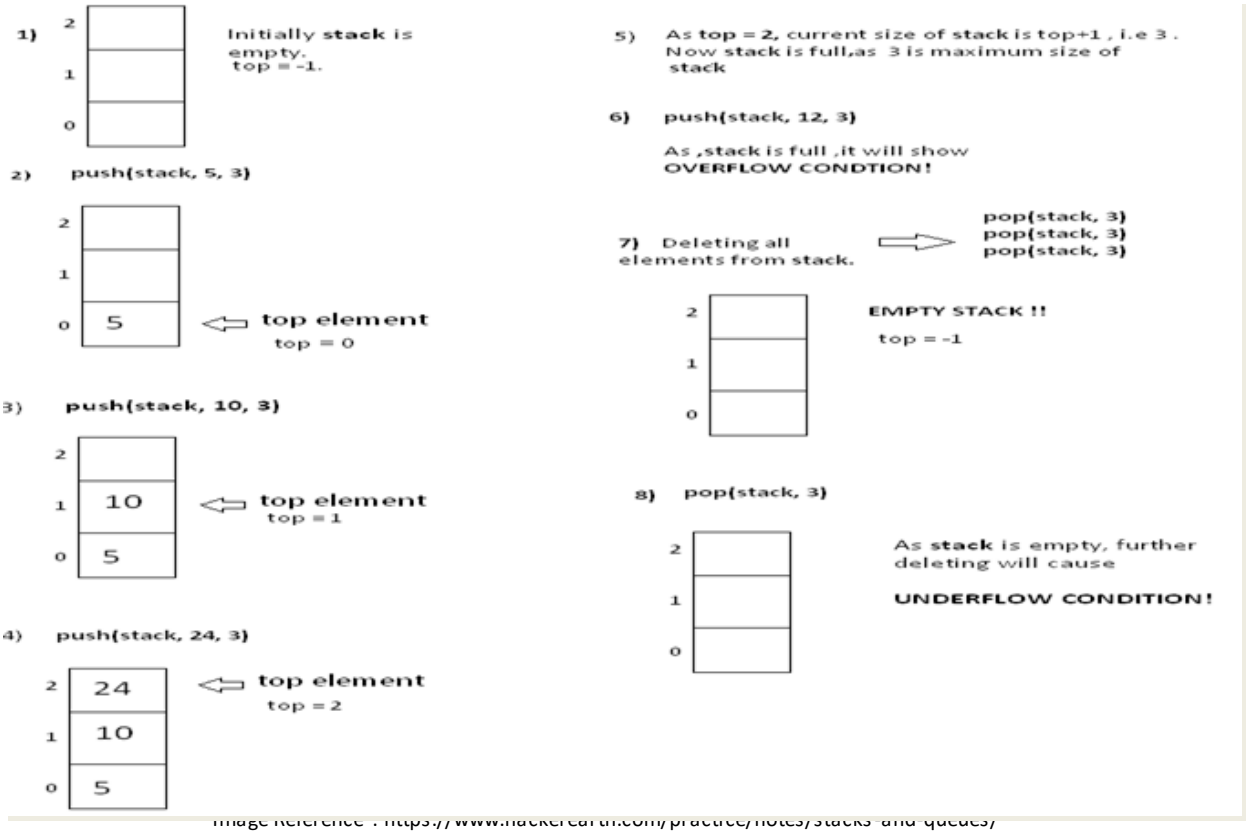


Image Reference : https://www.hackerearth.com/practice/notes/stacks-and-queues/

Cons Stack - Array

- The size of the array should be mentioned at the beginning of the implementation

- If more memory needed or less memory needed that time this array implementation is not useful

SESSION 2

Stack Implementation – Linked List

Pros Stack – Linked List

- A stack data structure can be implemented by using a linked list

- The stack implemented using linked list can work for the variable size of data. So, there is no need to fix the size at the beginning of the implementation

Stack – Linked list

- Dynamic Memory allocation of Linked list is followed
- The nodes are scattered and non-contiguously in the memory
- Each node contains a pointer to its immediate successor node in the stack
- Stack is said to be overflown if the space left in the memory heap is not enough to create a node

Stack – Linked list

- In linked list implementation of a stack, every new element is inserted as '**top**' element.
- That means every newly inserted element is pointed by '**top**'.
- To remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its previous node in the list.
- The **next** field of the **First** element must be always **NULL**.

Example Stack – Linked List

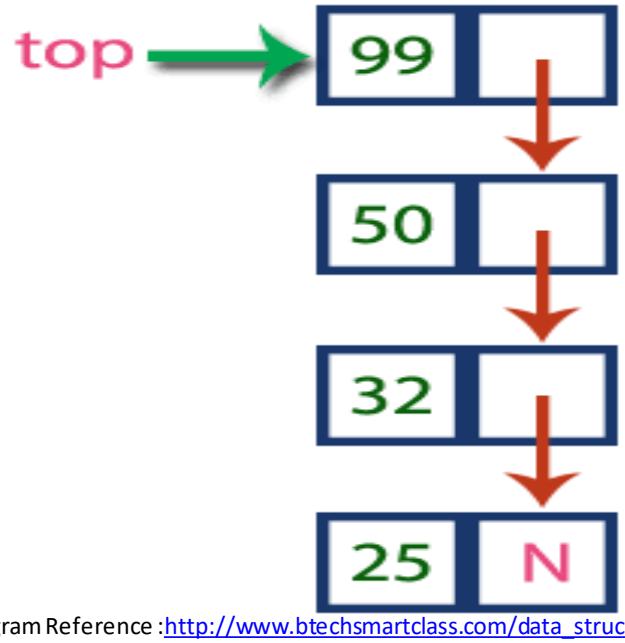


Diagram Reference :http://www.btechsmartclass.com/data_structures/stack-using-linked-list.html

Create Node – Linked List

1. Include all the **header files** which are used in the program. And declare all the **user defined functions**
2. Define a '**Node**' structure with two members **data** and **next**
3. Define a **Node** pointer '**top**' and set it to **NULL**
4. Implement the **main** method by displaying Menu with list of operations and make suitable function calls in the **main** method

push(value)
Inserting an element into the Stack

1. Create a **newNode** with given value

1. Check whether stack is **Empty** (**top** == **NULL**)

1. If it is **Empty**, then set **newNode → next** = **NULL**

1. If it is **Not Empty**, then set **newNode → next** = **top**

1. Finally, set **top** = **newNode**

pop()
Deleting an Element from a Stack

1. Check whether **stack** is **Empty** (**top == NULL**)

1. If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function

1. If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**‘

1. Then set '**top** = **top → next**‘

1. Finally, delete '**temp**'. (**free(temp)**)

display()
Displaying stack of elements

1. Check whether stack is **Empty** (**top** == **NULL**)
2. If it is **Empty**, then display **'Stack is Empty!!!'** and terminate the function
3. If it is **Not Empty**, then define a Node pointer **'temp'** and initialize with **top**
4. Display '**temp → data** --->' and move it to the next node. Repeat the same until **temp** reaches to the first node in the stack. (**temp → next** != **NULL**)
5. Finally! Display '**temp → data** ---> **NULL**'

Stack - Applications

1. Infix to Postfix Conversion
2. Postfix Evaluation
3. Balancing Symbols
4. Nested Functions
5. Tower of Hanoi

Infix to Postfix Conversion

Introduction

- Expression conversion is the most important application of stacks
- Given an infix expression can be converted to both prefix and postfix notations
- Based on the Computer Architecture either Infix to Postfix or Infix to Prefix conversion is followed

What is Infix, Postfix & Prefix?

- **Infix Expression** : The operator appears in-between every pair of operands. **operand1 operator operand2 (a+b)**

- **Postfix expression**: The operator appears in the expression after the operands. **operand1 operand2 operator (ab+)**

- **Prefix expression**: The operator appears in the expression before the operands. **operator operand1 operand2 (+ab)**

Example – Infix, Prefix & Postfix

Infix	Prefix	Postfix
(A + B) / D	**/ + A B D**	**A B + D /**
(A + B) / (D + E)	**/ + A B + D E**	**A B + D E + /**
(A - B / C + E)/(A + B)	**/ + - A / B C E + A B**	**A B C / - E + A B + /**
- 4 * A * C	**- ^ B 2 * * 4 A C**	**B 2 ^ 4 A * C * -**

Why postfix representation of the expression?

- Infix expressions are readable and solvable by humans because of easily distinguishable order of operators, but compiler doesn't have integrated order of operators.

- The compiler scans the expression either from left to right or from right to left.

Why postfix representation of the expression?

- Consider the below expression: a op1 b op2 c op3 d
  If op1 = +, op2 = *, op3 = +

a + b *c +d

- The compiler first scans the expression to evaluate the expression b * c, then again scan the expression to add a to it. The result is then added to d after another scan.

Why postfix representation of the expression?

- The repeated scanning makes it very in-efficient. It is better to convert the expression to postfix(or prefix) form before evaluation.

- The corresponding expression in postfix form is: abc*+d+.

- The postfix expressions can be evaluated easily in a single scan using a stack.

ALGORITHM

Step 1 : Scan the Infix Expression from left to right.

Step 2 : If the scanned character is an operand, append it with final Infix to Postfix string.

Step 3 : Else,

Step 3.1 : If the precedence order of the scanned(incoming) operator is greater than the precedence order of the operator in the stack (or the stack is empty or the stack contains a '('), push it on stack.

Step 3.2 : Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)

Step 4 : If the scanned character is an '(' , push it to the stack.

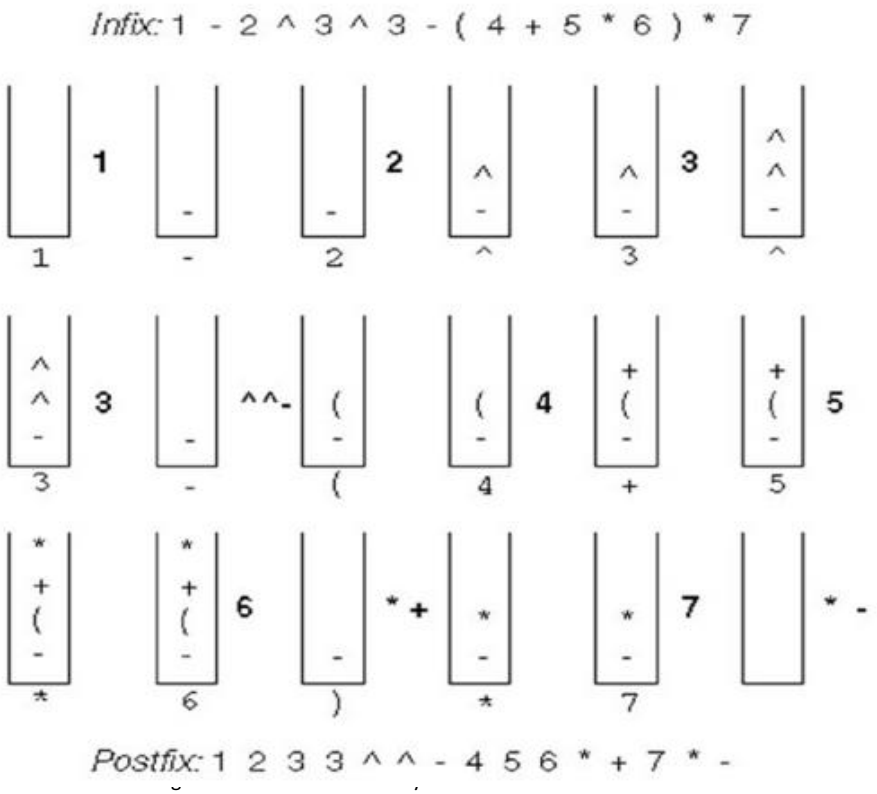
Step 5 : If the scanned character is an ')', pop the stack and and output it until a '(' is encountered, and discard both the parenthesis.

Step 6 : Repeat steps 2-6 until infix expression is scanned.

Step 7 : Print the output

Step 8 : Pop and output from the stack until it is not empty.

Example



Exercise Problems to Solve Infix to Postfix conversion

1. A+B-C
2. A+B*C
3. (A+B)*C
4. (A+B)*(C-D)
5. ((A+B)*C-(D-E))%(F+G)
6. A*(B+C/D)
7. ((A*B)+(C/D))
8. ((A*(B+C))/D)
9. (A*(B+(C/D)))
10. (2+((3+4)*(5*6)))
11. B ^ 2 - 4 * A * C
12. (A - B / C + E)/(A + B)

SESSION 3

Postfix Expression Evaluation

Postfix Expression

- A postfix expression is a collection of operators and operands in which the operator is placed after the operands.

Image Reference : http://www.btechsmartclass.com/data_structures/expressions.html

Postfix Expression Evaluation using Stack

Algorithm

1. Read all the symbols one by one from left to right in the given Postfix Expression
2. If the reading symbol is operand, then push it on to the Stack.
3. If the reading symbol is operator (+ , - , * , / etc.,), then perform TWO pop operations and store the two popped operands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 and push result back on to the Stack.
4. Finally! perform a pop operation and display the popped value as final result.

Example 1

Infix Expression (5 + 3) * (8 - 2)

Postfix Expression 5 3 + 8 2 - *

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Reading Symbol	Stack Operations	Evaluated Part of Expression
Initially	Stack is Empty	Nothing
5	push(5)	Nothing
3	push(3)	Nothing

+	value1 = pop()
	value2 = pop()
	result = value2 + value1
	push(result)
		value1 = pop(); // 3
		value2 = pop(); // 5
		result = 5 + 3; // 8
		Push(8)
		(5 + 3)

8	push(8)	(5 + 3)
2	push(2)	(5 + 3)

-	value1 = pop()
	value2 = pop()
	result = value2 - value1
	push(result)
		value1 = pop(); // 2
		value2 = pop(); // 8
		result = 8 - 2; // 6
		Push(6)
		(8 - 2)
		(5 + 3) , (8 - 2)

*	value1 = pop()
	value2 = pop()
	result = value2 * value1
	push(result)
		value1 = pop(); // 6
		value2 = pop(); // 8
		result = 8 * 6; // 48
		Push(48)
		(6 * 8)
		(5 + 3) * (8 - 2)

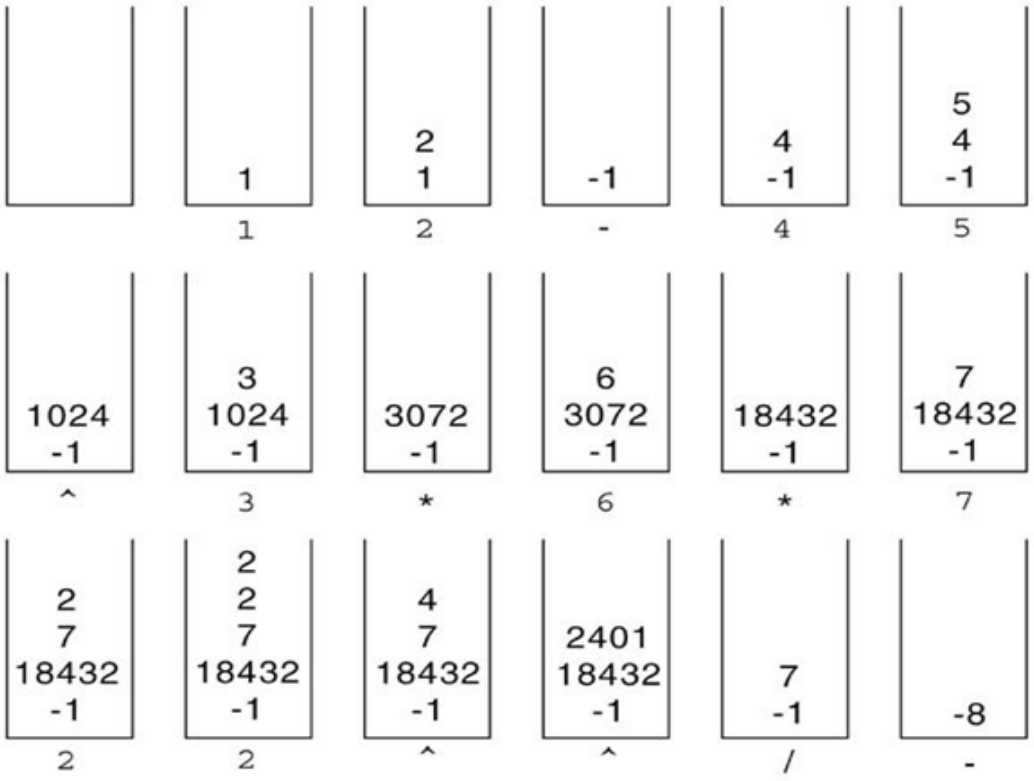
$ End of Expression	result = pop()	Display (result) 48 As final result

Infix Expression (5 + 3) * (8 - 2) = 48

Postfix Expression 5 3 + 8 2 - * value is 48

Example 2

Postfix Expression: 1 2 - 4 5 ^ 3 * 6 * 7 2 2 ^ ^ / -



Balancing Symbols

Introduction

- Stacks can be used to check if the given expression has balanced symbols or not.
- The algorithm is very much useful in compilers.
- Each time parser reads one character at a time.
  - If the character is an opening delimiter like ‘(‘ , ‘{‘ or ‘[‘ then it is PUSHED in to the stack.
  - When a closing delimiter is encountered like ‘)’ , ‘}’ or ‘]’ is encountered, the stack is popped.
  - The opening and closing delimiter are then compared.
  - If they match, the parsing of the string continues.
  - If they do not match, the parser indicates that there is an error on the line.

Balancing Symbols - Algorithm

- Create a stack
- while (end of input is not reached) {
  - If the character read is not a symbol to be balanced, ignore it.
  - If the character is an opening delimiter like (, { or [, PUSH it into the stack.
  - If it is a closing symbol like) , } ,] , then if the stack is empty report an error, otherwise POP the stack.
  - If the symbol POP-ed is not the corresponding delimiter, report an error.
- At the end of the input, if the stack is not empty report an error.

Example 1

Expression	Valid?	Description
(A+B) + (C-D)	Yes	The expression is having balanced symbol
((A+B) + (C-D)	No	One closing brace is missing.
((A+B) + [C-D])	Yes	Opening and closing braces correspond
((A+B) + [C-D]]	No	The last brace does not correspond with the first opening brace.

Example 2

Eg: [a+(b*c)+{(d-e)}]

[	Push [
[(	Push (
[	) and (matches, Pop (
[{	Push {
[{ (	Push (
[{	matches, pop (
[	Matches, pop {
	Matches, pop [

Thus, parenthesis match here

References

1. https://www.tutorialspoint.com/data_structures_algorithms/stack_algorithm.htm
2. https://www.codesdope.com/course/data-structures-stacks/
3. http://www.firmcodes.com/write-a-c-program-to-implement-a-stack-using-an-array-and-linked-list/
4. http://www.btechsmartclass.com/data_structures/stack-using-linked-list.html
5. http://www.exploredatabase.com/2018/01/stack-abstract-data-type-data-structure.html
6. https://www.geeksforgeeks.org/stack-set-2-infix-to-postfix/
7. https://www.hackerearth.com/practice/notes/stacks-and-queues/
8. https://github.com/niinpatel/Parenthesis-Matching-with-Reduce-Method
9. https://www.studytonight.com/data-structures/stack-data-structure
10. https://www.programming9.com/programs/c-programs/230-c-program-to-convert-infix-to-postfix-expression-using-stack

SESSION 6

Applications of Stack: Function Call and Return

- 2 types of Memory
  - Stack
  - Heap
- Stack memory stores the data (variables) related to each function.
- Why is it called stack memory?
  - The last function to be called is the first to return (LIFO)
  - The data related to a function are pushed into the stack when a function is called and popped when the function returns

Function Calls

- Function calls another function

->We are executing function A. In the course of its execution, function A calls another function B. Function B in turn calls another function C, which calls function D.

Function A

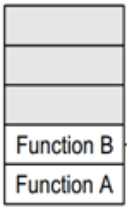
 Function B

 Function C

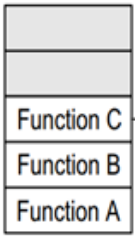
 Function D

- When A calls B, A is pushed on top of the system stack. When the execution of B is complete, the system control will remove A from the stack and continue with its execution.

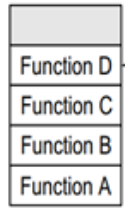
- When B calls C, B is pushed on top of the system stack. When the execution of C is complete, the system control will remove B from the stack and continue with its execution.

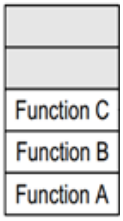


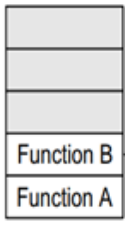
- When C calls D, C is pushed on top of the system stack. When the execution of D is complete, the system control will remove C from the stack and continue with its execution.

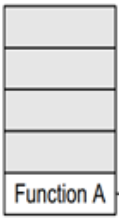


- When D calls E, D is pushed on top of the system stack. When the execution of E is complete, the system control will remove D from the stack and continue with its execution.
- When E has executed, D will be removed for execution.
- When C has executed, B will be removed for execution.
- When D has executed, C will be removed for execution.
- When B has executed, A will be removed for execution.









Nested Function Calls

- Consider the code snippet below:

```
main()
{
      ...
      foo(
);

      bar(
);
}
```

```
foo()
{
      ...
      bar(
);
      ...
}
```

```
bar()
{
      ...
}
```

Nested Function Calls and Returns in Stack Memory

- Stack memory when the code executes:

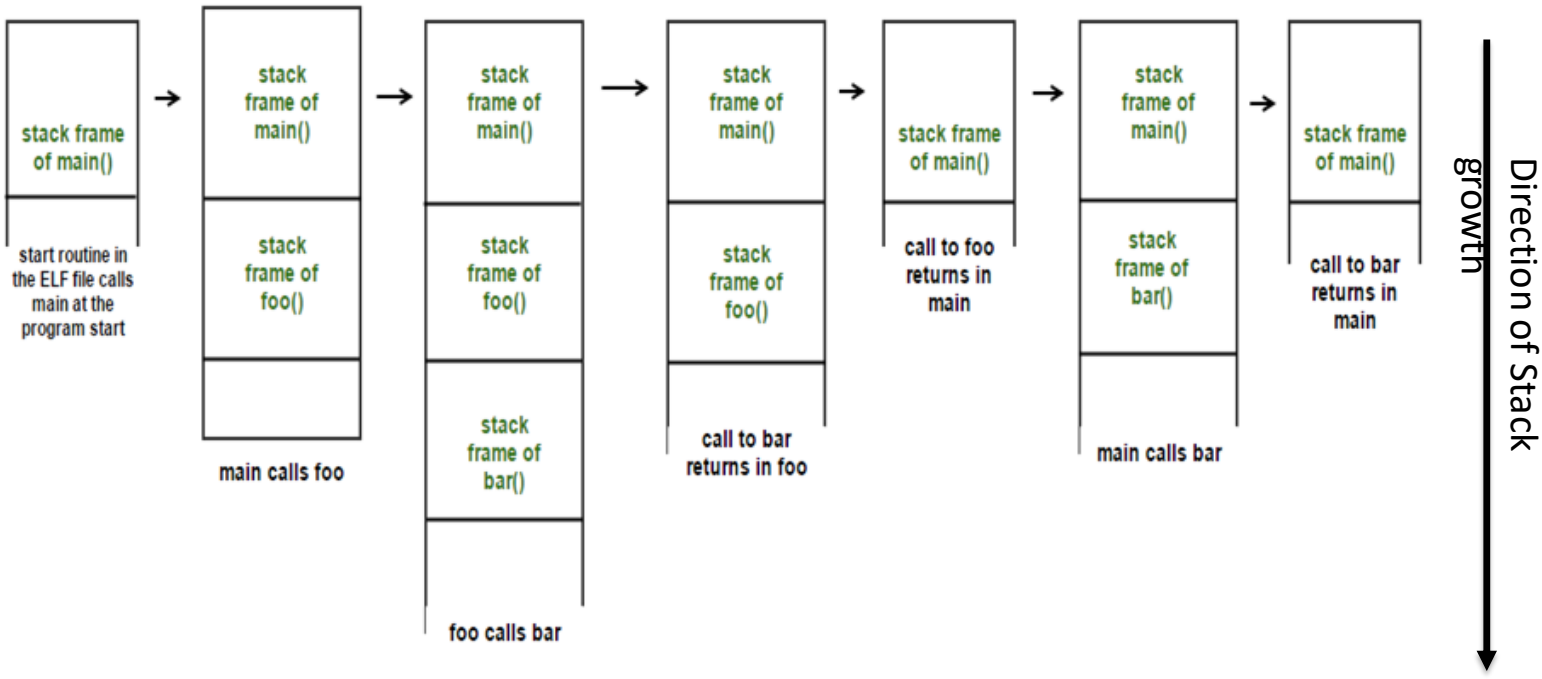


Image Source: https://loonytek.com/2015/04/28/call-stack-internals-part-1/

Function Call and Return in Stack Memory

- Each call to a function pushes the function's activation record (or stack frame) into the stack memory
- Activation record mainly consists of: local variables of the function and function parameters
- When the function finishes execution and returns, its activation record is popped

Recursion

- A recursive function is defined as a function that calls itself.
- Since a recursive function repeatedly calls itself, it makes use of the system stack to temporarily store the return address and local variables of the calling function.
- Every recursive solution has two major cases. They are
  - Base case, in which the problem is simple enough to be solved directly without making any further calls to the same function.
  - Recursive case, in which first the problem at hand is divided into simpler sub-parts. Second the function calls itself but with sub-parts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts.

Recursion

- A recursive function is a function that calls itself during its execution.
- Each call to a recursive function pushes a new activation record (a.k.a stack frame) into the stack memory.
- Each new activation record will consist of freshly created local variables and parameters for that specific function call.
- So, even though the same function is called multiple times, a new memory space will be created for each call in the stack memory.

Recursion Handling by Stack Memory

- Factorial program

```
int fact( int n)
{
   if (n==1)
      return 1;
   else
      return
(n*fact(n-1));
 }

 fact(4);
```

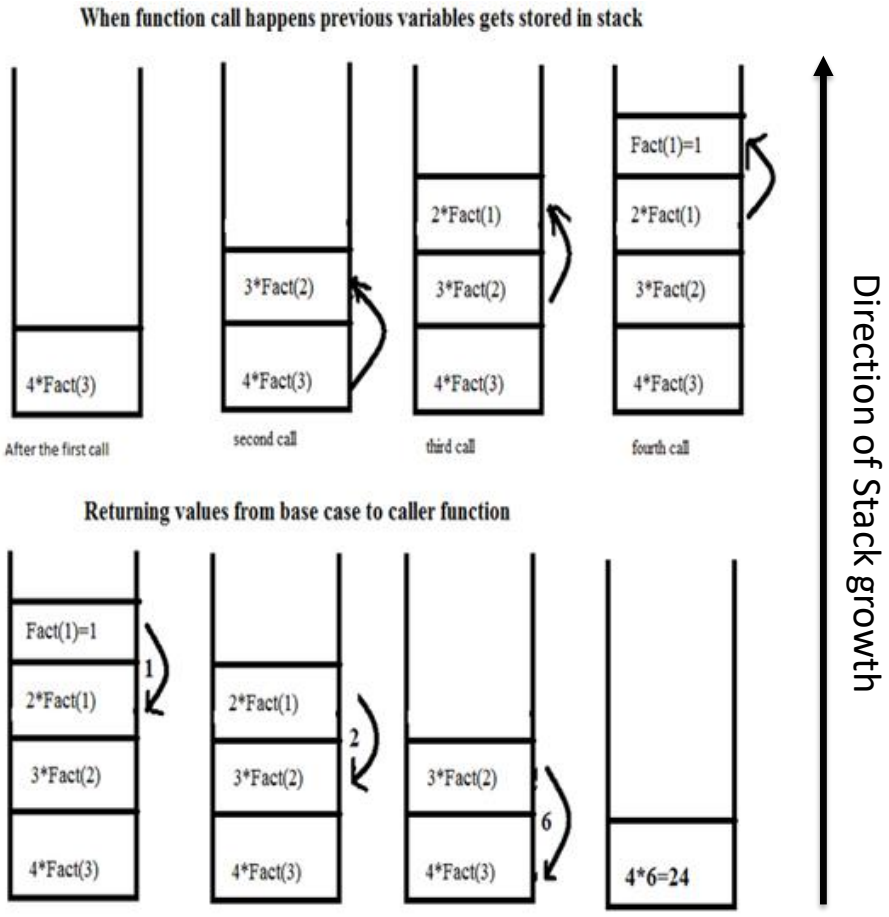


Image Source: https://stackoverflow.com/questions/19865503/can-recursion-be-named-as-a-simple-function-call

Example : Factorial Of a Number

To calculate n!, we multiply the number with factorial of the number that is 1 less than that number.

$$n! = n * (n–1)!$$

$5! = 5 * 4!$ Where $4! = 4 * 3!$

$= 5 * 4 * 3!$ Where $3! = 3 * 2!$

$= 5 * 4 * 3 * 2!$ Where $2! = 2*1!$

$= 5 * 4 * 3 * 2 * 1!$ Where $1! = 1$

$= 5 * 4 * 3 * 2 * 1$

$= 120$

Base case and Recursive Case

- Base case

When n = 1, because if n = 1, the result will be 1 as 1! = 1.

- Recursive case

Factorial function will call itself but with a smaller value of n, this case can be given as

$$factorial(n) = n \times factorial\ (n–1)$$

Advantages of using a recursive program

- Recursive solutions often tend to be shorter and simpler than non-recursive ones.
- Code is clearer and easier to use.
- Recursion works similar to the original formula to solve a problem.
- Recursion follows a divide and conquer technique to solve problems.
- In some (limited) instances, recursion may be more efficient.

Drawbacks/Disadvantages of using a recursive program

- For some programmers and readers, recursion is a difficult concept.
- Recursion is implemented using system stack. If the stack space on the system is limited,
- Recursion to a deeper level will be difficult to implement.
- Aborting a recursive program in midstream can be a very slow process.
- Using a recursive function takes more memory and time to execute as compared to its nonrecursive counterpart.
- It is difficult to find bugs, particularly while using global variables.

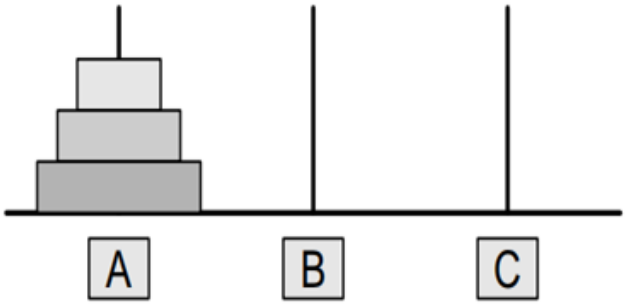
Factorial using recursion

Step 1: Specify the base case which will stop the function from making a call to itself.
Step 2: Check to see whether the current value being processed matches with the value of the base case. If yes, process and return the value.
Step 3: Divide the problem into smaller or simpler sub-problems.
Step 4: Call the function from each sub-problem.
Step 5: Combine the results of the sub-problems.
Step 6: Return the result of the entire problem.

```
int Fact(int n)
{
        if(n==1)
                return 1;
        else
            return (n * Fact(n-1));
}
```

Towers of Hanoi

- Figure below shows three rings mounted on pole A. The problem is to move all these rings from pole A to pole C while maintaining the same order. The main issue is that the smaller disk must always come above the larger disk



A-> Source pole

B-> Spare pole

C-> Destination pole

- To transfer all the three rings from A to C, we will first shift the upper

two rings (n–1 rings) from the source pole to the spare pole.

1) Source pole -> Destination pole

2) Source pole -> Spare pole

3) Destination pole -> Spare pole

- n–1 rings have been removed from pole A, the nth ring can be easily moved from the source pole (A) to the destination pole (C).

4) Source pole -> Destination pole

- The final step is to move the n–1 rings from the spare pole (B) to

the destination pole (C).

5) Spare pole -> Source pole

6) Spare pole -> Destination pole

7) Source pole -> Destination pole

Base case: if n=1

Move the ring from A to C using B as spare

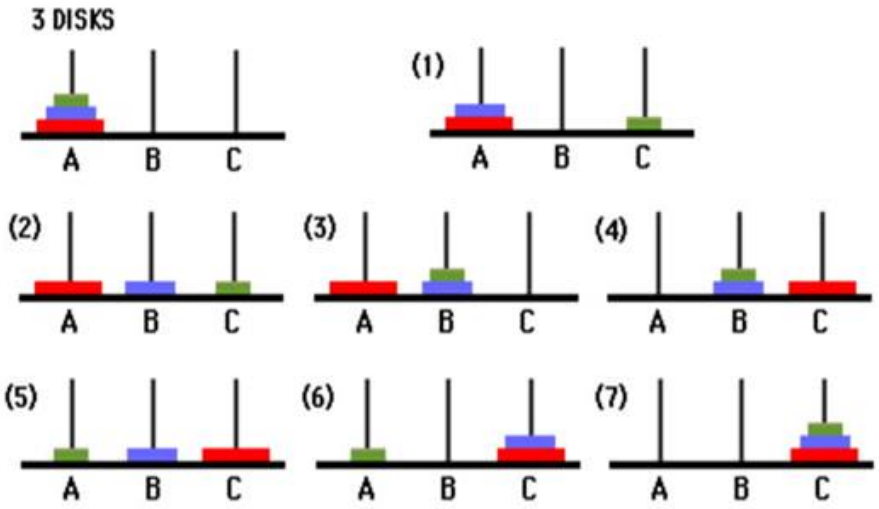
Recursive case:

Move n – 1 rings from A to B using C as spare

Move the one ring left on A to C using B as spare

Move n – 1 rings from B to C using A as spare

Step by Step Illustration



Code Snippet

- Function Call:

move(n,'A', 'C', 'B');

- Function Definition:

```
void move(int n, char source, char dest, char spare)
{
      if (n==1)
            printf("\n Move from %c to %c",source,dest);
      else
      {
            move(n-1,source,spare,dest);
            move(1,source,dest,spare);
            move(n-1,spare,dest,source);
      }
}
```

Session 7

Applications of Recursion: Towers of Hanoi

Towers of Hanoi Problem:

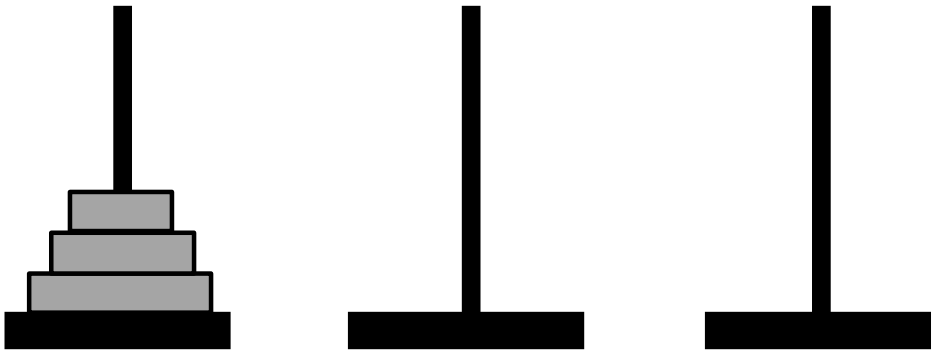
There are 3 pegs and n disks. All the n disks are stacked in 1 peg in the order of their sizes with the largest disk at the bottom and smallest on top. All the disks have to be transferred to another peg.

Rules for transfer:

- Only one disk can be moved at a time.
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
- No disk may be placed on top of a smaller disk.

Towers of Hanoi

- Initial Position:



- All disks from peg A have to be transferred to peg C

Towers of Hanoi Solution for 3 disks

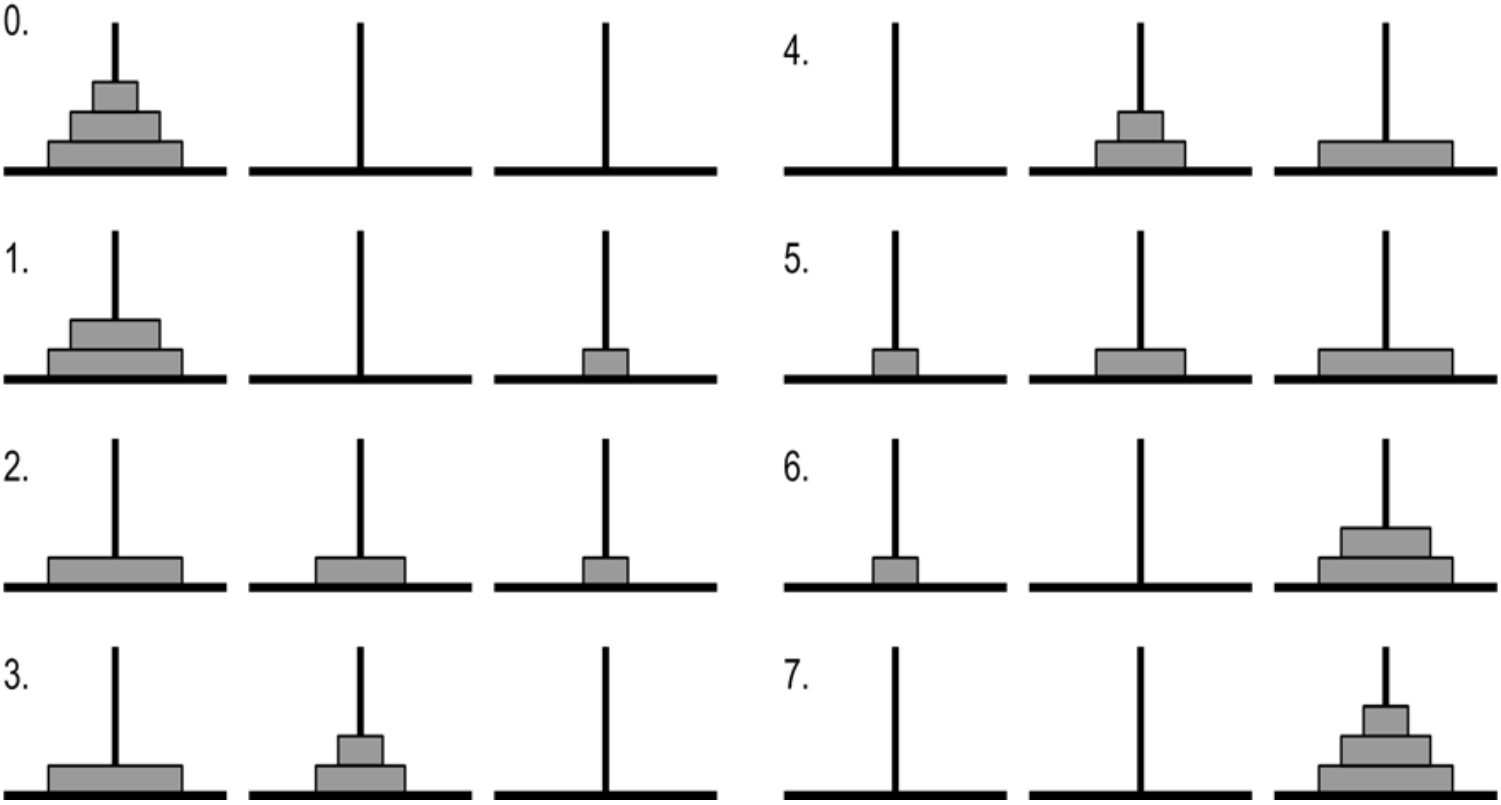


Image Source: https://www.includehelp.com/data-structure-tutorial/tower-of-hanoi-using-recursion.aspx

Towers of Hanoi – Recursive Solution


```
/*  N = Number of disks
    Beg, Aux, End are the pegs  */

Tower(N, Beg, Aux, End)
Begin
    if N = 1 then
        Print: Beg --> End;
    else
        Call Tower(N-1, Beg, End,
Aux);
        Print: Beg --> End;
        Call Tower(N-1, Aux, Beg,
End);
    endif
End
```

The Queue ADT

- With a queue, insertion is done at one end whereas deletion is performed at the other end.
- It is a First In First Out (FIFO) Structure. That is, the first element to enter into the queue will be the first element to get out of the queue.

Image Source: https://www.codesdope.com/course/data-structures-

QUEUE

- Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends.
- In a queue data structure, adding and removing elements are performed at two different positions.
- The insertion is performed at one end and deletion is performed at another end.
- In a queue data structure, the insertion operation is performed at a position which is known as '**rear**' and the deletion operation is performed at a position which is known as '**front**'.
- In queue data structure, the insertion and deletion operations are performed based on **FIFO (First In First Out)** principle.

QUEUE

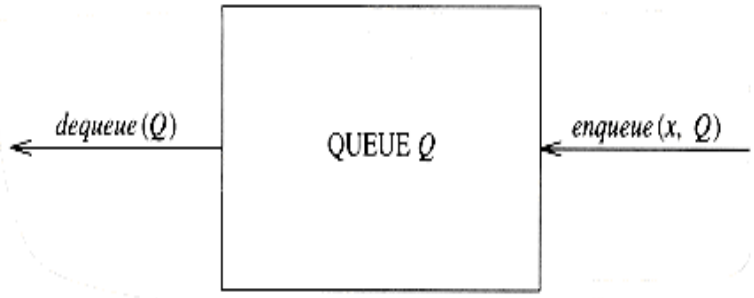
Queue

Let us explain the concept of queues using the analogies given below.

- People moving on an escalator. The people who got on the escalator first will be the first one to step out of it.
- People waiting for a bus. The first person standing in the line will be the first one to get into the bus.
- People standing outside the ticketing window of a cinema hall. The first person in the line will get the ticket first and thus will be the first one to move out of it.
- Luggage kept on conveyor belts. The bag which was placed first will be the first to come out at the other end.
- Cars lined at a toll bridge. The first car to reach the bridge will be the first to leave.

The basic operations on a Queue

- Enqueue: inserts an element at the end of the list (called the rear)
- Dequeue: deletes (and returns) the element at the start of the list (called the front).



- Peek: Get the front element

SESSION 8

Queue Representations

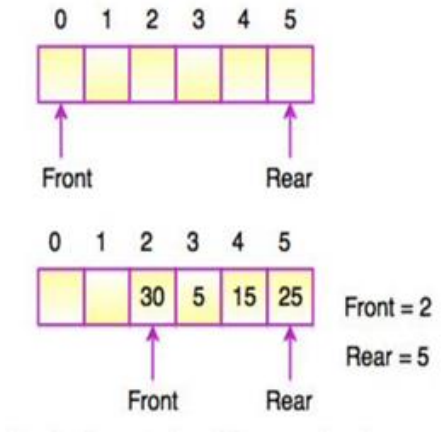
- Queues can be represented using:
  - Arrays
  - Linked Lists

ARRAY REPRESENTATION OF QUEUES

- Queues can be implemented by using either arrays or linked lists.

ARRAY REPRESENTATION OF QUEUES:

- **Array** is the easiest way to implement a queue. Queue can be also implemented using Linked List or Stack.
- Queues can be easily represented using linear arrays. As stated earlier, every queue has FRONT and REAR variables that point to the position from where deletions and insertions can be done, respectively.



Implementation of queue using an array

ARRAY REPRESENTATION OF QUEUES

Algorithm to insert an element in a queue

Step 1: IF REAR = MAX-1
Write OVERFLOW Goto step 4
[END OF IF]
Step 2: IF FRONT = -1 and REAR = -1
SET FRONT = REAR =
ELSE
SET REAR = REAR + 1
[END OF IF]
Step 3: SET QUEUE[REAR] = NUM
Step 4: EXIT

Algorithm to delete an element from a queue

Step 1: IF FRONT = -1 OR FRONT > REAR Write UNDERFLOW
ELSE
SET VAL = QUEUE[FRONT]
SET FRONT = FRONT + 1
[END OF IF]
Step 2: EXIT

Drawback of array implementation

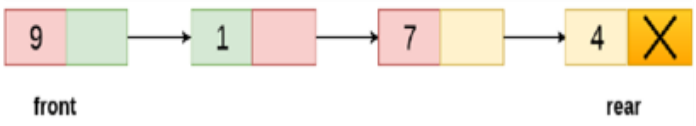
- Although, the technique of creating a queue is easy, but there are some drawbacks of using this technique to implement a queue.
- **Memory wastage:** The space of the array, which is used to store queue elements, can never be reused to store the elements of that queue because the elements can only be inserted at front end and the value of front might be so high so that, all the space before that, can never be filled.
- **Deciding the array size:** One of the most common problem with array implementation is the size of the array which requires to be declared in advance. Due to the fact that, the queue can be extended at runtime depending upon the problem, the extension in the array size is a time taking process and almost impossible to be performed at runtime since a lot of reallocations take place. Due to this reason, we can declare the array large enough so that we can store queue elements as enough as possible but the main problem with this declaration is that, most of the array slots (nearly half) can never be reused. It will again lead to memory wastage.

LINKED LIST REPRESENTATION OF QUEUES

- Due to the drawbacks discussed in the previous section, the array implementation can not be used for the large scale applications where the queues are implemented. One of the alternative of array implementation is linked list implementation of queue.
- The storage requirement of linked representation of a queue with n elements is o(n) while the time requirement for operations is o(1).
- In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.
- In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.
- Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.

LINKED LIST REPRESENTATION OF QUEUES

The linked representation of queue is shown in the following figure.



Linked Queue

Operations on Linked Queue

- There are two basic operations which can be implemented on the linked queues.
- The operations are Insertion and Deletion.

Operations on Linked Queue

Insertion operation

Algorithm

Step 1: Allocate the space for the new node PTR

Step2:SETPTR->DATA=VAL

Step 3: IF FRONT = NULL

SET FRONT = REAR = PTR

SET FRONT -> NEXT = REAR -> NEXT = NULL

ELSE

SET REAR -> NEXT = PTR

SET REAR = PTR

SET REAR -> NEXT = NULL

[END OF IF]

- Step 4: END

Deletion operation

Algorithm

Step 1: IF FRONT = NULL

Write " Underflow "

Go to Step 5

[END OF IF]

Step 2: SET PTR = FRONT

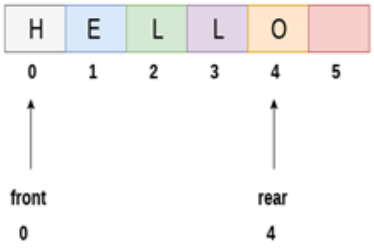
Step 3: SET FRONT = FRONT -> NEXT

Step 4: FREE PTR

Step 5: END

Array Implementation of Queues

- Queue can be implemented using a one-dimensional array.
- We need to keep track of 2 variables: *front* and *rear*
- *front* is the index in which the first element is present
- *rear* is the index in which the last element is present

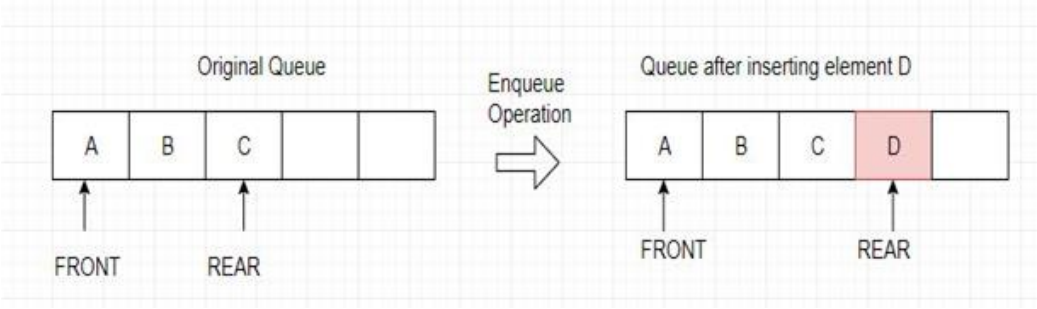


https://www.javatpoint.com/array-representation-of-queue

Enqueue in Array Implementation of Queue

```
//Basic Idea
enqueue(num):
        rear++
        queue[rear]=num
        return
```

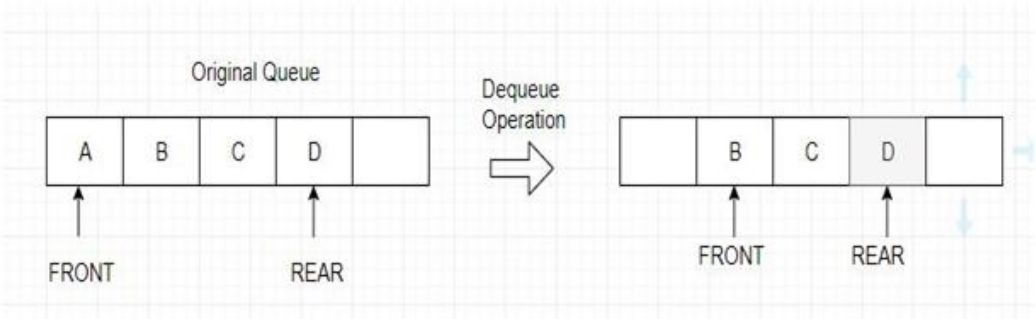
Time Complexity: O(1)



Dequeue in Array Implementation of Queue

```
//Basic Idea
dequeue():
        num = queue[front]
        front++
        return num
```

Time Complexity: O(1)
O(n) if elements are shifted



Enqueue and Dequeue in Queue

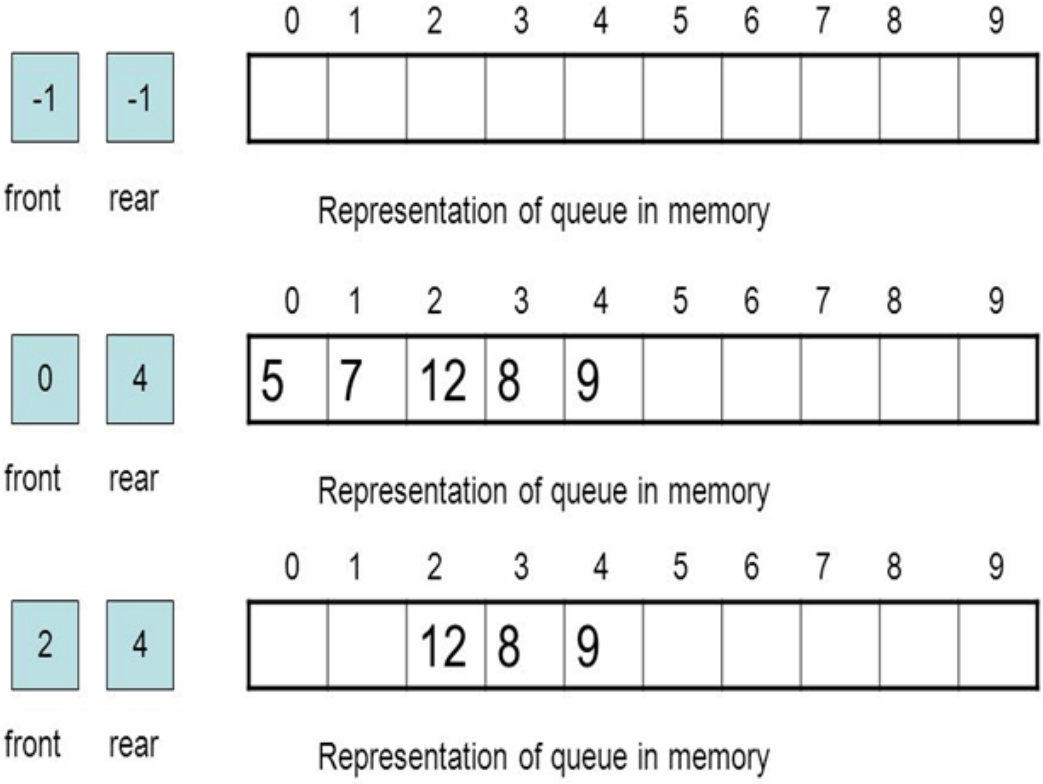
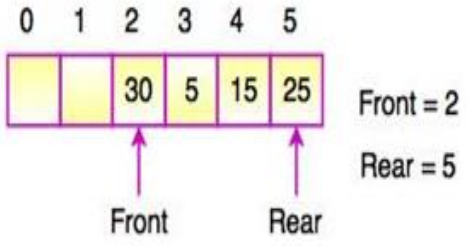


Image Source:
https://slideplayer.com/slide/4151401/

Disadvantage of Array Implementation of Queue

- Multiple enqueue and dequeue operations may lead to situation like this:

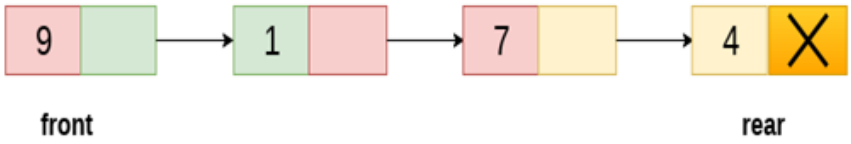


https://www.tutorialride.com/data-structures/queue-in-data-structure.htm

- No more insertion is possible (since rear cannot be incremented anymore) even though 2 spaces are free.
- Workaround: Circular Queue, Shifting elements after each dequeue

Linked List Implementation of Queue

- In the linked list implementation of queue, two pointers are used, namely, *front* and *rear*
- New nodes are created when an element has to be inserted.
- Each node consists of queue element and a pointer to the next node



```
void insert(struct node *ptr, int item; )
{


    ptr = (struct node *) malloc (sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW\n");
        return;
    }
    else
    {
        ptr -> data = item;
        if(front == NULL)
        {
            front = ptr;
            rear = ptr;
            front -> next = NULL;
            rear -> next = NULL;
        }
        else
        {
            rear -> next = ptr;
            rear = ptr;
            rear->next = NULL;
        }
    }
}
```

Dequeue Operation in Linked List Implementation of Queue

- Dequeue basic code:

```
ptr = front;
front = front -> next;
free(ptr);
```

Time Complexity: O(1)

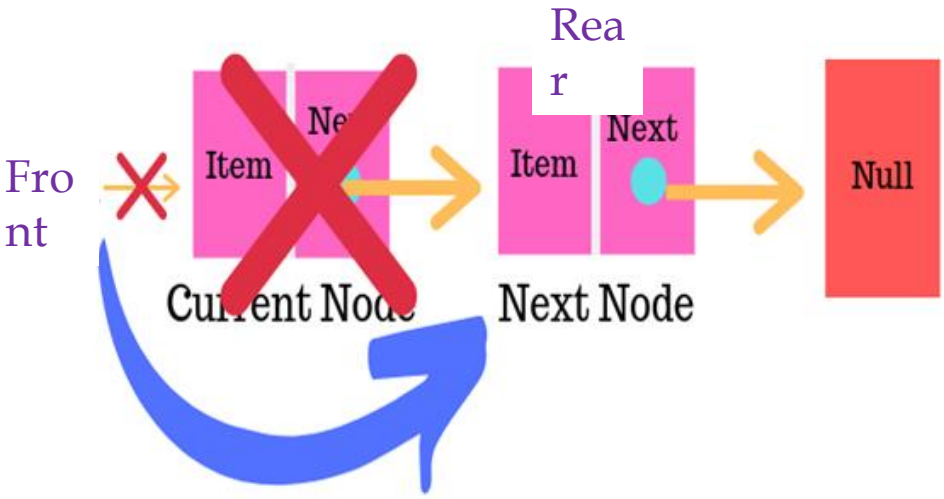


Image Source: https://learnersbucket.com/tutorials/data-structures/implement-queue-using-linked-list/

```
void delete (struct node *ptr)
{
   if(front == NULL)
   {
      printf("\nUNDERFLOW\n");
      return;
   }
   else
   {
      ptr = front;
      front = front -> next;
      free(ptr);
   }
}
```

References

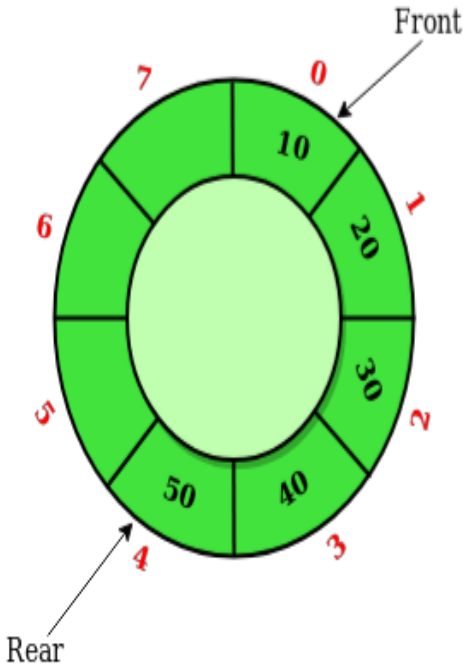
- Seymour Lipschutz, Data Structures, Schaum's Outlines, 2014.

SESSION 11

Circular Queue

CIRCULAR QUEUES

- Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called 'Ring Buffer'.
- In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.
- Deletions and insertions can only be performed at front and rear end respectively, as far as linear queue is concerned.



CIRCULAR QUEUES

Consider the queue shown in the following figure.

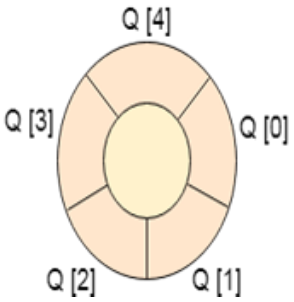
Queue

- The Queue shown in above figure is completely filled and there can't be inserted any more element due to the condition rear == max - 1 becomes true.
- One of the solution of this problem is circular queue. In the circular queue, the first index comes right after the last index. You can think of a circular queue as shown in the following figure.
- However, if we delete 2 elements at the front end of the queue, we still can not insert any element since the condition rear = max -1 still holds.
- This is the main problem with the linear queue, although we have space available in the array, but we can not insert any more element in the queue. This is simply the memory wastage and we need to overcome this problem.

Queue after deletion of first 2 elements

CIRCULAR QUEUES

- One of the solution of this problem is circular queue. In the circular queue, the first index comes right after the last index. You can think of a circular queue as shown in the following figure.



- Circular queue will be full when front = -1 and rear = max-1. Implementation of circular queue is similar to that of a linear queue. Only the logic part that is implemented in the case of insertion and deletion is different from that in a linear queue.

CIRCULAR QUEUE OPERATIONS

Insertion in Circular queue

- There are three scenario of inserting an element in a queue.
- If (rear + 1)%maxsize = front, the queue is full. In that case, overflow occurs and therefore, insertion can not be performed in the queue.
- If rear != max - 1, then rear will be incremented to the mod(maxsize) and the new value will be inserted at the rear end of the queue.
- If front != 0 and rear = max - 1, then it means that queue is not full therefore, set the value of rear to 0 and insert the new element there.

Algorithm to insert an element in circular queue

Step 1: IF (REAR+1)%MAX = FRONT
Write " OVERFLOW "
Goto step 4
[End OF IF]
Step2:IF FRONT=-1andREAR=-1
SET FRONT = REAR = 0
ELSE
IF REAR = MAX - 1 and FRONT ! = 0
SET REAR = 0
ELSE
SET REAR = (REAR + 1) % MAX
[END OF IF]
Step 3: SET QUEUE[REAR] = VAL o Step 4: EXIT

CIRCULAR QUEUE OPERATIONS

Deletion in Circular queue

- To delete an element from the circular queue, we must check for the three following conditions.
- If front = -1, then there are no elements in the queue and therefore this will be the case of an underflow condition.
- If there is only one element in the queue, in this case, the condition rear = front holds and therefore, both are set to -1 and the queue is deleted completely.
- If front = max -1 then, the value is deleted from the front end the value of front is set to 0.
- Otherwise, the value of front is incremented by 1 and then delete the element at the front end.

Algorithm to delete an element in circular queue

Step 1: IF FRONT = -1
Write " UNDERFLOW " Goto Step 4
[END of IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
SET FRONT = REAR = -1 ELSE
IF FRONT = MAX -1
SET FRONT = 0
ELSE
SET FRONT = FRONT + 1
[END of IF]
[END OF IF]
Step 4: EXIT

Operations Involved

- Enqueue
- Dequeue

Variables Used

- MAX- Number of entries in the array
- Front – is the index of front queue entry in an array (Get the front item from queue)
- Rear – is the index of rear queue entry in an array.(Get the last item from queue)

Concepts of Circular Queue

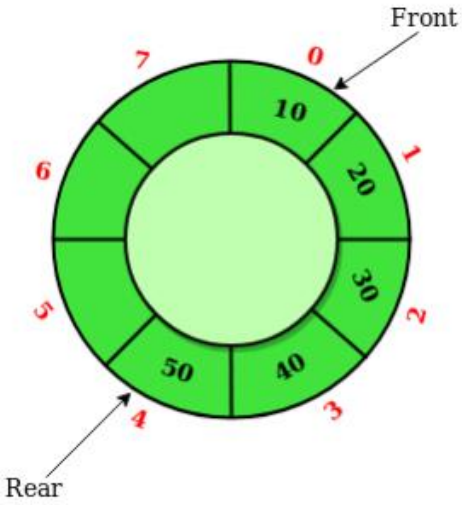
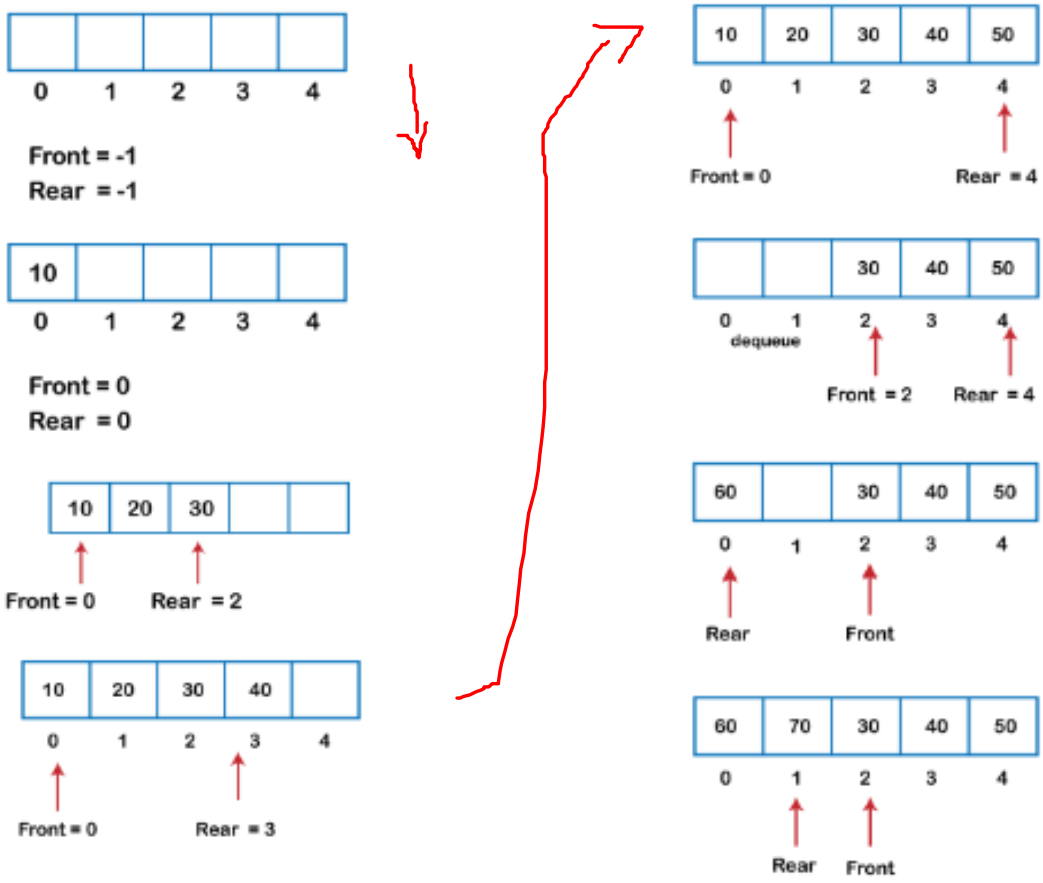


Illustration of Enqueue and Dequeue



```
#include <stdio.h>
# define max 6
int queue[max];  // array declaration
int front=-1;
int rear=-1;
// function to insert an element in a circular queue
void enqueue(int element)
{
   if(front==-1 && rear==-1)   // condition to check queue is empty
   {
      front=0;
      rear=0;
      queue[rear]=element;
   }
   else if((rear+1)%max==front)  // condition to check queue is full
   {
      printf("Queue is overflow..");
   }
   else
   {
      rear=(rear+1)%max;      // rear is incremented
      queue[rear]=element;    // assigning a value to the queue at the rear position.
   }
}
```

```
int dequeue()
{
   if((front==-1) && (rear==-1))  // condition to check queue is empty
   {
      printf("\nQueue is underflow..");
   }
  else if(front==rear)
{
   printf("\nThe dequeued element is %d", queue[front]);
   front=-1;
   rear=-1;
}
else
{
    printf("\nThe dequeued element is %d", queue[front]);
   front=(front+1)%max;
}
}
```

Applications of Queue

- Queue, as the name suggests is used whenever we need to manage 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:
- Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
- In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.
- Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served.

Review Questions

1. Explain the concept of a circular queue? How is it better than a linear queue?
2. Draw the queue structure in each case when the following operations are performed on an empty queue.
(a)Add A, B, C, D, E, F. (b) Delete two letters
(c) Add G (d) Add H
(e) Delete four letters (f) Add I
3. The circular queue will be full only when _____.
(a) FRONT=MAX-1 and REAR=MAX-1 (b) FRONT=0 and REAR=MAX-1
(c) FRONT=MAX-1 and REAR=0. (d) FRONT=0 and REAR=0
4. The function that deletes values from a queue is called _____.
(a) Enqueue (b)dequeue
(c) Pop (d) peek
5. New nodes are added at_____of the queue.
6. _____ allows insertion of elements at either ends but not in the middle.
7. A queue is a _____ data structure in which the element that is inserted first is the first one to be taken out.
8. In a circular queue, the first index comes after the _____ index.
9. In the computer's memory, queues can be implemented using _____ and _____.
10. The storage requirement of linked representation of queue with n elements is ____ and the typical time requirement for operations is _____.

Difference between Queue and Circular Queue

- In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.

- The unused memory locations in the case of ordinary queues can be utilized in circular queues.

SESSION 12

Circular Queue Example-ATM

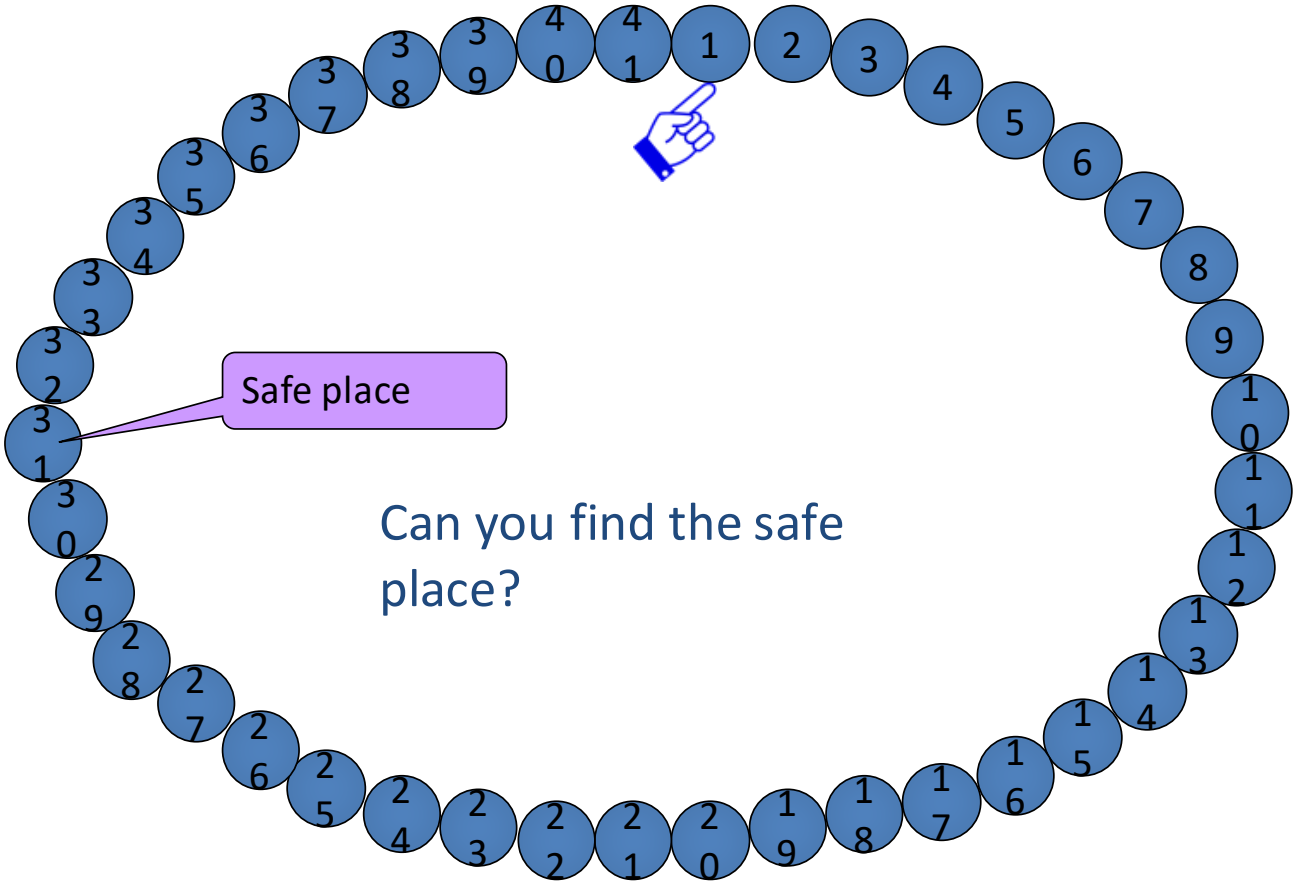
ATM is the best example for the circular queue. It does the following using circular queue

1. ATM sends request over private network to central server
2. Each request takes some amount of time to process.
3. More request may arrive while one is being processed.
4. Server stores requests in queue if they arrive while it is busy.
5. Queue processing time is negligible compared to request processing time.

Josephus problem

Flavius Josephus is a Jewish historian living in the 1st century. According to his account, he and his 40 comrade soldiers were trapped in a cave, surrounded by Romans. They chose suicide over capture and decided that they would form a circle and start killing themselves using a step of three. As Josephus did not want to die, he was able to find the safe place, and stayed alive with his comrade, later joining the Romans who captured them.

Can you find the safe place?



The first algorithm: Simulation

- We can find $f(n)$ using simulation.
  - Simulation is a process to imitate the real objects, states of affairs, or process.
  - We do not need to "kill" anyone to find $f(n)$.
- The simulation needs

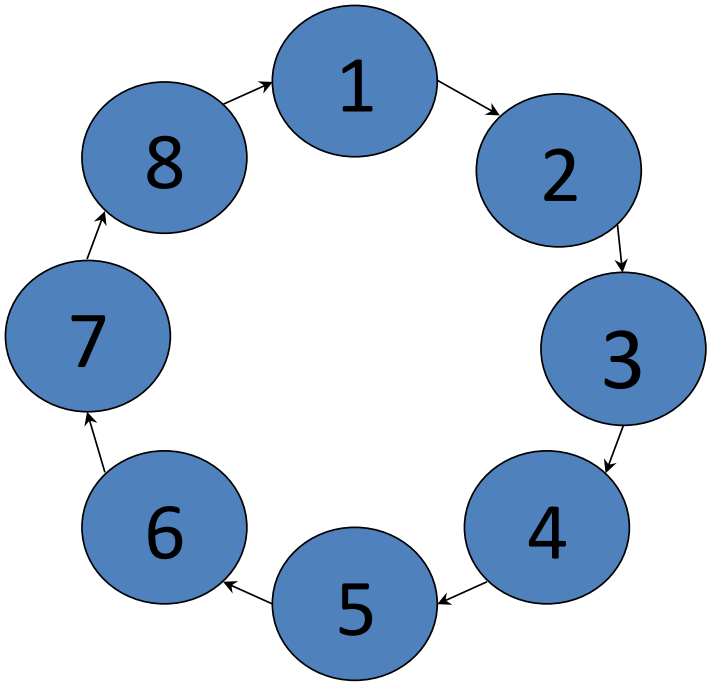
  (1) a **model** to represents "n people in a circle"

  (2) a way to **simulate** "kill every 2nd person"

  (3) knowing when to stop

Model *n* people in a circle

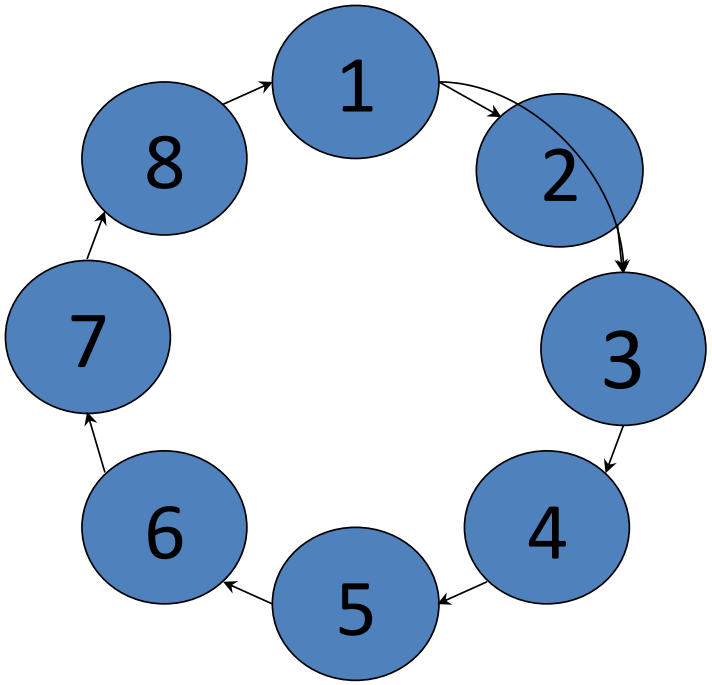
- We can use "data structure" to model it.
- This is called a "circular linked list".
  - Each node is of some "**struct**" data type
  - Each link is a "**pointer**"




```
struct node {
    int ID;
    struct node *next;
}
```

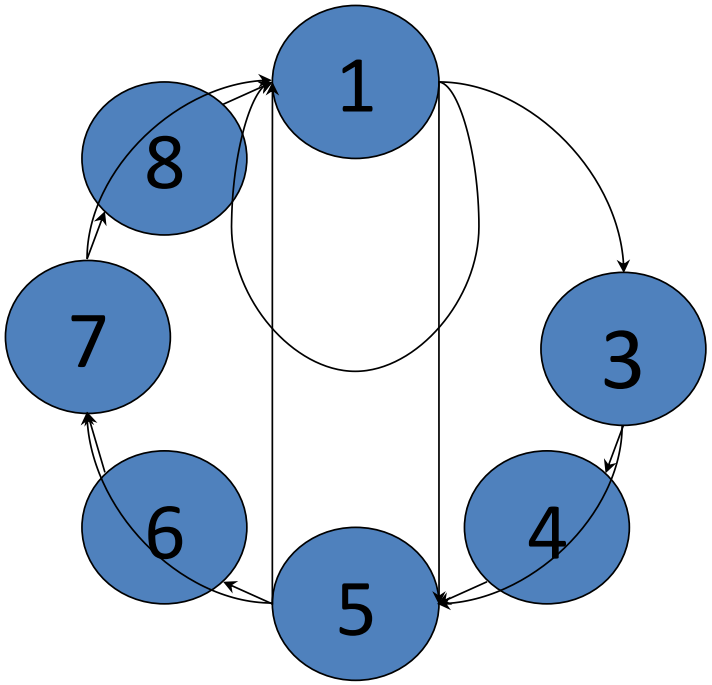
Kill every 2nd person

- Remove every 2nd node in the circular liked list.
  - You need to maintain the circular linked structure after removing node 2
  - The process can continue until …



Knowing when to stop

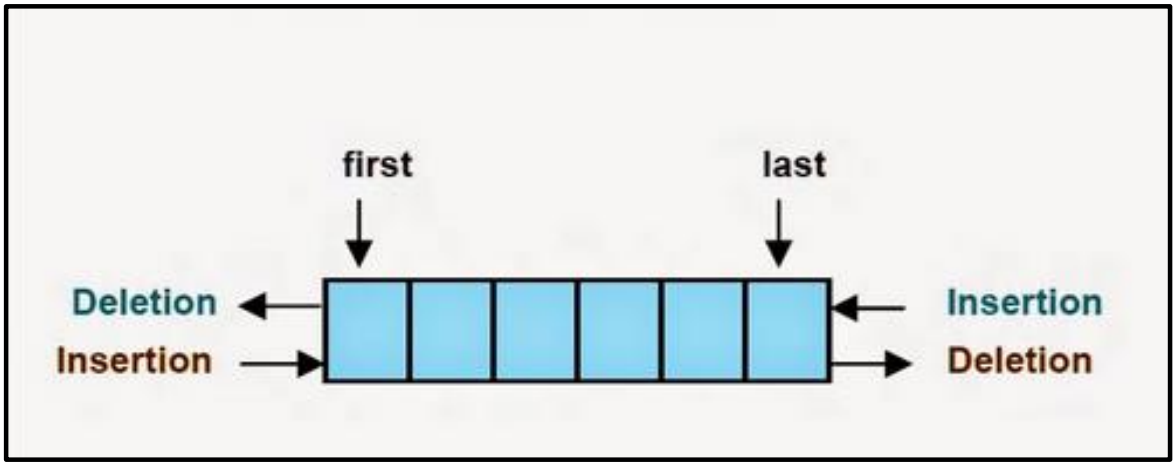
- Stop when there is only one node left
  - How to know that?
  - When the *next is pointing to itself
  - It's ID is f(n)
    - f(8) = 1



Double Ended Queue

- Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (**front** and **rear**).

Double Ended Queue

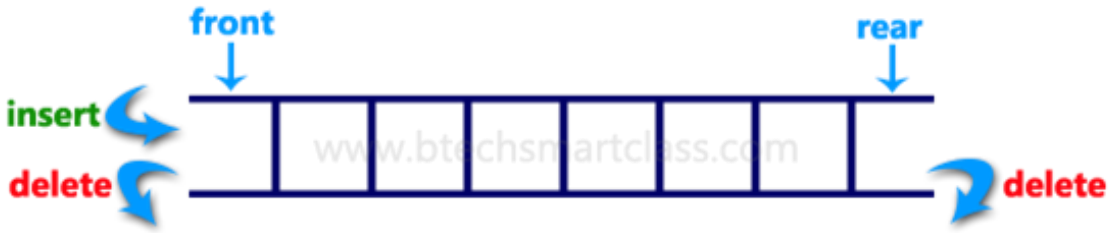


Double Ended Queue can be represented inTWO ways

- Input Restricted Double Ended Queue
- Output Restricted Double Ended Queue

Input Restricted Double Ended Queue

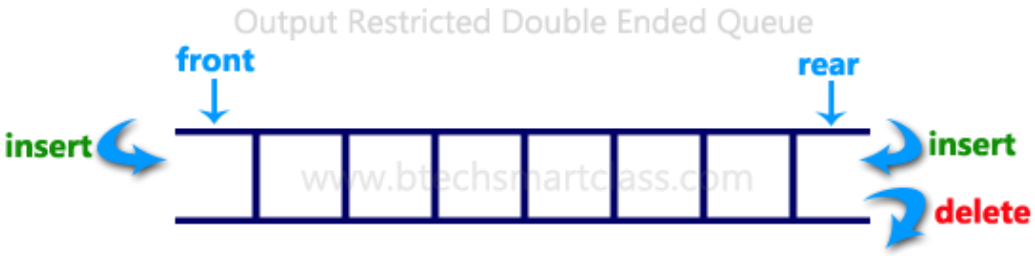
In input restricted double ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the ends.



Input Restricted Double Ended Queue

Output Restricted Double Ended Queue

In output restricted double ended queue, the deletion operation is performed at only one end and insertion operation is performed at both the ends



SESSION 13

Priority Queue

- Priority Queue is an extension of a queue with following properties.
- Every item has a priority associated with it.
- An element with high priority is dequeued before an element with low priority.
- If two elements have the same priority, they are served according to their order in the queue.

Priority Queue Example

Example:

X	Y	A	M	B	C	N	O	P	Z
1	1	2	3	2	2	3	3	3	1

Operations Involved

- **insert(item, priority):** Inserts an item with given priority.

- **getHighestPriority():** Returns the highest priority item.

- ***pull_highest_priority_element:*** remove the element from the queue that has the *highest priority*, and return it.

Implementation of priority queue

- Circular array
- Multi-queue implementation
- Double Link List
- Heap Tree

Node of linked list in priority queue

It comprises of three parts

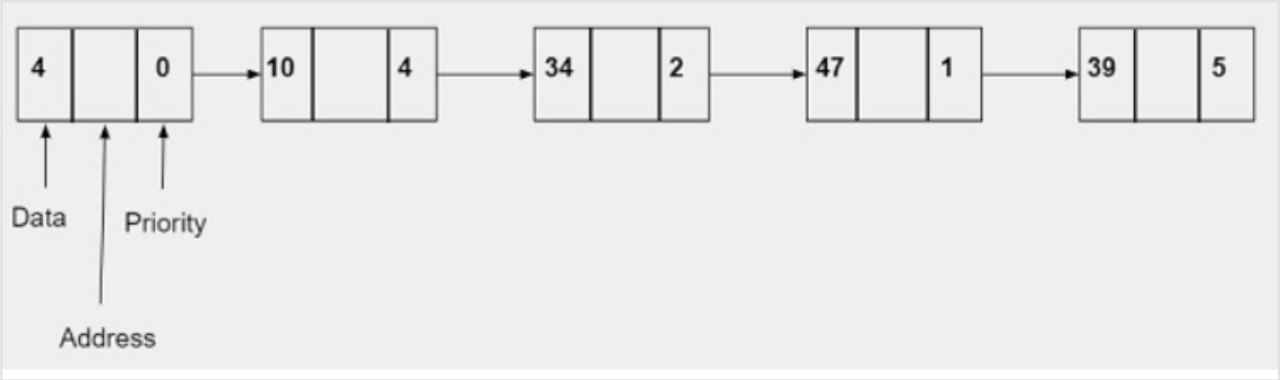
- **Data** − It will store the integer value.

- **Address** − It will store the address of a next node

- **Priority** −It will store the priority which is an integer value. It can range from 0-10 where 0 represents the highest priority and 10 represents the lowest priority.

Example

Input



Output