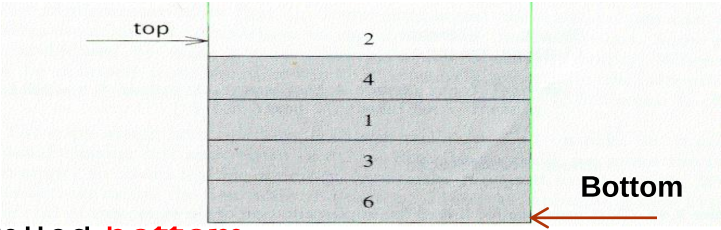
Lecture 5: Stack and Queue

Data Structure and Algorithm Analysis

The Stack ADT

- A stack is a list with the restriction
  - insertions and deletions can only be performed at the *top* of the list



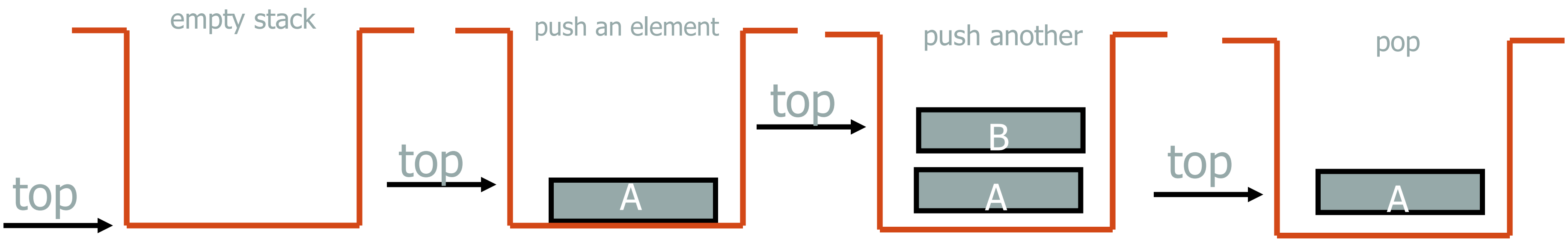
  - The other end is called bottom
- Stacks are less flexible
  - ✓ but are more efficient and easy to implement
- Stacks are known as LIFO (Last In, First Out) lists.
  - The last element inserted will be the first to be retrieved

Stack ADT

- Fundamental operations:
  - Push: Equivalent to an insert
    - Add an element to the top of the stack
  - Pop: Equivalent to delete
    - Removes the most recently inserted element from the stack
    - In other words, removes the element at the top of the stack
  - Top/peek: Examines the most recently inserted element
    - Retrieves the top element from the stack

Push and Pop

- Example



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
- Pop
  - (1) Set return value to 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`.

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

Push Stack

- `void Push(const double x);`
  - Increment top by 1
  - Check if stack is not full
    - Push an element onto the stack
  - If the stack is full, print the error information.
- Note `top` always represents the index of the top element.

```
void 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()`----Pop and return the element at the top of the stack
  - If the stack is empty, print the error information. (In this case, the return value is useless.)
  - Else, delete the top element
  - 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);
}
```

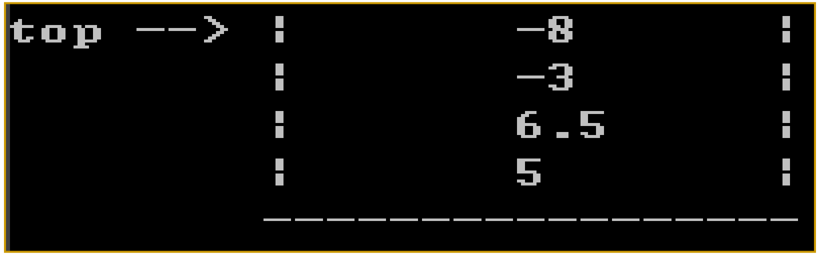
Stack Top

- `double Top()`
  - Return the top element of the stack
  - 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;

        stack.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

- Need not know the maximum size
- Add/Access/Delete in the beginning, O(1)
- Need several memory access, deletions

<u>Create the stack</u>

```
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 so, go to step-3 else 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(node *newnode)
{
Cout<<"Add data"<<endl;
Cin>>newnode-> item ;
newnode-> next = NULL;
if( topOfStack = = NULL){
topOfStack = newnode;
}
else {
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 next pointer
  - Step-3: Free the topOfstack node;
  - Step-4: Make the node pointed by "target" as your TOP most element

Pop operation

```
int pop( ) {
int pop_val= 0;
if(topOfStack = = NULL)
cout<<"Stack Underflow";
else {
node *temp= topOfStack;
pop_val= temp->data;
topOfStack =topOfStack-> next;
delete temp;
}
return(pop_val);
}
```

Application of stack Data Structure

- **Balancing Symbols:-** 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 if the stack is empty, report an error

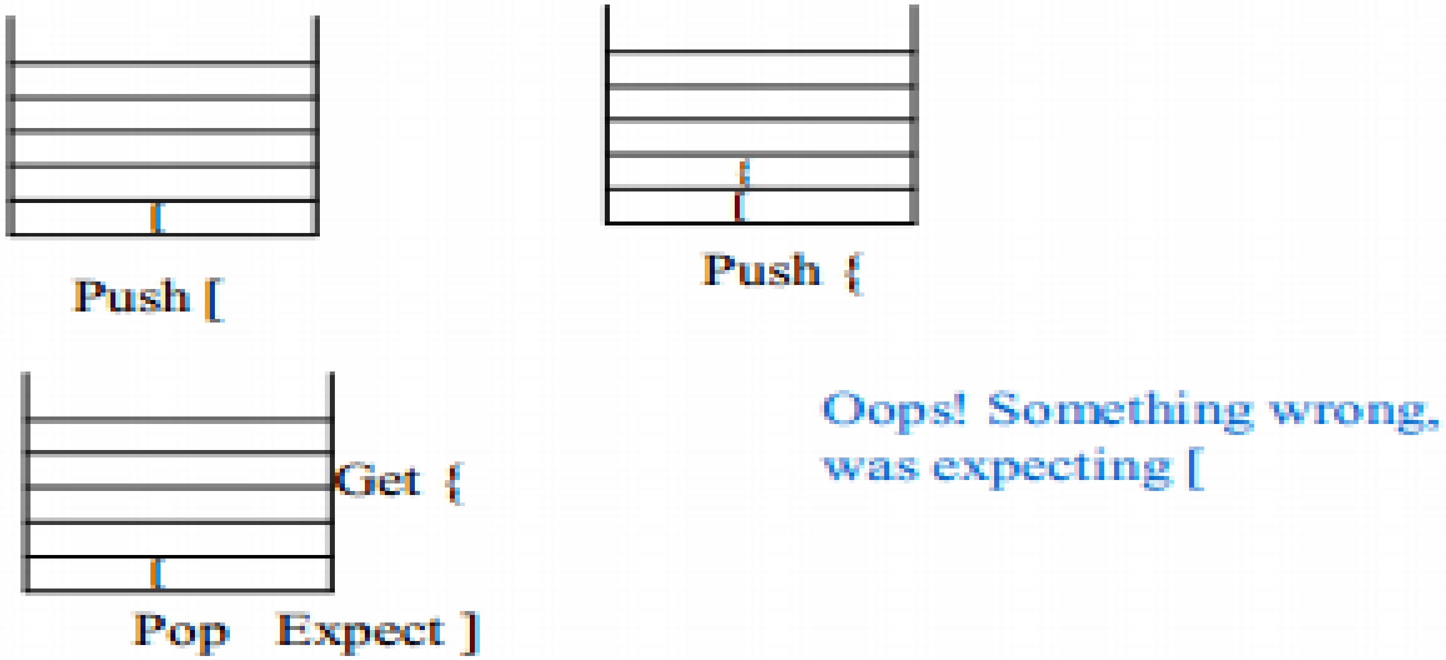
iii. Otherwise, pop the stack. If the symbol popped is not the corresponding opening symbol, then report an error

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

Example

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

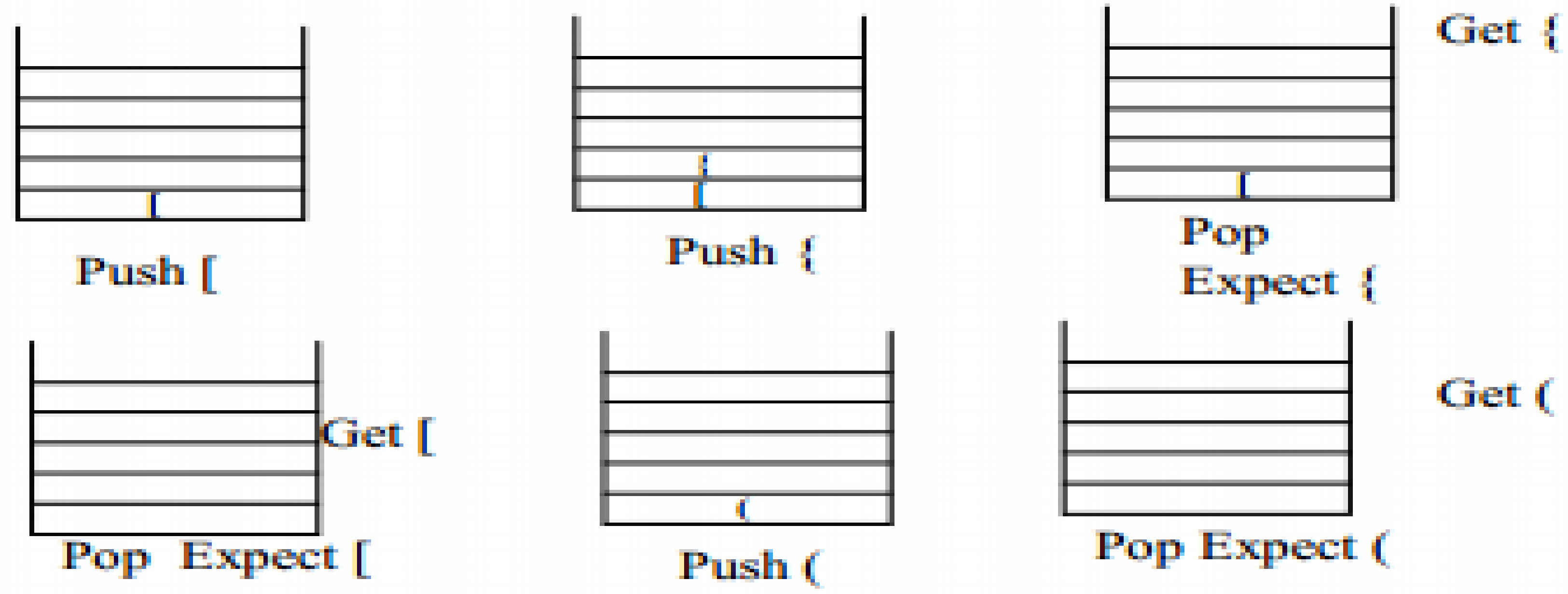
Push [, Push {, Pop, Pop, Push (, Pop



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 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

- Examine the next element in the input.
- If it is operand, output it.
- If it is opening parenthesis, push it on stack.
- If it is an operator, then
  - If stack is empty, push operator on stack.
  - If the top of stack is opening parenthesis, push operator on stack
  - If it has higher priority than the top of stack, push operator on stack.
  - Else pop the operator from the stack and output it, repeat step 4
- 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.
- If there is more input go to step 1
- 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*+

More Example:

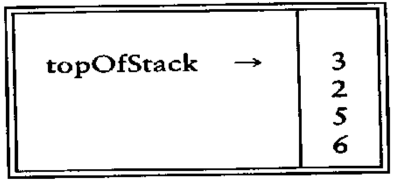
Suppose we want to convert 2*3/(2-1)+5*3 into Postfix form

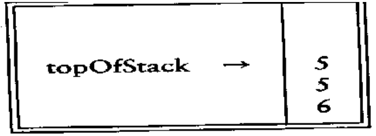
Expression	Stack	Output
2	Empty	2
*	*	2
3	*	23
/	/	23*
(	/(	23*
2	/(	23*2
-	/(-	23*2
1	/(-	23*21
)	/	23*21-
+	+	23*21-/
5	+	23*21-/5
*	+*	23*21-/53
3	+*	23*21-/53
	Empty	23*21-/53*+

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

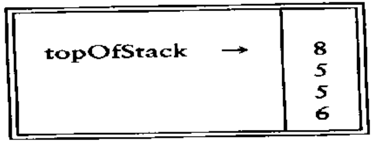
Postfix Expressions

- Calculate 4 * 5 + 6 * 7
  - Need to know the precedence rules
- Postfix (reverse Polish) expression
  - 4 5 * 6 7 * +
- Use stack to evaluate postfix expressions
  - 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

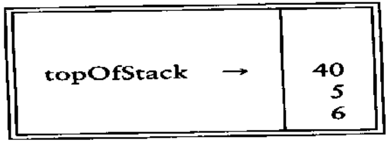




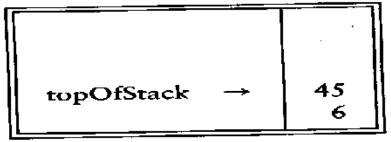
Next 8 is pushed.



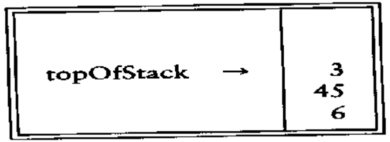
Now a '*' is seen, so 8 and 5 are popped and 5 * 8 = 40 is pushed.



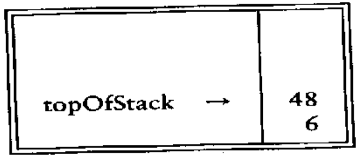
Next a '+' is seen, so 40 and 5 are popped and 5 + 40 = 45 is pushe



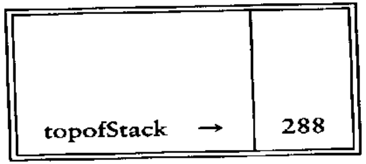
Now, 3 is pushed.



Next '+' pops 3 and 45 and pushes 45 + 3 = 48.



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



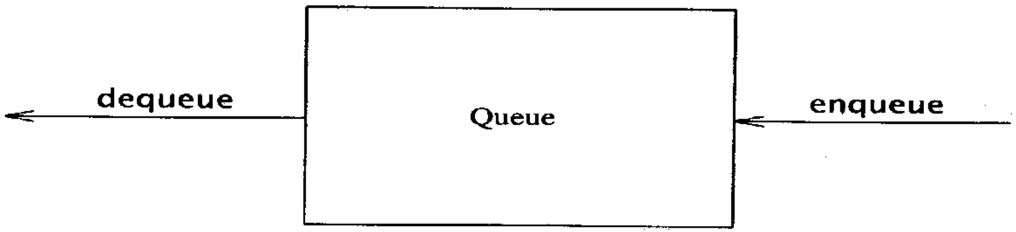
Queue

Queue ADT

- Like a stack, a *queue* is also a list.
  - However, with a queue, insertion is done at one end, while deletion is performed at the other end.
- 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



- First-in First-out (FIFO) list

Enqueue and Dequeue

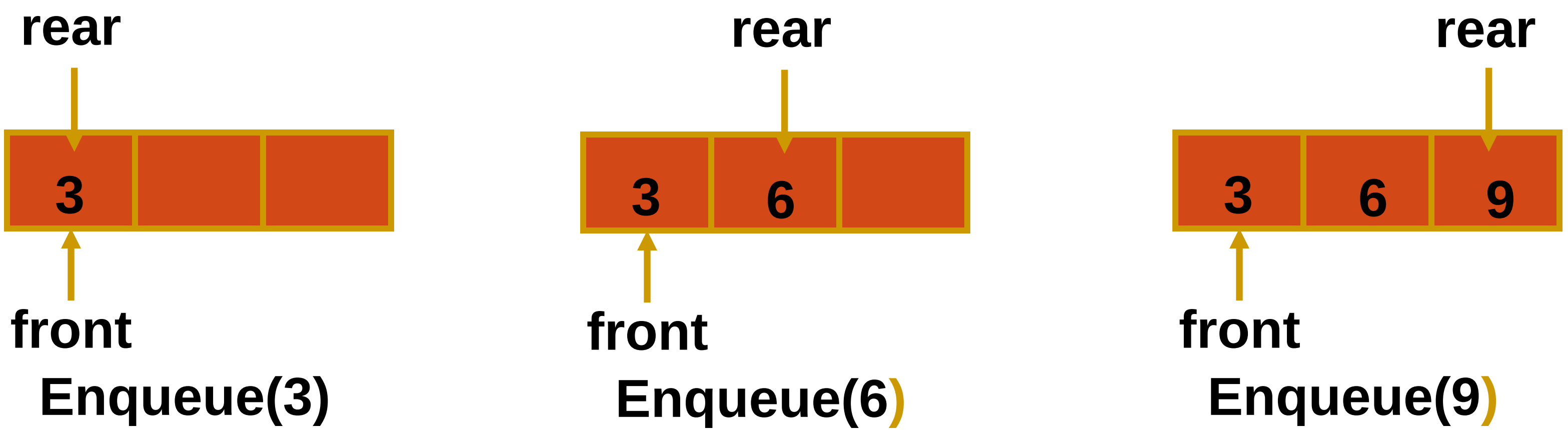
- Like check-out lines in a store, a queue has a front and a rear.

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

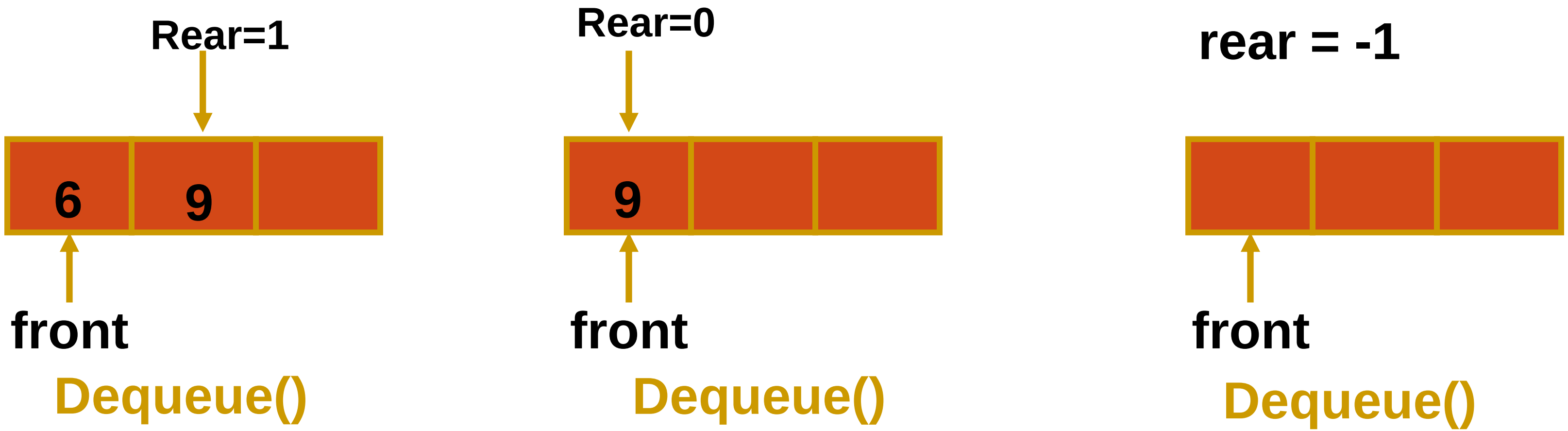
Array Implementation of Queue

- There are several different algorithms to implement Enqueue and Dequeue
- Naïve way
  - 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 front index is always fixed and the rear index moves forward in the array.
  - When dequeuing, the element at the front of the queue is removed. Move all the elements after it by one position. (Inefficient!!!)



Array Implementation of Queue

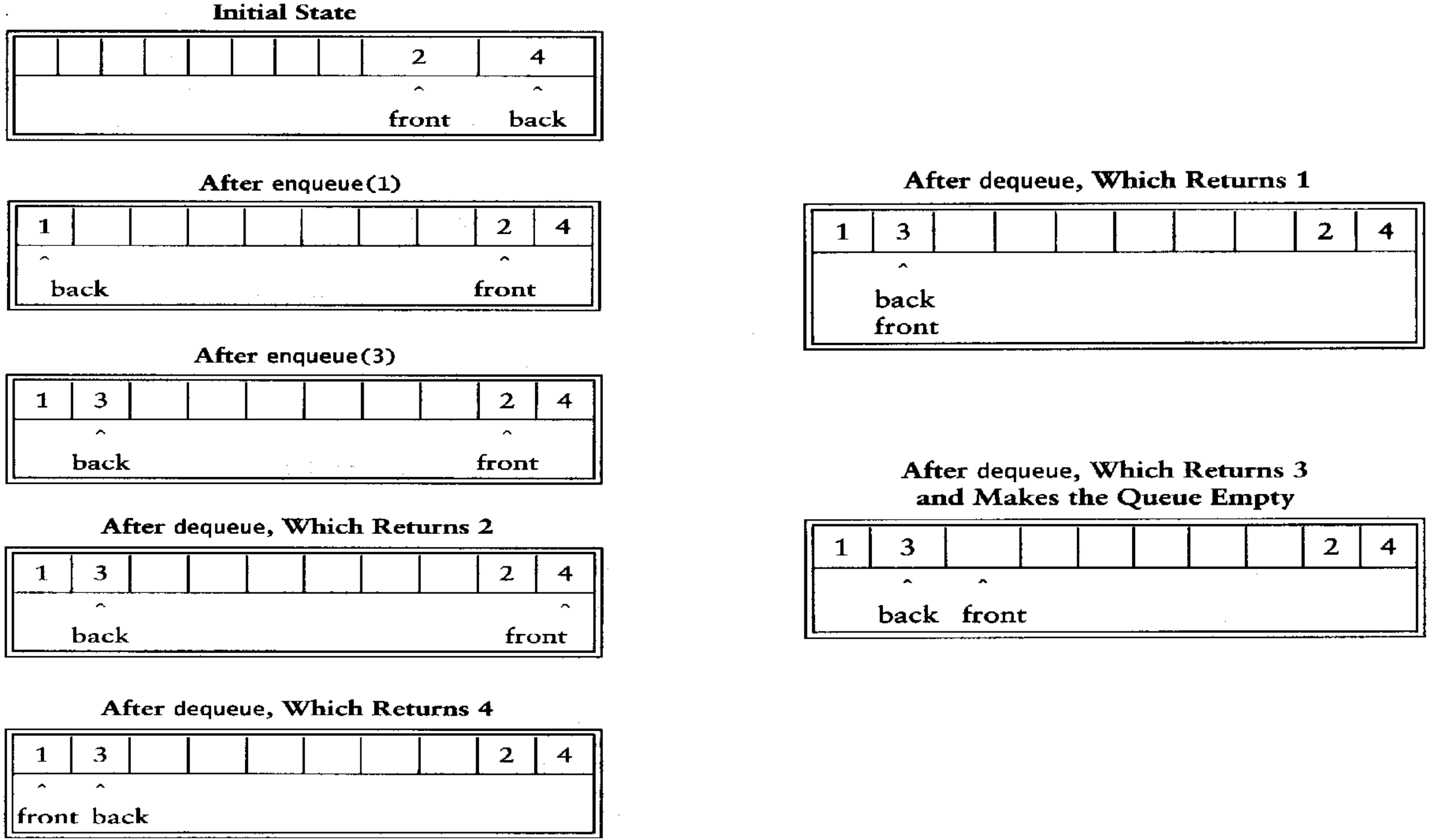
- Better way (Non-Naïve Way)
  - When an item is enqueued, make the rear index move forward.
  - When an item is dequeued, the front index moves by one element towards the back of the queue (thus removing the front item, so no copying to neighboring elements is needed).

```
(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 <u>rear index</u> moves from 3 to 4.


Initial State
2
4
front
back
After enqueue(1)
1
2
4
back
front
After enqueue(3)
1
3
2
4
back
front
After dequeue, Which Returns 2
1
3
2
4
back
front
After dequeue, Which Returns 4
1
3
2
4
front back
After dequeue, Which Returns 1
1
3
2
4
back
front
After dequeue, Which Returns 3
and Makes the Queue Empty
1
3
2
4
back front

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 number of elements 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.

```
Queue::Queue(int size /* = 10 */) {
        values          =       new double[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 Queue::IsEmpty() {
      if (counter)             return false;
      else                     return true;
}
bool Queue::IsFull() {
      if (counter < maxSize)   return false;
      else                     return true;
}
```

Enqueue

```
bool Queue::Enqueue(double 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
		values[rear]	= x;
		// update counter
		counter++;
		return true;
	}
}
```

Dequeue

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

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 << values[(front + i) % maxSize];
		if (i != counter - 1)
			cout << endl;
		else
			cout << "\t<-- rear" << endl;
	}
}
```

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 again..." <<
		endl;
	queue.Enqueue(5);
	queue.DisplayQueue();
	double value;
	queue.Dequeue(value);
	cout << "Retrieved element = " << value << endl;
	queue.DisplayQueue();
	queue.Enqueue(7);
	queue.DisplayQueue();
	return 0;
}
```

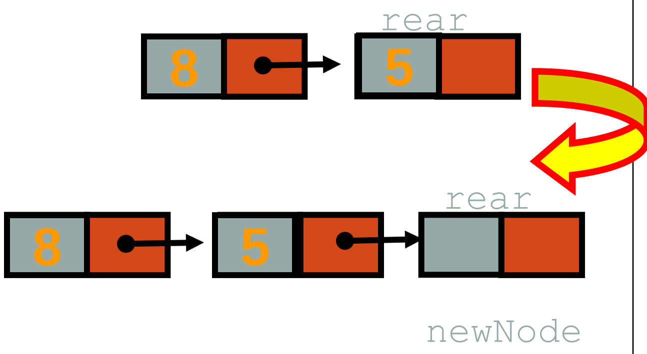
```
Enqueue 5 items.
Now attempting to enqueue again...
Error: the queue is full.
front -->       0
                1
                2
                3
                4       <-- rear
Retrieved element = 0
front -->       1
                2
                3
                4       <-- rear
front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

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
};
```

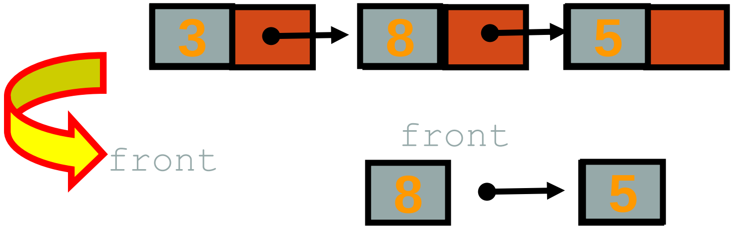
Enqueue

```
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
```

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 Lecture 5

Next Lecture:-Trees