Deepak Mitra CDAC-COE

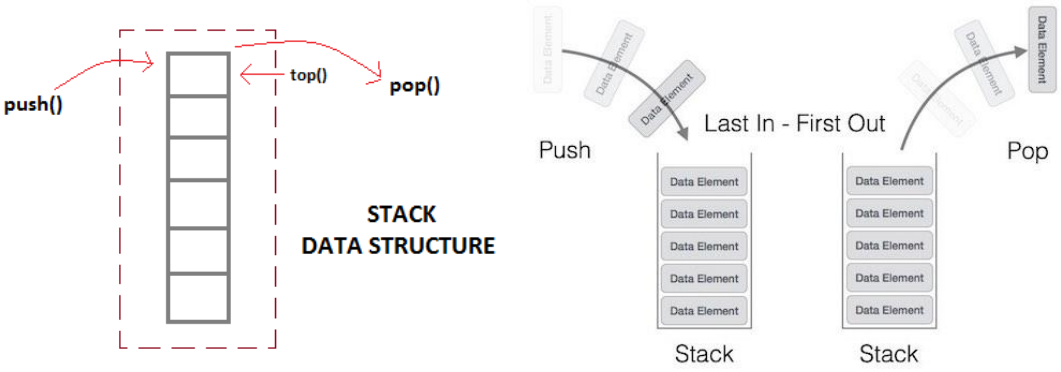
STACKS AND QUEUES

STACK

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, the only element that can be removed is the element that was at the top of the stack, just like a pile of objects.

- Stack is a last-in, first-out (LIFO) data structure. i.e. the element added last to the stack will be the one to be removed first.
- Typical use cases of stacks include:- (1) During debugging it is quite common to examine the function call stack during panics. (2) In RTOS like Symbian there are concepts like cleanup stack to avoid memory leaks.
- Some of the common terminology associated with stacks include PUSH, POP and TOP of the stack.
- PUSH refers to adding an element to the stack.
- POP refers to removing an element from the stack.
- TOP refers to the first element that could be POPed (or) the last element PUSHed.

Diagram below explains the stack.



Basic features of Stack

1. Stack is an ordered list of similar data type.
2. Stack is a **LIFO** structure. (Last in First out).
3. **push()** function is used to insert new elements into the Stack and **pop()** is used to delete an element from the stack. Both insertion and deletion 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.

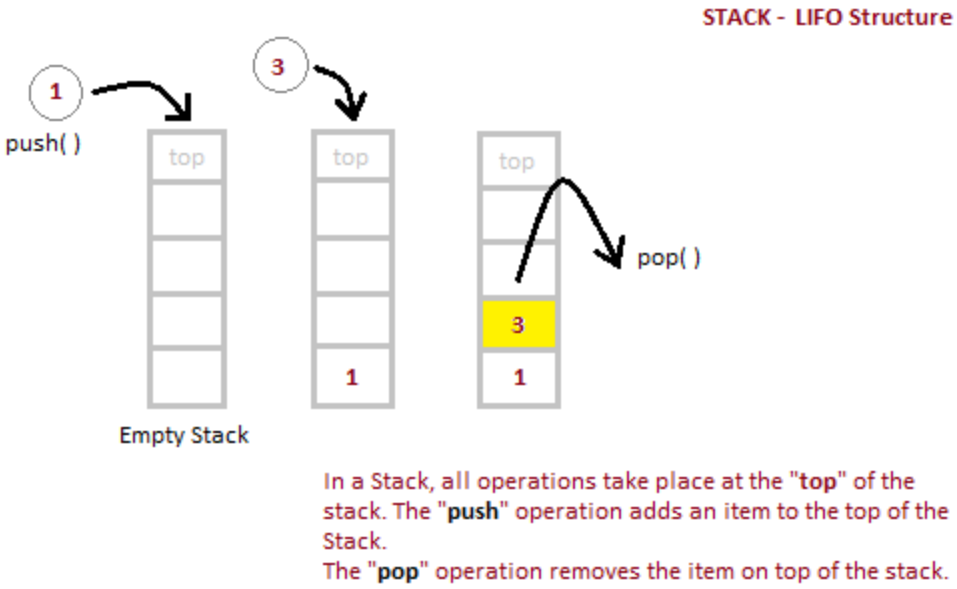
Applications of Stack

The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.

There are other uses also like : **Parsing**, **Expression Conversion**(Infix to Postfix, Postfix to Prefix etc) and many more.

Implementation of Stack

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 de-allocate, but is not limited in size. Here we will implement Stack using array.



/* Below program is written in C++ language */

```
Class Stack
{
  int top;
  public:
  int a[10];    //Maximum size of Stack
  Stack()
  {
    top = -1;
  }
};

void Stack::push(int x)
{
  if( top >= 10)
  {
    cout << "Stack Overflow";
  }
  else
  {
```

```
    a[++top] = x;
    cout << "Element Inserted";
  }
}

int Stack::pop()
{
  if(top < 0)
  {
    cout << "Stack Underflow";
    return 0;
  }
  else
  {
    int d = a[--top];
    return d;
  }
}

void Stack::isEmpty()
{
  if(top < 0)
  {
    cout << "Stack is empty";
  }
  else
  {
    cout << "Stack is not empty";
  }
}
```

Position of Top	Status of Stack
-1	Stack is Empty
0	Only one element in Stack
N-1	Stack is Full
N	Overflow state of Stack

Analysis of Stacks

Below mentioned are the time complexities for various operations that can be performed on the Stack data structure.

- **Push Operation** : O(1)
- **Pop Operation** : O(1)
- **Top Operation** : O(1)
- **Search Operation** : O(n)

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 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
```

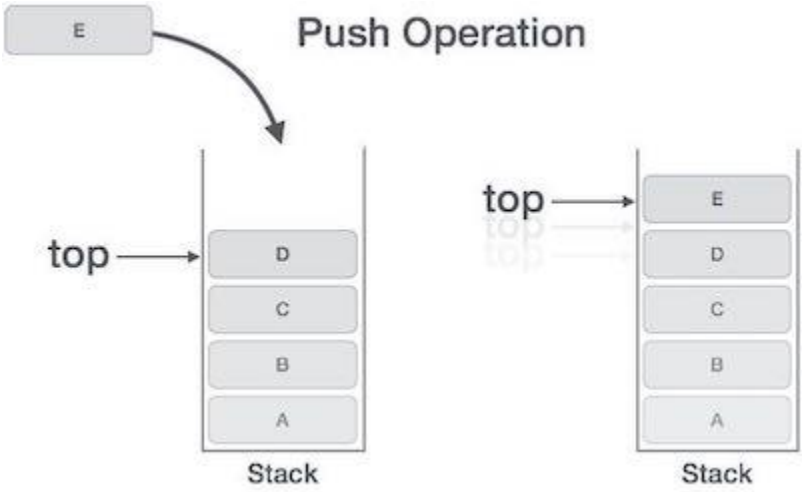
Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as index in array starts from 0. So we check if top is below zero or -1 to determine if stack is empty. Here's the code –

```
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 **PUSH**Operation. Push operation involves series of steps –

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



if linked-list is used to implement stack, then in step 3, we need to allocate space dynamically.

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 below code –

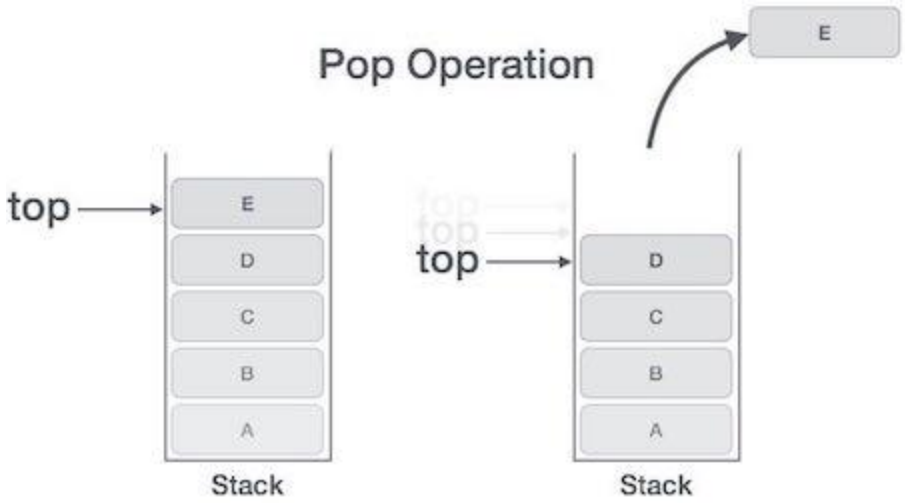
```
void push(int data) {
   if(!isFull()) {
      top = top + 1;
      stack[top] = data;
   }else {
      printf("Could not insert data, Stack is full.\n");
   }
}
```

Pop Operation

Accessing the content while removing it from stack, is known as pop operation. In array implementation of pop() operation, data element is not actually removed, instead **top** is decremented to a lower position in stack to point to 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** – Check if stack is empty.
- **Step 2** – If stack is empty, produce error and exit.
- **Step 3** – If stack is not empty, access the data element at which **top**is pointing.
- **Step 4** – Decrease the value of top by 1.
- **Step 5** – return 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 shown below –

```
int pop(int data) {
   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   }else {
```

```
    printf("Could not retrieve data, Stack is empty.\n");
  }
}
```

Demonstrate the implementation of a simple stack using arrays

```
#include <iostream>
using namespace std;
const int MAX_SIZE = 100;
class StackOverFlowException
{
   public:
      StackOverFlowException()
      {
         cout << "Stack overflow" << endl;
      }
};
class StackUnderFlowException
{
   public:
      StackUnderFlowException()
      {
         cout << "Stack underflow" << endl;
      }
};
class ArrayStack
{
   private:
      int data[MAX_SIZE];
      int top;
  public:
     ArrayStack()
     {
```

```
        top = -1;
    }
    void Push(int element)
    {
        if ( top >= MAX_SIZE )
            {
                throw new StackOverFlowException();
            }
            data[++top] = element;
    }
    int Pop()
    {
        if ( top == -1 )
            {
                throw new StackUnderFlowException();
            }
            return data[top--];
    }
    int Top()
    {
        return data[top];
    }
    int Size()
    {
        return top + 1;
    }
    bool isEmpty()
    {
        return ( top == -1 ) ? true : false;
    }
};
int main()
```

```
{
   ArrayStack s;
   try {
      if ( s.isEmpty() )
         {
         cout << "Stack is empty" << endl;
         }
      // Push elements
      s.Push(100);
      s.Push(200);
      // Size of stack
      cout << "Size of stack = " << s.Size() << endl;
      // Top element
      cout << s.Top() << endl;
      // Pop element
      cout << s.Pop() << endl;
      // Pop element
      cout << s.Pop() << endl;
      // Pop element
      cout << s.Pop() << endl;
   }
   catch (...) {
      cout << "Some exception occured" << endl;
   }
}
```

OUTPUT:-

Stack is empty

Size of stack = 2

200

200

100

Stack underflow

Some exception occured

Demonstrate the implementation of a simple stack using linked lists

```
#include <iostream>
using namespace std;
class StackUnderFlowException
{
   public:
      StackUnderFlowException()
      {
         cout << "Stack underflow" << endl;
      }
};
struct Node
{
   int data;
   Node* link;
};
class ListStack
{
   private:
      Node* top;
      int count;
  public:
     ListStack()
     {
        top = NULL;
        count = 0;
   }
   void Push(int element)
   {
```

```
        Node* temp = new Node();
        temp->data = element;
        temp->link = top;
        top = temp;
        count++;
    }
    int Pop()
    {
        if ( top == NULL )
            {
                throw new StackUnderFlowException();
            }
            int ret = top->data;
            Node* temp = top->link;
            delete top;
            top = temp;
            count--;
            return ret;
    }
    int Top()
    {
        return top->data;
    }
    int Size()
    {
        return count;
    }
    bool isEmpty()
    {
        return ( top == NULL ) ? true : false;
    }
};
```

```
int main()
{
   ListStack s;
   try {
      if ( s.isEmpty() )
         {
         cout << "Stack is empty" << endl;
         }
      // Push elements
      s.Push(100);
      s.Push(200);
      // Size of stack
      cout << "Size of stack = " << s.Size() << endl;
      // Top element
      cout << s.Top() << endl;
      // Pop element
      cout << s.Pop() << endl;
     // Pop element
      cout << s.Pop() << endl;
     // Pop element
      cout << s.Pop() << endl;
   }
   catch (...) {
      cout << "Some exception occured" << endl;
   }
}
```

OUTPUT:-

```
Stack is empty

Size of stack = 2

200

200
```

100

Stack underflow

Some exception occured

Queue

Queue is also an abstract data type or a linear data structure, in which the first element is inserted from one end called **REAR**(also called tail), and the deletion of existing element takes place from the other end called as **FRONT**(also called head). This makes queue as FIFO data structure, which means that element inserted first will also be removed first.

Queue is an abstract data structure, somewhat similar to stack. In contrast to stack, queue is opened at both end. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

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

Queue Representation

As we now understand that in queue, we access both ends for different reasons, a diagram given below tries to explain queue representation as data structure –

Same as stack, queue can also be implemented using Array, Linked-list, Pointer and Structures. For the sake of simplicity we shall implement queue using one-dimensional array.

Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it and then completing erasing it from memory. Here we shall try to understand 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 above mentioned queue operation efficient. These are −

- **peek()** − get the element at front of the queue without removing it.
- **isfull()** − checks if queue is full.
- **isempty()** − checks if queue is empty.

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

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

peek()

Like stacks, this function helps to see the data at the **front** of the queue. Algorithm of peek() function −

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 queue is full. In case we maintain 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 value of **front** is less than MIN or 0, it tells that queue is not yet initialized, hence empty.

Here's the C programming code −

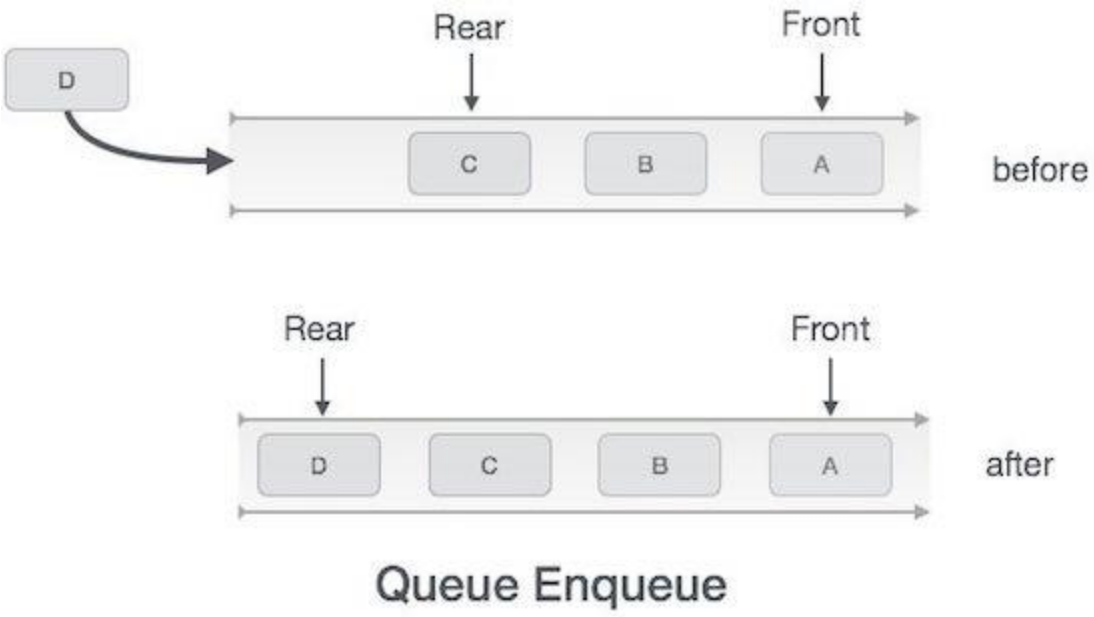
```
bool isempty() {
  if(front < 0 || front > rear)
    return true;
  else
    return false;
}
```

Enqueue Operation

As queue maintains two data pointers, **front** and **rear**, its operations are comparatively more difficult to implement than stack.

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

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



Queue Enqueue

Sometimes, we also check that if 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;

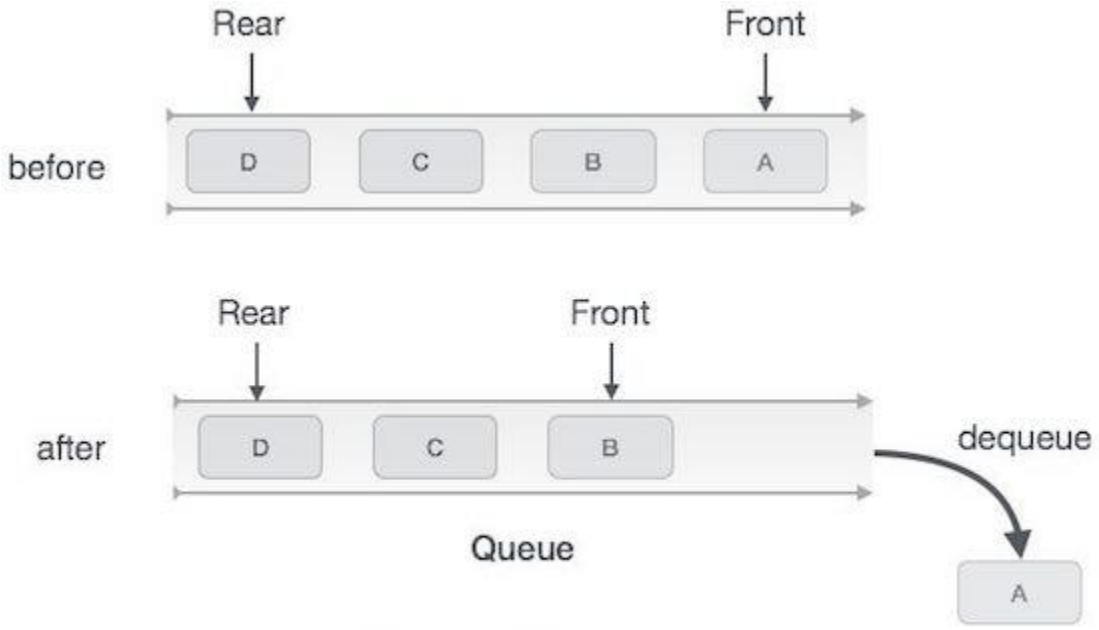
return 1;

end procedure

Dequeue Operation

Accessing data from 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 queue is empty.
- **Step 2** – If queue is empty, produce underflow error and exit.
- **Step 3** – If queue is not empty, access data where **front** is pointing.
- **Step 3** – Increment **front** pointer to point next available data element.
- **Step 5** – return success.



Queue Dequeue

Algorithm for dequeue operation –

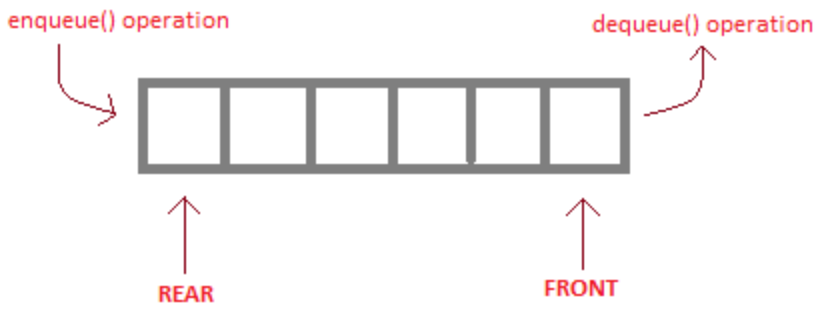
procedure dequeue

```
   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;
}
```

The process to add an element into queue is called **Enqueue** and the process of removal of an element from queue is called **Dequeue**.



enqueue( ) is the operation for adding an element into Queue.
dequeue( ) is the operation for removing an element from Queue .

QUEUE DATA STRUCTURE

Basic features of Queue

1. Like Stack, Queue is also an ordered list of elements of similar data types.
2. Queue is a FIFO(First in First Out) structure.
3. Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
4. **peek( )** function is oftenly used to return the value of first element without dequeuing it.

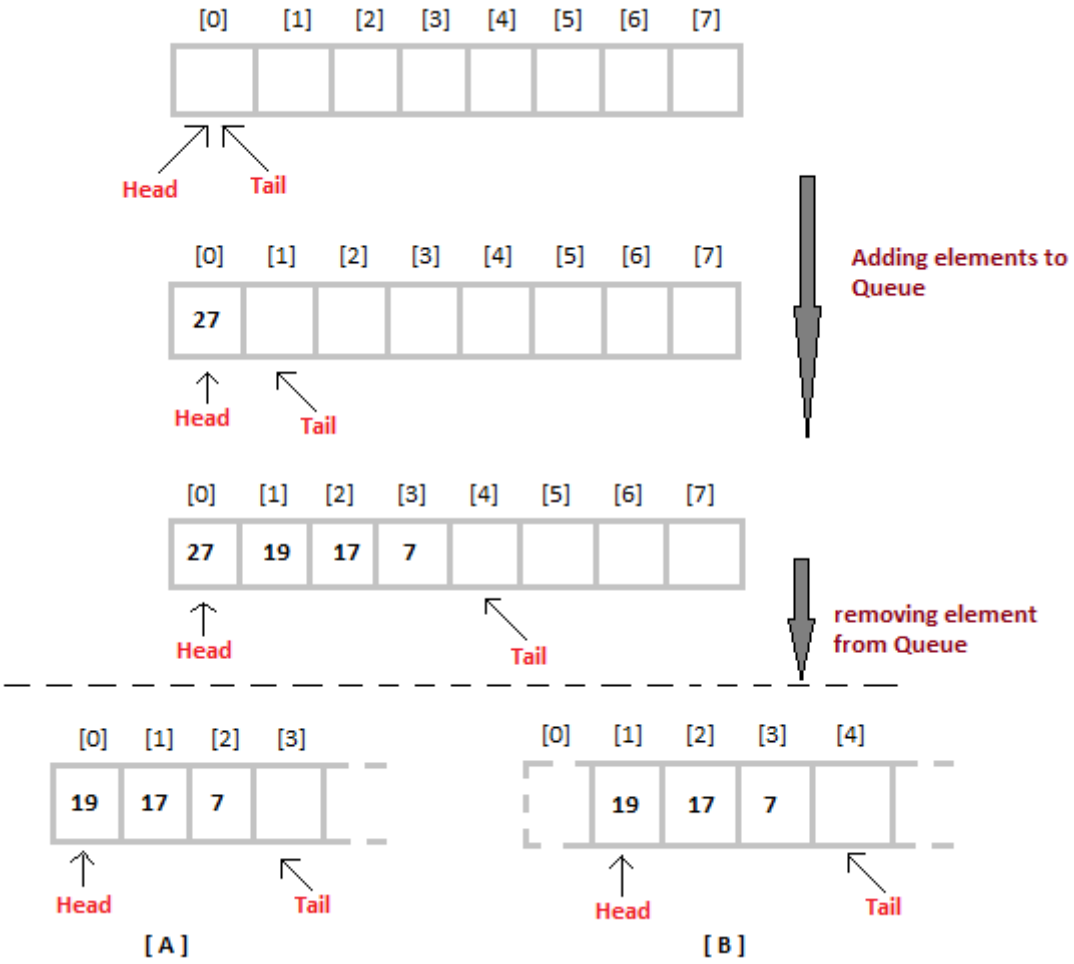
Applications of Queue

Queue, as the name suggests is used whenever we need to have any group of objects in an order in which the first one coming in, also gets out first while the others wait for there turn, like in the following scenarios :

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.

Implementation of Queue

Queue can be implemented using an Array, Stack or Linked List. The easiest way of implementing a queue is by using an Array. Initially the **head**(FRONT) and the **tail**(REAR) of the queue points at the first index of the array (starting the index of array from 0). As we add elements to the queue, the tail keeps on moving ahead, always pointing to the position where the next element will be inserted, while the head remains at the first index.



When we remove element from Queue, we can follow two possible approaches (mentioned [A] and [B] in above diagram). In [A] approach, we remove the element at **head** position, and then one by one move all the other elements on position forward. In approach [B] we remove the element from **head** position and then move **head** to the next position.

In approach [A] there is an overhead of shifting the elements one position forward every time we remove the first element. In approach [B] there is no such overhead, but whener we move head one position ahead, after removal of first element, the size on Queue is reduced by one space each time.

/* Below program is wtitten in C++ language */

```
#define SIZE 100
class Queue
{
  int a[100];
  int rear;     //same as tail
  int front;    //same as head
  public:
  Queue()
  {
    rear = front = -1;
  }
  void enqueue(int x);     //declaring enqueue, dequeue and display functions
  int dequeue();
  void display();
}
void Queue :: enqueue(int x)
{
  if( rear = SIZE-1)
  {
    cout << "Queue is full";
  }
  else
  {
    a[++rear] = x;
  }
}
int queue :: dequeue()
```

```
{
  return a[++front];     //following approach [B], explained above
}
void queue :: display()
{
  int i;
  for( i = front; i <= rear; i++)
  {
    cout << a[i];
  }
}
```

To implement approach [A], you simply need to change the dequeue method, and include a for loop which will shift all the remaining elements one position.

```
return a[0];      //returning first element
for (i = 0; i < tail-1; i++)      //shifting all other elements
{
  a[i]= a[i+1];
  tail--;
}
```

Analysis of Queue

- Enqueue : **O(1)**
- Dequeue : **O(1)**
- Size : **O(1)**

Queue Data Structure using Stack

A Queue is defined by its property of FIFO, which means First in First Out, i.e the element which is added first is taken out first. Hence we can implement a Queue using Stack for storage instead of array.

For performing **enqueue** we require only one stack as we can directly **push** data into stack, but to perform**dequeue** we will require two Stacks, because we need to follow queue's FIFO property and if we directly **pop**any data element out of Stack, it will follow LIFO approach(Last in First Out).

Implementation of Queue using Stacks

In all we will require two Stacks, we will call them InStack and OutStack.

```
class Queue {
  public:
  Stack S1, S2;
  //defining methods

  void enqueue(int x);

  int dequeue();
}
```

We know that, Stack is a data structure, in which data can be added using **push()** method and data can be deleted using **pop()** method.

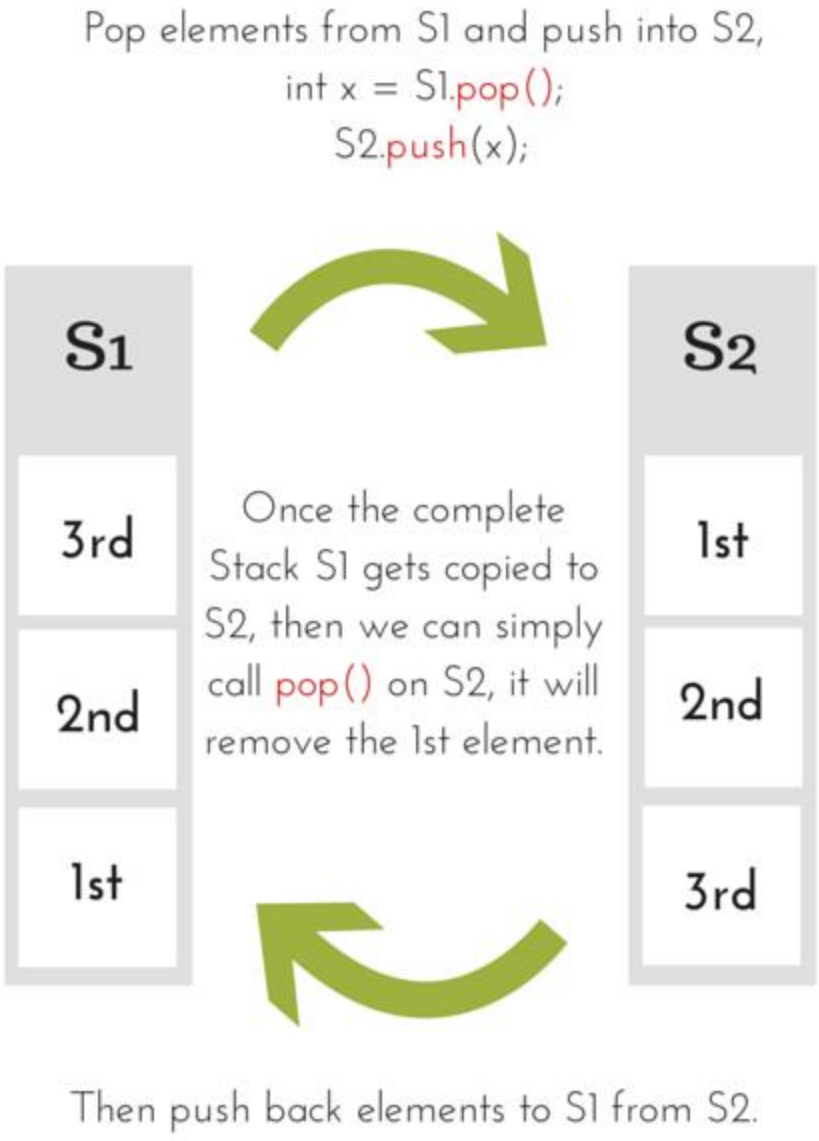
Adding Data to Queue

As our Queue has Stack for data storage in place of arrays, hence we will be adding data to Stack, which can be done using the push() method, hence :

```
void Queue :: enqueue(int x) {
  S1.push(x);
}
```

Removing Data from Queue

When we say remove data from Queue, it always means taking out the First element first and so on, as we have to follow the FIFO approach. But if we simply perform S1.pop() in our dequeue method, then it will remove the Last element first. So what to do now?




```
int Queue :: dequeue() {
  while(S1.isEmpty()) {
    x = S1.pop();
    S2.push();
  }
```

```
  //removing the element
  x = S2.pop();
  while(!S2.isEmpty()) {
    x = S2.pop();
    S1.push(x);
  }
  return x;
}
```

Demonstrate the implementation of a simple queue using arrays

```
#include <iostream>
#include <cstdlib>
using namespace std;

const int MAX_SIZE = 100;

class QueueOverFlowException
{
public:
   QueueOverFlowException()
   {
      cout << "Queue overflow" << endl;
   }
};
class QueueEmptyException
{
public:
   QueueEmptyException()
   {
      cout << "Queue empty" << endl;
```

```
  }
};
class ArrayQueue
{
private:
  int data[MAX_SIZE];
  int front;
  int rear;
public:
  ArrayQueue()
  {
     front = -1;
     rear = -1;
  }
  void Enqueue(int element)
  {
     // Don't allow the queue to grow more
     // than MAX_SIZE - 1
     if ( Size() == MAX_SIZE - 1 )
        throw new QueueOverFlowException();

     data[rear] = element;

     // MOD is used so that rear indicator
     // can wrap around
     rear = ++rear % MAX_SIZE;
  }
  int Dequeue()
  {
     if ( isEmpty() )
        throw new QueueEmptyException();
```

```
        int ret = data[front];

        // MOD is used so that front indicator
        // can wrap around
        front = ++front % MAX_SIZE;

        return ret;
    }
    int Front()
    {
        if ( isEmpty() )
            throw new QueueEmptyException();

        return data[front];
    }
    int Size()
    {
        return abs(rear - front);
    }
    bool isEmpty()
    {
        return ( front == rear ) ? true : false;
    }
};

int main()
{
    ArrayQueue q;
    try {
        if ( q.isEmpty() )
        {
            cout << "Queue is empty" << endl;
```

```
    }
    // Enqueue elements
    q.Enqueue(100);
    q.Enqueue(200);
    q.Enqueue(300);
    // Size of queue
    cout << "Size of queue = " << q.Size() << endl;
    // Front element
    cout << q.Front() << endl;
    // Dequeue elements
    cout << q.Dequeue() << endl;
    cout << q.Dequeue() << endl;
    cout << q.Dequeue() << endl;
  }
  catch (...) {
    cout << "Some exception occured" << endl;
  }
}
```

OUTPUT:-

```
Queue is empty
Size of queue = 3
100
100
200
300
```

Demonstrate the implementation of a simple queue using linked lists

```
#include <iostream>
using namespace std;
class QueueEmptyException
{
public:
```

```
  QueueEmptyException()
  {
     cout << "Queue empty" << endl;
  }
};
class Node
{
public:
  int data;
  Node* next;
};
class ListQueue
{
private:
  Node* front;
  Node* rear;
  int count;
public:
  ListQueue()
  {
     front = NULL;
     rear = NULL;
     count = 0;
  }
  void Enqueue(int element)
  {
     // Create a new node
     Node* tmp = new Node();
     tmp->data = element;
     tmp->next = NULL;

     if ( isEmpty() ) {
```

```
        // Add the first element
        front = rear = tmp;
    }
    else {
        // Append to the list
        rear->next = tmp;
        rear = tmp;
    }

    count++;
}
int Dequeue()
{
    if ( isEmpty() )
        throw new QueueEmptyException();
    int ret = front->data;
    Node* tmp = front;
    // Move the front pointer to next node
    front = front->next;
    count--;
    delete tmp;
    return ret;
}
int Front()
{
    if ( isEmpty() )
        throw new QueueEmptyException();

    return front->data;
}
int Size()
{
```

```
        return count;
    }
    bool isEmpty()
    {
        return count == 0 ? true : false;
    }
};

int main()
{
    ListQueue q;
    try {
        if ( q.isEmpty() )
        {
            cout << "Queue is empty" << endl;
        }
        // Enqueue elements
        q.Enqueue(100);
        q.Enqueue(200);
        q.Enqueue(300);
        // Size of queue
        cout << "Size of queue = " << q.Size() << endl;
        // Front element
        cout << q.Front() << endl;
        // Dequeue elements
        cout << q.Dequeue() << endl;
        cout << q.Dequeue() << endl;
        cout << q.Dequeue() << endl;
    }
    catch (...) {
        cout << "Some exception occured" << endl;
    }
```

```
}
OUTPUT:-
Queue is empty
Size of queue = 3
100
100
200
300
```

Demonstrate the queue library usage in standard template library (STL)

```
#include <iostream>
#include <queue>
using namespace std;
int main()
{
  queue<int> q;
  q.push(100);
  q.push(200);
  q.push(300);
  q.push(400);
  cout << "Size of the queue: " << q.size() << endl;
  cout << q.front() << endl;
  q.pop();
  cout << q.front() << endl;
  q.pop();
  cout << q.front() << endl;
  q.pop();
```

```
   cout << q.front() << endl;
   q.pop();
   if ( q.empty() ) {
   cout << "Queue is empty" << endl;
   }
}
```

OUTPUT:-

Size of the queue: 4

100

200

300

400

Queue is empty

Difference Between stack and Queue

STACK: Stack is defined as a list of element in which we can insert or delete elements only at the top of the stack

Stack is used to pass parameters between function. On a call to a function, the parameters and local variables are stored on a stack.

A stack is a collection of elements, which can be stored and retrieved one at a time. Elements are retrieved in reverse order of their time of storage, i.e. the latest element stored is the next element to be retrieved. A stack is sometimes referred to as a Last-In-First-Out (LIFO) or First-In-Last-Out (FILO) structure. Elements previously stored cannot be retrieved until the latest element (usually referred to as the 'top' element) has been retrieved.

QUEUE:

Queue is a collection of the same type of element. It is a linear list in which insertions can take place at one end of the list,called rear of the list, and deletions can take place only at other end, called the front of the list

A queue is a collection of elements, which can be stored and retrieved one at a time. Elements are retrieved in order of their time of storage, i.e. the first element stored is the next element to be retrieved. A queue is sometimes referred to as a First-In-First-Out (FIFO) or Last-In-Last-Out (LILO) structure. Elements subsequently stored cannot be retrieved until the first element (usually referred to as the 'front' element) has been retrieved.

Expression Parsing

The way to write arithmetic expression is known as notation. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of expression. These notations are −

- **Infix Notation**
- **Prefix (Polish) Notation**
- **Postfix (Reverse-Polish) Notation**

These notations are named as how they use operator in expression.

Infix Notation

We write expression in infix notation, e.g. a-b+c, where operators are used in-between operands. It is easy for us humans to read, write and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

Prefix Notation

In this notation, operator is prefixed to operands, i.e. operator is written ahead of operands. For example +ab. This is equivalent to its infix notation a+b. Prefix notation is also known as Polish Notation.

Postfix Notation

This notation style is known as Reversed Polish Notation. In this notation style, operator is postfixed to the operands i.e., operator is written after the operands. For example ab+. This is equivalent to its infix notation a+b.

The below table briefly tries to show difference in all three notations –

S.n.	Infix Notation	Prefix Notation	Postfix Notation
1	a + b	+ a b	a b +
2	(a + b) * c	* + a b c	a b + c *
3	a * (b + c)	* a + b c	a b c + *
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	(a + b) * (c + d)	* + a b + c d	a b + c d + *
6	((a + b) * c) - d	- * + a b c d	a b + c * d -

Parsing Expressions

As we have discussed, it is not very efficient way to design an algorithm or program to parse infix notations. Instead, these infix notations are first converted into either postfix or prefix notations and then computated.

To parse any arithmetic expression, we need to take care of operator precedence and associativity also.

Precedence

When an operand is in between two different operator, which operator will take the operand first, is decided by the precedence of an operator over others.

For example –

As multiplication operation has precedence over addition, b * c will be evaluated firs. A table of operator precedence is provided later.

Associativity

Associativity describes the rule where operators with same precedence appear in an expression. For example, in expression a+b−c, both + and − has same precedence, then which part of expression will be evaluated first, is determined by associativity of those operators. Here, both + and − are left associative, so the expression will be evaluated as (a+b)−c.

Precedence and associativity, determines the order of evaluation of an expression.

An operator precedence and associativity table is given below (highest to lowest) −

S.n.	Operator	Precedence	Associativity
1	Esponentiation ^	Highest	Right Associative
2	Multiplication (*) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (−)	Lowest	Left Associative

The above table shows the default behavior of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis.

For example

In a+b*c, the expression part b*c will be evaluated first, as multiplication as precedence over addition. We here use parenthesis to make a+b be evaluated first, like (a+b)*c.

Postfix Evaluation Algorithm

We shall now look at the algorithm on how to evaluate postfix notation −

Step 1 – scan the expression from left to right

Step 2 – if it is an operand push it to stack

Step 3 – if it is an operator pull operand from stack and perform operation

Step 4 – store the output of step 3, back to stack

Step 5 – scan the expression until all operands are consumed

Step 6 – pop the stack and perform operation

Infix, Postfix and Prefix

Infix, Postfix and Prefix notations are three different but equivalent ways of writing expressions. It is easiest to demonstrate the differences by looking at examples of operators that take two operands.

Infix notation: X + Y

Operators are written in-between their operands. This is the usual way we write expressions. An expression such as A * (B + C) / D is usually taken to mean something like: "First add B and C together, then multiply the result by A, then divide by D to give the final answer."

Infix notation needs extra information to make the order of evaluation of the operators clear: rules built into the language about operator precedence and associativity, and brackets ( ) to allow users to override these rules. For example, the usual rules for associativity say that we perform operations from left to right, so the multiplication by A is assumed to come before the division by D. Similarly, the usual rules for precedence say that we perform multiplication and division before we perform addition and subtraction.

Postfix notation (also known as "Reverse Polish notation"): X Y +

Operators are written after their operands. The infix expression given above is equivalent to A B C + * D /

The order of evaluation of operators is always left-to-right, and brackets cannot be used to change this order. Because the "+" is to the left of the "*" in the example above, the addition must be performed before the multiplication.

Operators act on values immediately to the left of them. For example, the "+" above uses the "B" and "C". We can add (totally unnecessary) brackets to make this explicit: ((A (B C +) *) D /)

Thus, the "*" uses the two values immediately preceding: "A", and the result of the addition. Similarly, the "/" uses the result of the multiplication and the "D".

Prefix notation (also known as "Polish notation"): + X Y

Operators are written before their operands. The expressions given above are equivalent to / * A + B C D

As for Postfix, operators are evaluated left-to-right and brackets are superfluous. Operators act on the two nearest values on the right. I have again added (totally unnecessary) brackets to make this clear: (/ (* A (+ B C)) D)

Although Prefix "operators are evaluated left-to-right", they use values to their right, and if these values themselves involve computations then this changes the order that the operators have to be evaluated in. In the example above, although the division is the first operator on the left, it acts on the result of the multiplication, and so the multiplication has to happen before the division (and similarly the addition has to happen before the multiplication).

Because Postfix operators use values to their left, any values involving computations will already have been calculated as we go left-to-right, and so the order of evaluation of the operators is not disrupted in the same way as in Prefix expressions.

In all three versions, the operands occur in the same order, and just the operators have to be moved to keep the meaning correct. (This is particularly important for asymmetric operators like subtraction and division: A - B does not mean the same as B - A; the former is equivalent to A B - or - A B, the latter to B A - or - B A).

Examples:

Infix	Postfix	Prefix	Notes
A * B + C / D	A B * C D / +	+ * A B / C D	multiply A and B, divide C by D, add the results
A * (B + C) / D	A B C + * D /	/ * A + B C D	add B and C, multiply by A, divide by D
A * (B + C / D)	A B C D / + *	* A + B / C D	divide C by D, add B, multiply by A

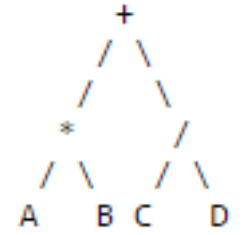
Converting between these notations

The most straightforward method is to start by inserting all the implicit brackets that show the order of evaluation e.g.:

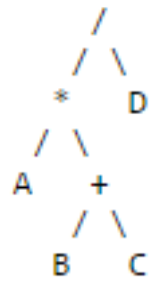
Infix	Postfix	Prefix
((A * B) + (C / D))	((A B *) (C D /) +)	(+ (* A B) (/ C D))
((A * (B + C)) / D)	((A (B C +) *) D /)	(/ (* A (+ B C)) D)
(A * (B + (C / D)))	(A (B (C D /) +) *)	(* A (+ B (/ C D)))

You can convert directly between these bracketed forms simply by moving the operator within the brackets e.g. (X + Y) or (X Y +) or (+ X Y). Repeat this for all the operators in an expression, and finally remove any superfluous brackets.

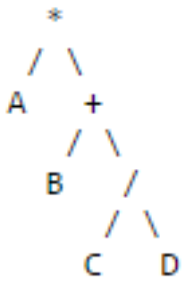
You can use a similar trick to convert to and from parse trees - each bracketed triplet of an operator and its two operands (or sub-expressions) corresponds to a node of the tree. The corresponding parse trees are:



((A*B)+(C/D))



((A*(B+C))/D)



(A*(B+(C/D)))