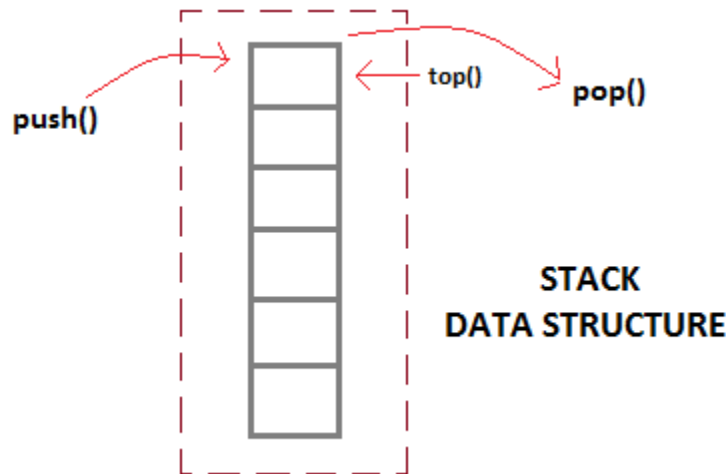


UNIT III- STACK & QUEUE

WHAT IS STACK DATA STRUCTURE?

Stack is an abstract data type with a bounded(predefined) capacity. It is a simple data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the **top** of the stack and the only element that can be removed is the element that is at the top of the stack, just like a pile of objects.



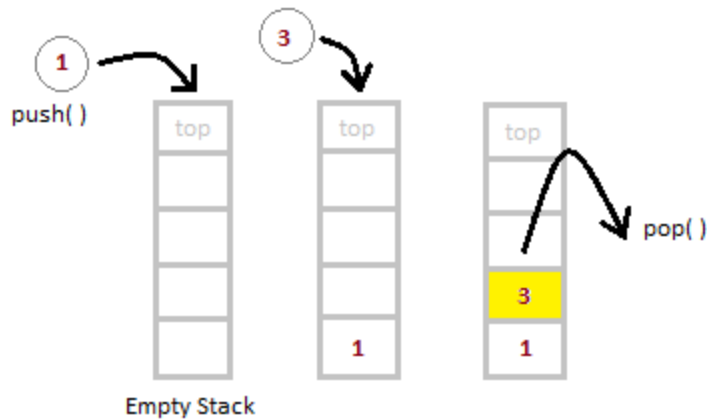
Basic features of Stack

1. Stack is an **ordered list** of **similar data type**.
2. Stack is a **LIFO**(Last in First out) structure or we can say **FILO**(First in Last out).
3. **push()** function is used to insert new elements into the Stack and **pop()** function is used to remove an element from the stack. Both insertion and removal are allowed at only one end of Stack called **Top**.
4. Stack is said to be in **Overflow** state when it is completely full and is said to be in **Underflow** state if it is completely empty.

IMPLEMENTATION OF STACK DATA STRUCTURE

Stack can be easily implemented using an Array or a Linked List. Arrays are quick, but are limited in size and Linked List requires overhead to allocate, link, unlink, and deallocate, but is not limited in size. Here we will implement Stack using array.

STACK - LIFO Structure



In a Stack, all operations take place at the "top" of the stack. The "push" operation adds an item to the top of the Stack.
The "pop" operation removes the item on top of the stack.

Implementation of Stack using Array

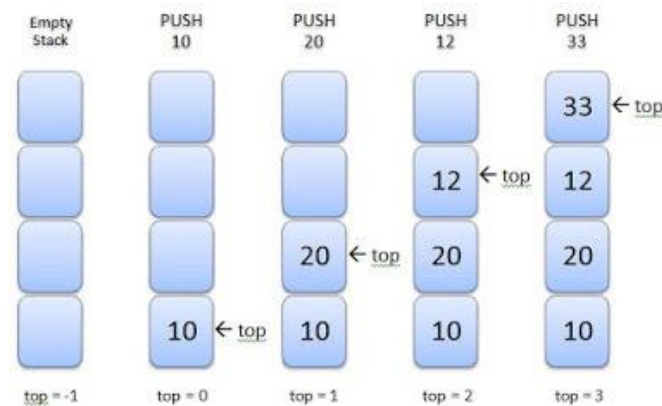
- A stack data structure can be implemented using a one-dimensional array
- Array stores only a fixed number of data values
- Implementation is very simple
- Define a one dimensional array of specific size
 - insert or delete the values into the array by using **LIFO principle** with the help of a variable called '**top**'
 - Initially, the top is set to -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

Stack Array Creation

- Include all the **header files** which are used in the program and define a constant '**SIZE**' with specific value.
- Declare all the **functions** used in stack implementation.
- Create a one dimensional array with fixed size (**int stack[SIZE]**)
- Define a integer variable '**top**' and initialize with '**-1**'. (**int top = -1**)
- In main method, display menu with list of operations and make suitable function calls to perform operation selected by the user on the stack.

Push Algorithm

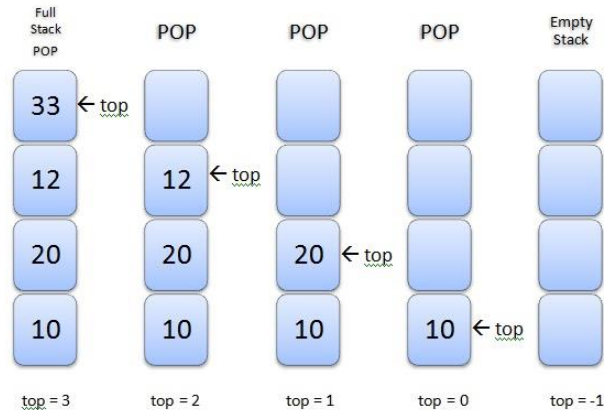
- `push(val)` is a function used to insert a new element into stack at **top** position.
- Push function takes one integer value as parameter
 1. Check whether **stack** is **FULL**. (**top == SIZE-1**)
 2. If it is **FULL**, then display "**Stack is FULL!!! Insertion is not possible!!!**" and terminate the function.
 3. If it is **NOT FULL**, then increment **top** value by one (**top++**) and set `stack[top]` to value (**`stack[top] = value`**)



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`.

Pop Algorithm

- `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**. (**top == -1**)
 2. If it is **EMPTY**, then display "**Stack is EMPTY!!! Deletion is not possible!!!**" and terminate the function
 3. If it is **NOT EMPTY**, then delete `stack[top]` and decrement **top** value by one (**`top--`**)



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

Display Alorithm

- **display() - Displays the elements of a Stack**
 1. Check whether **stack** is **EMPTY**. (**top == -1**)
 2. If it is **EMPTY**, then display "**Stack is EMPTY!!!**" and terminate the function
 3. If it is **NOT EMPTY**, then define a variable '**i**' and initialize with **top**.
Display **stack[i]** value and decrement **i** value by one (**i--**)
 4. Repeat above step until **i** value becomes '0'.

Disadvantages of Array

- The amount of data must be specified at the beginning of the implementation
- Stack implemented using an array is not suitable, when the size of data is unknown

C Program CODE

```
#include<stdio.h>
int stack[100],choice,n,top,x,i;
void push(void);
void pop(void);
void display(void);
int main()
{
    top=-1;
    printf("\n Enter the size of STACK[MAX=100]:");
    scanf("%d",&n);
    printf("\n\t STACK OPERATIONS USING ARRAY");
    printf("\n\t-----");
    printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");
    do
```

```

{
    printf("\n Enter the Choice:");
    scanf("%d",&choice);
    switch(choice)
    {
        case 1:
        {
            push();
            break;
        }
        case 2:
        {
            pop();
            break;
        }
        case 3:
        {
            display();
            break;
        }
        case 4:
        {
            printf("\n\t EXIT POINT ");
            break;
        }
        default:
        {
            printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");
        }
    }
}
while(choice!=4);
return 0;
}
void push()
{
    if(top>=n-1)
    {
        printf("\n\tSTACK is over flow");
    }
    else
    {
        printf(" Enter a value to be pushed:");
        scanf("%d",&x);
    }
}

```

```

        top++;
        stack[top]=x;
    }
}
void pop()
{
    if(top<=-1)
    {
        printf("\n\t Stack is under flow");
    }
    else
    {
        printf("\n\t The popped elements is %d",stack[top]);
        top--;
    }
}
void display()
{
    if(top>=0)
    {
        printf("\n The elements in STACK \n");
        for(i=top; i>=0; i--)
            printf("\n%d",stack[i]);
        printf("\n Press Next Choice");
    }
    else
    {
        printf("\n The STACK is empty");
    }
}
}

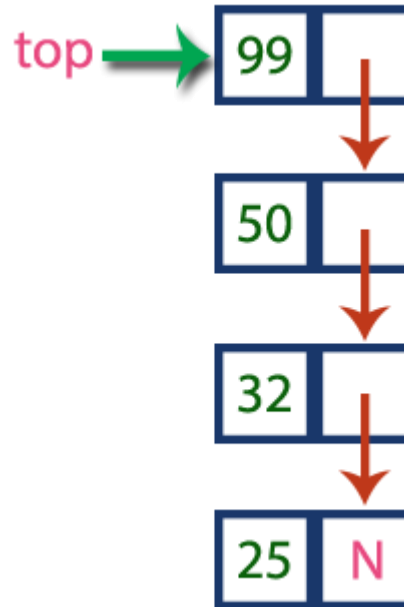
```

Implementation of Stack using Linked List

A stack data structure can be implemented by using a linked list data structure. The stack implemented using linked list can work for an unlimited number of values. That means, stack implemented using linked list works for the variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

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**'. Whenever we want 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



In the above example, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32, 50 and 99.

Stack Operations using Linked List

To implement a stack using a linked list, we need to set the following things before implementing actual operations.

- **Step 1** - Include all the **header files** which are used in the program. And declare all the **user defined functions**.
- **Step 2** - Define a '**Node**' structure with two members **data** and **next**.
- **Step 3** - Define a **Node** pointer '**top**' and set it to **NULL**.
- **Step 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

We can use the following steps to insert a new node into the stack...

- **Step 1** - Create a **newNode** with given value.
- **Step 2** - Check whether stack is **Empty** (**top == NULL**)
- **Step 3** - If it is **Empty**, then set **newNode** → **next = NULL**.
- **Step 4** - If it is **Not Empty**, then set **newNode** → **next = top**.
- **Step 5** - Finally, set **top = newNode**.

pop() - Deleting an Element from a Stack

We can use the following steps to delete a node from the stack...

- **Step 1** - Check whether **stack** is **Empty** (**top == NULL**).
- **Step 2** - If it is **Empty**, then display "**Stack is Empty!!! Deletion is not possible!!!**" and terminate the function
- **Step 3** - If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.
- **Step 4** - Then set '**top = top → next**'.
- **Step 5** - Finally, delete '**temp**'. (**free(temp)**).

display() - Displaying stack of elements

We can use the following steps to display the elements (nodes) of a stack...

- **Step 1** - Check whether stack is **Empty** (**top == NULL**).
- **Step 2** - If it is **Empty**, then display '**Stack is Empty!!!**' and terminate the function.
- **Step 3** - If it is **Not Empty**, then define a Node pointer '**temp**' and initialize with **top**.
- **Step 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**).
- **Step 5** - Finally! Display '**temp → data ---> NULL**'.

C Program Code

```
#include<stdio.h>
```

```
#include<conio.h>
```

```
struct Node
```

```
{
    int data;
    struct Node *next;
}*top = NULL;
```

```
void push(int);
```

```
void pop();
```

```
void display();
```

```
void main()
```

```
{
    int choice, value;
    clrscr();
```



```

printf("\n:: Stack using Linked List ::\n");
while(1){
    printf("\n***** MENU *****\n");
    printf("1. Push\n2. Pop\n3. Display\n4. Exit\n");
    printf("Enter your choice: ");
    scanf("%d",&choice);
    switch(choice){
        case 1: printf("Enter the value to be insert: ");
                scanf("%d", &value);
                push(value);
                break;
        case 2: pop(); break;
        case 3: display(); break;
        case 4: exit(0);
        default: printf("\nWrong selection!!! Please try again!!!\n");
    }
}
}

void push(int value)
{
    struct Node *newNode;
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    if(top == NULL)
        newNode->next = NULL;
    else
        newNode->next = top;
    top = newNode;
    printf("\nInsertion is Success!!!\n");
}

```

```

void pop()
{
    if(top == NULL)
        printf("\nStack is Empty!!!\n");
    else{
        struct Node *temp = top;
        printf("\nDeleted element: %d", temp->data);
        top = temp->next;
        free(temp);
    }
}

void display()
{
    if(top == NULL)
        printf("\nStack is Empty!!!\n");
    else{
        struct Node *temp = top;
        while(temp->next != NULL){
            printf("%d--->",temp->data);
            temp = temp -> next;
        }
        printf("%d--->NULL",temp->data);
    }
}

```

Array vs Linked List

ARRAY	LINKED LIST

Array is a collection of elements of similar data type.	Linked List is an ordered collection of elements of same type, which are connected to each other using pointers.
<p>Array supports Random Access, which means elements can be accessed directly using their index, like <code>arr[0]</code> for 1st element, <code>arr[6]</code> for 7th element etc.</p> <p>Hence, accessing elements in an array is fast with a constant time complexity of $O(1)$.</p>	<p>Linked List supports Sequential Access, which means to access any element/node in a linked list, we have to sequentially traverse the complete linked list, upto that element.</p> <p>To access nth element of a linked list, time complexity is $O(n)$.</p>
In an array, elements are stored in contiguous memory location or consecutive manner in the memory.	<p>In a linked list, new elements can be stored anywhere in the memory.</p> <p>Address of the memory location allocated to the new element is stored in the previous node of linked list, hence forming a link between the two nodes/elements.</p>
In array, Insertion and Deletion operation takes more time, as the memory locations are consecutive and fixed.	<p>In case of linked list, a new element is stored at the first free and available memory location, with only a single overhead step of storing the address of memory location in the previous node of linked list.</p> <p>Insertion and Deletion operations are fast in linked list.</p>
Memory is allocated as soon as the array is declared, at compile time . It's also known as Static Memory Allocation .	Memory is allocated at runtime , as and when a new node is added. It's also known as Dynamic Memory Allocation .
In array, each element is independent and can be accessed using its index value.	In case of a linked list, each node/element points to the next, previous, or maybe both nodes.
Array can be single dimensional , two dimensional or multidimensional	Linked list can be Linear(Singly) , Doubly or Circular linked list.
Size of the array must be specified at time of array declaration.	Size of a Linked list is variable. It grows at runtime, as more nodes are added to it.

Array gets memory allocated in the **Stack** section.

Whereas, linked list gets memory allocated in **Heap** section.

APPLICATIONS OF STACK

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

INFIX TO POSTFIX CONVERSION

- 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)

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

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.
- 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.
- 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.

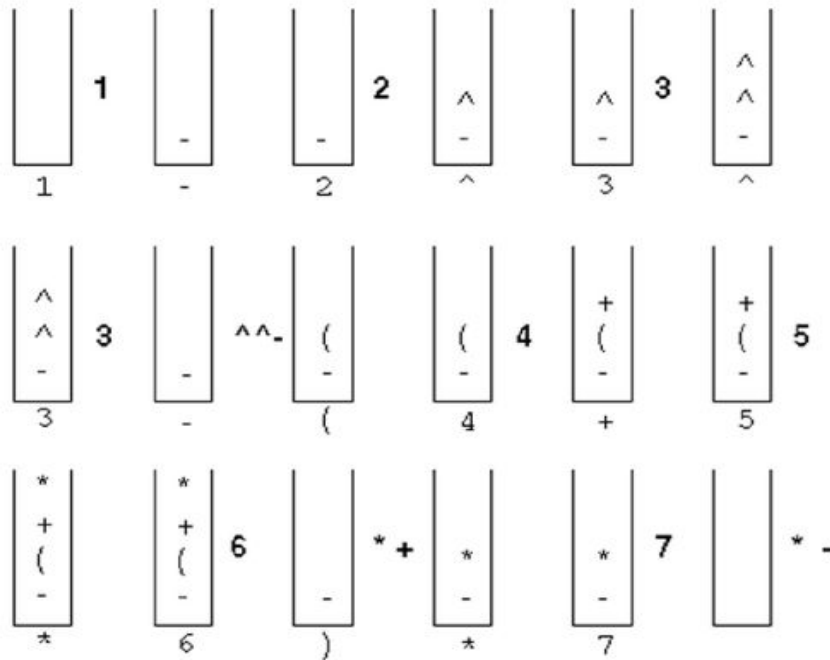
Step 5 : If the scanned character is an ')', pop the stack and output it until a '(' is encountered, and discard both the parenthesis.

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.

Infix: 1 - 2 ^ 3 ^ 3 - (4 + 5 * 6) * 7












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

Postfix Expression Evaluation

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.

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		

BALANCING SYMBOLS

- Stacks can be used to check if the given expression has balanced symbols or not.
- The algorithm is very much useful in compilers.

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

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

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

Thus, parenthesis match here

Applications of Stack: Function Call and Return

2 types of Memory

Stack Memory

Stack Memory is a special region of the computer's memory that stores temporary variables created by each function (including the `main()` function). The stack is a "LIFO" (last in, first out) data structure, that is managed and optimized by the CPU quite closely. Every time a function declares a new variable, it is "pushed" onto the stack. Then every time a function exits, **all** of the variables pushed onto the stack by that function, are popped (that is to say, they are deleted). Once a stack variable is freed, that region of memory becomes available for other stack variables.

A key to understanding the stack is the notion that **when a function exits**, all of its variables are popped off of the stack (and hence lost forever). Thus stack variables are **local** in nature. This is related to a concept we saw earlier known as **variable scope**, or local vs global variables.

Heap Memory

The heap is a region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU. It is a more free-floating region of memory (and is larger). To allocate memory on the heap, you must use `malloc()` or `calloc()`, which are built-in C functions. Once you have allocated memory on the heap, you are responsible for using `free()` to deallocate that memory once you don't need it any more. If you fail to do this, your program will have what is known as a **memory leak**. That is, memory on the heap will still be set aside (and won't be available to other processes). As we will see in the debugging section, there is a tool called `valgrind` that can help you detect memory leaks.

Unlike the stack, variables created on the heap are accessible by any function, anywhere in your program. Heap variables are essentially global in scope.

Nested Function Calls

Consider the code snippet below:

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

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

bar()
{
    ...
}
```

The growth and shrinking of stack as the program executes is given below:

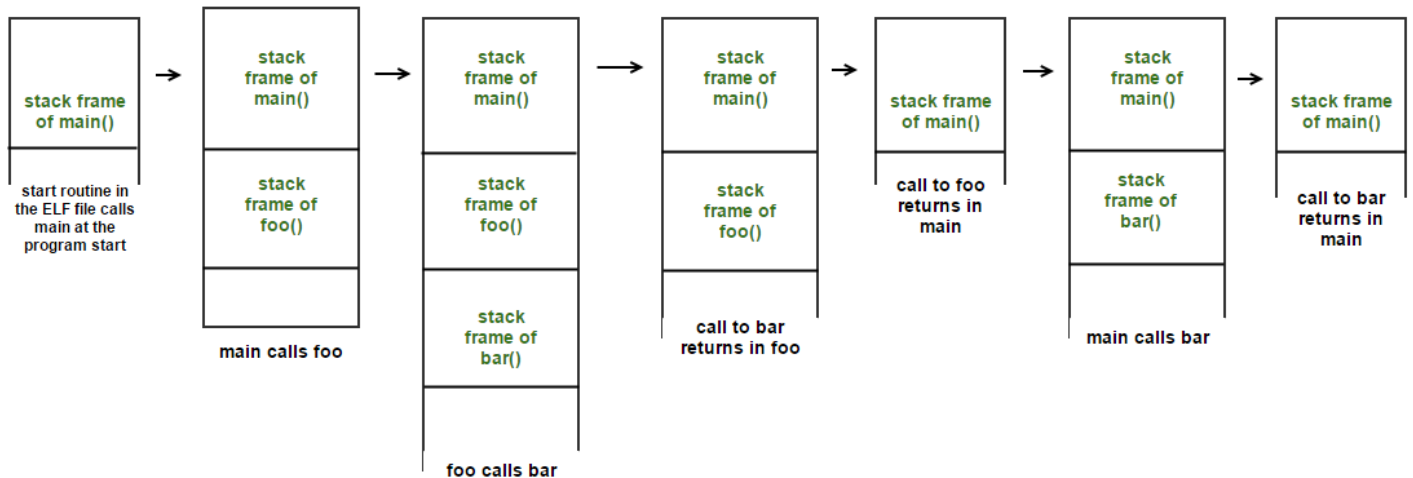


Image Source: <https://loonytek.com/2015/04/28/call-stack-internals-part-1/>

- 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 a function that calls itself during its execution.
- Each call to a recursive function pushes a new activation record (or 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.

Handling of Recursion by Stack Memory

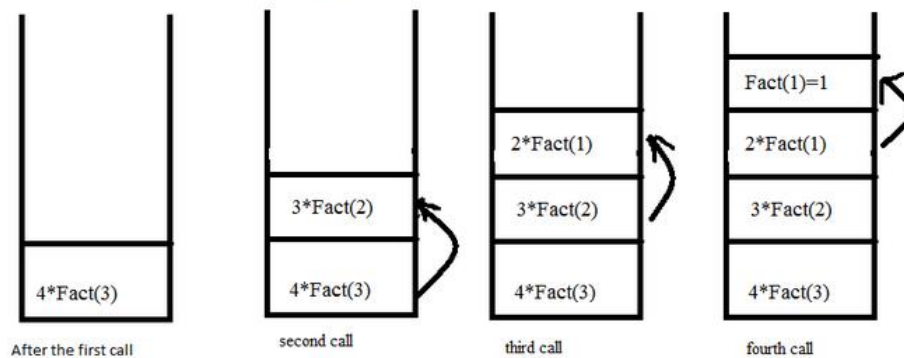
When any function is called from `main()`, the memory is allocated to it on the stack. A recursive function calls itself, the memory for a called function is allocated on top of memory allocated to calling function and different copy of local variables is created for each function call. When the base case is reached, the function returns its value to the function by whom it is called and memory is de-allocated and the process continues.

Example:

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

fact(4);
```

When function call happens previous variables gets stored in stack



Returning values from base case to caller function

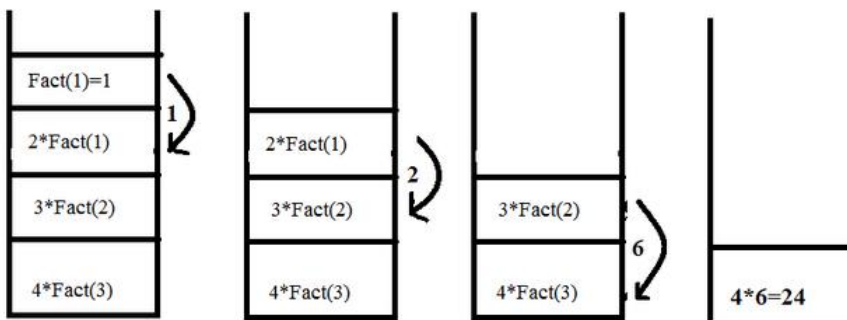


Image Source: <https://stackoverflow.com/questions/19865503/can-recursion-be-named-as-a-simple-function-call>

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.

Basic Solution to the problem:

Step 1 – Move $n-1$ disks from **source** to **aux**

Step 2 – Move n^{th} disk from **source** to **dest**

Step 3 – Move $n-1$ disks from **aux** to **dest**

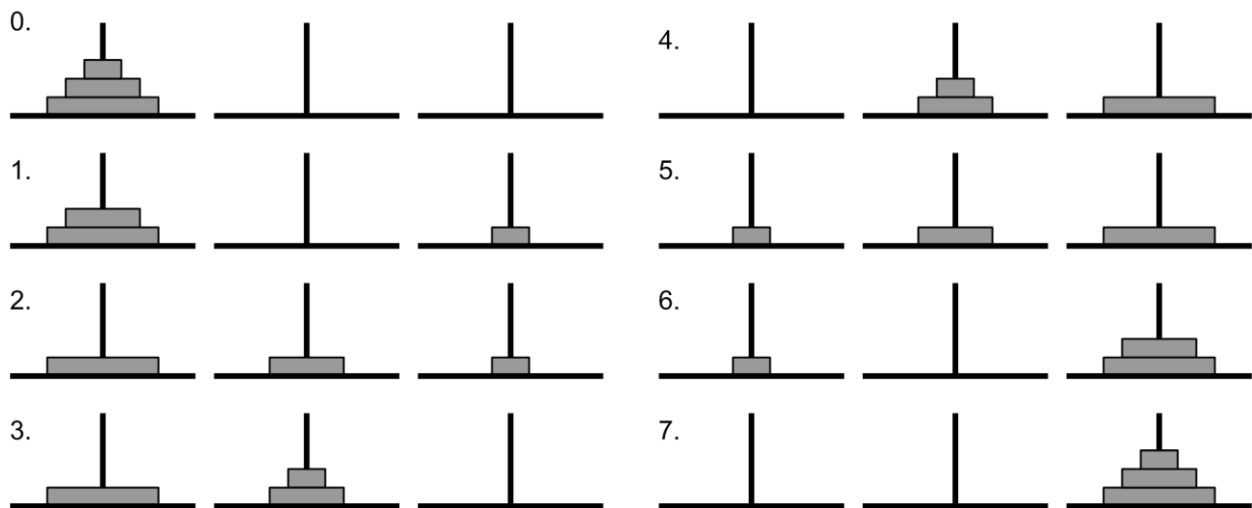


Image Source: <https://www.includehelp.com/data-structure-tutorial/tower-of-hanoi-using-recursion.aspx>

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

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.



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

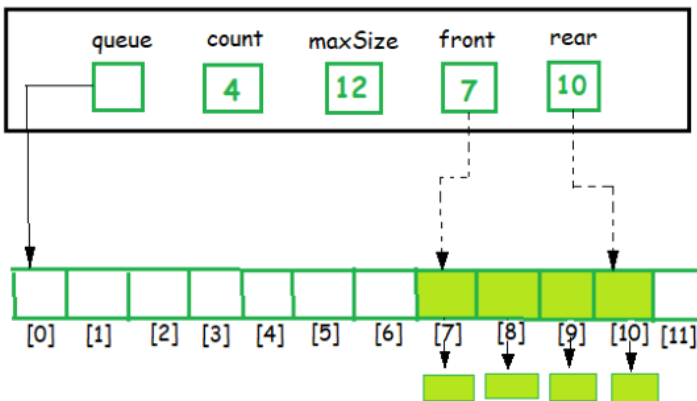
In a queue data structure, the insertion operation is performed using a function called "**enqueue()**" and deletion operation is performed using a function called "**dequeue()**".

Common Queue Operations

- **enqueue()** – Insert an element at the end of the queue.
- **dequeue()** – Remove and return the first element of the queue, if the queue is not empty.
- **peek()** – Return the element of the queue without removing it, if the queue is not empty.
- **size()** – Return the number of elements in the queue.
- **isEmpty()** – Return true if the queue is empty, otherwise return false.
- **isFull()** – Return true if the queue is full, otherwise return false.

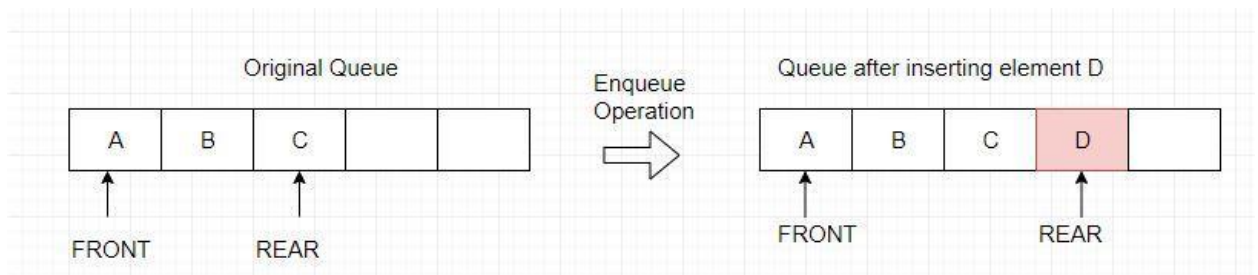
Array Implementation of Queue

In queue, insertion and deletion happen at the opposite ends.

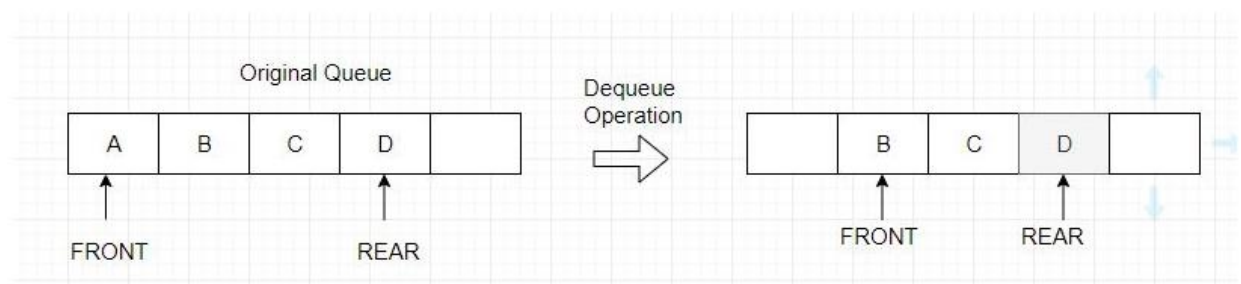


To implement a queue using array, create an array *arr* of size *n* and take two variables *front* and *rear* both of which will be initialized to 0 which means the queue is currently empty. Element *rear* is the index upto which the elements are stored in the array and *front* is the index of the first element of the array. Now, some of the implementation of queue operations are as follows:

1. **Enqueue:** Addition of an element to the queue. Adding an element will be performed after checking whether the queue is full or not. If $rear < n$ which indicates that the array is not full then store the element at $arr[rear]$ and increment *rear* by 1 but if $rear == n$ then it is said to be an Overflow condition as the array is full.



2. **Dequeue:** Removal of an element from the queue. An element can only be deleted when there is at least an element to delete i.e. $rear > 0$. Now, element at $arr[front]$ can be deleted but all the remaining elements have to shifted to the left by one position in order for the dequeue operation to delete the second element from the left on another dequeue operation.



3. **Front:** Get the front element from the queue i.e. $arr[front]$ if queue is not empty.
4. **Display:** Print all element of the queue. If the queue is non-empty, traverse and print all the elements from index $front$ to $rear$.

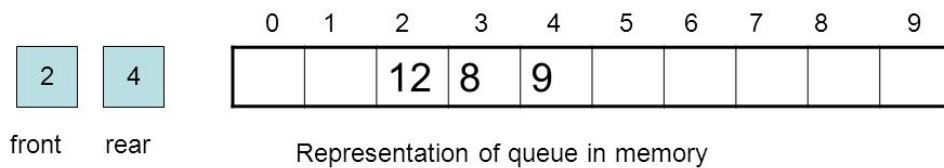
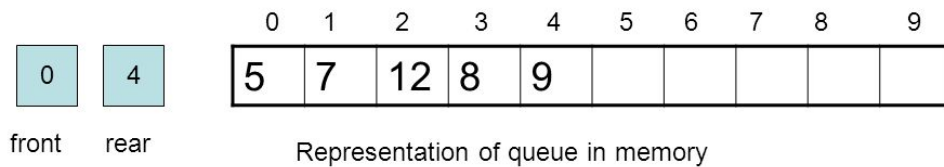
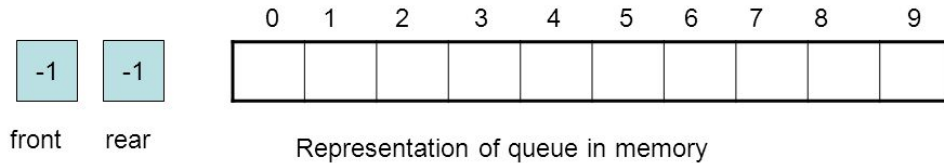


Image Source: <https://slideplayer.com/slide/4151401/>

Enqueue Operation:

```
void Enqueue(int data)
{
    // check queue is full or not
    if (capacity == rear) {
        printf("\nQueue is full\n");
        return;
    }

    // insert element at the rear
    else {
        queue[rear] = data;
        rear++;
    }
    return;
}
```

Dequeue Operation:

```
void Dequeue()
{
    // if queue is empty
    if (front == rear) {
        printf("\nQueue is empty\n");
        return;
    }

    // shift all the elements from index 2 till rear
    // to the left by one
    else {
```

```

        for (int i = 0; i < rear - 1; i++) {
            queue[i] = queue[i + 1];
        }

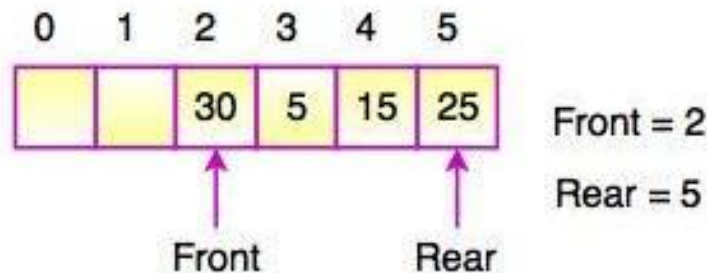
        // decrement rear
        rear--;
    }
    return;
}

```

Disadvantage of Array Implementation of Queue

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.



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

- **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 Implementation of Queue

Due to the drawbacks discussed in the previous section, the array implementation cannot 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.

Node Creation:

```
struct Node
{
    int data;
    struct Node *next;
};
```

Enqueue Operation:

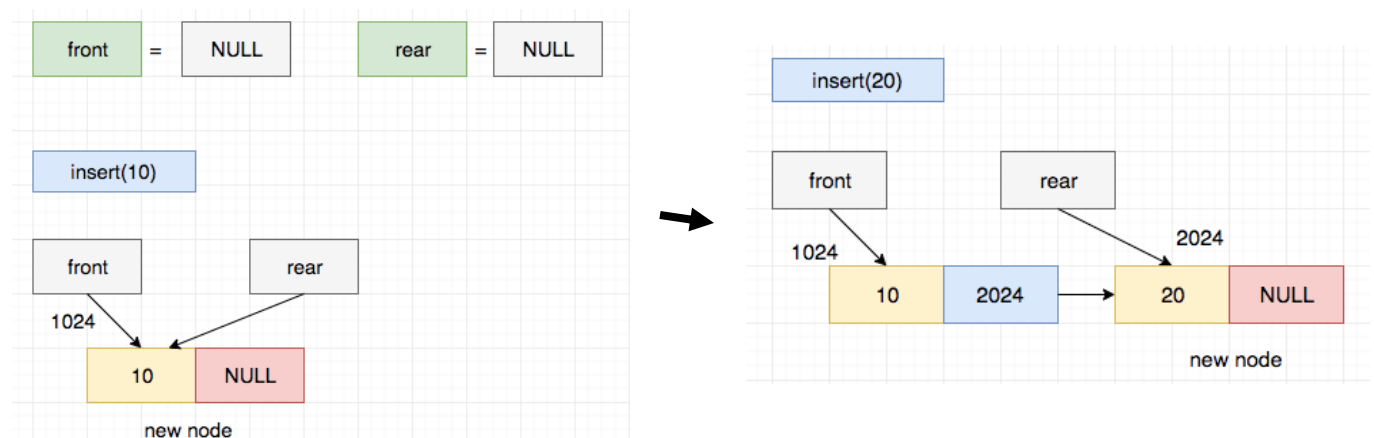


Image Source: <https://www.log2base2.com/data-structures/queue/queue-using-linked-list.html>

```
void enqueue(int value){
    struct Node *newNode;
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    if(front == NULL)
        front = rear = newNode;
    else{
        rear->next = newNode;
        rear = newNode;
    }
}
```

Dequeue Operation:

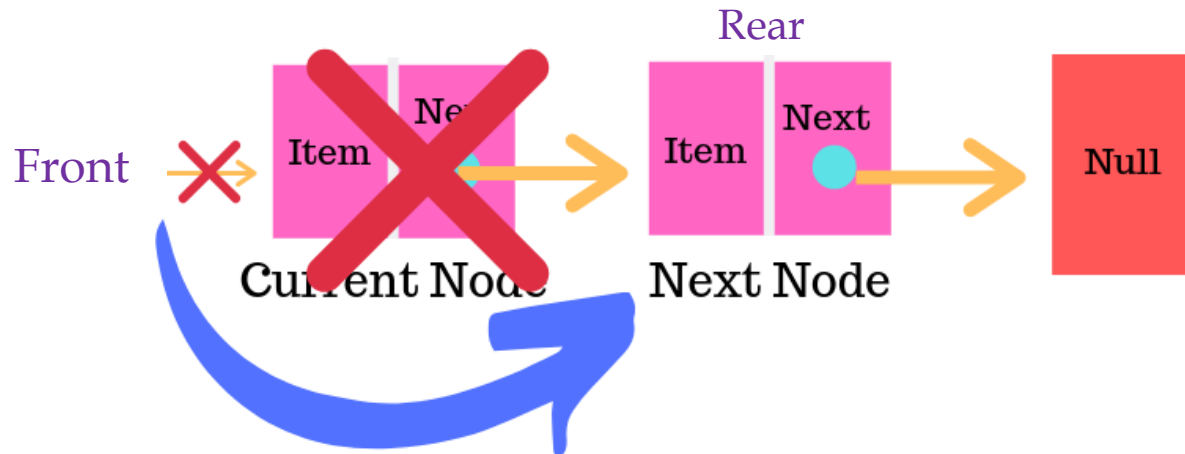


Image Source: <https://learnersbucket.com/tutorials/data-structures/implement-queue-using-linked-list/>

```
void dequeue ()
{
    if(front == NULL)
        printf("\nQueue is Empty!!!\n");
    else{
        struct Node *temp = front;
        front = front -> next;
        free(temp);
    }
}
```

CIRCULAR QUEUE

Circular Queue is a linear data structure in which first position of the queue is connected with last position of the queue.

In other words, The queue is considered as a circular queue when the positions 0 and MAX-1 are adjacent. It is also referred as RING BUFFER.

Principle Used

FIFO (First In First Out) (First entered element is processed first) is used for performing the operation.

Operations Involved

- Enqueue- is the process of inserting an element in REAR end
- Dequeue – is the process of removing element from FRONT end

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

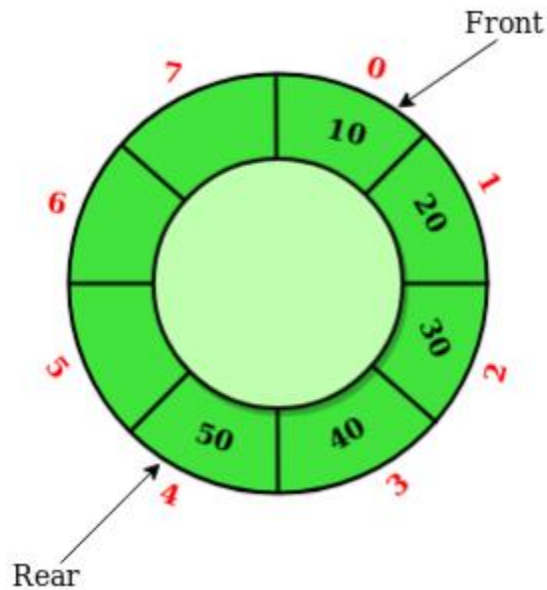
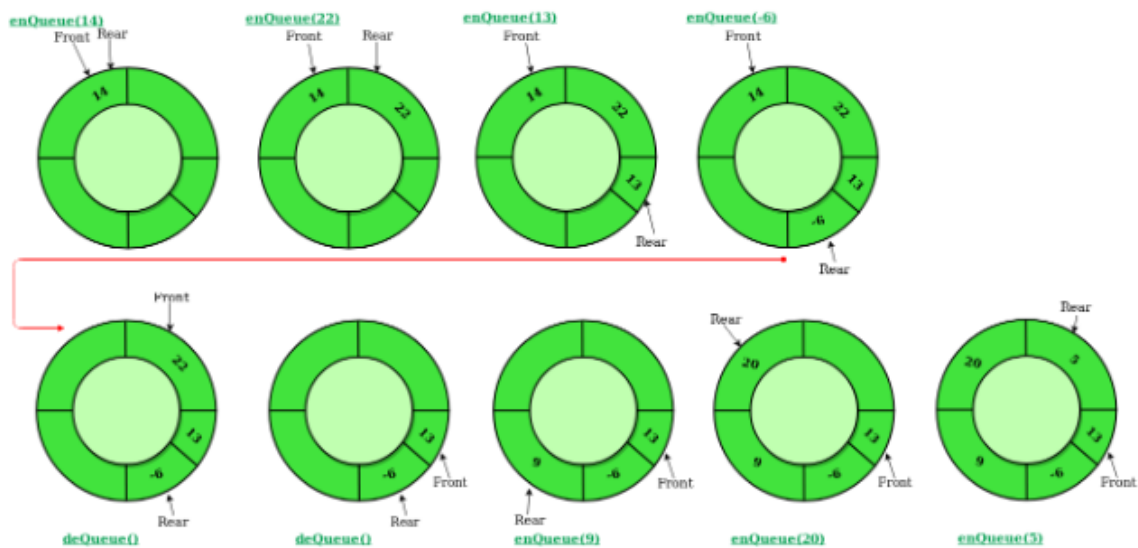


Illustration of Enqueue and Dequeue



Conditions used in Circular Queue

- Front must point to the first element.
- The queue will be empty if $\text{Front} = \text{Rear}$.
- When a new element is added the queue is incremented by value one ($\text{Rear} = \text{Rear} + 1$).
- When an element is deleted from the queue the front is incremented by one ($\text{Front} = \text{Front} + 1$).

Steps Involved in Enqueue

Check whether queue is Full or not by using the following condition

$$((\text{rear} == \text{SIZE}-1 \ \&\& \ \text{front} == 0) \ || \ (\text{rear} == \text{front}-1))$$

If queue is full, display the queue is full else we can insert an element by incrementing rear pointer.

Steps Involved in Dequeue

1. Check whether queue is Empty or not by using the following condition

$$\text{if}(\text{front} == -1).$$

2. If it is empty then display Queue is empty, else we can delete the element.

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.

Circular Queue Example-ATM

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

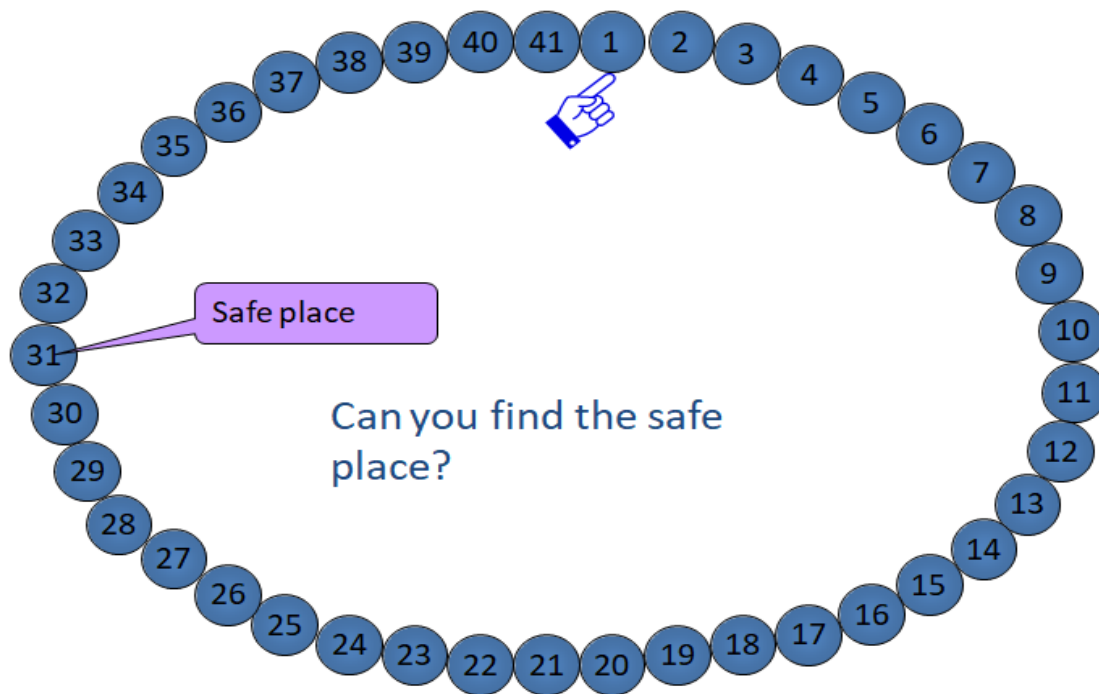
1. ATM sends request over private network to central server
2. Each request takes some amount of time to process.
3. More request may arrive while one is being processed.
4. Server stores requests in queue if they arrive while it is busy.

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”

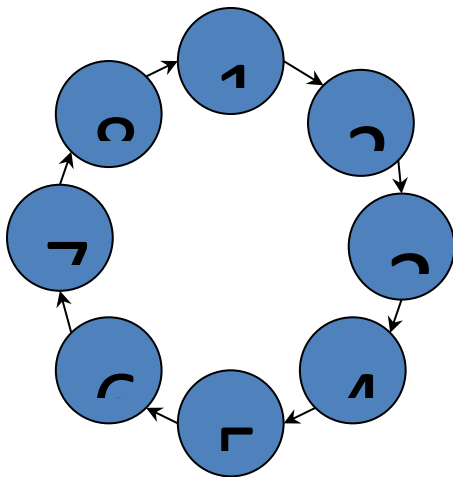
(2) a way to **simulate** “kill every 2nd person”

(3) knowing when to stop

Model n people in a circle

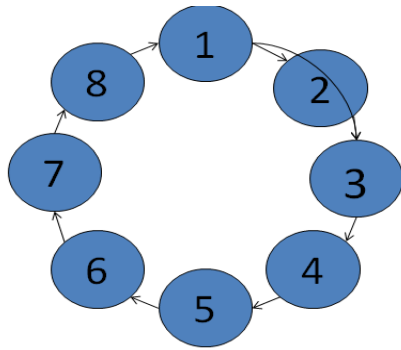
- We can use “data structure” to model it.
- This is called a “circular linked list”.
 - Each node is of some “**struct**” data type
 - Each link is a “**pointer**”

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



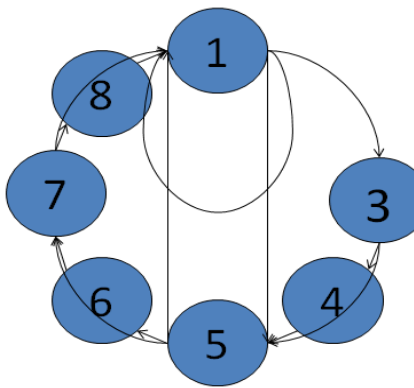
Kill every 2nd person

- Remove every 2nd node in the circular linked 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 $f(n)$ $f(8) = 1$



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

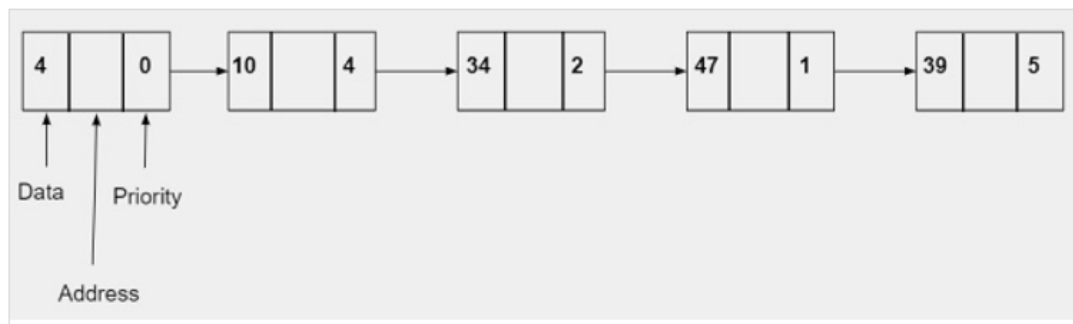
Node of linked list in priority queue

It comprises of three parts

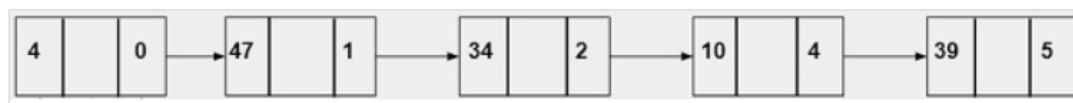
- **Data** – It will store the integer value.
- **Address** – It will store the address of a next node
- **Priority** – It will store the priority which is an integer value. It can range from 0–10 where 0 represents the highest priority and 10 represents the lowest priority.

Example

Input



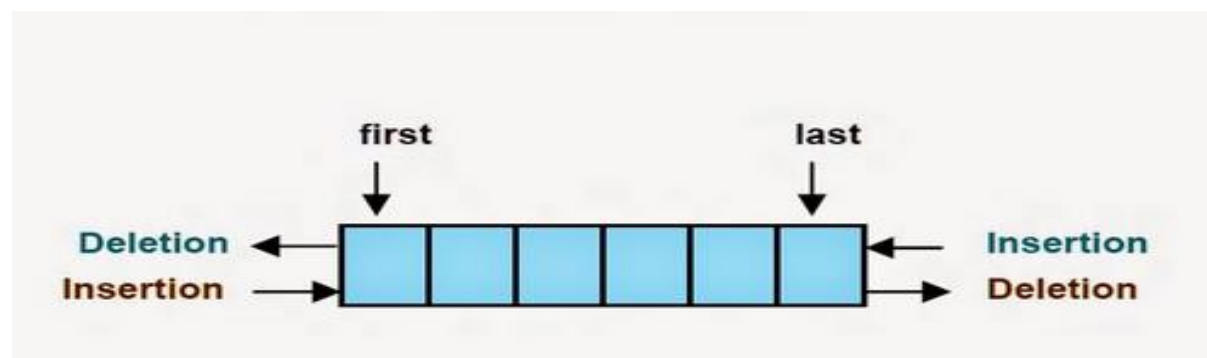
Output



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**).

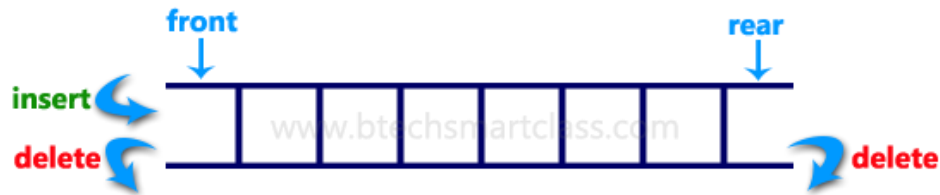
Double Ended Queue



Double Ended Queue can be represented in TWO ways

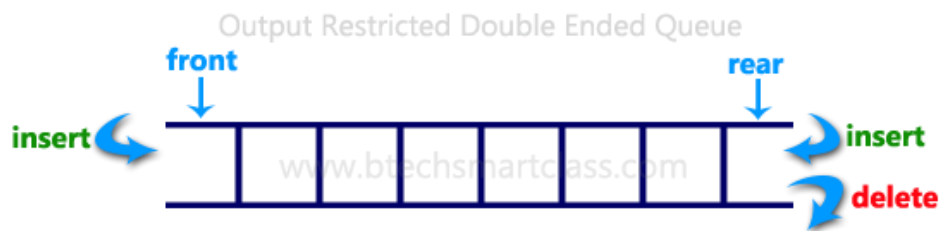
Input Restricted Double Ended Queue

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



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



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/Paranthesis-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>
11. Data Structures, Seymour Lipschutz, Schaum's Outline, 2014.
12. https://gribblelab.org/CBootCamp/7_Memory_Stack_vs_Heap.html
13. <https://www.geeksforgeeks.org/recursion/>
14. <https://www.javatpoint.com/linked-list-implementation-of-queue>
15. Book Reference for Circular queue: Referred from Seymour Lipschutz, Data Structures with C