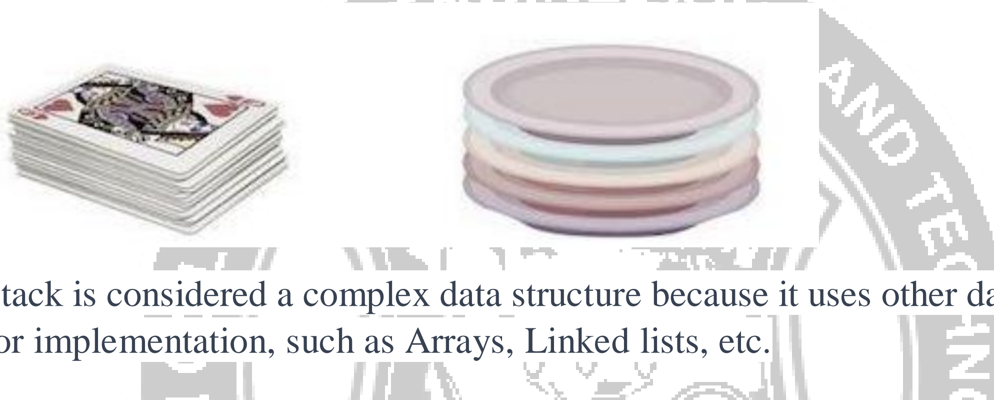
1.4. STACK – EXPRESSION EVALUATION, SYNTAX CHECKING, QUEUE – CIRCULAR QUEUE, PRIORITY QUEUE, APPLICATIONS IN SCHEDULING

A stack is a **linear data structure** where elements are stored in the LIFO (Last In First Out) principle where the last element inserted would be the first element to be deleted. A stack is an Abstract Data Type (ADT), that is popularly used in most programming languages. It is named stack because it has the similar operations as the real-world stacks, for example − a pack of cards or a pile of plates, etc. Stack is considered a complex data structure because it uses other data structures for implementation, such as Arrays, Linked lists, etc.

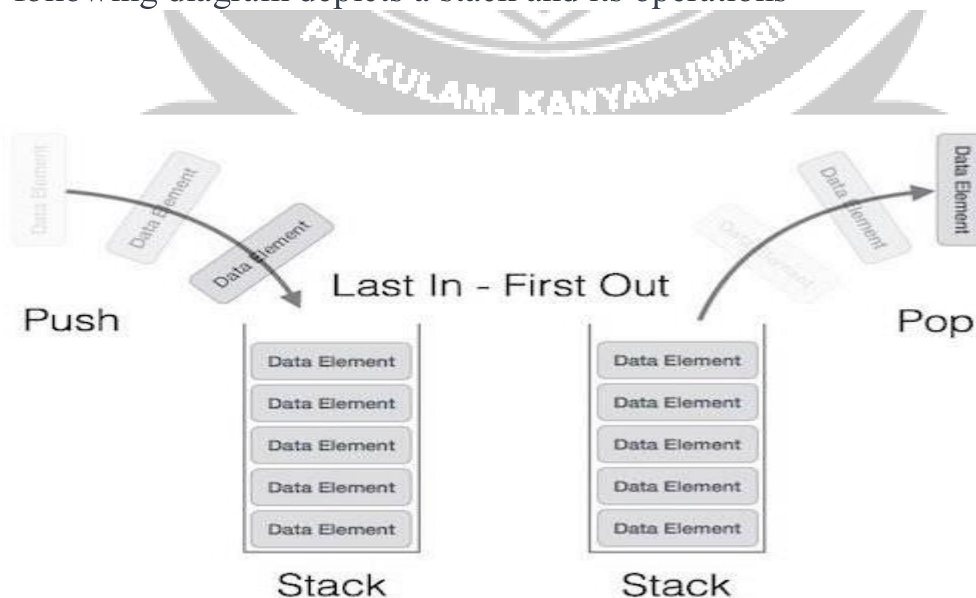


Stack is considered a complex data structure because it uses other data structures for implementation, such as Arrays, Linked lists, etc.

Stack Representation

A stack allows all data operations at one end only. At any given time, we can only access the top element of a stack.

The following diagram depicts a stack and its operations −



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic

resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

- **peek()** − get the top data element of the stack, without removing it.
- **isFull()** − check if stack is full.
- **isEmpty()** − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

First we should learn about procedures to support stack functions −

peek()

Algorithm of peek() function −

```
begin procedure peek

   return stack[top]

end procedure
```

Implementation of peek() function in C programming language −

```
int peek() {
   return stack[top];
}
```

isfull()

Algorithm of isfull() function –

```
begin procedure isfull

   if top equals to MAXSIZE
      return true
   else
      return
   false endif

end procedure
```

Implementation of isfull() function in C programming language –

```
bool isfull() {
   if(top == MAXSIZE)
      return true;
   else
      return false;
}
```

isempty()

Algorithm of isempty() function –

```
begin procedure isempty

   if top less than 1
      return true
   else
      return
   false endif

end procedure
```

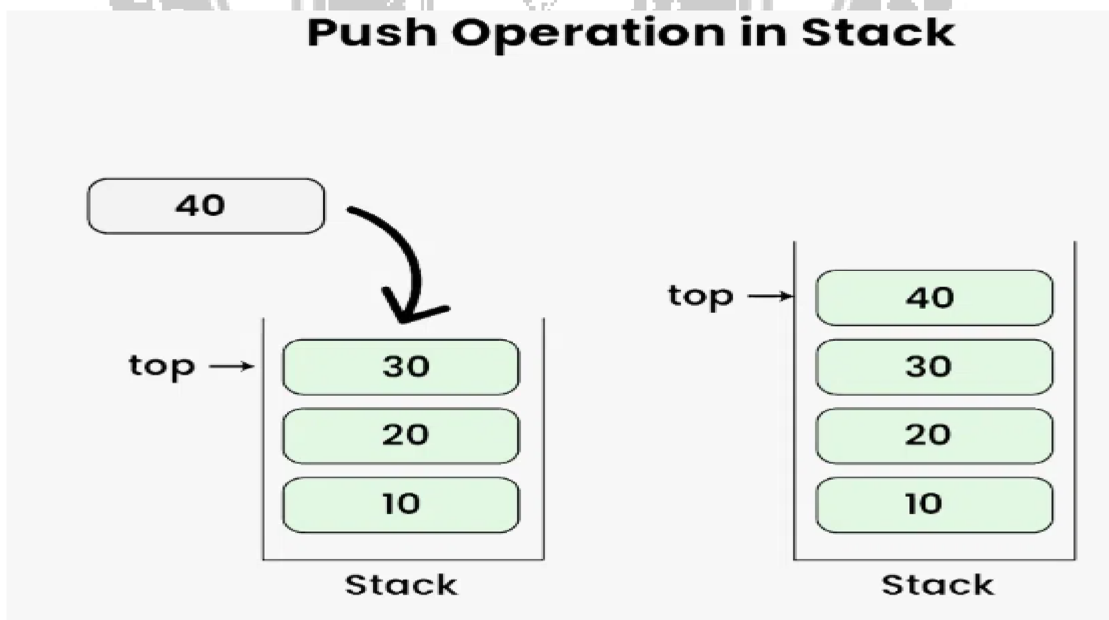
Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code –

```
bool isempty() {
   if(top == -1)
      return true;
   else
      return false;
}
```

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps –

- **Step 1** – Checks if the stack is full.
- **Step 2** – If the stack is full, produces an error and exit.
- **Step 3** – If the stack is not full, increments **top** to point next empty space.
- **Step 4** – Adds data element to the stack location, where top is pointing.
- **Step 5** – Returns success.



Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows –

```
begin procedure push: stack, data

   if stack is full
      return null
   endif

   top ← top + 1

   stack[top] ← data

end procedure
```

Implementation of this algorithm in C, is very easy. See the following code –

```
void push(int data) {
   if(!isFull()) {
      top = top + 1;
      stack[top] = data;
   }else {
      printf("Could not insert data, Stack is full.\n");
   }
}
```

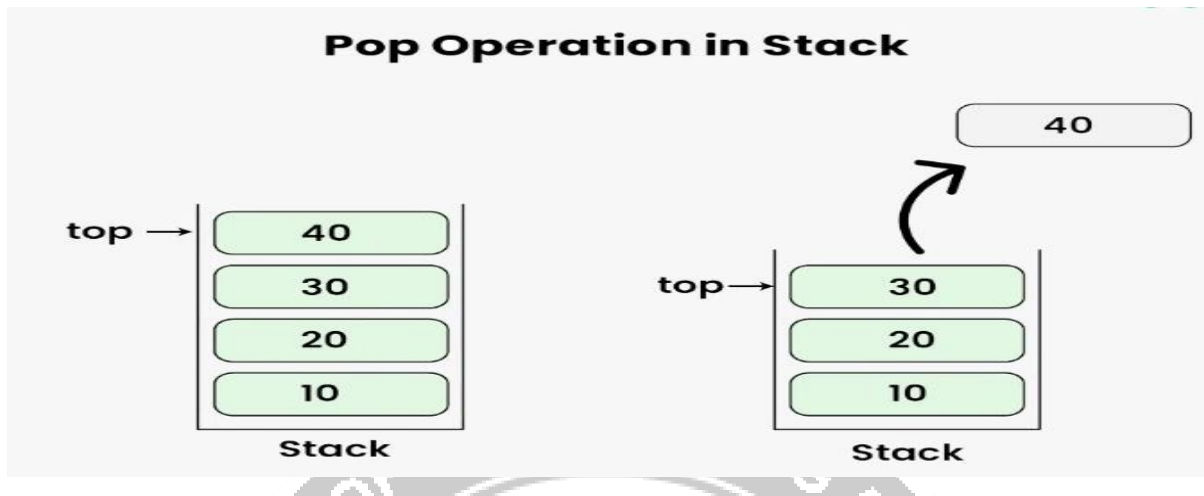
Activate W
Go to Setting

Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps –

- **Step 1** – Checks if the stack is empty.
- **Step 2** – If the stack is empty, produces an error and exit.
- **Step 3** – If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** – Decreases the value of top by 1.
- **Step 5** – Returns success.



Algorithm for Pop Operation

A simple algorithm for Pop operation can be derived as follows –

```
begin procedure pop: stack

   if stack is empty
      return null
   endif

   data ← stack[top]

   top ← top - 1

   return data

end procedure
```

Implementation of this algorithm in C, is as follows –

```
int pop(int data) {

   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   }else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
}
```

Stack Program in C

We shall see the stack implementation in C programming language here. You can try the program by clicking on the Try-it button. To learn the theory aspect of stacks, click on visit previous page.

Implementation in C

```
#include <stdio.h>


int MAXSIZE = 8;
int stack[8];
int top = -1;


int isempty() {


   if(top == -1)
      return 1;
   else
      return 0;
}
```

```
int isfull() {

   if(top == MAXSIZE)
      return 1;
   else
      return 0;
}

int peek() {
   return stack[top];
}


int pop() {
   int data;

   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   }else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
}
int push(int data) {

   if(!isfull()) {
      top = top + 1;
      stack[top] =
   data; }else {
      printf("Could not insert data, Stack is full.\n");
   }
}
```

```
int main() {
   // push items on to the
   stack push(3);
   push(5);
   push(9);
   push(1);
   push(12);
   push(15);

   printf("Element at top of the stack: %d\n" ,peek());
   printf("Elements: \n");

    // print stack data
    while(!isempty()) {
      int data = pop();
      printf("%d\n",data);
   }

   printf("Stack full: %s\n" , isfull()?"true":"false");
   printf("Stack empty: %s\n" , isempty()?"true":"false");

   return 0;
```

If we compile and run the above program, it will produce the following result –

```
Element at top of the stack: 15
Elements:
15
12
1
9
5
3
Stack full: false

Stack empty: true
```

Queue

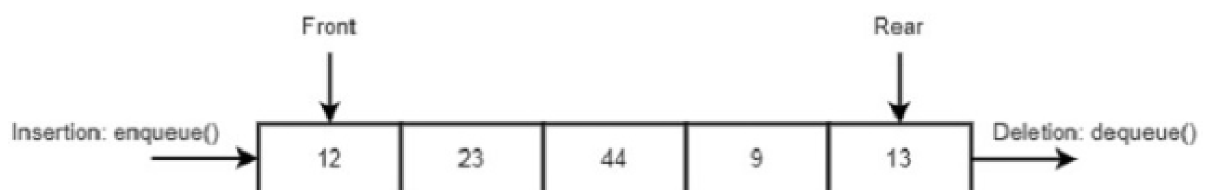
What is a Queue?

A **queue** is a linear data structure where elements are stored in the FIFO (First In First Out) principle where the first element inserted would be the first element to be accessed. A queue is an Abstract Data Type (ADT) similar to stack, the thing that makes queue different from stack is that a queue is open at both its ends. The data is inserted into the queue through one end and deleted from it using the other end. Queue is very frequently used in most programming languages.

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Representation of Queues

Similar to the stack ADT, a queue ADT can also be implemented using arrays, linked lists, or pointers. As a small example in this tutorial, we implement queues using a one-dimensional array.



Queue: FIFO Operation

Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues –

- **enqueue()** – add (store) an item to the queue.
- **dequeue()** – remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are –

- **peek()** – Gets the element at the front of the queue without removing it.
- **isfull()** – Checks if the queue is full.
- **isempty()** – Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueing (or storing) data in the queue we take help of **rear** pointer.

Let's first learn about supportive functions of a queue –

peek()

This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows –

```
begin procedure peek

   return queue[front]

end procedure
```

Implementation of peek() function in C programming language –

```
int peek() {
   return queue[front];
}
```

isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function –

```
begin procedure isfull

   if rear equals to MAXSIZE
      return true
   else
```

```
return false
   endif

end procedure
```

Implementation of isfull() function in C programming language –

```
bool isfull() {
   if(rear == MAXSIZE - 1)
      return true;
   else
      return false;
}
```

isempty()

Algorithm of isempty() function –

```
begin procedure isempty

   if front is less than MIN OR front is greater than rear
      return true
   else
      return
   false endif

end procedure
```

If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Here's the C programming code –

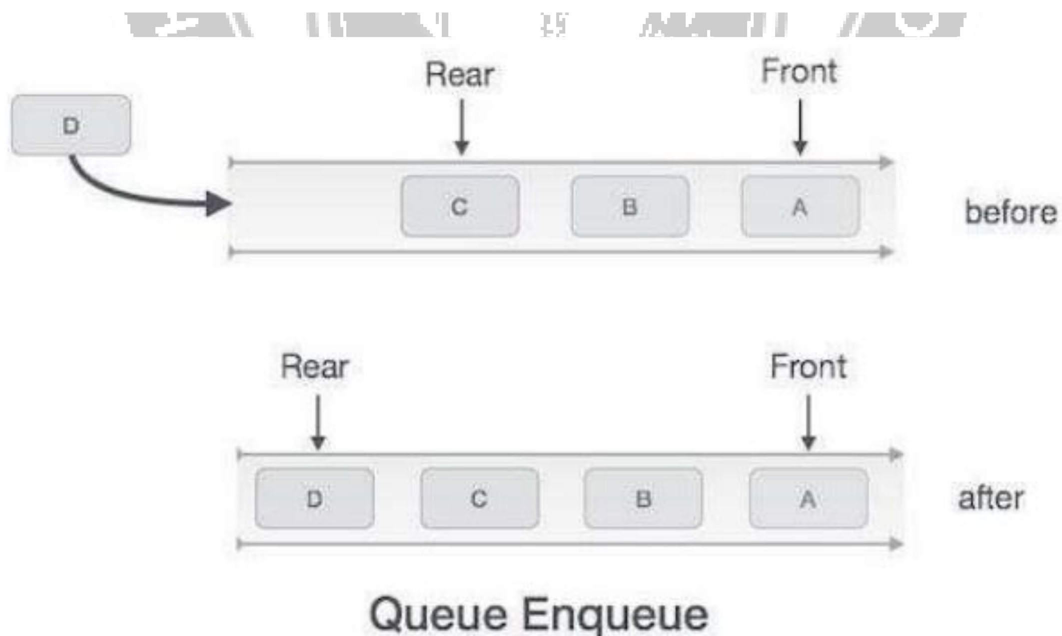
```
bool isempty() {
   if(front < 0 || front > rear)
      return true;
   else
      return false;
}
```

Enqueue Operation

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue –

- **Step 1** – Check if the queue is full.
- **Step 2** – If the queue is full, produce overflow error and exit.
- **Step 3** – If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** – Add data element to the queue location, where the rear is pointing.
- **Step 5** – Return success.



Queue Enqueue

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

Algorithm for enqueue Operation

```
procedure enqueue(data)
   if queue is full
      return
   overflow endif

   rear ← rear + 1
   queue[rear] ← data
   return true

end procedure
```

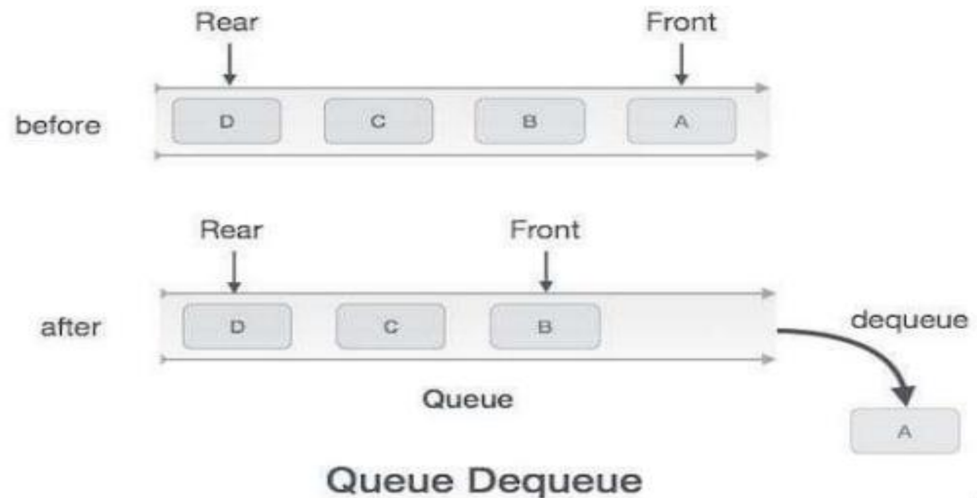
Implementation of enqueue() in C programming language –

```
int enqueue(int data)
   if(isfull())
      return 0;

   rear = rear + 1;
   queue[rear] = data;

   return 1;
end procedure
```

Dequeue Operation

Accessing data from the queue is a process of two tasks – access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation –

- **Step 1** – Check if the queue is empty.
- **Step 2** – If the queue is empty, produce underflow error and exit.
- **Step 3** – If the queue is not empty, access the data where **front** is pointing.
- **Step 4** – Increment **front** pointer to point to the next available data element.
- **Step 5** – Return success.



Queue Dequeue

Algorithm for dequeue Operation

```
procedure dequeue if
   queue is empty
      return
   underflow end if

   data = queue[front]
   front ← front + 1

   return true
end procedure
```

Implementation of dequeue() in C programming language –

```
int dequeue() {
   if(isempty())
      return 0;
   int data = queue[front];
   front = front + 1;

   return data;
}
```

Queue Program in C

We shall see the stack implementation in C programming language here. You can try the program by clicking on the Try-it button. To learn the theory aspect of stacks, click on visit previous page.

Implementation in C

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

#define MAX 6

int intArray[MAX];
int front = 0;
int rear = -1;
int itemCount = 0;

int peek(){
   return intArray[front];
}
```

```
bool isEmpty(){
   return itemCount == 0;
}
bool isFull(){
   return itemCount == MAX;
}

int size(){
   return itemCount;
}

void insert(int data){

   if(!isFull()){
```

```
        if(rear == MAX-
           1){ rear = -1;
        }

        intArray[++rear] = data;
        itemCount++;
    }
}

int removeData(){
    int data = intArray[front++];

    if(front == MAX){
        front = 0;
    }
    itemCount--;
    return data;
}

int main() {
    /* insert 5 items */
    insert(3);
    insert(5);
    insert(9);
    insert(1);
    insert(12);

    // front : 0
    // rear  : 4
    // ------------------
    // index : 0 1 2 3 4
    // ------------------
    // queue :  3  5  9  1
    12 insert(15);

    // front : 0
    // rear  : 5
```

```
// ---------------------
// index : 0 1 2 3 4  5
// ---------------------
// queue : 3 5 9 1 12 15

if(isFull()){
   printf("Queue is full!\n");
}

// remove one item
int num = removeData();

printf("Element removed: %d\n",num);
// front : 1
// rear  : 5
// -------------------
// index : 1 2 3 4  5
// -------------------
// queue : 5 9 1 12 15

// insert       more
items insert(16);

// front : 1
// rear  : -1
// ---------------------
// index : 0 1 2 3 4 5 //
---------------------
// queue : 16 5 9 1 12 15

// As  queue  is  full,  elements  will  not  be
inserted. insert(17);
insert(18);

// ---------------------
// index : 0 1 2 3 4 5 //
---------------------
```

```
   // queue : 16 5 9 1 12 15 printf("Element
   at front: %d\n",peek());

   printf("---------------------
   \n"); printf("index : 5 4 3 2 1
   0\n"); printf("-------------------
   ---\n"); printf("Queue: ");

   while(!isEmpty()){
      int n = removeData();
      printf("%d ",n);
   }
}
```

If we compile and run the above program, it will produce the following result –

```
Queue is full!
Element removed: 3
Element at front: 5
----------------------
index : 5 4 3 2 1 0
----------------------
Queue: 5 9 1 12 15 16
```

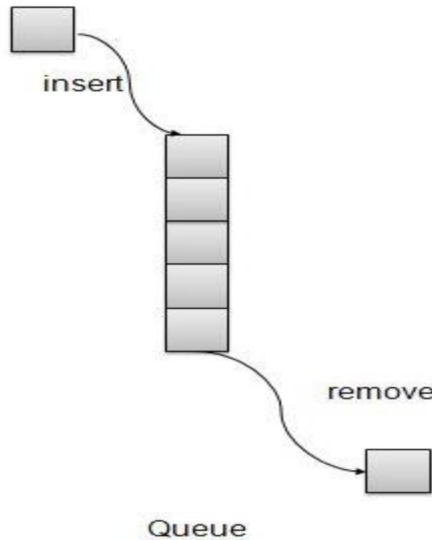
PRIORITY QUEUE

Priority Queue is more specialized data structure than Queue. Like ordinary queue, priority queue has same method but with a major difference. In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa. So we're assigned priority to item based on its key value. Lower the value, higher the priority. Following are the principal methods of a Priority Queue..

Basic Operations

- **insert / enqueue** – add an item to the rear of the queue.
- **remove / dequeue** – remove an item from the front of the queue.

Priority Queue Representation

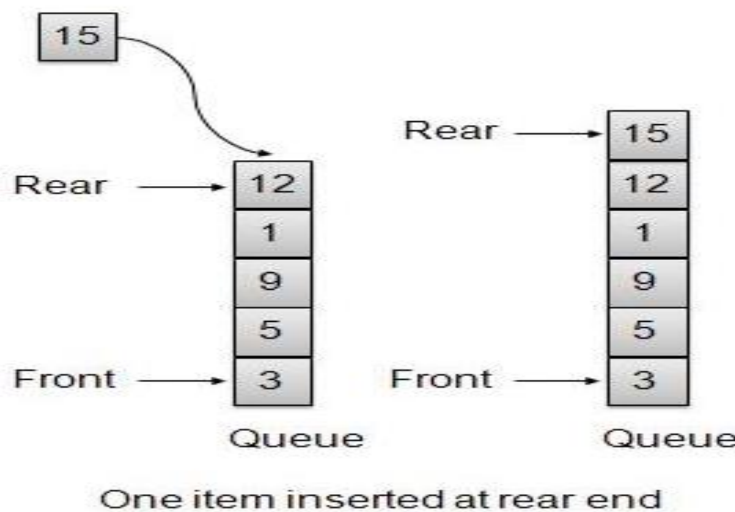


We're going to implement Queue using array in this article. There is few more operations supported by queue which are following.

- **Peek** – get the element at front of the queue.
- **isFull** – check if queue is full.
- **isEmpty** – check if queue is empty.

Insert / Enqueue Operation

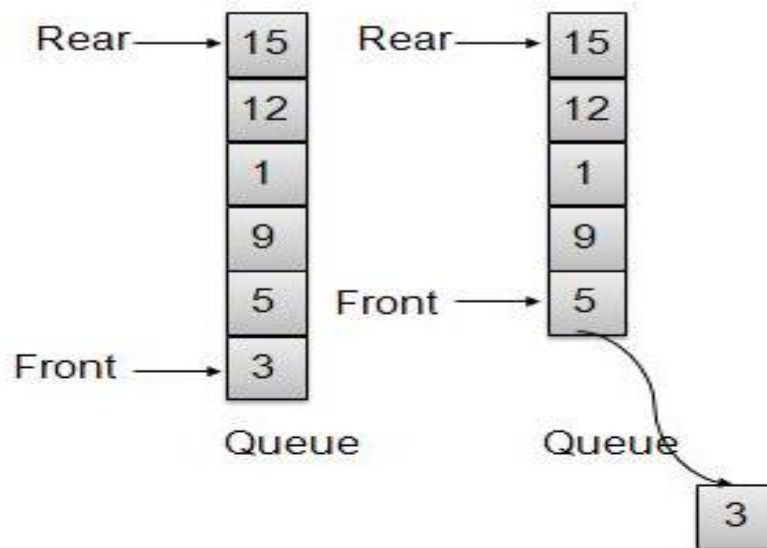
Whenever an element is inserted into queue, priority queue inserts the item according to its order. Here we're assuming that data with high value has low priority.



One item inserted at rear end

Remove / Dequeue Operation

Whenever an element is to be removed from queue, queue get the element using item count. Once element is removed. Item count is reduced by one.



C Program for Priority Queue Using Array

```
#include <stdio.h>
#define MAX 5
struct PriorityQueue {
    int data[MAX];
    int priority[MAX];
    int size;
};
void insert(struct PriorityQueue *pq, int value, int pr) {
    if (pq->size == MAX) {
        printf("Priority Queue is Full\n");
        return;
    }
    pq->data[pq->size] = value;
    pq->priority[pq->size] = pr;
```

PALKULAM, KANYAKUMARI
OBSERVE OPTIMIZE OUTSPREAD

```
    pq->size++;
    printf("Inserted %d with priority %d\n", value, pr);
}

void delete(struct PriorityQueue *pq) {
    if (pq->size == 0) {
        printf("Priority Queue is Empty\n");
        return;
    }
    int highest = 0;
    for (int i = 1; i < pq->size; i++) {
        if (pq->priority[i] > pq->priority[highest]) {
            highest = i;
        }
    }
    printf("Deleted element: %d\n", pq->data[highest]);

    for (int i = highest; i < pq->size - 1; i++) {
        pq->data[i] = pq->data[i + 1];
        pq->priority[i] = pq->priority[i + 1];
    }
    pq->size--;
}
void display(struct PriorityQueue *pq) {
    if (pq->size == 0) {
        printf("Priority Queue is Empty\n");
        return;
```

```
    }
    printf("\nElement\tPriority\n");
    for (int i = 0; i < pq->size; i++) {
        printf("%d\t%d\n", pq->data[i], pq->priority[i]);
    }
}
int main() {
    struct PriorityQueue pq;
    pq.size = 0;
    insert(&pq, 10, 2);
    insert(&pq, 20, 5);
    insert(&pq, 30, 1);
    insert(&pq, 40, 4);
    display(&pq);
    delete(&pq);
    display(&pq);
    return 0;
}
```

OUTPUT:

```
Inserted 10 with priority 2
Inserted 20 with priority 5
Inserted 30 with priority 1
Inserted 40 with priority 4
Element  Priority
10       2
20       5
30       1
```

40 4

Deleted element: 20

Element Priority

10 2

30 1

40 4

EXPRESSION EVALUATION

In the C programming language, an expression is evaluated based on the operator precedence and associativity. When there are multiple operators in an expression, they are evaluated according to their precedence and associativity. The operator with higher precedence is evaluated first and the operator with the least precedence is evaluated last.

Operator Precedence

Operator precedence is used to determine the order of operators evaluated in an expression. In c programming language every operator has precedence (priority). When there is more than one operator in an expression the operator with higher precedence is evaluated first and the operator with the least precedence is evaluated last.

Operator Associativity

Operator associativity is used to determine the order of operators with equal precedence evaluated in an expression. In the c programming language, when an expression contains multiple operators with equal precedence, we use associativity to determine the order of evaluation of those operators.

Precedence	Operator	Operator Meaning	Associativity
1	() [] → .	function call array reference structure member access structure member access	Left to Right
2	! ~ -	logical not 1's complement Unary minus (-5)	Right to Left

	++ -- & * sizeof (type)	increment operator decrement operator address of operator pointer returns size of a variable type conversion	
3	* / %	multiplication division remainder	Left to Right
4	+ -	addition subtraction	Left to Right
5	<< >>	left shift right shift	Left to Right
6	< <= > >=	less than less than or equal to greater than greater than or equal to	Left to Right
7	== !=	equal to not equal to	Left to Right
8	&	bitwise AND	Left to Right
9	^	bitwise EXCLUSIVE OR	Left to Right
10	|	bitwise OR	Left to Right
11	&&	logical AND	Left to Right
12	||	logical OR	Left to Right
13	?:	conditional operator	Left to Right
14	= *= /= %= += -= &= ^= |= <<= >>=	assignment assign multiplication assign division assign remainder assign addition assign subtraction assign bitwise AND assign bitwise XOR assign bitwise OR assign left shift assign right shift	Right to Left
15	,	separator	Left to Right

To understand expression evaluation in c, let us consider the following simple example expression...

6*2/ (2+1 * 2/3 + 6) + 8 * (8/4)

Evaluation of expression	**Description of each operation**

6*2/(2+ **1 * 2** /3 +6) +8 * (8/4)	An expression is given.
6*2/(2+**2/3** + 6) + 8 * (8/4)	1 is multiplied by 2, giving value 2.
6*2/(**2+0+6**) + 8 * (8/4)	2 is divided by 3, giving value 0.
6*2/ 8+ 8 * (**8/4**)	2 is added to 6, giving value 8.
6*2/8 + 8 * 2	8 is divided by 4, giving value 2.
12/8 +8 * 2	6 is multiplied by 2, giving value 12.
1 + **8 * 2**	12 is divided by 8, giving value 1.
1 + 16	8 is multiplied by 2, giving value 16.
17	1 is added to 16, giving value 17.

SYNTAX CHECKING

What is Syntax Checking?

Syntax checking is the process of verifying whether a C program follows the grammatical rules (syntax) of the C language.

The **compiler** performs syntax checking before converting the source code into machine code.

If the program violates C language rules, the compiler generates **syntax errors**.

Why is Syntax Checking Important?

- Detects programming mistakes early.
- Prevents program execution with invalid code.
- Improves code reliability.
- Helps developers write correct programs.

Example 1: Correct Syntax

```
#include <stdio.h>

int main()

{

   int a = 10;

   int b = 20;

   printf("Sum = %d", a + b);

   return 0;
```

}

Output

Sum = 30

The compiler finds no syntax errors because:

- Semicolons are present.
- Braces are balanced.
- Statements follow C syntax rules.

Example 2: Missing Semicolon

```
#include <stdio.h>

int main()

{

    int a = 10

    printf("%d", a);

    return 0;

}
```

Compiler Error

error: expected ';' before 'printf'

Correction

int a = 10;

COMMON SYNTAX ERRORS IN C

Error Type	Example
Missing Semicolon	int a = 10
Missing Braces	{ ...
Missing Parentheses	main(

Error Type	Example
Incorrect Quotes	printf(Hello)
Misspelled Keywords	pritnf()
Missing Header Files	Using printf() without #include <stdio.h>
Unmatched Brackets	(a+b]

APPLICATIONS IN SCHEDULING

Applications of Priority Queue in Scheduling

A **Priority Queue** is widely used in scheduling because it allows tasks with higher priority to be processed before lower-priority tasks.

1. CPU Process Scheduling

Operating systems use priority queues to decide which process should run next.

Example

- Process P1 → Priority 3
- Process P2 → Priority 8
- Process P3 → Priority 5

Execution Order:

```
P2 → P3 → P1
```

The process with the highest priority gets CPU time first.

2. Printer Scheduling

When multiple print jobs are waiting, urgent jobs can be assigned higher priority.

Example

- Student Document → Priority 2
- Office Report → Priority 5
- Exam Paper → Priority 10

Printing Order:

```
Exam Paper → Office Report → Student Document
```

3. Hospital Emergency Systems

Patients are treated based on the severity of their condition.

Example

- Minor Injury → Priority 2
- Broken Bone → Priority 6
- Heart Attack → Priority 10

Treatment Order:

```
Heart Attack → Broken Bone → Minor Injury
```

4. Network Packet Scheduling

Routers prioritize important network packets to ensure smooth communication.

Example

- Video Call Data → High Priority
- Email Data → Medium Priority
- File Download → Low Priority

Video call packets are transmitted first to reduce delays.

5. Job Scheduling in Batch Systems

Large computing systems schedule jobs according to importance or deadlines.

Example

- Payroll Processing → High Priority
- Backup Task → Low Priority

The payroll job is executed first.

Advantages in Scheduling

- Faster handling of critical tasks.
- Efficient resource utilization.
- Reduced waiting time for important jobs.
- Better system performance.