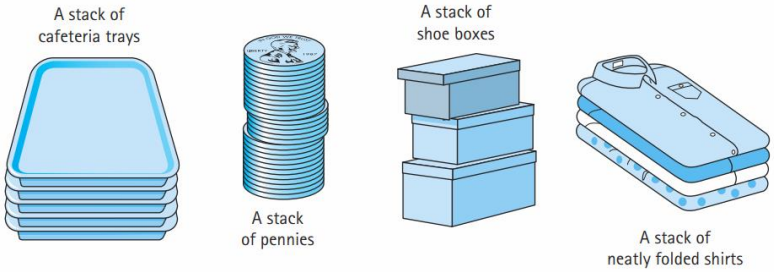
Stack & Queue

In Chapters 2 & 3 we covered the built-in data structures in C++. We did not cover any **A**bstract **D**ata **T**ype (ADT) yet. In this chapter, we are going to learn about two of the finest and simplest but most important ADTs out there: *Stack* & *Queue*. We will start with stack and explore it in detail. We are going to explore the array-based implementation of stack with both *static* and *dynamic* array in C++. Later, we will also go through the queue ADT in detail with the array-based implementation which will include both *linear queue* and *circular queue* variants in C++. Finally, we end the chapter with some applications of stack & queue. After finishing this chapter, you should be able to define stack & queue by yourself as well as learn when to apply or use them to solve problems more efficiently.

1.1 Stack

A stack is a version of a list (or array) that is particularly useful in applications involving the reversing of data sequence. In a stack data structure, all insertions and deletions of entries are made at one end, called the top of the stack. Due to this behavior stack is called a LIFO (**L**ast **I**n, **F**irst **O**ut) data structure where last inserted data remains at the top of the stack and is available to be out first.

A helpful analogy is to think of a stack of trays or plates sitting on the counter in a busy cafeteria. Throughout the lunch hour, customers take trays off the top of the stack, and employees place returned trays back on top of the stack. The tray most recently put on the stack is the first one taken off. The bottom tray is the first one put on, and the last one to be used.



A stack has two principal operations: *push* and *pop*. Inserting or adding a new element to the stack takes place from the top of the stack and thus it is called push. Similarly, an element is removed or deleted from the top of the stack making it a pop operation. Apart from these two main operations, we are going to also learn some more for a complete implementation of a stack, such as checking emptiness/ fullness of a stack, retrieving the top value/ element of the stack, and finally printing/ showing the stack elements from top to bottom.

We are going to explore an array-based implementation of stack in this chapter. This implementation opens up two possible types of stacks in terms of their size: bounded and unbounded (or non-bounded).

In case of a bounded stack, if the stack is full and does not contain enough space to accept an entity to be pushed, the stack is then considered to be in an overflow state. A pop either reveals previously concealed items or results in an empty stack – which means no items are present in stack to be removed.

On the other hand, an unbounded stack is a stack where memory for the stack is allocated dynamically. As a result, there is no chance of stack overflow to take place in such stack.

1.1.1 Implementation of Bounded Stack in C++

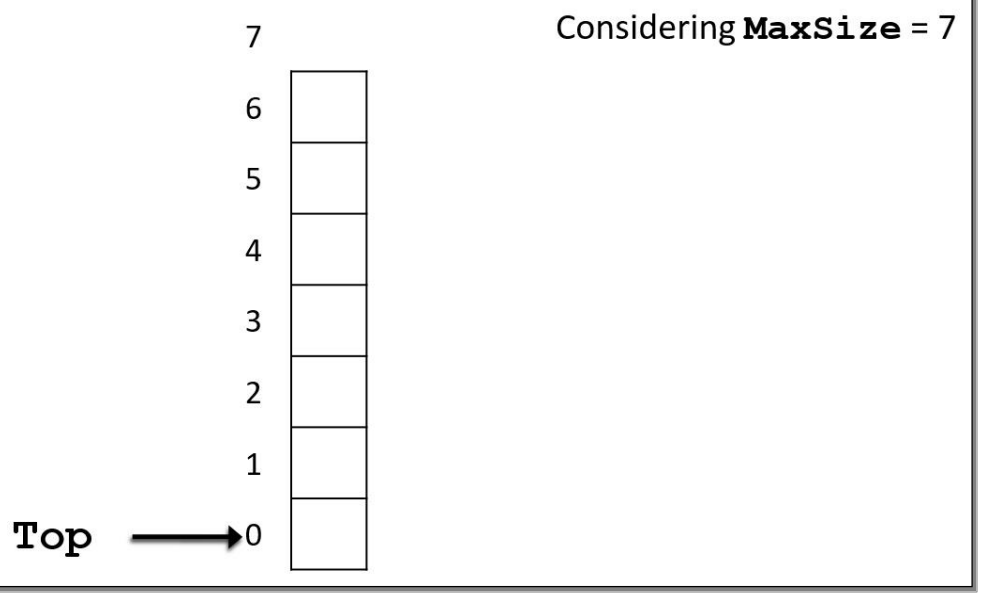
The *stack* data structure can be implemented in C++ with the help of an array. All the stack operations we mentioned earlier, can be implemented too with that approach. Below, you will find the list of functions in C++ representing those operations.

```
int Stack[100], Top=0, MaxSize=100;//Stack[] holds the elements;
Top is the index of Stack[] always holding the whereabouts of the
first/top element of the stack

bool isEmpty(); //returns True if stack has no element

bool isFull(); //returns True if stack full

bool push(int Element); //inserts Element at the top of the stack

bool pop(); //deletes top element from stack into Element

int topElement(); //gives the top element in Element

void show(); //prints the whole stack
```

Here, the array `int Stack[100]` serves the purpose of being the stack. In this case, it is an integer stack (meaning we can only hold integer values/ elements in this stack). An integer variable called `Top` keeps track of the index of the array where the top element of the stack will reside. In the beginning, the value of `Top` is 0 (zero), meaning the stack is empty but if an element were to be pushed in the stack now, they will be on the 0^{th} index of the `Stack` array. The `MaxSize` variable of type `int` holds the maximum elements that the stack can hold before it overflows as this is a bounded implementation.

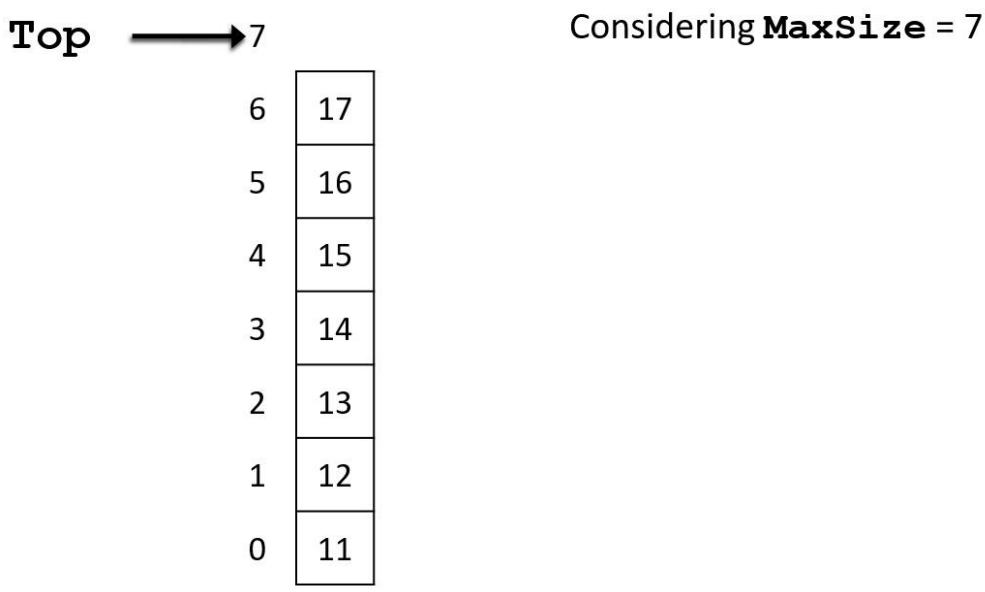
Before anything is popped from the stack, it should be checked whether the stack has any element in it in the first place. If it does not, no pop operation can be performed at all. Therefore, we need to have the ability to check whether the stack is empty in our implementation. We can achieve that by writing a function called `isEmpty`. The only purpose of this function is to return `true` if the stack is empty (when the value of `Top` is `0`) and `false` if it is not. Therefore, the return type of this function is `bool`. Below we can see an illustration of an empty stack and when the function `isEmpty` will return `true` with the help of the implementation in C++:

```
bool isEmpty(){
        /*returns True if stack is empty*/
        return (Top == 0);
}
```



Like `isEmpty`, we also need the mechanism to check whether the stack is full. If it is, then no more elements can be pushed into the stack making the stack overflow. If it is not full, then at least one more element can be pushed. We can write a function `isFull` that will do this check every time before anything is pushed into the stack. The return type of `isFull` is `bool` as it will return `true` if the stack is full (when the value of `Top` is the same as the value of `MaxSize`) and `false` otherwise. Below, let us take a look at the C++ implementation of the function along with an illustration that depicts the stack to be full:

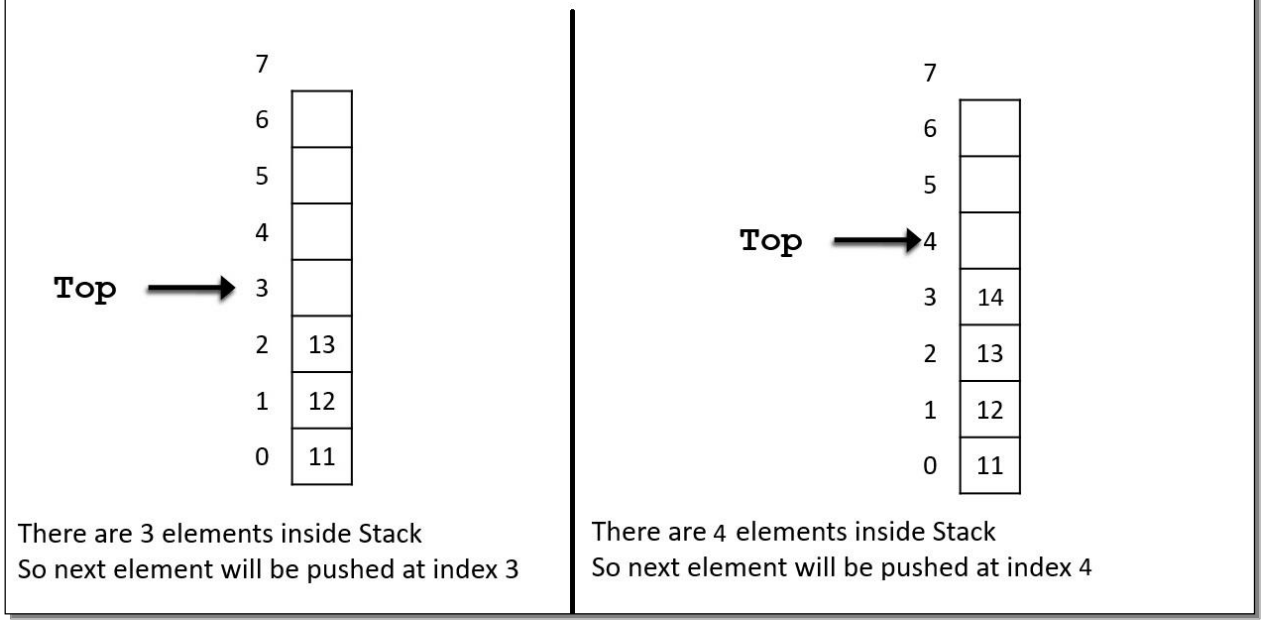
```
bool isFull(){
        /*returns True if stack is full*/
        return (Top == MaxSize);
}
```



As discussed earlier, to add or insert something into the stack, we call it *push*. We can write a function `push` in C++ to perform that operation. The function takes the element that needs to be pushed into the stack as a parameter `int Element`. The function definition will include a check for the fullness of the stack with the help of the `isFull` function defined earlier. Only when it is confirmed that the function is not full, we are then able to push new element into the stack. So, where should this new element be in the array `Stack`? Which index should it occupy in that array?

As we know, the variable `Top` keeps track of the top of the stack by remembering where the next element of the stack will be pushed. That means, if any new element is added to the stack, the index of the `Stack` array that it will occupy is what the value of `Top` is. For example, in the following illustration, the left stack is before when the element 14 is added into the stack. As it is seen there, the value of Top is then 3, meaning the next element that will be pushed into the stack will be pushed in the index 3 of the `Stack` array. After the element is pushed, we increase the value of `Top` by 1 and then as you can see in the right stack in the below-given illustration, the value of `Top` is then 4 (which means the next element that will be pushed into the stack, will occupy index 4 of the `Stack` array). So, in our implementation, we have to capture this phenomenon. And this can be done fairly simply. When we push the `Element` into the `Stack` array the statement `Stack[Top++] = Element` is the one that represents it. So, what happens here? This statement means, we are pushing the `Element` into the stack that will occupy the index `Top` (3 in the example below) of the `Stack` array. Then the index will be increased by 1 to make the value of `Top` 1 higher (4 in the example below).

```
bool push(int Element ){
     // inserts Element at the top of the stack
     if( isFull( ) ) { cout << "Stack is Full\n"; return false; }
     // push element if there is space
     Stack[ Top++ ] = Element;
     return true;
}
```



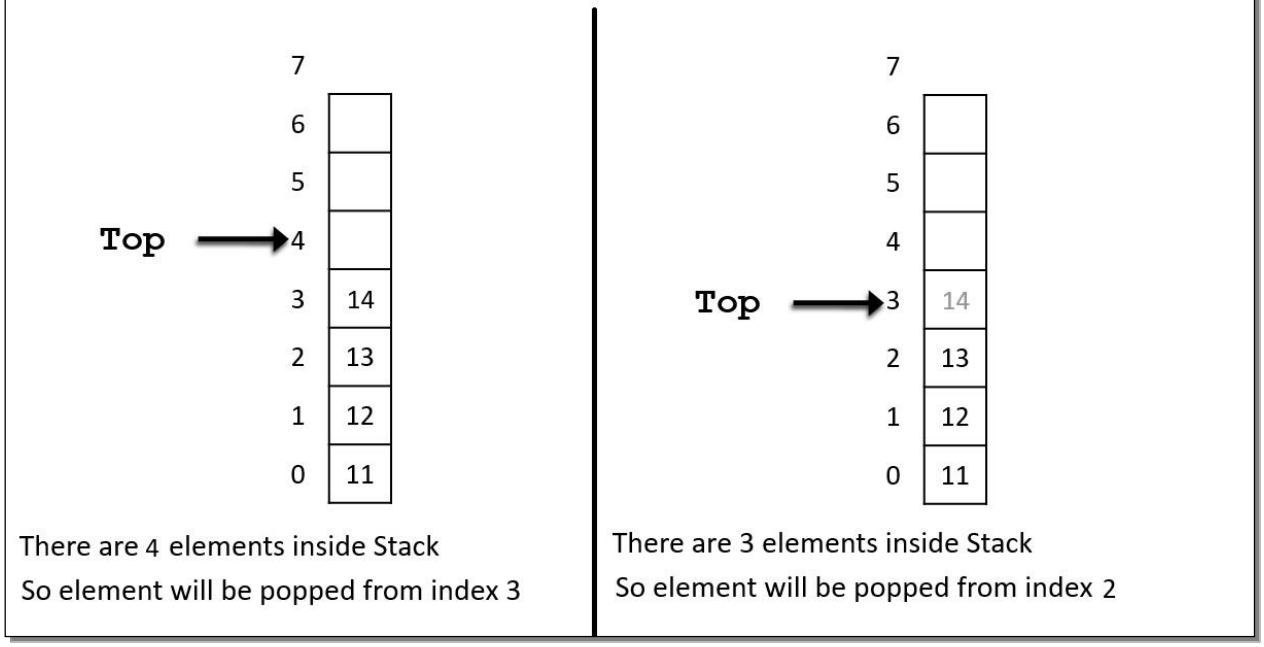
The operation of removing an element from the stack is called *pop*. We can capture this operation in our implementation with the help of a function called `pop`. The following illustration with a code-snippet shows the basics of this operation.

When the function `pop` is called, an element from the top of the stack should be removed. This will make the previous element (that was second to top of the stack before the pop operation took place) the top element of the stack now. Let us take a look at how that occurs in terms of the C++ implementation of the function.

Every time anything is popped from the stack, we need to have a check first whether the stack is empty. Because if it is, then there is nothing to pop from the stack and we get out of the `pop` function by returning `false`. That means nothing was popped from the stack. This check is done by calling the function `isEmpty`. Now, if the stack is not empty, we proceed to pop the element at the top of the stack. As this is an array-based implementation of stack and we know that we cannot delete an array element entirely by removing or freeing its memory here, we just make sure that the value of the variable `Top` is decreased by 1. That will simply mean there is one less element in the stack now. We can see in the illustration below that the stack on the left has 4 elements. After popping once, it has only 3 elements (stack on the right) and the value of `Top` is decreased by 1. Note, the element `14` remains in the `Stack` array but is not part of this particular stack data structure anymore theoretically because of where the `Top` is pointing to (as we know

the value of `Top` represents the index of the `Stack` array where the next element will be stored after being pushed. That means the index that the `Top` points to, is theoretically an empty slot for the stack data structure).

```
bool pop(){
     // removes top element from stack and puts it in
     if( isEmpty() ) { cout << "Stack empty\n"; return false; }
     Top--;
     return true;
}
```

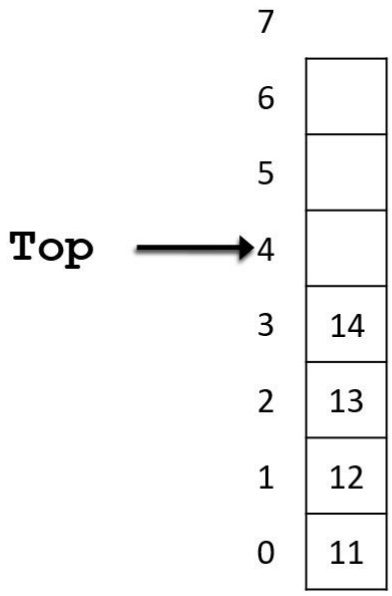


Another useful operation of a stack is to get the top value of the stack. It can be done fairly easily in C++. As `Top-1` index in the `Stack` array stores the value for the top of the stack, we just get the value of that index and then `return` it from the function. The code snippet to this is given below:

```
int topElement(){
     return Stack[ Top - 1 ];
}
```

Last but not the least, we have the operation of showing all the elements in the stack. We do this by printing all the elements in the `Stack` array from the `Top-1` index to the `0` index. We do need to check at first whether the stack is empty. For the below-given illustration, elements of the stack will be printed like `14 13 12 11`.

```
void show(){
     //prints the whole stack from top to bottom
     if(isEmpty()) { cout << "Stack empty\n"; return; }
     for( int i=Top-1; i>=0; i-- ) cout << Stack[i] << endl;
}
```



Considering **MaxSize** = 7
There are 4 elements inside Stack
So top element will be at index 3

Thus far we have explored the implementation of a bounded stack in C++. However, it is not an object-oriented implementation. This limits the use of this implemented stack in several ways. One of them is we have to be contented by just creating only one stack. But if we perform the implementation in an object-oriented manner, we can create multiple stacks by creating multiple objects of the stack class.

The first step in implementing the stack data structure in C++ with object-orientation is, there needs to be a class representing the stack data structure. The code snippet given below for such implementation has the name for that class as `MyStack`. This class will have some member variables such as `Top`, `MaxSize`, and the `Stack` array. Some member functions will also be there. These functions are the ones that we implemented previously for each operation. The newest function that will be added here is the constructor of the class. As we know a constructor initializes certain members of a class when an instance or object is created. In this scenario, when a stack object is created, generally the stack should be empty at the beginning. This will require the constructor to initialize `Top = 0` as well as assigning a size for the stack by initializing `MaxSize`. As we have already declared the `Stack` array with a size earlier, we cannot initialize `MaxSize` with a value more than the size of the array here. The member functions are not fully defined in the below-give skeleton code. Only their prototype is given. It is your task to complete this implementation.

```
class MyStack{
    int Stack[100], Top, MaxSize;
public:
    //Initializing stack
    MyStack( int Size = 100 ){ MaxSize = Size; Top = 0;}
    bool isEmpty();
    bool isFull();
    bool push(int Element);
    bool pop();
    int topElement();
    void show();
};
```

1.1.2 Dynamic Stack

Up to this point in the chapter, we have covered all the basics of a stack and gone through the implementation of it in C++. The implementation was array-based as well as of bounded stack. We are about to enhance the bounded stack to an unbounded (non-bounded) one. That will require the implementation of stack in C++ to be changed a little.

Most of the common functionalities will remain the same as before. We only have to take care of the unboundedness of the stack. And we can achieve that by making sure the array that we are basing the stack on is a *dynamic* one. That is right, we are going to be taking the help of dynamic memory allocation in order to have the capability to make the Stack array unbounded by increasing its size whenever we need in runtime. Below, we can have a look at the skeleton code of such implementation.

```
class MyStack{
    int *Stack, Top, MaxSize;
public:
    MyStack(int);
    ~MyStack();
    bool isEmpty();
    bool isFull();
    bool push(int Element);
    bool pop();
    bool topElement();
    void show();
    void resize( int size); //Resize the stack
};
```

From the above skeleton code of the *dynamic stack* implementation, it is apparent what things are new here. First of all, we are now using a pointer variable `*Stack` for the stack array, not a predefined array with size because dynamic memory allocation deals with the concept of pointer. There is a *destructor* `~MyStack`, which we did not need in earlier implementation. There is also a new function called `resize` which takes a parameter `int size`. Although it is not apparent from the skeleton code, the constructor definition is also different here which we will be dissecting now with the definition of the destructor as well. Another function that will have a slight change in definition is the `push` function.

```
MyStack::MyStack( int Size = 100 ){
    MaxSize = Size; //get Size
    Stack = new int[MaxSize]; //create array accordingly
    Top = 0; //start the stack
}

MyStack::~MyStack(){
    delete [] Stack; //release the memory for stack
}
```

As can be seen from the definition of the constructor above, we are not only initializing `Top` and `MaxSize` like bounded stack implementation but also creating the `Stack` array at runtime with the help of dynamic memory allocation. Whatever we dynamically create, we must delete it at the end of its use. Therefore, in the destructor, we are freeing up the space we created with the `new` operator using the `delete` operator.

Now, let us explore what changes are there to be made for the `push` function. Recall, this is an unbounded stack. That means, it cannot have stack overflow. Whenever there is not space in the stack for the next element to be pushed, we resize the stack by enlarging it with extra size. When it is enlarged (resized), the next element then can be pushed. This way, we avoid stack overflow and it remains unbounded. From this description, we can conclude that the job for enlarging the stack by increasing its size should be a separate function. And we can call it `resize` in this implementation. Simply put, while pushing an element into the stack, if we find the stack is full, we call the `resize` function, and that way the stack is enlarged. Then the new element can be pushed into the stack. The code snippet for the modified push function is given below.

```
void push( int Element ){
    //inserts Element at the top of the stack
    if( isFull( ) )    resize( ); //increase size if full
    Stack[ Top++ ] = Element;
}
```

Now, the only thing that remains, is to implement the `resize` function. Let us take a look at the resize function's definition by checking out the C++ code snippet of it below. Then we can discuss how it works with an illustration that should make things more clear.

```
void resize( int Size = 100 ){
    //creates a new stack with a new capacity, MaxSize + Size
    int *tempStk = new int[ MaxSize + Size ];
    //copy the elements from old to new stack
    for( int i=0; i<MaxSize; i++ ) tempStk[i] = Stack[i];
    MaxSize += Size; //MaxSize increases by Size
    delete [] Stack; //release the old stack
    Stack = tempStk; //assign Stack with new stack
}
```

Let us go through what is happening in the above code snippet by breaking it down step-by-step:

Step 1: A new `tempStk` array is created. We can assume this is a temporary stack. It is larger than our existing `Stack` array (by 100 in this example but that number can be any valid integer).

Step 2: Now, all the elements from the main stack (`Stack` array) are copied to the temporary stack `tempStk`. That means, we have two stacks now, with the same elements but one is full and smaller (`Stack`) in its total size, another is not full and larger (`tempStk`) in its total size.

Step 3: The plan was all along to increase the `MaxSize` of the Stack array so that it can hold more elements later when pushed into it. Therefore, now we increase the size of `MaxSize` with the extra size that we want the main stack (`Stack` array) to hold.

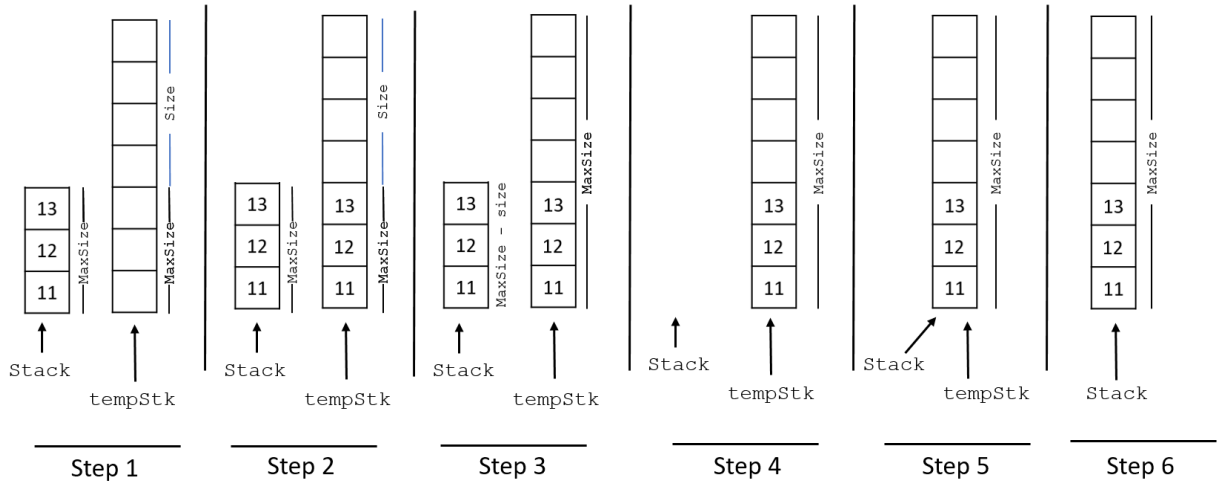
Step 4: As mentioned earlier, we have now two stacks with the same elements. But we do not need the elements of the smaller stack anymore as we have already transferred them to a much larger `tempStack`. Therefore, we release the memory where the `Stack` variable was pointing to using the `delete` operator. Recall, `Stack` is a global pointer variable that was dynamically created as an array in our implementation. Now, the `Stack` pointer is not pointing anywhere as the sequence of memory it was pointing to, was deleted.

Step 5: Both `Stack` and `tempStk` are integer pointer variables. However, `tempStk` is pointing to a sequence of memory holding all the stack elements and some extra memory. `Stack` is pointing nowhere as in the previous step that memory was removed. Recall, `Stack` is global which is working as our main stack in this implementation, and `tempStk` is a local variable to the `resize` function. That means `tempStk` is unavailable to all the functions and everything else in the implementation outside the `resize` function. And as soon as the end of the `resize` function is reached and we get out of it, all the local variable to that function will be destroyed. So, `tempStk` will also be destroyed. It only makes sense to point `Stack` to the memory that the `tempStk` is pointing to. That way, `Stack` will become an array again with all the elements and

the extra memory. That is exactly what happens when the statement `Stack = tempStk` is executed.

Step 6: We get out of the `resize` function and `tempStk` is destroyed but `Stack` remains.

Let us now take a look at an example illustration below depicting the steps above:



By following this above illustration with the steps mentioned earlier, it should be understandable how the stack gets resized.

1.2 Queue

Like stack, queue is also an ADT and is one of the simplest but most useful ones. In ordinary English, a queue is defined as a waiting line, like a line of people waiting to purchase tickets, where the first person in line is the first person served. For computer applications, we similarly define a queue to be a list in which all additions to the list are made at one end (rear), and all deletions from the list are made at the other end (front). Queues are also called first-in, first-out lists, or **FIFO** for short. An example of a real-life queue is given below.

Applications of queues are even more common than applications of stacks, since in performing tasks by computer, as in all parts of life, it is often necessary to wait for one's turn before having access to something. Within a computer system, there may be queues of tasks waiting for the printer, for access to disk storage, or even, with multitasking, for use of the CPU. Within a single program, there may be multiple requests to be kept in a queue, or one task may create other tasks, which must be done in turn by keeping them in a queue. Like a stack, a queue is a holding structure for data that we use later.

The entry in a queue ready to be served, that is, the first entry that will be removed from the queue, is called the front of the queue (or, sometimes, the head of the queue). Similarly, the last entry in the queue, that is, the one most recently added, is called the rear (or the tail) of the queue.

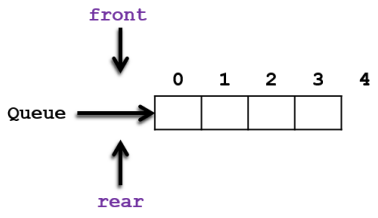
Queue has some similar operations like stack. Adding a new element in the queue is called *enqueue*. Removing an element from the queue is called *dequeue*. While implementing queue in a programming language like C++, some other helping operations are needed. For example, checking whether the queue is empty, or full. Also, we need to get the element that front of the queue for which we can write a function to get it. Then finally we need to view all the elements in the queue starting from front to rear.

We will be focusing on an array-based implementation of queue. Let us assume the array that will be used is called `Queue`. The maximum size of `Queue` is represented by a variable `MaxSize`. To denote the beginning of the queue, let us use a variable called `front`. The job of `front` is to store the index number of the `Queue` array where the first element of the queue resides. And to denote the end of the queue, let us use a variable called `rear`. The job of `rear` is to store the index number of the `Queue` array where the last element of the queue resides. With all these settled, let us take a look at the illustration of operations of *linear queue* below with the pseudo-code for each operation. But first, we have to initialize the queue as empty:

```
Initialize Queue[maxSize]; front = rear = -1;
```

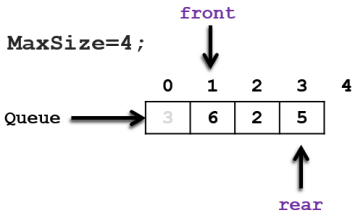
When a queue is empty, the value of `front` and `rear` is set to `-1`. This means there is no element in the `queue` array. Therefore, if we wanted to check whether a queue is empty, we could just check whether both `front` and `rear` have the value of `-1`. This can be done in C++ if the below-given pseudo-code is followed (an illustration of an empty queue is also provided).

```
isEmpty(){
 if ((front==-1) and (rear==-1)):
   then return true;
}
```



Similar to `isEmpty`, we need another function `isFull` for checking whether the queue is full. When the value of `rear` is `1` less than the value of `MaxSize`, we say the queue is full. See the pseudocode with illustration for a better understanding of this.

```
isFull(){
 if rear==(maxSize-1):
   then return true;
}
```



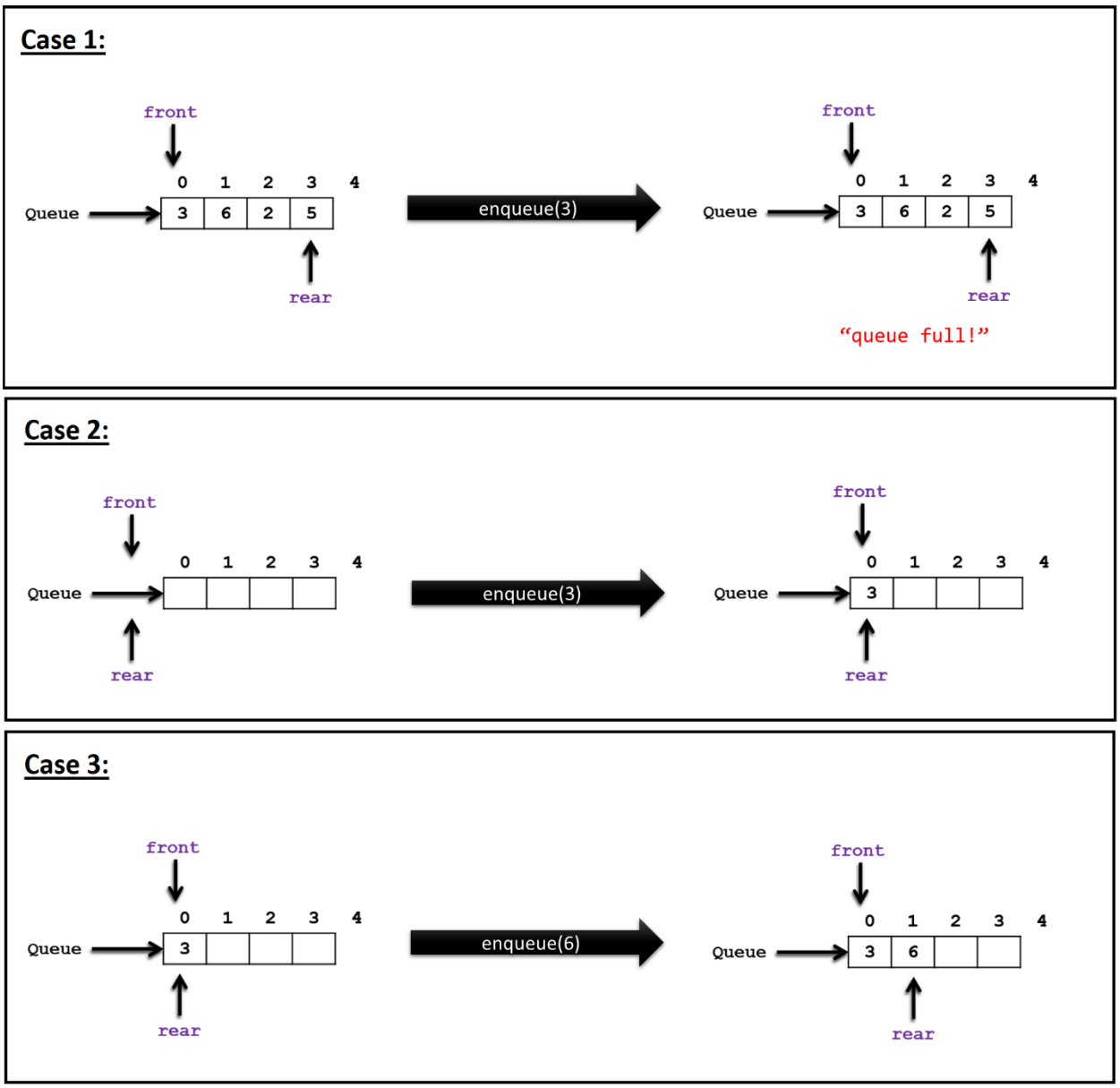
The operation of adding a new element into the queue is called *enqueue*. A new element is added at the end or `rear` of the queue. There are three cases for adding elements into the queue.

Firstly, if the queue is full, as the above illustration, it is not possible to perform enqueue operation. So, nothing gets added to the queue.

Secondly, if the queue is empty, then adding a new element would mean that is the only element in the whole queue. So, `front` and `rear` will point to the same index, which is `0` (zero).

Thirdly, if the queue is neither full nor empty, adding a new element simply means value of `rear` will be increased by 1 and that updated value of `rear` would be the index number where the new element will be added. For these three different cases, three different illustrations are provided below with the help of a pseudocode for the `enqueue` function.

```
enqueue(x){
  if(queue full): {error: "queue full!";}
  otherwise if(queue is empty): {front=rear=0; insert x in queue[rear]}
  otherwise: rear++; insert x in queue[rear];
}
```



Similar to *enqueue*, there are three cases for *dequeue* too. Recall, removing an element from the queue is called *dequeue*. An element can only be removed from the front of the queue.

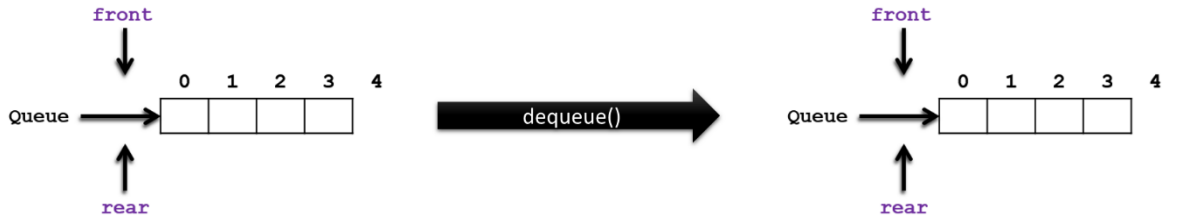
Among the three cases, firstly, if the queue is empty, nothing can be removed from the queue.

Secondly, if there is only one element in the queue (`front` and `rear` will be pointing to the same index, meaning their value would be the same), after removing that element from the queue, the queue will become empty. If that happens, the statement `front = rear = -1;` will be executed because this is what it means for the queue to be empty.

Thirdly, if there are more than one element in the queue, removing one element from the queue would mean increasing the value of `front` by `1`. See the below-given illustration with the pseudo code for a dequeue function.

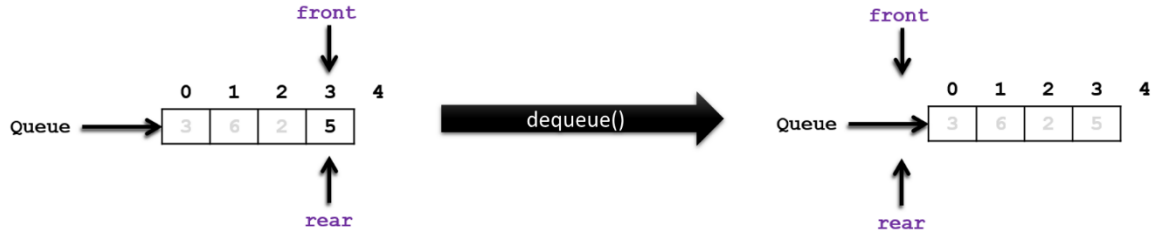
```
dequeue(){
  if(queue empty): {error: "queue is empty! dequeue not possible"}
  otherwise if (front and rear are equal): {front=rear=-1;}
  otherwise: {front++;}
}
```

Case 1:

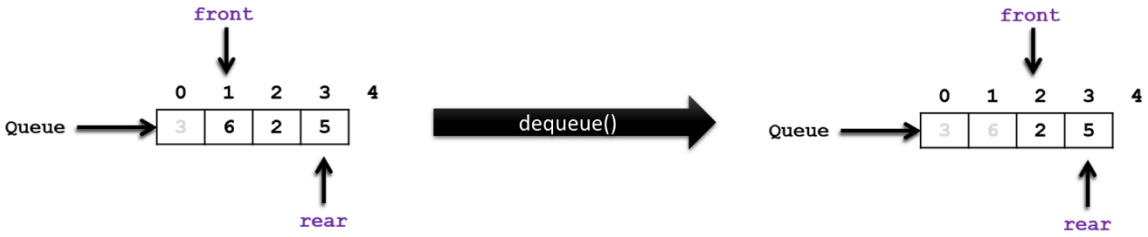


"queue is empty! dequeue not possible"

Case 2:



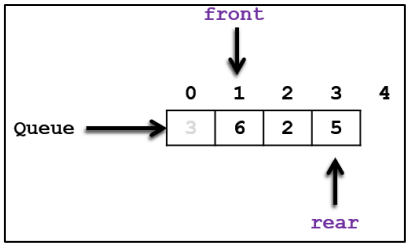
Case 3:



Returning the front element of the queue is also an operation. To perform that, we can write a function `frontElement` like below:

```
frontElement(){
  return queue[front];
}
```

We can do all kinds of operations on a queue, but if we are unable to show/ print all the elements of that queue, it will remain incomplete. To show all the elements of a queue, we can write a function `showQueue` which will print the queue from front to rear. For the following illustration of a queue, if it is printed from front to rear it will print `6 2 5`.

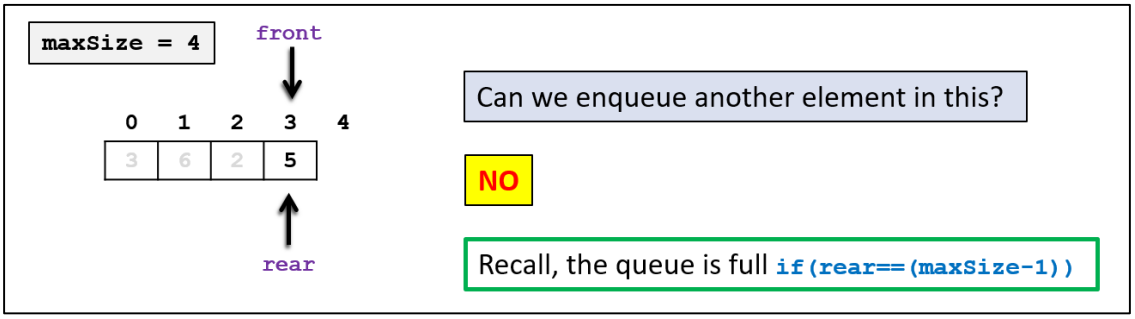



```
showQueue(){
  if (queue empty)
    error: "cannot show queue because it is empty";
  otherwise:
    for: i=front; i<=rear; i++
      output: queue[front];
}
```

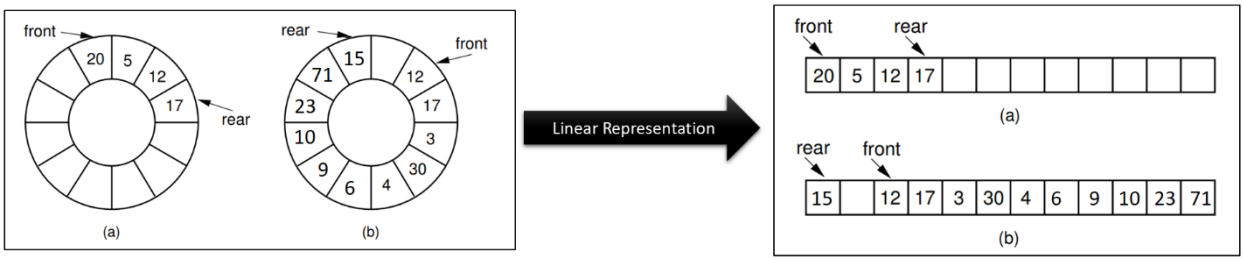
1.3 Circular Queue

The variation of queue data structure that we have studied in the last section is particularly called a *linear queue* (FIFO). That variation raises a new problem. Assume that the first element of the queue is initially at position 0 and that elements are added to successively higher-numbered positions in the array. When elements are removed from the queue, the `front` index increases. Over time, the entire queue will drift toward the higher-numbered positions in the array. Once an element is inserted into the highest-numbered (`maxSize-1`) position in the array, the queue has run out of space.

This happens even though there might be free positions at the low end of the array where elements have previously been removed from the queue. For example, in the illustration below of a linear queue, such a scenario occurs.



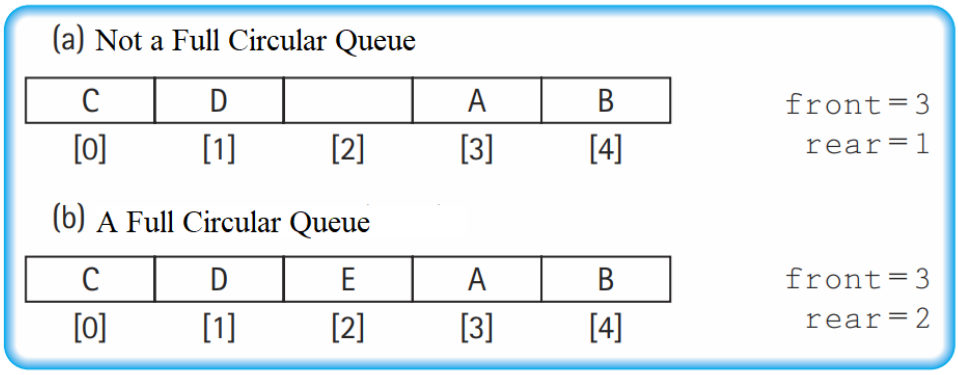
Due to this, the previously dequeued positions of the queue remain unused and memory is wasted. Thus, inefficient use of space occurs in the *linear queue*. This "drifting queue" problem can be overcome simply by thinking of the array (that represents the queue) as a circle rather than a straight line (see the illustration below). In this way, as entries are added and removed from the queue, the front will continually chase the rear around the array. At different times, the queue will occupy different parts of the array, but we never need to worry about running out of space unless the array is fully occupied, in which case we truly have an overflow.



In terms of operations, like the linear queue, circular queue also has the same ones. However, while implementing them in programming language pr pseudo code, some of the definitions for the functions need to be changed because of the structural differences between a circular and a linear queue. There is nothing to be changed in the `isEmpty` and the `frontElement` functions. They work okay for both linear and circular queue just the way they are. However, for the other functions, there are some changes to be made.

Let us first start with what it means for a circular queue to be full. In linear queue, we learned that if `(rear == (maxSize-1))` is `true`, then it is full. However, the circular queue does not rely on the highest or maximum index of the array for the end of the queue. As we learned above (with some illustrations), circular queue can circle back to the beginning of the array and fill-up unused spaces. Therefore, in the array, sometimes `rear` can appear before `front`. So, how do we measure fullness of a circular queue? We do that by changing the statement `(rear == (maxSize-1))` to `(((rear+1)%maxSize) == front)` in the definition of the `isFull` function. Now this version of the function works for the circular queue.

```
isFull(){
 if (((rear+1)%maxSize)==front):
   then return true;
}
```



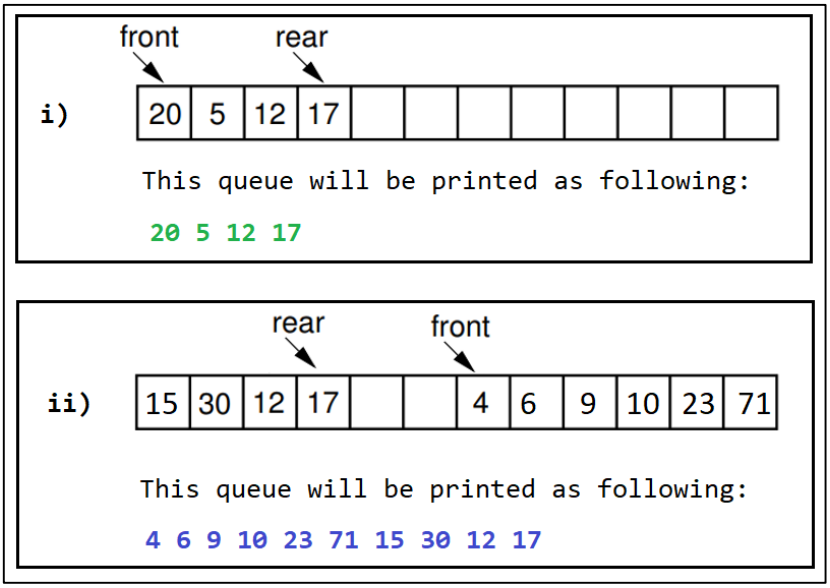
Slight changes are also in order for the functions `enqueue` and `dequeue`. As in circular queue, there is a chance of circling back to the beginning of the array, we are going to take the help of the module (`%`) operator to change certain statements in the previous definitions of `enqueue` and `dequeue` to make them suitable for circular queue. Instead of `rear++` in enqueue, we are going to use the statement `rear = (rear + 1)%maxSize`. And instead of using `front++` in dequeue, we are going to use the statement `front = (front+1)%maxSize`. Therefore, these function definitions can be re-written followingly for circular queue.

```
enqueue(x){
  if(queue full): {error: "queue full!";}
  otherwise if(queue is empty): {front=rear=0; insert x in queue[rear]}
  otherwise: rear=(rear+1)%maxSize; insert x in queue[rear];
}
```

```
dequeue(){
  if(queue empty): {error: "queue is empty! dequeue not possible"}
  otherwise if (front and rear are equal): {front=rear=-1;}
  otherwise: {front=(front+1)%maxSize;}
}
```

For printing all the elements of the linear queue, we defined the `showQueue` function earlier. The definition of this also needs to be modified to make it support the printing for a circular queue. Why do we need to change it? Because, in the circular queue, there are mainly two cases when it comes to printing all the elements of the queue: i) when `front` index is before or the same as `rear` index, ii) when `rear` index is before `front` index. We can see how both cases can be encoded in the pseudo-code and an illustration.

```
showQueue(){
  if (queue is empty)
    error: "cannot show queue because it is empty!";
  otherwise:
    if(front <= rear):
      for: i=front; i<=rear; i++
        output: queue[front];

    otherwise:
      for: i=front; i<=(maxSize-1); i++
        output: queue[front];

      for: i=0; i<=rear; i++
        output: queue[front];
}
```



1.4 Stack & Queue Applications

Now that we know how stack and queue ADTs work, we need to apply them appropriately in different problem-solving scenarios in computer science. We can use stack to solve problems such as syntax parsing, parentheses check, expression evaluation and expression conversion, banking transaction view, backtracking and implementation of recursive function, calling function, towers of Hanoi, undo mechanism in text editors, etc. Some of the applications of queue are keeping track of printing jobs, CPU task scheduling, in any customer service solution, etc. Queue can also be used alongside stack for expression evaluation and conversion. In this chapter, we will particularly be concentrated on algebraic expression conversion and evaluation (infix to postfix and postfix evaluation). We will also be learning how applying stack can detect correct and incorrect use of parentheses in an expression.

1.4.1 Algebraic Expression

Let us revisit the land of algebraic expressions shortly before we jump into their conversion or evaluation. An algebraic expression is a legal combination of operands and operators. An operand is a quantity (unit of data) on which a mathematical operation is performed. An operand can be a variable like x,y,z, or a constant like 5,4,0,9,1, etc. An operator is a symbol that signifies a mathematical or logical operation between the operands. Example of familiar operators include +, -, *, /, ^, %. Considering these definitions of operands and operators now we can write an example of expression as `x + y * z`.

1.4.2 Infix, Postfix, and Prefix Expressions

The expressions in which operands surround the operator, i.e. the operator is in between the operands. For example, `x+y`, `6*3` etc. The infix notation is the general way we write an expression. To our surprise INFIX notations are not as simple as they seem especially while evaluating them. To evaluate an infix expression we need to consider Operators' Precedence and Associative property. For example expression 3+5*4 evaluates to:

```
32 = (3+5)*4 - Wrong
```
or
```
23 = 3+(5*4) - Correct
```

Operator precedence and associativity govern the evaluation order of expression. An operator with higher precedence is applied before an operator with lower precedence. The same precedence order operator is evaluated according to their associativity order. C++ has its operator precedence and associativity chart[1].

Postfix is also known as Reverse Polish Notation (RPN) where the operator comes after the operands, i.e. operator comes post of the operands, so the name postfix. e.g. xy+, xyz+*, etc.

Prefix is also known as Polish Notation (PN) where the operator comes before the operands, i.e. operator comes pre of the operands, so the name prefix. e.g. +xy, *+xyz, etc.

For computers, infix expressions are hard to parse because operator priorities, tiebreakers, and delimiters are needed. This makes the evaluation of expression more difficult than is necessary for the processor. Both prefix and postfix notations have an advantage over infix that while evaluating an expression in prefix or postfix form we need not consider the Precedence and Associative property. The expression is scanned from the user in infix form; it is converted into prefix or postfix form and then evaluated without considering the parentheses and priorities of the operators. So, it is easier (complexity wise) for the processor to evaluate expressions that are in these forms. Let us juggle our memory a little by going through the following examples of conversions from infix to postfix and prefix and their evaluations.

[1] https://en.cppreference.com/w/cpp/language/operator_precedence

Infix	PostFix	Prefix
A+B	AB+	+AB
(A+B) * (C + D)	AB+CD+*	*+AB+CD
A-B/(C*D^E)	ABCDE^*/-	-A/B*C^DE

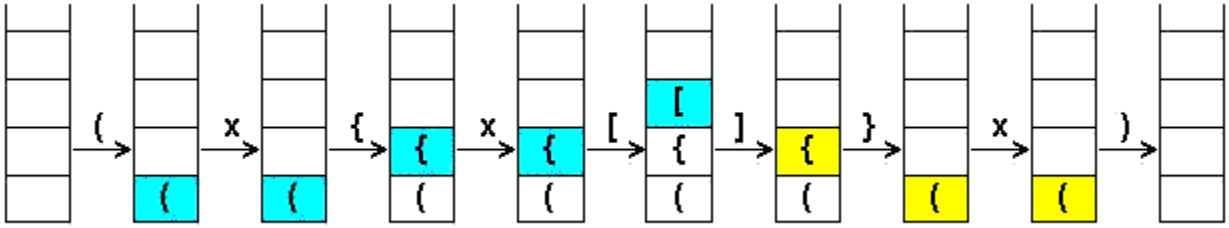
A-B/(C*D^E)		ABCDE^*/-		-A/B*C^DE	
A-B/(C*F)		ABCF*/-	D^E	-A/B*CF	D^E
A-B/G		ABG/-	C*F	-A/BG	C*F
A-H		AH-	B/G	-AH	B/G
I		I	A-H	I	A-H

1.4.3 Parentheses Check Using Stack

Using stack, we can check whether an expression has its parenthesis properly placed; i.e., whether it is opening and closing parentheses match with each other. For example, let's take the expression `(x{x[]}x)`. We will read the expression as a string starting from the first character to the last and for each character, we will do the following three things:

- Whenever we get an opening parenthesis, we will push it into the stack.
- When we get a closing parenthesis we will check that with the top of the stack. If the top of the stack has the same type of opening parenthesis, we will pop it.
- We skip the character in the string which is not a parenthesis.

Finally, if we reach the end of the expression and the stack is also empty, that means the expression is "well-formed". In any other case, the expression is "not well-formed". Let us take a look at the illustration of this process below for the parentheses check for: `(x{x[]}x)`.



This expression is well-formed.

1.4.4 Infix to Postfix Conversion

In this section of the chapter, we are going to learn the technique of converting an infix expression to a postfix one. We will be using a stack to process the input infix expression (which will be input as a string) and use a queue to store the resultant postfix expression. This can be achieved by following the pseudo code given below:

```
Add ')' to the end of Infix;     Push( '(' );
do{
  OP = next symbol from left of Infix;
  if OP is OPERAND then EnQueue( OP );
  else if OP is OPERATOR then{
    if OP = '(' then Push( OP );
    else if OP = ')'  then{
        while TopElement() != '(' do{
          Enqueue(TopElement());
          Pop();
        }
          Pop();
    }else{
    while Precedence( OP ) <= Precedence( TopElement() ) do{
          Enqueue(TopElement());
          Pop();
    }
    Push( OP );
    }
}while !IsEmpty();
```

We can translate the above-mentioned pseudo-code in English following some steps:

a) Add a parenthesis ‘)’ to the end of the infix expression we took as an input string (let us call it **I**).
b) Push a parenthesis ‘(’ in the stack (let us call it **S**).
c) Now we start reading from the start of **I** character by character. For each character, we will be following the instructions of the next steps.
d) If the character is an *operand*, we enqueue it to the output postfix queue (let us call it **Q**).
e) Otherwise, if it is an *operator*, then we check which kind of *operator* it is.
    - If it is an opening parenthesis ‘(’ then we push it in **S**.
    - Otherwise, if it is a closing parenthesis ‘)’ then we do the following:
        - enqueue the top element of **S** into **Q**
        - pop it from **S**
        - repeat this process until we find the top element of **S** to be an opening parenthesis ‘(’
    - Otherwise, we check the precedence of it with the precedence of the element top of **S** (let us call it **t**). Then we do the following:
        - if the precedence of **t** is higher or equal, enqueue **t** to **Q** and pop it.
        - repeat this process until the precedence of **t** is lower. Then we push the operator (that we were reading from the infix expression) in **S**.
f) When the stack is empty, we are done converting the infix expression to a postfix one. When we print **Q** we will have found the postfix expression.

We can take a look at the following example (simulation) of converting an infix expression to a postfix one using the above-mentioned technique. Here, infix expression: `2*6/(4-1)+5*3`. If

we add ‘)’ at the end of it: `2*6/(4-1)+5*3)`, then a ‘(’ will be added to stack from the beginning for that.

Symbol	Stack	Postfix (Queue)
2	(	2
*	(*	2
6	(*	26
/	(/	26*
(	(/(	26*
4	(/(	26*4
-	(/(-	26*4
1	(/(-	26*41
)	(/	26*41-
+	(+	26*41-/
5	(+	26*41-/5
*	(+*	26*41-/5
3	(+*	26*41-/53
)		26*41-/53*+

Suffice to say, this conversion technique will only work for single character operands when implemented using a programming language.

1.4.5 Evaluation of Postfix Expression

The technique of evaluating a postfix expression in mathematics is shown in section 4.3.2 of this chapter. However, in this section, we will be using a stack in combination with a queue for the evaluation. We will follow the below-given pseudo code for this. This technique will only work for single character operands when implemented using a programming language.

```
Postfix Expression: 26*41-/53*+
EnQueue( ')' );
while ( FrontElement() != ')' ) do{
    OP = FrontElement();
    DeQueue();
    if OP is OPERAND then Push( OP );
    else if OP is OPERATOR then{
          OperandRight = TopElement();
          Pop();
          OperandLeft = TopElement();
          Pop();
          x = Evaluate(OperandLeft, OP, OperandRight);
          Push(x);
    }
}
Result = TopElement();
Pop();
cout << Result;
```

An example simulation of how we can use the above-mentioned technique in practice can be seen below. Let us take the postfix expression `2,6,*,4,1,-,/,5,3,*,+` for evaluation. This postfix expression is treated as a queue, where each symbol of it is a queue element. We are using commas to separate the numbers so that there is no confusion.

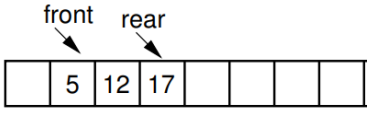
Symbol	Stack
2	2,
6	2,6
*	12
4	12,4
1	12,4,1
-	12,3
/	4
5	4,5
3	4,5,3
*	4,15
+	19
)	19

Result =19

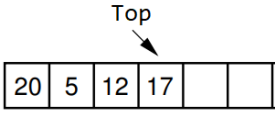
1.5 Exercises

1. Implement a Generic Stack in C++.
2. With the help of the pseudo-code available for different operations of the *linear queue*, implement it using object orientation in C++.

3. With the help of the pseudo-code available for different operations of the *circular queue*, implement it using object orientation in C++.
4. Explain the differences between Stack and Queue with appropriate examples.
5. Explain the need for a *circular queue* with appropriate examples.
6. You are given a *linear queue* of maximum size = 8. The state of the queue currently is:



   Perform simulation on this queue for the following operations:
   enqueue(3), dequeue(), enqueue(53), dequeue(), enqueue(23), dequeue(), enqueue(33), dequeue(), enqueue(32), enqueue(13).
7. Now assume the queue given in question **6** is circular. Perform the simulation of question **6** on it.
8. You are given a bounded stack of maximum size = 6. The state of the stack currently is:



   Perform simulation on this stack for the following operations:
   push(21), pop(), pop(), pop(), push(22), pop(), push(92), push(23), pop(), pop(), pop().
9. Show the simulation of coverting each of the following infix expressions to postfix:
   - `a+(b*((c-d)/c))-(x+y*z)`
   - `(a*b*c)/(d+e+f)-x-y+z`
   - `((a/b)+(c*d))*((a*b)-(c/d))`
10. Show the simulation for evaluating each of the following postfix expressions (operators and operands are comma-separated for clarity):
    - `7,4,3,*,1,5,+,/,*`
    - `5,7,+,6,2,-,*`
    - `4,2,+,3,5,1,-,*,+`
    - `5,3,2,*,+,4,-,5,+`

1.6 References

1. Data structures and Program Design in C++ by Robert L.Kruse, Alexander J.Ryba. Chapter 2, 3.
2. Nell Dale - C++ Plus Data Structures, Third Edition (2002, Jones and Bartlett Publishers, Inc.). Chapter 4.
3. Dr. Clifford A. Shaffer - Data Structures and Algorithm Analysis in C++, 3rd Edition -Dover Publications (2011). Chapter 4.2, 4.3.
4. "Advanced Data Structures", Peter Brass, Cambridge University Press, 2008. [Chapter 1: 1.1, 1:1.2]
5. https://en.wikipedia.org/wiki/Stack_(abstract_data_type)
6. https://www.cs.usfca.edu/~galles/visualization/StackArray.html
7. https://en.wikipedia.org/wiki/Queue_(abstract_data_type)
8. https://www.cs.usfca.edu/~galles/visualization/QueueArray.html
9. https://en.wikipedia.org/wiki/Circular_buffer

10. http://www.cs.uregina.ca/Links/class-info/210/Stack/
11. http://www.cs.csi.cuny.edu/~zelikovi/csc326/data/assignment5.htm