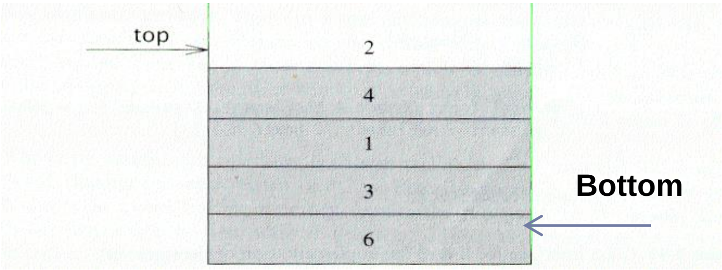
# *chapter 4- Stack and Queue*

College of computing Science,
Department of Software Engineering, DBU
Instructor:  Bekele M.(MSc in Software Eng.)
Address: Email: bekele18@gmail.com
December 2023 E.C

# The Stack ADT

- A stack is a list with the restriction
  - The restriction is:
    - insertions and deletions can only be performed at only one end called the *top* of the stack.



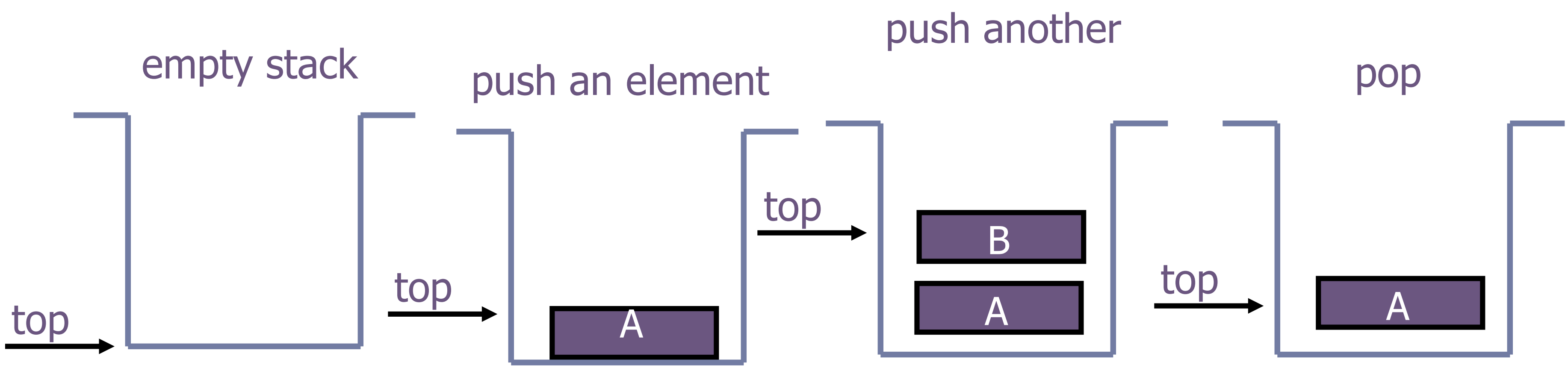

  - The other end is of the stack is called bottom
- Fundamental operations:
  - Push: Equivalent to an insert
  - Pop: Deletes the most recently inserted element
  - Top: Examines the most recently inserted element

# Stack ADT

- Other utility operations stack might support are:
  - Isempty: determines whether the stack has anything in it
  - Isfull returns true if the stack is full otherwise returns false.
  - Size: returns the number of items in the stack
- A common model of a stack is a plate or coin stacker. Plates are "pushed" onto to the top and "popped" off the top. Hence
  - Stacks are less flexible
  - Easy to implement
  - Stacks are known as LIFO (Last In, First Out) lists.
    - Because, the last element inserted will be the first to be retrieved

# Push and Pop

- Primary operations of stack are : Push and Pop
- Push
  - Add an element to the top of the stack
- Pop
  - Remove the element at the top of the stack
- Top – points the top element of the stack



# Implementation of Stacks

- Any list implementation could be used to implement a stack
  - Arrays (static: the size of stack is given initially)
  - Linked lists (dynamic: never become full)
- We will explore implementations based on array and linked list
- Let's see how to use an array to implement a stack first

# Array Implementation

- Need to declare an array size ahead of time
- Associated with each stack is topOfStack
  - for an empty stack, set topOfStack to -1
- Push
  - (1) Increment topOfStack by 1.
  - (2) Set Stack[topOfStack] = X //x is the data we add to the stack
- Pop
  - (1) Set return value to Stack[TopOfStack] // value=stack[topofstack]
  - (2) Decrement topOfStack by 1
- These operations are performed in very fast constant time

# `Stack` attributes and Operations

- Attributes of `Stack`
  - `maxTop`: the max size of stack
  - `top`: the index of the top element of stack
  - `values`: element/point to an array which stores elements of stack
- Operations of `Stack`
  - `IsEmpty`: return true if stack is empty, return false otherwise
  - `IsFull`: return true if stack is full, return false otherwise
  - `Top`: return the element at the top of stack
  - `Push`: add an element to the top of stack
  - `Pop`: delete the element at the top of stack
  - `DisplayStack`: print all the data in the stack

# Create Stack

- Initialize the `Stack`
  - Allocate a stack array of `size`. Example, `size= 10`.
  - Initially `top` is set to -1.
    - It means the stack is empty.
  - When the stack is full, `top` will have value `size – 1`.

```
int Stack[size]
maxTop = size - 1;
int top = -1;
```

# Push Stack

- `void Push(const double x);`
  - Increment top by 1
  - If the stack is full,
    - print the error information.
  - Else
    - Push an element onto the stack

```
push(int item)
{
 top = top+ 1;
 if(top<= maxTop)
 //Put the new element in the stack
  stack[top] = item;
 else
  cout<<"Stack Overflow";
}
```

# Pop Stack

- `int Pop()`
  - If the stack is empty,
    - print the error information. (In this case, the return value is useless.)
  - Pop and return the element at the top of the stack
  - decrement `top`

```
int pop()
{
int del_val= 0;
if(top= = -1)
   cout<<"Stack underflow";
else {
   del_val= stack[top];//Store the top most value in del_val
   stack[top] = NULL; //Delete the top most value
   top = top -1;
}
return(del_val);
}
```

# Stack Top

- `double Top()`
  - If the stack is empty,
    - print the error information.
  - Else
    - Return the top element of the stack
- Note:- Unlike `Pop`, this function does not remove the top element

```
double Top() {
      if (top==-1) {
            cout << "Error: the stack is empty." << endl;
            return -1;
      }
      else
            return stack[top];
}
```

# Printing all the elements

- `void DisplayStack()`
  - Print all the elements


```
void DisplayStack() {
      cout << "top -->";
      for (int i = top; i >= 0; i--)
            cout << "\t|\t" << stack[i] << "\t|" << endl;
      cout << "\t|---------------|" << endl;
}
```

# Using Stack

```
int main(void) {
      Push(5.0);
      Push(6.5);
      Push(-3.0);
      Push(-8.0);
      DisplayStack();
      cout << "Top: " <<Top() << endl;

      Pop();
      cout << "Top: " <<Top() << endl;
      while (top!=-1)
            Pop();
      DisplayStack();
      return 0;
}
```

**result**

```
top --> |        -8       |
        |        -3       |
        |        6.5      |
        |        5        |
        |-----------------|
Top: -8
Top: -3
top --> |-----------------|
```

# Linked-List implementation of stack

- We perform a ***push*** by inserting at the ***front of the list***.
- We perform a ***pop*** by deleting the element at the ***front of the list***
- A ***top*** operation merely examines the element at the ***front of the list***, returning its value.

**Create the stack**

```
struct Node{
int item;
node *next;
};
Node *topOfStack= NULL;
```

# Linked List push Stacks

- Algorithm
  - Step-1:Create the new node
  - Step-2: Check whether the top of Stack is empty or not
    - if empty, go to step-3
    - Otherwise, go to step-4
  - Step-3:Make your "topOfstack" pointer point to it and quit.
  - Step-4:Assign the topOfstackpointer to the newly attached element.

# Push operation

```
push(int item)
{
    Node newnode= new Node; // create new node (step 1)
    newnode-> item = item; // add data to the node (step 1)

    if( topOfStack = = NULL){ //check if stack is empty (step 2 )
                topOfStack = newnode;        //make topOfStack point to the new
(step 3)                  nodenewnode-> next = NULL;
    }
    else {      // if there were nodes in the stack, rearrange the pointers (step 4)
                newnode-> next = topOfStack;
                topOfStack = newnode;
    }
}
```

# The POP Operation

- Algorithm:
  - Step-1:If the Stack is empty then give an alert message "Stack Underflow" and quit; else proceed
  - Step-2:Make "target" point to topOfstack
  - Step-3: Store the value of the node at topOfStack
  - Step-4: Make topOfstack point topOfStack's next
  - Step-5: Free the target node;

# Pop operation

```
int pop( ) {
int pop_val= 0;
if(topOfStack = = NULL)
    cout<<"Stack Underflow";
else {
    Node *target = topOfStack // Make the first node a target
    pop_val= topOfStack-> item; // return data from the target node to be deleted
    topOfStack= topOfStack->next;  //make the top pointer point to the next node
    delete target;
}
return(pop_val);
}
```

# Algorithm Analysis

- Array Implementation and Linked list implementation
  - push O(?)
  - pop O(?)
  - isEmpty O(?)
  - isFull O(?)
  - Top O(?)
  - displayStack O(?)

# Application of stack Data Structure

- Compiler's syntax check for matching symbols is implemented by using stack.
  - Balancing symbols ( ), [ ], { }
- Evaluating expression
- To reverse a word.
  - Push a given word to stack - letter by letter - and then pop letters from the stack.
- "undo" mechanism in text editors;
  - this operation is accomplished by keeping all text changes in a stack.
- Function calls
  - space for function return address, parameters and local variables is created internally using a stack.

# Balancing Symbols

- Compilers check your programs for syntax errors like missing symbols.
- Such programs use ***stack*** to check that every right brace, bracket, and parentheses must correspond to its left counterpart
  - e.g. [( )] is legal, but [( ] ) is illegal
- Algorithm

(1) Make an empty stack.

(2) Read characters until end of file

   i. If the character is an opening symbol, push it onto the stack

   ii. **If** it is a closing symbol, then

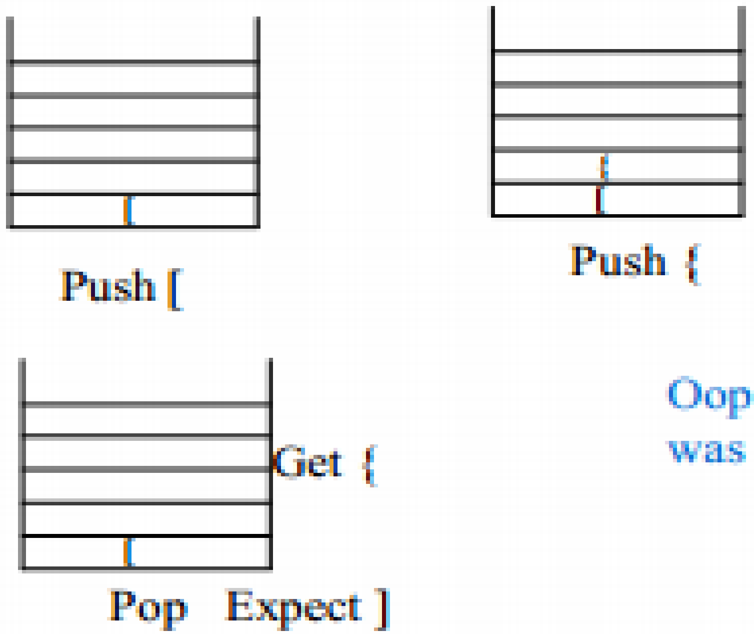
      1. if the stack is empty, report an error and stop
      2. **else**, pop the stack.
         1. If the symbol popped is not the corresponding opening symbol, then report an error

(3) At end of file, if the stack is not empty, report an error and stop

# Example

Check brace, bracket parentheses matching [a+b{1*2]9*1}+(2-1)

Push [, Push {, Pop, Pop, Push (, Pop

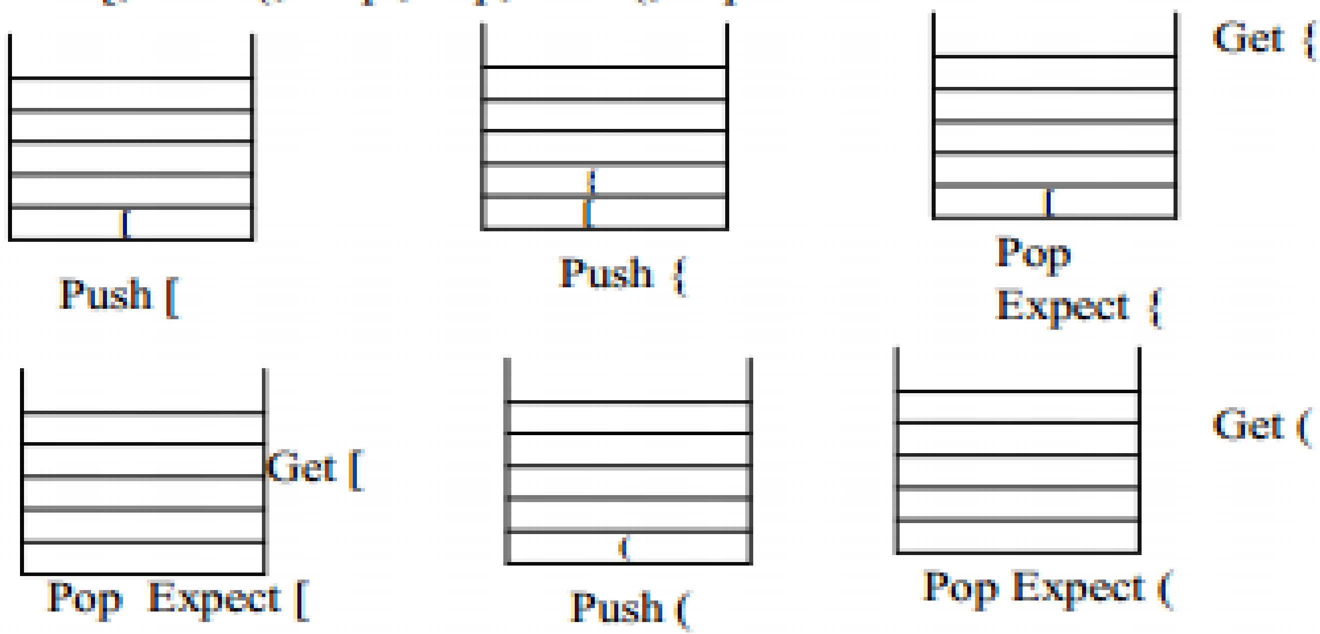



Oops! Something wrong,
was expecting [

# Example

Check brace, bracket parentheses matching [a+b{1*2}9*1]+(2-1)

Push [, Push {, Pop , Pop, Push (, Pop



# Expression evaluation

- There are three common notations to represent arithmetic expressions
  - **Infix:-**operators are between operands. Ex. A + B
  - **Prefix (polish notation):-** operators are before their operands.
    - Example. + A B
  - **Postfix (Reverse polish notation):-** operators are after their operands
    - Example : A B +
- Though infix notation is convenient for human beings, postfix notation is much cheaper and easy for machines
  - Therefore, computers change the infix to postfix notation first
  - Then, the post-fix expression is evaluated

# Algorithm for Infix to Postfix

1. Examine the next element in the input.
2. If it is operand, output it.
3. If it is opening parenthesis, push it on stack.
4. If it is an operator, then
   A. If stack is empty, push operator on stack.
   B. If the top of stack is opening parenthesis, push operator on stack
   C. If it has higher priority than the top of stack, push operator on stack.
   D. Else pop the operator from the stack and output it, repeat step A through D
5. If it is a closing parenthesis, pop operators from stack and output them until an opening parenthesis is encountered. pop and discard the opening parenthesis.
6. If there is more input go to step 1, otherwise go to step 7.
7. If there is no more input, pop the remaining operators to output.

# Examples

A * B + C

| Current symbol | Operator stack | Postfix expression |
| --- | --- | --- |
| A | | A |
| * | * | A |
| B | * | AB |
| + | + | AB* |
| C | + | AB*C |
| | | AB*C+ |

A + B * C

| Current symbol | Operator stack | Postfix expression |
| --- | --- | --- |
| A | | A |
| + | + | A |
| B | + | AB |
| * | + * | AB |
| C | + * | ABC |
| | | ABC*+ |

# Some more examples

| Infix | Postfix |
|---|---|
| A+B | A b+ |
| A+(B+ C) | ABC++ |
| (A+B)+ C | AB+C+ |
| A+BxC | ABCx+ |
| A+BxC | AB+Cx |

**convert 2*3/(2-1)+5*3 into Postfix form**

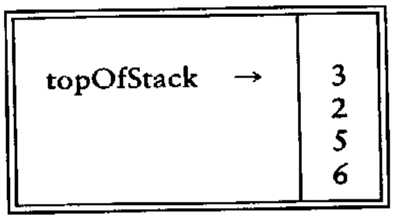
| Expression | Stack | Output |
|---|---|---|
| 2 | Empty | 2 |
| * | * | 2 |
| 3 | * | 23 |
| / | / | 23* |
| ( | /( | 23* |
| 2 | /( | 23*2 |
| - | /(- | 23*2 |
| 1 | /(- | 23*21 |
| ) | / | 23*21- |
| + | + | 23*21-/ |
| 5 | + | 23*21-/5 |
| * | +* | 23*21-/53 |
| 3 | +* | 23*21-/53 |
| | Empty | 23*21-/53*+ |

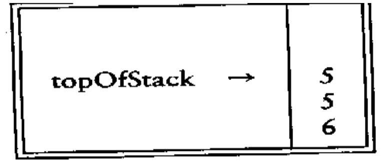
So, the Postfix Expression is 23*21-/53*+

# Postfix Expressions

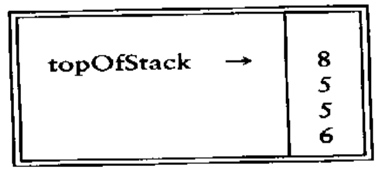
- To Calculate 4 * 5 + 6 * 7, we need to know the precedence rules
- But its Postfix (reverse Polish) equivalent doesn't require that
  - 4 5 * 6 7 * +
- Instead, use stack to evaluate postfix expressions as follows
  - When a number is seen, it is pushed onto the stack
  - When an operator is seen,
    - The operator is applied to the 2 numbers that are popped from the stack.
    - The result is pushed onto the stack
- Example
  - Evaluate 6 5 2 3 + 8 * + 3 + *
- The time to evaluate a postfix expression is O(N)
  - Processing each element in the input consists of stack operations and thus takes constant time

**Evaluate: 6 5 2 3 + 8 * + 3 + ***

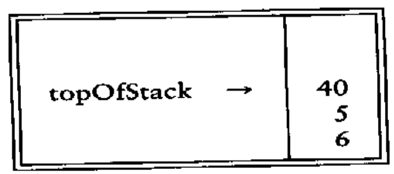


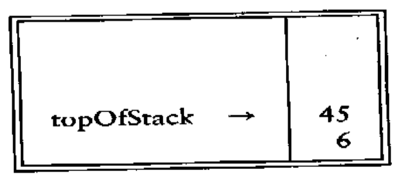

Next 8 is pushed.



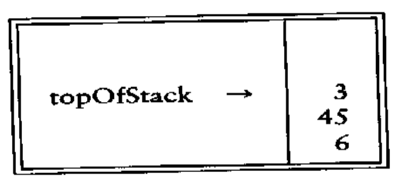

Now a ‘*’ is seen, so 8 and 5 are popped and 5 * 8 = 40 is pushed.



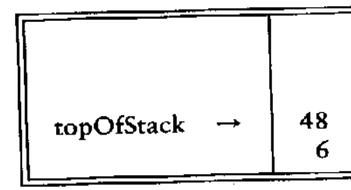

Next a ‘+’ is seen, so 40 and 5 are popped and 5 + 40 = 45 is pushe



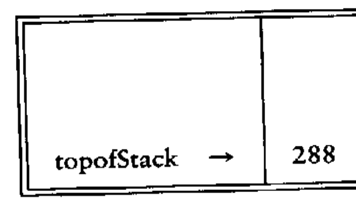

Now, 3 is pushed.




Next ‘+’ pops 3 and 45 and pushes 45 + 3 = 48.




and 48 and 6 are popped; the result, 6 * 4



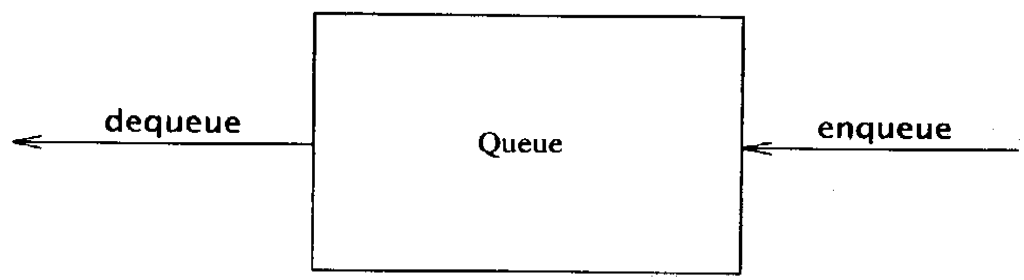
# *Queue*

# Queue ADT

- Like a stack, a *queue* is also a list.
  - However, with a queue, insertion is done at one end(rear of the queue), while deletion is performed at the other end (front of the queue).
- Accessing the elements of queues follows a First In, First Out (FIFO) order.
  - Like customers standing in a check-out line in a shop, the first customer in is the first customer served.

# The Queue ADT

- Basic operations:
  - enqueue: insert an element at the rear of the list
  - dequeue: delete the element at the front of the list
- Other operations:
  - `IsEmpty`: return true if queue is empty, return false otherwise
  - `IsFull`: return true if queue is full, return false otherwise
  - `DisplayQueue`: print all the data



# Implementation of Queue

- Just as stacks can be implemented as arrays or linked lists, so with queues.
- Dynamic queues have the same advantages over static queues as dynamic stacks have over static stacks

# Simple array implementation

- There are several different algorithms to implement Enqueue and Dequeue using arrays
  - Simple implementation
  - Naïve implementation
  - Circular array implementation
- The following shall be declared global
  - int FRONT =-1; //index of the front element
  - int REAR =-1; //index of the rear element
  - int QUEUESIZE=0; // number of elements in the array
  - int Max_Size =100; //Defines the maximum array size
  - int myQueue[Max_Size ]; the queue

# Simple array implementation of enqueue and dequeue

```
void enqueue(int x){
        if(Rear<MAX_SIZE-1)
        {
                REAR++;
                myQueue [REAR]=x;
                QUEUESIZE++;
                if(FRONT = = -1)
                        FRONT++;
        }
        else
                cout<<"Queue
Overflow";
        }
```

```
int dequeue(){
        int x;
         if(QUEUESIZE>0)   {
                x=myQueue [FRONT];
                FRONT++;
                 QUEUESIZE--;
          }
          else
            cout<<"Queue Underflow";

      return(x);
}
```

# Drawback of simple array implementation

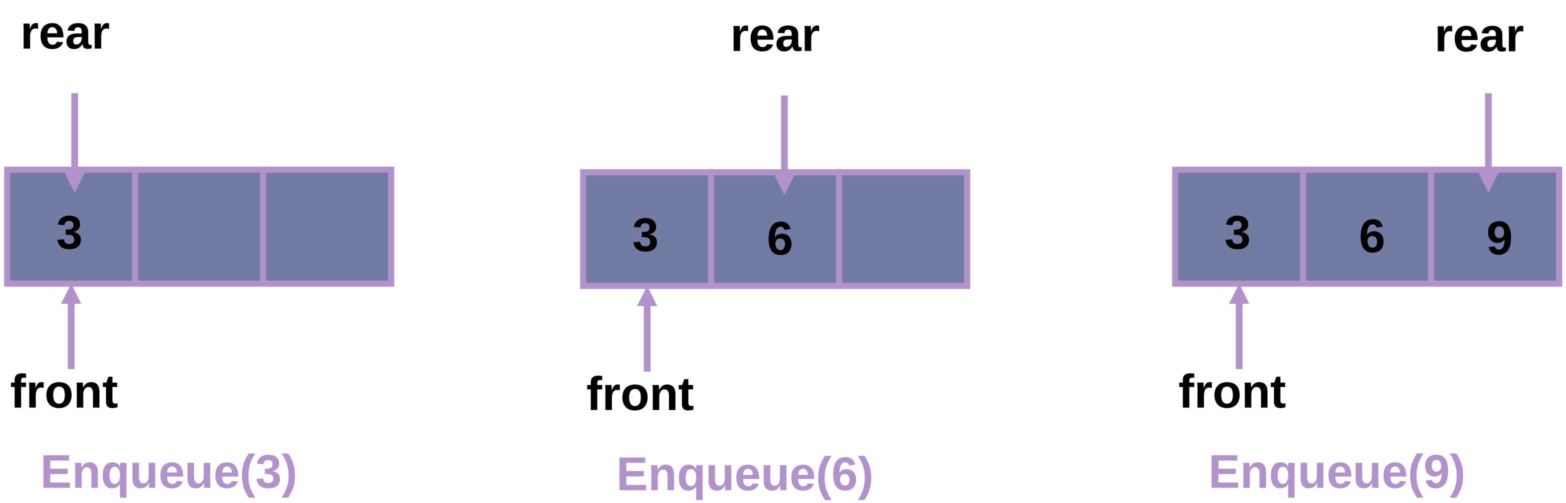
- Example: Consider a queue with MAX_SIZE = 4

| Operation | Simple array | | | | | | |
|---|---|---|---|---|---|---|---|
| | Content of the array | | | | Content of the Queue | QUEUE SIZE | Message |
| Enqueue(B) | B | | | | B | 1 | |
| Enqueue(C) | B | C | | | BC | 2 | |
| Dequeue() | | C | | | C | 1 | |
| Enqueue(G) | | C | G | | CG | 2 | |
| Enqueue (F) | | C | G | F | CGF | 3 | |
| Dequeue() | | | G | F | GF | 2 | |
| Enqueue(A) | | | G | F | GF | 2 | Overflow |
| Dequeue() | | | | F | F | 1 | |
| Enqueue(H) | | | | F | F | 1 | Overflow |
| Dequeue () | | | | | Empty | 0 | |
| Dequeue() | | | | | Empty | 0 | Underflow |

**A problem with simple arrays is we run out of space even if the queue never reaches the size of the array.**

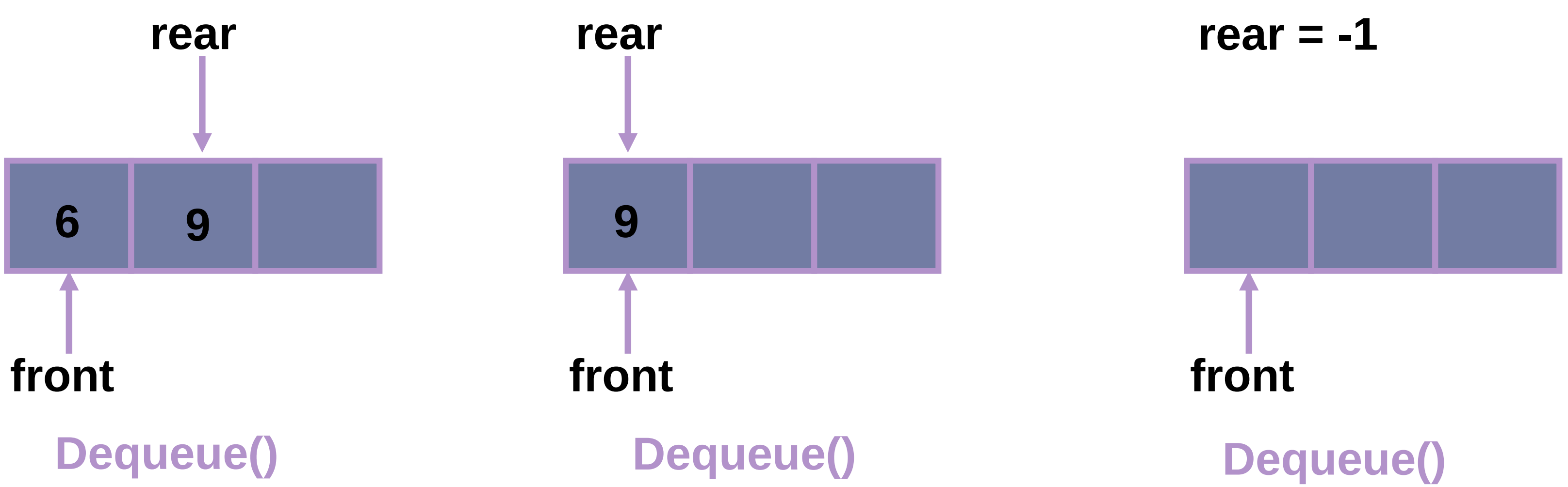
# Array Implementation of Queue

- Naïve way
  - In order to solve the space wastage, move the elements by a position after each dequeue
  - When enqueuing, the <u>front index</u> is always fixed and the <u>rear index</u> moves forward in the array.



# Array Implementation of Queue

- Naïve way
  - When enqueuing, the <u>front index</u> is always fixed and the <u>rear index</u> moves forward in the array.
  - When dequeuing, the element at the front the queue is removed. Move all the elements after it by one position. (Inefficient!!!)



# Array Implementation of Queue

- Better way
  - When an item is enqueued, make the <u>rear index</u> move forward.
  - When an item is dequeued, the <u>front index</u> moves by one element towards the back of the queue (thus removing the front item, so no copying to neighboring elements is needed).

(front) XXXXOOOOO (rear)
OXXXXOOOO (after 1 dequeue, and 1 enqueue)
OOXXXXXOO (after another dequeue, and 2 enqueues)
OOOOXXXXX (after 2 more dequeues, and 2 enqueues)

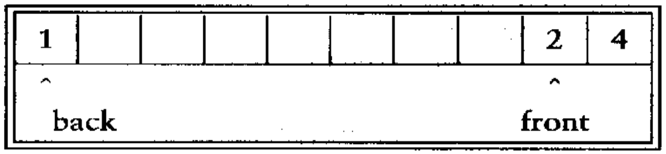
**The problem here is that the rear index cannot move beyond the last element in the array.**

# Implementation using Circular Array

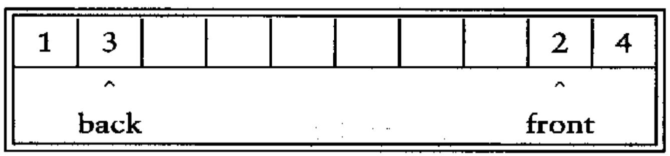
- Using a circular array
- When an element moves past the end of a circular array, it wraps around to the beginning, e.g.
  - OOOOO7963 → 4OOOO7963 (after Enqueue(4))
  - After Enqueue(4), the rear index moves from 3 to 4.

**Initial State**

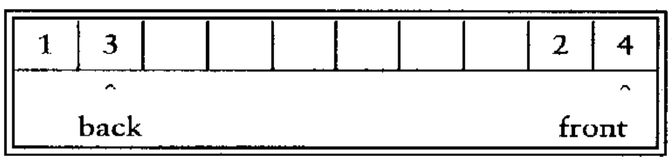

**After enqueue(1)**



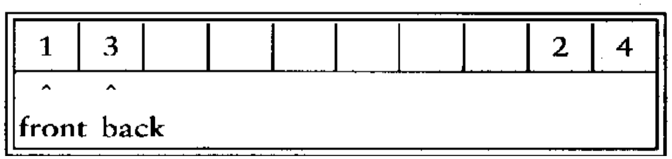

**After enqueue(3)**



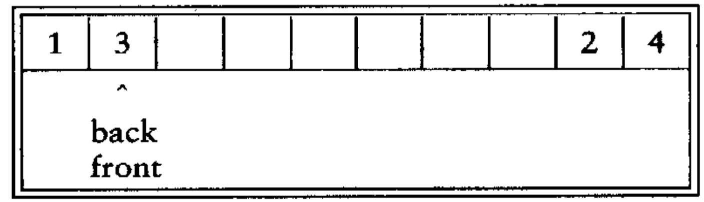

**After dequeue, Which Returns 2**



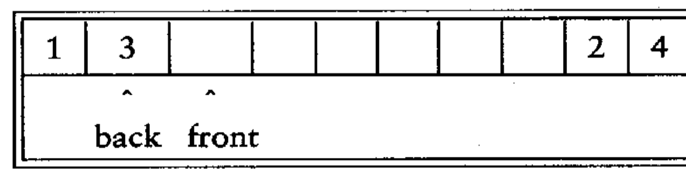

**After dequeue, Which Returns 4**




**After dequeue, Which Returns 1**




**After dequeue, Which Returns 3 and Makes the Queue Empty**



# Empty or Full?

- Empty queue
  - back = front - 1
- Full queue?
  - We need to count to know if queue is full
- Solutions
  - Use a boolean variable to say explicitly whether the queue is empty or not
  - Make the array of size n+1 and only allow n elements to be stored
  - Use a counter of the <u>number of elements</u> in the queue

# `Queue` Class

- Attributes of `Queue`
  - `front/rear`: front/rear index
  - `counter`: number of elements in the queue
  - `maxSize`: capacity of the queue
  - `values`: point to an array which stores elements of the queue
- Operations of `Queue`
  - `IsEmpty`: return true if queue is empty, return false otherwise
  - `IsFull`: return true if queue is full, return false otherwise
  - `Enqueue`: add an element to the rear of queue
  - `Dequeue`: delete the element at the front of queue
  - `DisplayQueue`: print all the data

# Create Queue

- `Queue(int size = 10)`
  - Allocate a queue array of `size`. By default, `size = 10`.
  - `front` is set to `0`, pointing to the first element of the array
  - `rear` is set to `-1`. The queue is empty initially.

```
int myQueue[size];
maxSize = size;
front=    0;
rear =    -1;
counter = 0;
```

# `IsEmpty` & `IsFull`

- Since we keep track of the number of elements that are actually in the queue: `counter`, it is easy to check if the queue is empty or full.

```
bool IsEmpty() {
    if (counter)    return false;
    else            return true;
}
bool IsFull() {
    if (counter < maxSize)    return false;
    else                      return true;
}
```

# Enqueue

```
bool Enqueue(int x) {
      if (IsFull()) {
            cout << "Error: the queue is full." << endl;
            return false;
      }
      else {
            // calculate the new rear position (circular)
            rear                = (rear + 1) % maxSize;
            // insert new item
            myQueue[rear]       = x;
            // update counter
            counter++;
            return true;
      }
}
```

# Dequeue

```
Int Dequeue( ) {
        if (IsEmpty()) {
                cout << "Error: the queue is empty." << endl;
                return -1;
        }
        else {
                // retrieve the front item
                int x         = values[front];
                // move front
                front  = (front + 1) % maxSize;
                // update counter
                counter--;
                return x;
        }
}
```

# Printing the elements

```
front -->       0
                1
                2
                3
                4       <-- rear
```

```
void Queue::DisplayQueue() {
    cout << "front -->";
    for (int i = 0; i < counter; i++) {
        if (i == 0) cout << "\t";
        else        cout << "\t\t";
        cout << myQueue[(front + i) % maxSize];
        if (i != counter - 1)
            cout << endl;
        else
            cout << "\t<-- rear" << endl;
    }
}
```

# Using Queue

```
Enqueue 5 items.
Now attempting to enqueue again...
Error: the queue is full.
front -->       0
                1
                2
                3
                4       <-- rear
Retrieved element = 0
front -->       1
                2
                3
                4       <-- rear
front -->       1
                2
                3
                4
                7       <-- rear
```

```
int main(void) {

      cout << "Enqueue 5 items." << endl;
      for (int x = 0; x < 5; x++)
            Enqueue(x);
      cout << "Now attempting to enqueue ag
      Enqueue(5);
      DisplayQueue();
      int value;
      value = Dequeue();
      cout << "Deleted element = " << value << endl;
      DisplayQueue();
      Enqueue(7);
      DisplayQueue();
      return 0;
}
```

# Queue Implementation based on Linked List

- First lets define the node that has data and a link

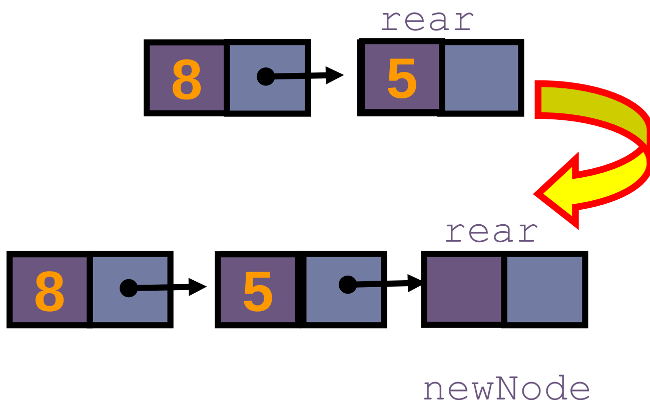
```
struct Node
{
    int data;
    Node *next;
};
```

# Queue Implementation based on Linked List

```
class Queue {
public:
        Queue() {                   // constructor
                front = rear = NULL;
                counter  = 0;
        }
        ~Queue() {                  // destructor
                double value;
                while (!IsEmpty()) Dequeue(value);
        }
        bool IsEmpty() {
                if (counter)        return false;
                else                return true;
        }
        void Enqueue(double x);
        bool Dequeue(double & x);
        void DisplayQueue(void);
private:
        Node* front;        // pointer to front node
        Node* rear;         // pointer to last node
        int counter;        // number of elements
};
```

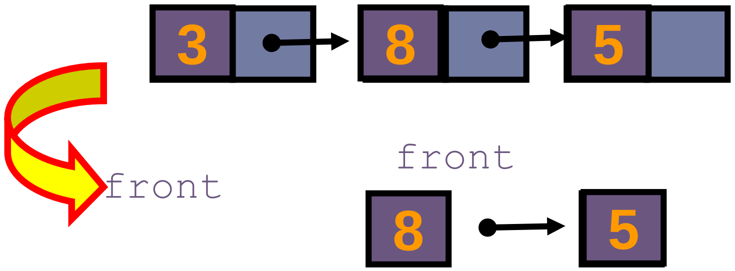
# Enqueue

```
void Queue::Enqueue(double x) {
      Node* newNode    =     new Node;
      newNode->data    =     x;
      newNode->next    =     NULL;
      if (IsEmpty()) {
            front      =     newNode;
            rear       =     newNode;
      }
      else {
            rear->next =     newNode;
            rear       =     newNode;
      }
      counter++;
}
```



# Dequeue

```
bool Queue::Dequeue(double & x) {
      if (IsEmpty()) {
            cout << "Error: the queue is empty." << endl;
            return false;
      }
      else {
            x                 =     front->data;
            Node* nextNode    =     front->next;
            delete front;
            front             =     nextNode;
            counter--;
      }
}
```



# Printing all the elements

```
void Queue::DisplayQueue() {
      cout << "front -->";
      Node* currNode   =     front;
      for (int i = 0; i < counter; i++)  {
            if (i == 0) cout << "\t";
            else        cout << "\t\t";
            cout << currNode->data;
            if (i != counter - 1)
                  cout << endl;
            else
                  cout << "\t<-- rear" << endl;
            currNode   =     currNode->next;
      }
}
```

```
Enqueue 5 items.
Now attempting to enqueue again..
front -->       0
                1
                2
                3
                4
                5       <-- rear
Retrieved element = 0
front -->       1
                2
                3
                4
                5       <-- rear
front -->       1
                2
                3
                4
                5
                7       <-- rear
```

# Using Queue

```
int main(void) {
      Queue queue(5);
      cout << "Enqueue 5 items." << endl;
      for (int x = 0; x < 5; x++)
            queue.Enqueue(x);
      cout << "Now attempting to enqueue ag
      queue.Enqueue(5);
      queue.DisplayQueue();
      double value;
      queue.Dequeue(value);
      cout << "Retrieved element = " << value << endl;
      queue.DisplayQueue();
      queue.Enqueue(7);
      queue.DisplayQueue();
      return 0;
}
```

```
Enqueue 5 items.
Now attempting to enqueue again...
Error: the queue is full.
front -->       0
                1
                2
                3
                4       <-- rear
Retrieved element = 0
front -->       1
                2
                3
                4       <-- rear
front -->       1
                2
                3
                4
                7       <-- rear
```

# Result

- Queue implemented using linked list will be never full

```
Enqueue 5 items.
Now attempting to enqueue again...
Error: the queue is full.
front -->       0
                1
                2
                3
                4       <-- rear
Retrieved element = 0
front -->       1
                2
                3
                4       <-- rear
front -->       1
                2
                3
                4
                7       <-- rear
```

**based on array**

```
Enqueue 5 items.
Now attempting to enqueue again..
front -->       0
                1
                2
                3
                4
                5       <-- rear
Retrieved element = 0
front -->       1
                2
                3
                4
                5       <-- rear
front -->       1
                2
                3
                4
                5
                7       <-- rear
```

**based on linked list**

***End of chapter 4***