Chapter 6
Stack and Queue

Content

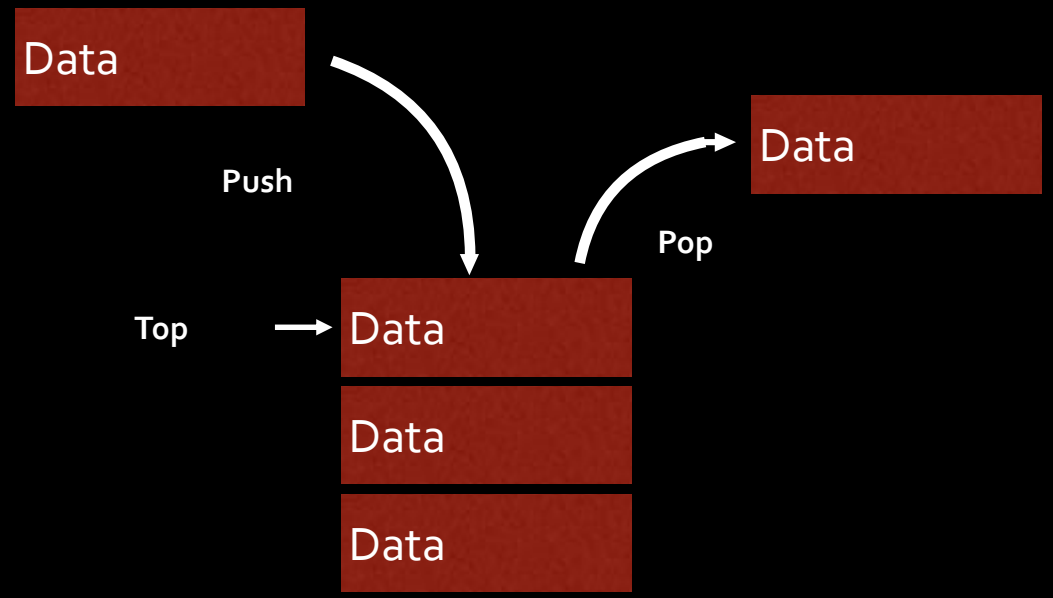
- Stack
  - Stack ADT?
  - Implementation of stack(Array, Linked list)
  - Application
- Queue
  - Queue ADT
  - Applications
  - Implementation (Array, Linked list)

Stack

- A **stack** is an Abstract Data Type (ADT), commonly used in most programming languages.
  - It is named **stack** as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

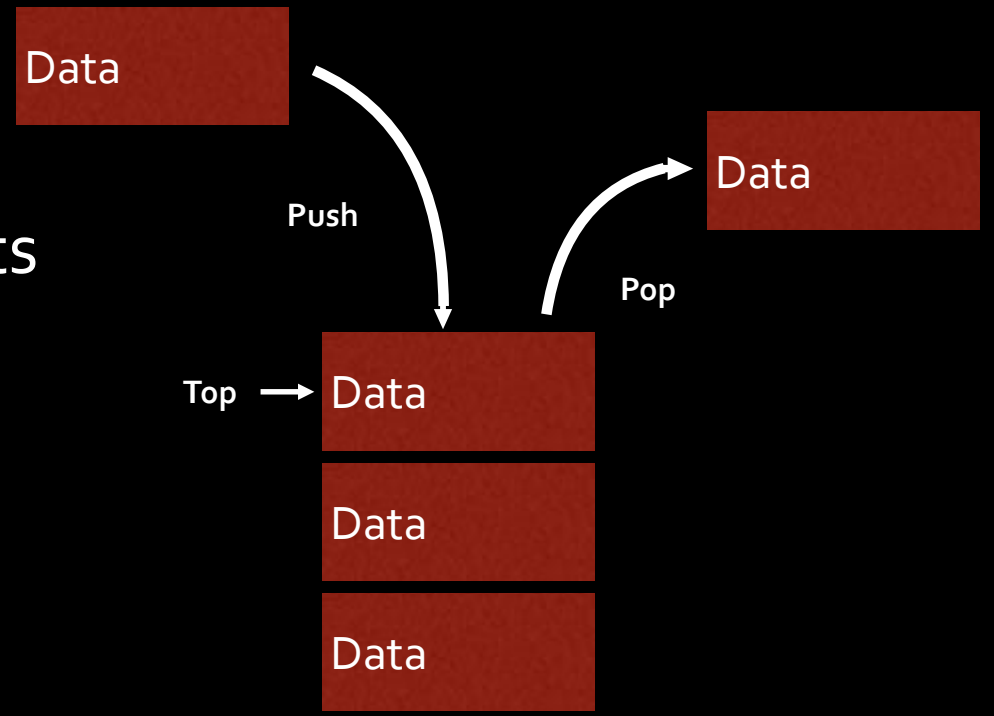
What is a Stack?

A stack is a one-ended linear data structure which models a real world stack by having two primary operations, namely **push** and **pop**.



The Stack ADT

- In other words, **a stack is a list with the restriction**
  - Because insertions and deletions can only be performed at only one end called the *top* of the stack.
- Hence:
  - Stacks are less flexible and
  - easy to implement compared to lists



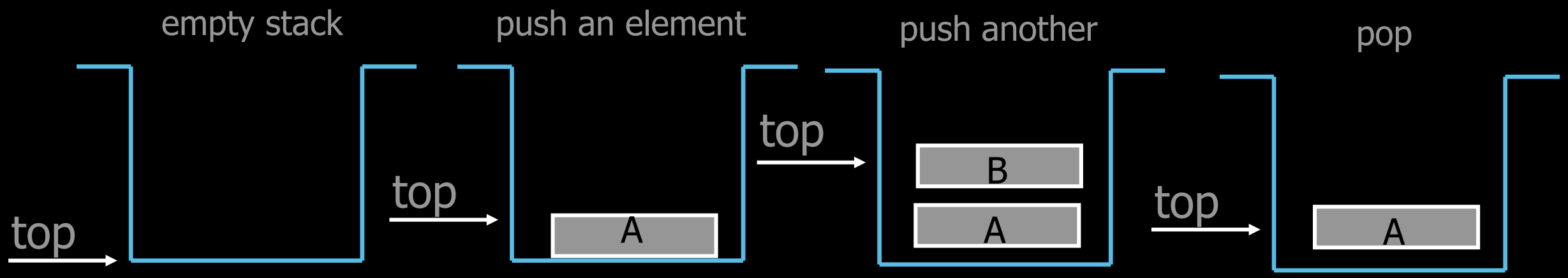
Stack ADT

- Fundamental operations:
  - Push: Equivalent to an insert
  - Pop: Deletes the most recently inserted element
  - Top/Peek: Examines the most recently inserted element
- 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

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/ peek()/** – Returns the top element of the stack without deleting it



Implementation of 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

`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`
  - Create a "stack array" of some `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`.

  Sample code:

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

Push Stack

Algorithm

- 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. After pushing an element, increment `top`.

Sample code

```
push(int item)
{

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

Pop Stack

Algorithm

- 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.)
- Don't forgot to decrement `top`

Sample code

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

Stack Top

Algorithm

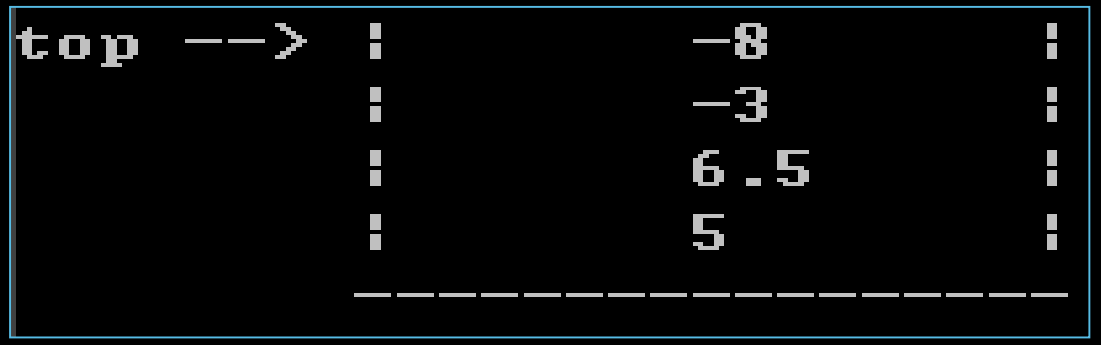
- Return the top element of the stack
- Unlike `Pop`, this function does not remove the top element

Sample code

```
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



Sample code

```
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

- 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 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(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(step2)
    topOfStack = newnode; //topOfStack point to newnode(S3)
     newnode-> next = NULL;
   }
   else {// if stack is not empty, rearrange the pointers(step4)
     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);
}
```

Complexity

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

Complexity

	Static	Dynamic
Pushing	O(1)	O(1)
Popping	O(1)	O(1)
Peeking	O(1)	O(1)
Searching	O(n)	O(n)
Size	O(1)	O(n)
Display	O(n)	O(n)

Application of stack Data Structure

Application of stack Data Structure

- Compiler's syntax check for matching symbols is implemented by using stack.
  - Balancing symbols ( ), [ ], { }
- Evaluating algebraic 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.

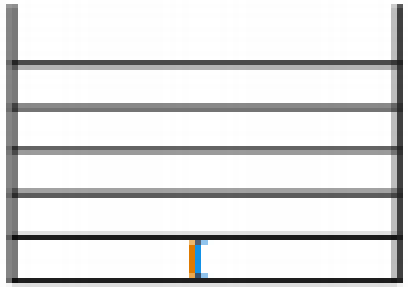
Balancing Symbols

- 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 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

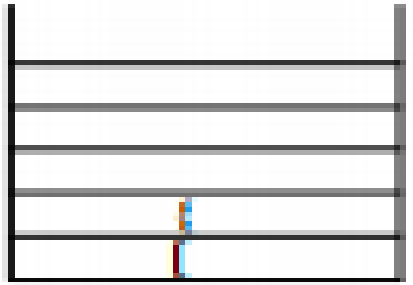
Example

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

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



Push [



Push {

Get {

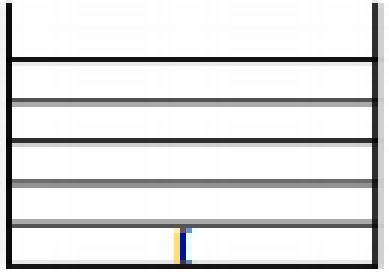
Pop Expect [

Oops! Something wrong, was expecting [

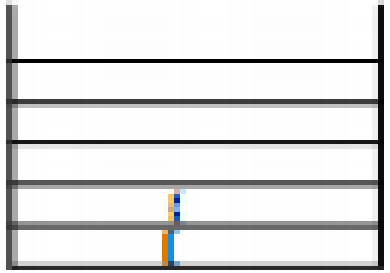
Example

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

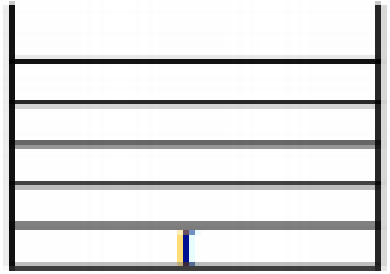
Push [, Push {, Pop , Pop, Push (, Pop



Push [



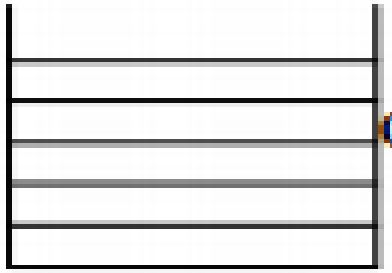
Push {



Get {

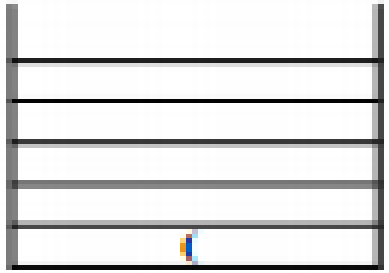
Pop

Expect {

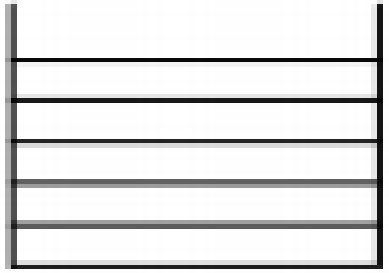


Get [

Pop Expect [



Push (



Get (

Pop Expect (

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, 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

Some Examples

Sr. No.	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 -

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

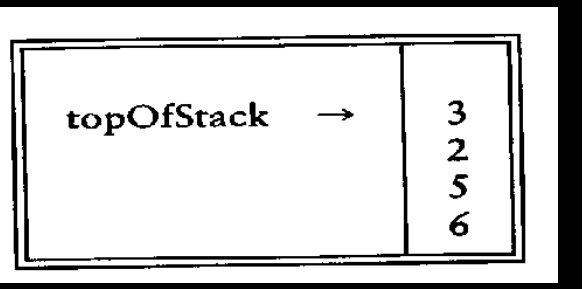
Expression	Stack	Output
2	Empty	2
*	*	2
3	*	23
/	/	23*
(	/(	23*
2	/(	23*2
-	/(-	23*2
1	/(-	23*21
)	/	23*21-
+	+	23*21-/
5	+	23*21-/5
*	+*	23*21-/53
3	+*	23*21-/53
	Empty	23*21-/53*+

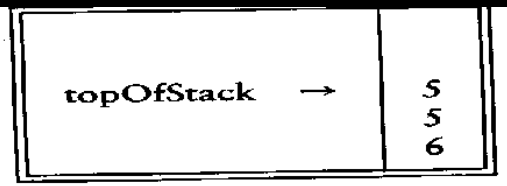
So, the Postfix Expression is 23*21-/53*+

Postfix Expressions

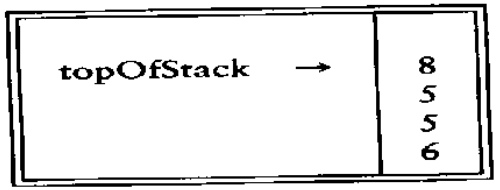
- To Calculate 4 * 5 + 6 * 7, we need to know the precedence rules
- But its Postfix (reverse Polish) equivalent doesn't require that
  - 4 5 * 6 7 * +
- Instead, use stack to evaluate postfix expressions as follows
  - When a number is seen, it is pushed onto the stack
  - When an operator is seen, 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

6 5 2 3 + 8 * + 3 + *

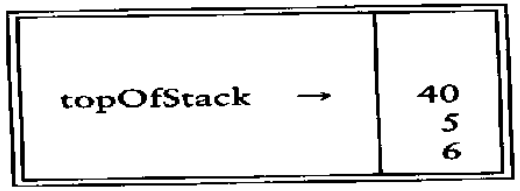




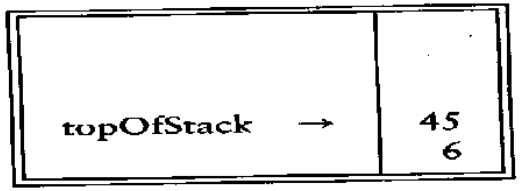
Next 8 is pushed.



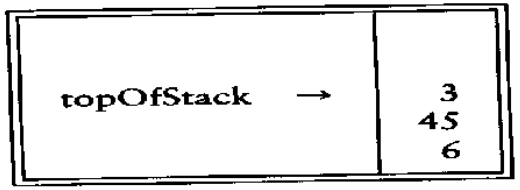
Now a '*' is seen, so 8 and 5 are popped and 5 * 8 = 40 is pushed.



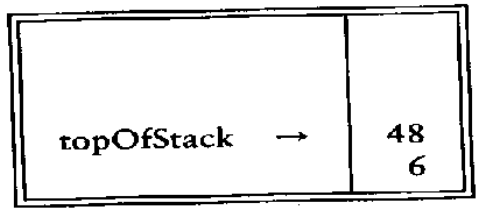
Next a '+' is seen, so 40 and 5 are popped and 5 + 40 = 45 is pushe



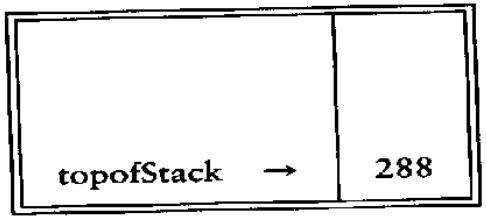
Now, 3 is pushed.



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



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



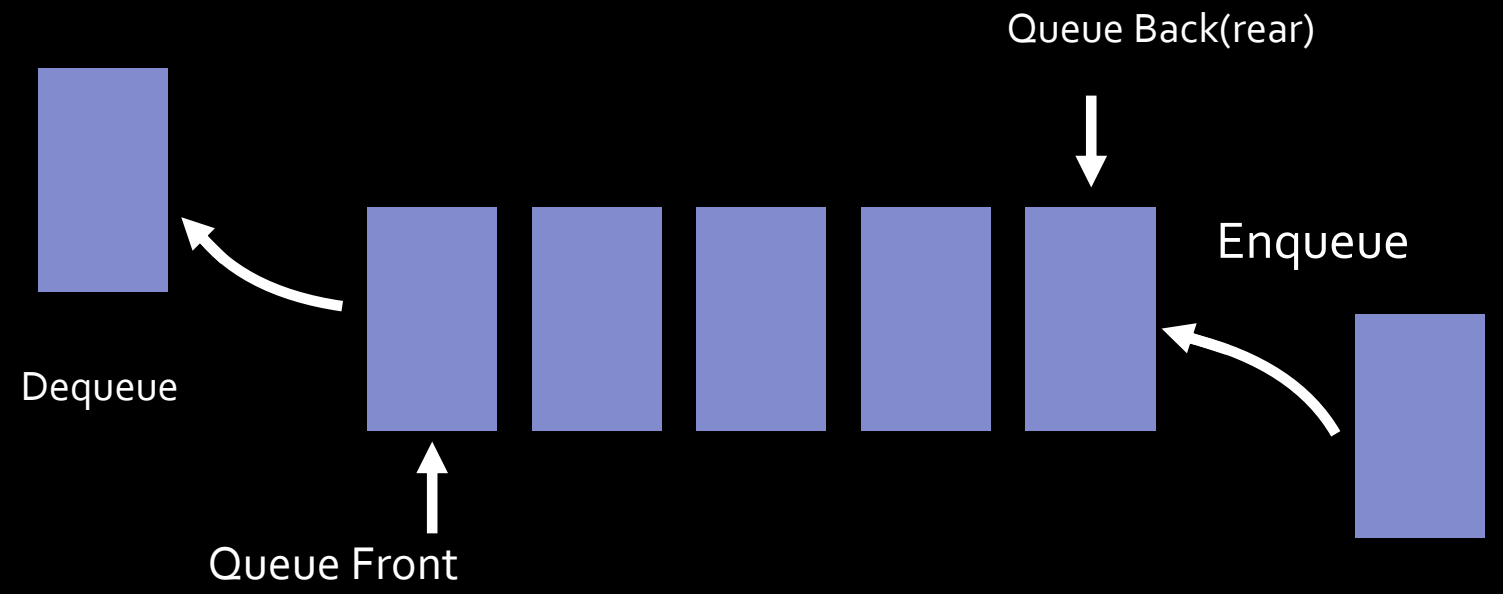
More Examples

- E.g. 1
  - Infix Expression : 1*(2+3)
  - Postfix expression : 123+*
    - Result is : 5
- E.g. 2
  - Infix Expression : (1+2)*(3+4)
  - Postfix expression : 12+34+*
    - Result is : 21

Queue ADT

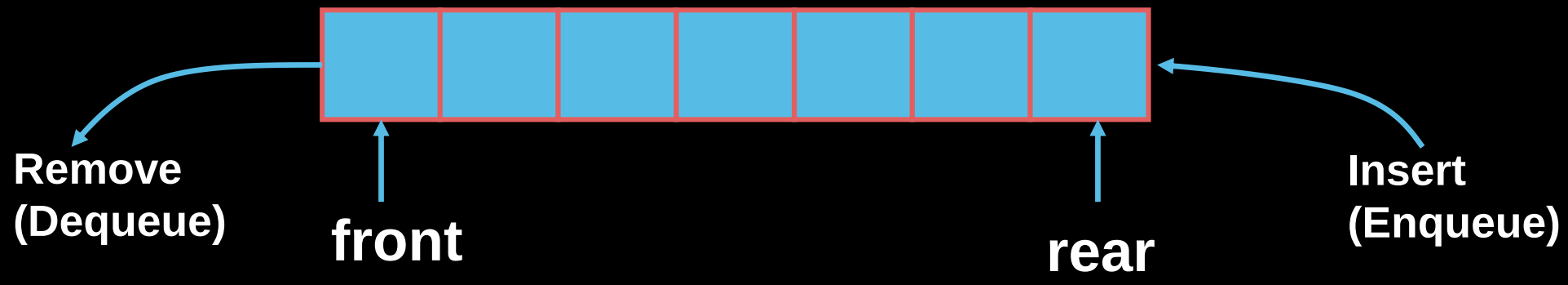
What is a Queue?

A queue is a linear data structure which models real world queues by having two primary operations, namely **enqueue** and **dequeue**.



Queue ADT

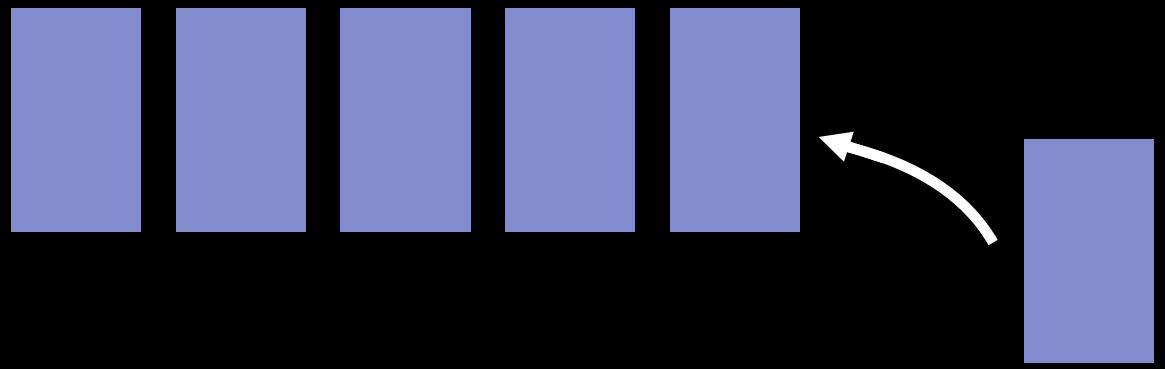
- Like a stack, a *queue* is also a list.
  - However, with a queue, insertion is done at one end(rear of the queue), while deletion is performed at the other end (front of the queue).
- Accessing the elements of queues follows a First In, First Out (FIFO) order.
  - Like customers standing in a check-out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



Queue Terminology

There does not seem to be consistent terminology for inserting and removing elements from queues.

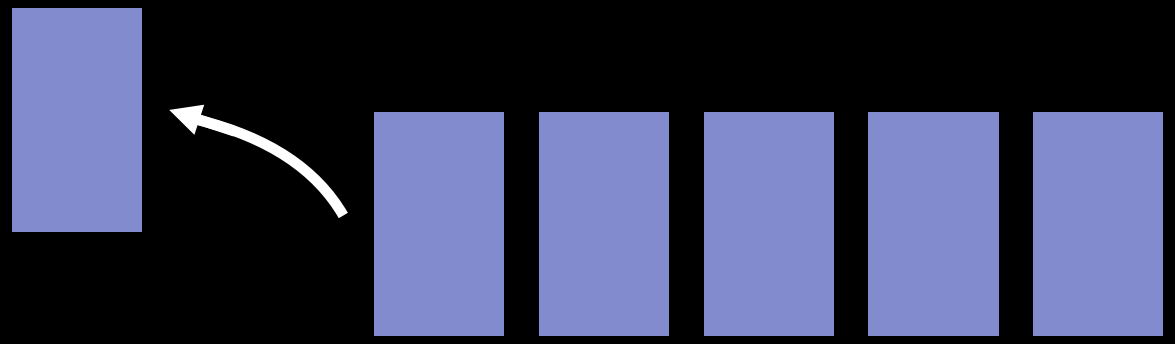
Enqueue = Adding = Offering



Queue Terminology

There does not seem to be consistent terminology for inserting and removing elements from queues.

Dequeue = Polling = *Removing*



When and where is a Queue used?

- Any waiting line models a queue, for example a lineup at a movie theatre.
- Can be used to efficiently keep track of the ***x*** most recently added elements.
- Web server request management where you want first come first serve.
- **Games**: Used to implement players' turn in multi-player games
- Breadth first search (BFS) graph traversal.

Operations of Queue ADT

- Basic operations:
  - Enqueue(): insert an element at the rear of the list
  - Dequeue(): delete the element at the front of the list
  - Peek() – Gets the element at the front of the queue without removing it.
- 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

Array implementation(Static queue)

Array implementation

- There are several different ways to implement Enqueue and Dequeue using arrays
  - Simple implementation
    - Increment rear, add element
    - Increment front, dequeue element
  - Naïve implementation
    - Don't move front during enqueue
  - Circular array implementation
    - Rear and front may wrap to the front of the queue

Simple 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

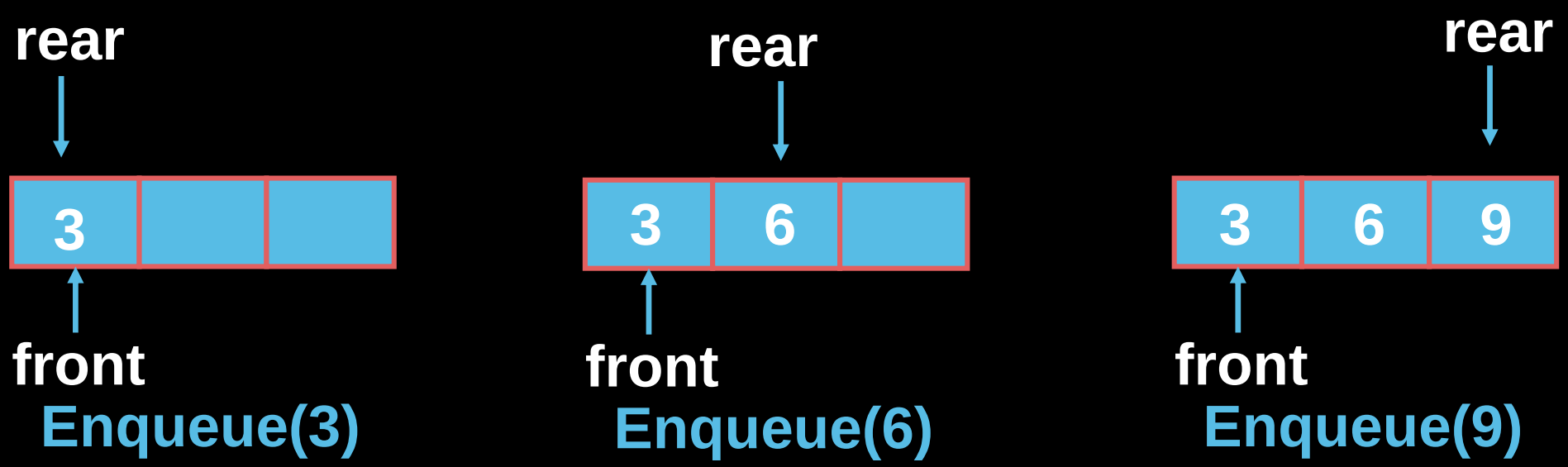
- Example: Consider a queue with MAX_SIZE = 4

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.

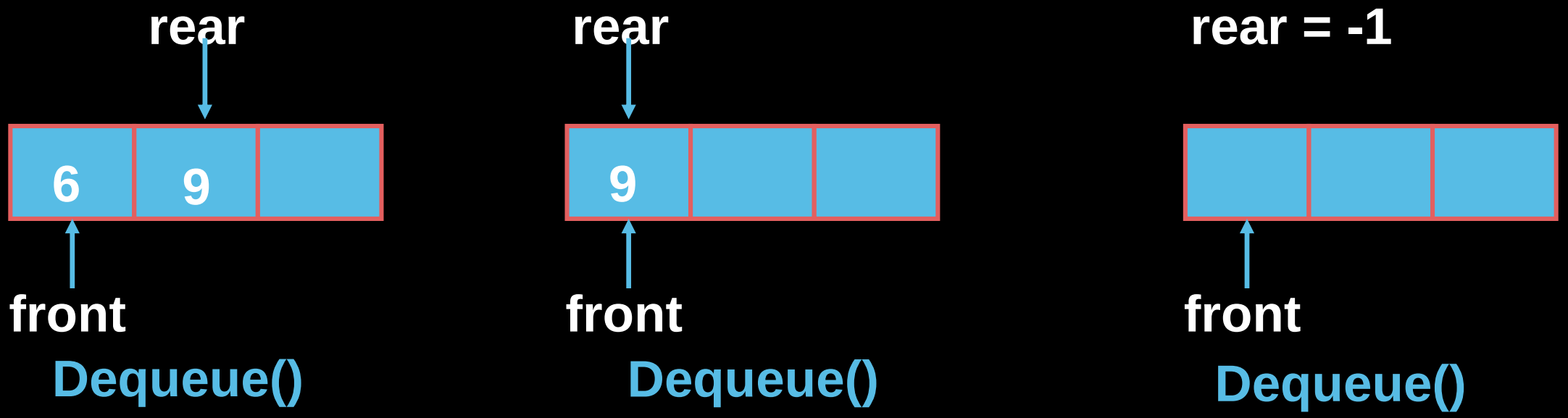
Array Implementation of Queue

- Naïve way
  - In order to solve the space wastage, move the elements by a position after each dequeue
  - When enqueuing, the front index is always fixed and the rear index 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!!!)



Naïve 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];
      For (j = o; j < QUEUESIZE ; j++)
        A[j] = A[j+1];

      QUEUESIZE--;
      REAR--;
     }
     else
      cout<<"Queue Underflow";
     return(x);}
```

Circular Array Implementation of Queue

Better 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

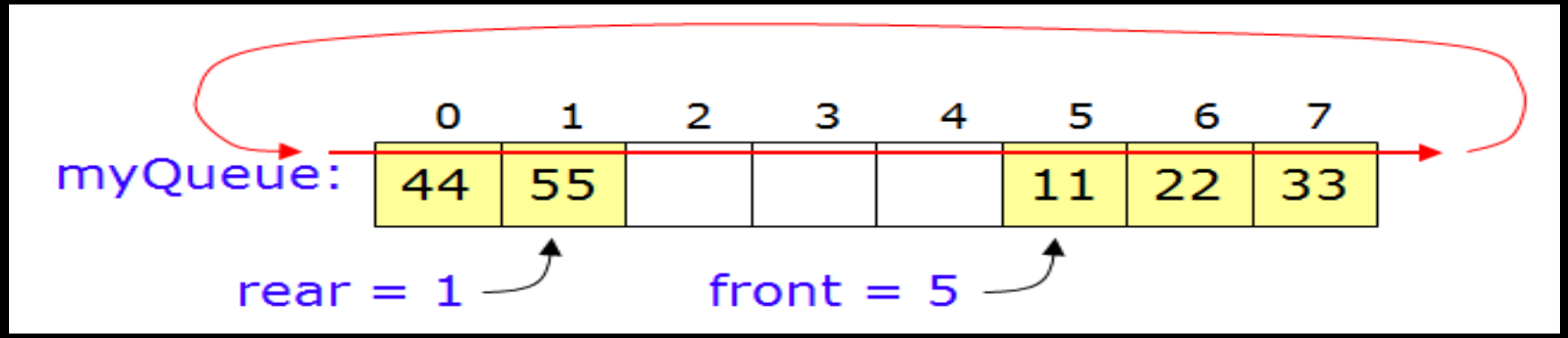
Therefore, when removing the front item, so no copying to neighboring elements is needed).

(front) XXXXOOOOOO (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.

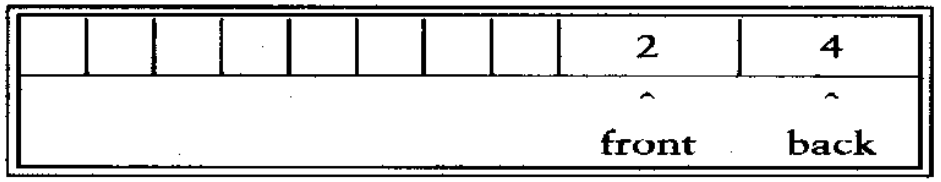
Circular arrays

- We can treat the array holding the queue elements as circular (joined at the ends)
- Elements were added to this queue in the order 11, 22, 33, 44, 55, and will be removed in the same order

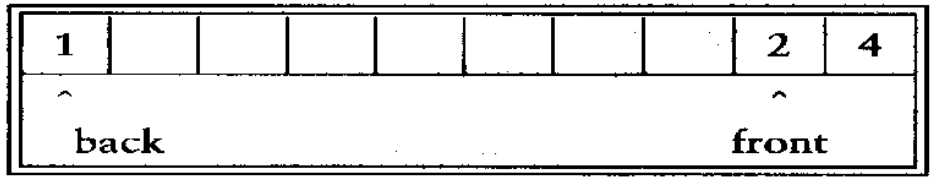


- Use: `front = (front + 1) % myQueue_length;`
  and: `rear = (rear + 1) % myQueue_length;`

Initial State

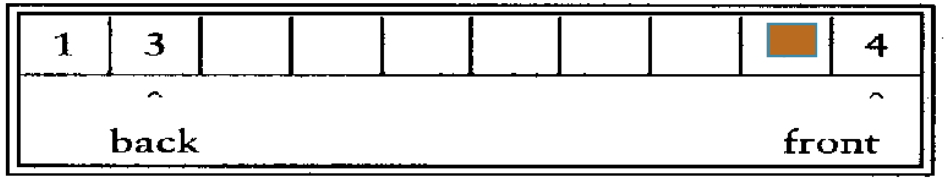


After enqueue(1)



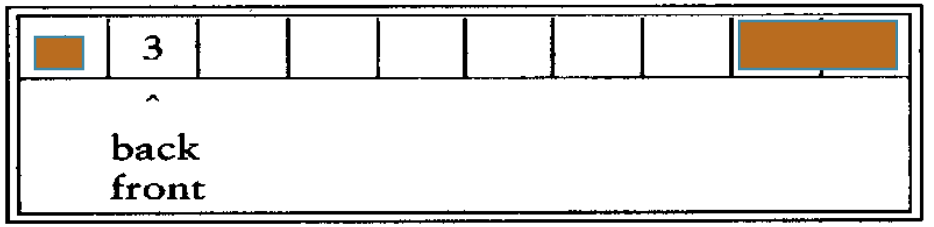
After enqueue(3)

After dequeue, Which Returns 2

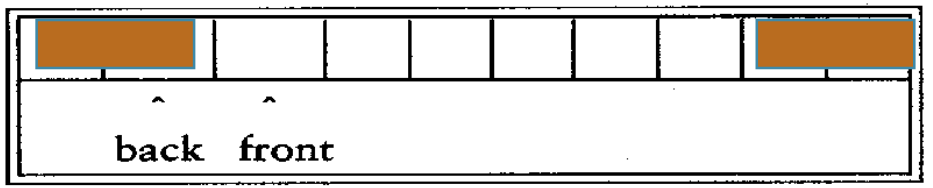


After dequeue, Which Returns 4

After dequeue, Which Returns 1

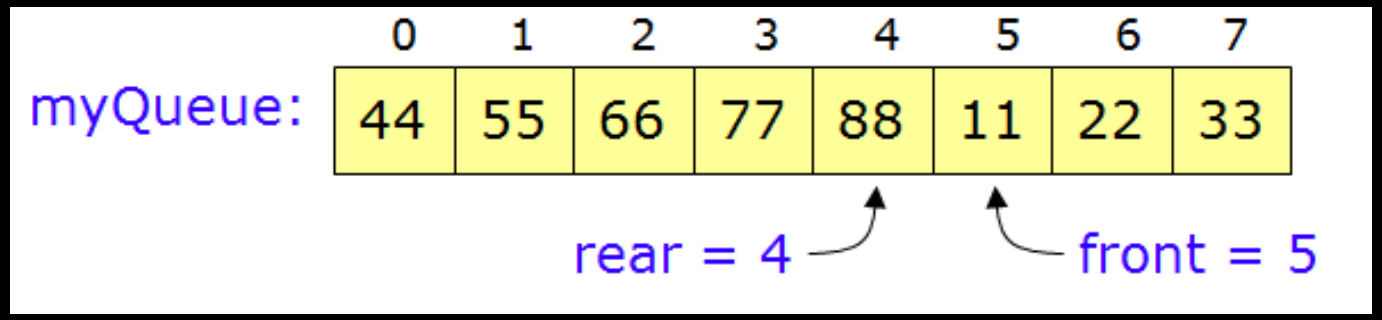


After dequeue, Which Returns 3 and Makes the Queue Empty

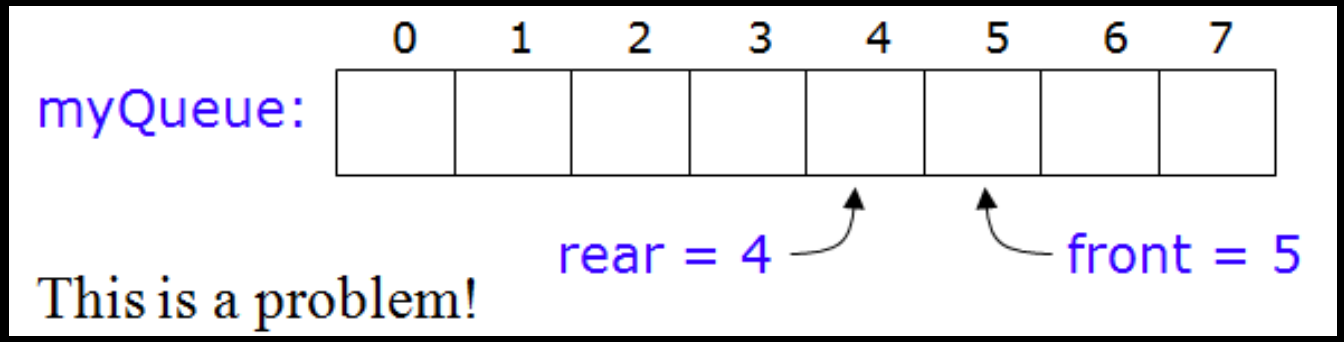


How to check if queue is Empty or Full?

- If the queue were to become completely full, it would look like this:

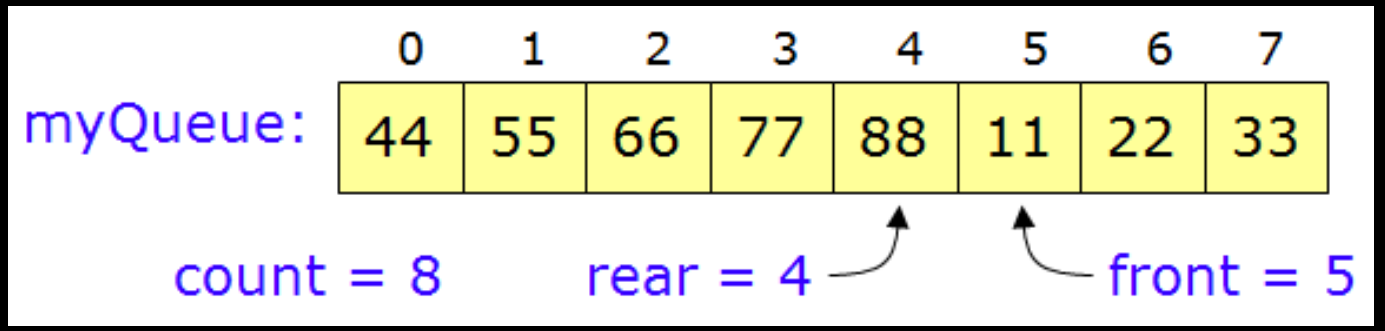


- If we were then to remove all eight elements, making the queue completely empty, it would look like this:

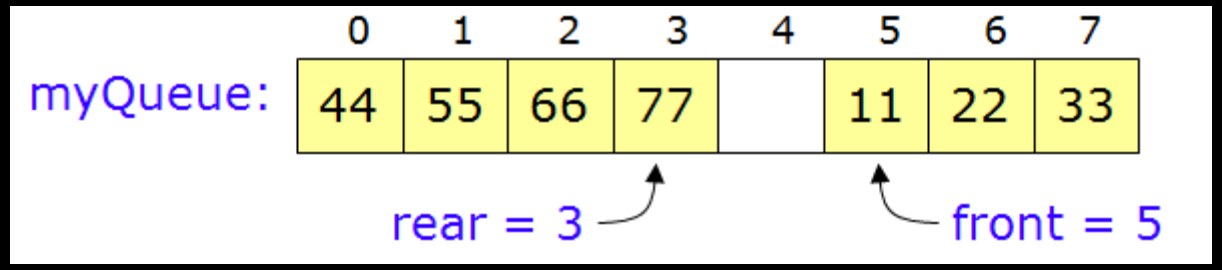


Full and empty queues: solutions

- **Solution #1:** Keep an additional counter variable



- **Solution #2:** (Slightly more efficient) Keep a gap between elements: consider the queue full when it has `n-1` elements



Circular Array Implementation by Keeping a gap

- **Shared data**

```
int queue[QUEUE_SIZE];
int rear = 0;
int front = 0;
```

- Solution is correct, but can only use **QUEUE_SIZE -1** elements

Circular Array Implementation by Keeping a gap

```
void enqueue(int data){
  /* Check if queue is not full*/
  if ((rear + 1) % QUEUE_SIZE) == front) {
      cout<<“Full”;
      return;}
   else{
      queue[rear] = data;
      rear = (rear + 1) % QUEUE_SIZE;
} }
```

Circular Array Implementation by Keeping a gap

```
int  dequeue(){
     int item=INT_MIN; //the minimum integer value

     if (rear == front){
          cout<<"Empty";
     }
     else{
          item = buffer[front];
          front = (front + 1) % BUFFER_SIZE;
     }
          return item;
}
```

Circular Array Implementation using **counter variable**

- Attributes of `Queue`
  - `front/rear`: front/rear index
  - `counter`: number of elements in the queue
  - `maxSize`: capacity of the queue
  - `myQueue`: 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

- Allocate a queue array of `size`. In this particular example, `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.

```
const int size = 5;
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 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

```
int main(void) {

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

Linked-list implementation of queues

- In a queue, insertions occur at one end, deletions at the other end
  - Operations at the front of a singly-linked list (SLL) are O(1), but at the other end they are O(n)
    - Because you have to find the last element each time
  - To use a singly-linked list to implement both insertions and deletions in O(1) time:
    - Keep pointers to *both* the **front** and the **rear** of the SLL

Queue Implementation based on Linked List

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

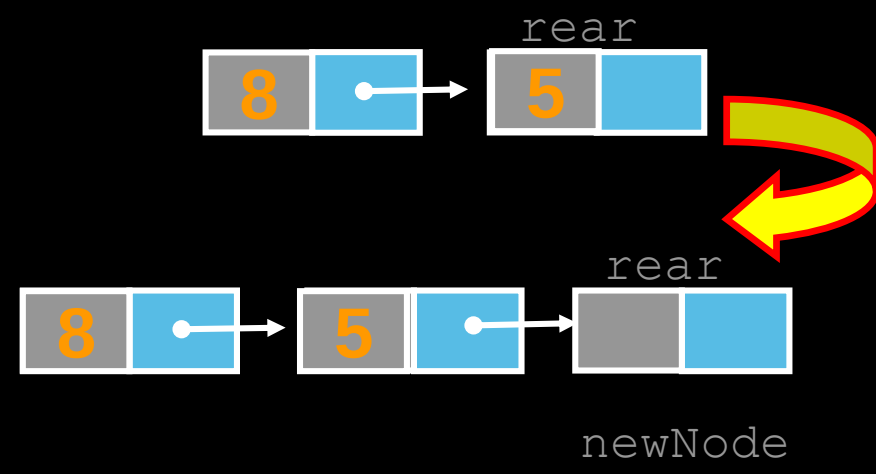
```
struct Node
{
        int data;
        Node *next;
};
```

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
};
```

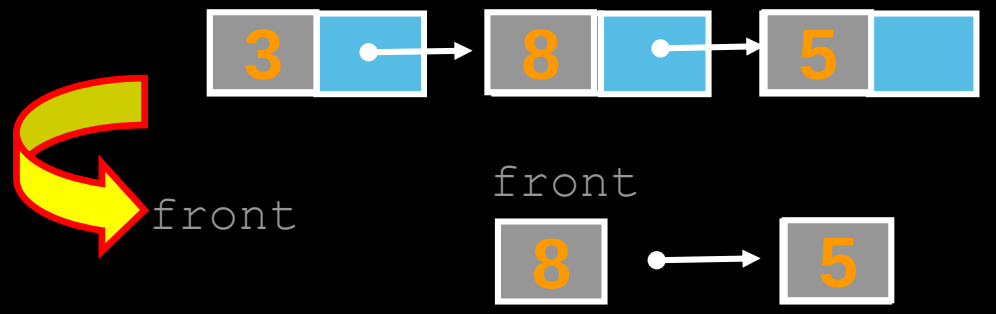
Enqueue

```
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 again
	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

```
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

Queue implementation notes

- With an array implementation:
  - You can have both overflow and underflow
  - You should set deleted elements to **null**

- With a linked-list implementation:
  - You can have underflow
  - Overflow is a global out-of-memory condition
  - There is no reason to set deleted elements to **null**

Complexity

Enqueue	O(1)
Dequeue	O(1)
Peeking	O(1)
Contains	O(n)
Removal	O(n)
Is Empty	O(1)

Exercise

Operation	Content of queue
Enqueue(B)	
Enqueue(C)	
Dequeue()	
Enqueue(G)	
Enqueue (F)	
Dequeue()	
Enqueue(A)	
Dequeue()	

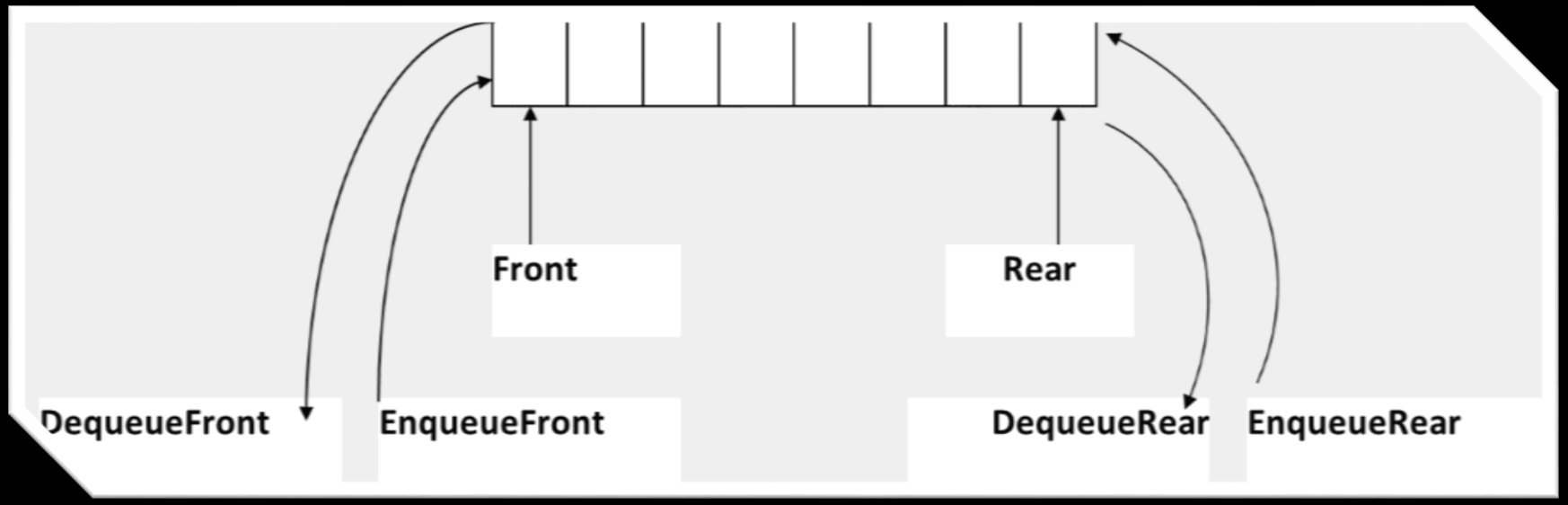
Variations of queque (The Exam doesn’t cover this part)

Deques

- A **deque** is a double-ended queue
  - **Insertions** and **deletions** can occur at ***either*** **end**
- Deque has the following basic operations
  - EnqueueFront – inserts data at the front of the list
  - DequeueFront – deletes data at the front of the list
  - EnqueueRear – inserts data at the end of the list
  - DequeueRear – deletes data at the end of the list

- Implementation is similar to that of queues
- It is best implemented using **doubly linked list**

Deques



- Deques are **not heavily used**
- You should know what a deque is, but we won't explore them much further.

Priority Queues

- A stack is first in, last out
- A queue is first in, first out
- A **priority queue** is a queue where **each data has an associated key** that is provided at the time of insertion.
  - If it is *least-first-out then*
    - The "smallest" element is the first one removed
  - It may also be a ***largest-first-out*** **priority queue**) in which case
    - The "largest" element is the first one removed
  - The definition of "smallest"/"largest/ is up to the programmer.

Examples

- *Consider the following queue of persons where females have higher priority than males (**gender is the key to give priority**).*

Abebe	Alemu	Aster	Belay	Kedir	Meron	Yonas
Male	Male	Female	Male	Male	Female	Male

Examples

- Dequeue()- deletes Aster

Abebe	Alemu	Belay	Keder	Meron	Yonas
Male	Male	Male	Male	Female	Male

- Dequeue()- deletes Meron

Abebe	Alemu	Belay	Kedr	Yonas
Male	Male	Male	Male	Male

  - Now the queue has data having equal priority and dequeue operation deletes the front element like in the case of ordinary queues

- Dequeue()- deletes Abebe

Alemu	Belay	Kedr	Yonas
Male	Male	Male	Male

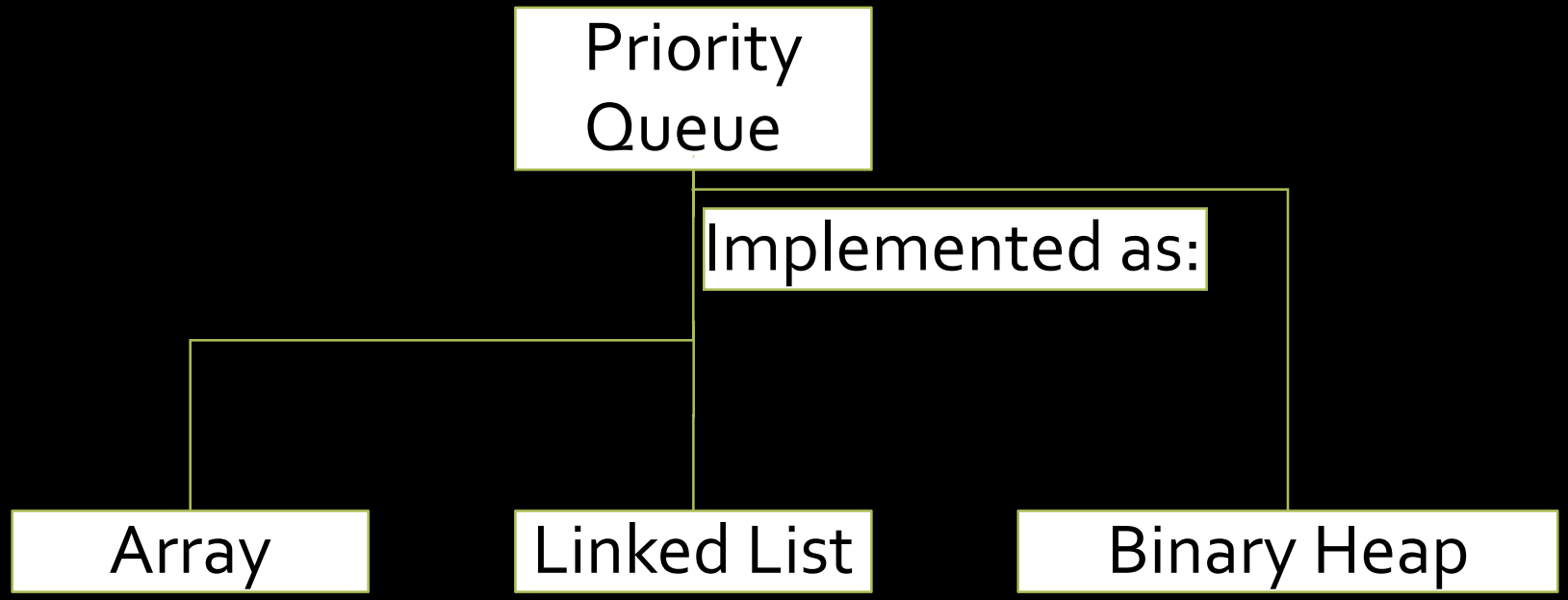
Priority Queues: Assumptions

- The highest priority can be either the ***minimum*** value of all the *items*, or the ***maximum***.
  - We will assume the highest priority is the ***maximum***.
- Assume the priority queue has *n* members

Operations on Priority Queues

- **Create**: Create a new PQ of a given maximum size
- **Add**: Add an element to a PQ based on its priority (key)
- **Remove**: Remove and return the element in a PQ with highest priority
- **Full**: Return whether or not the PQ is full
- **Empty**: Return whether or not the PQ is empty

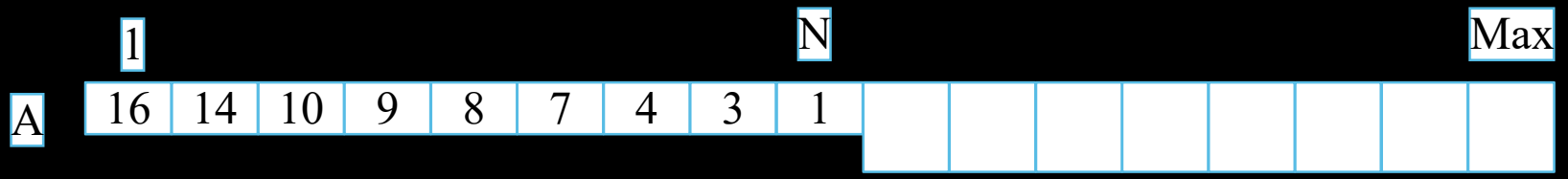
Implementations of Priority Queues



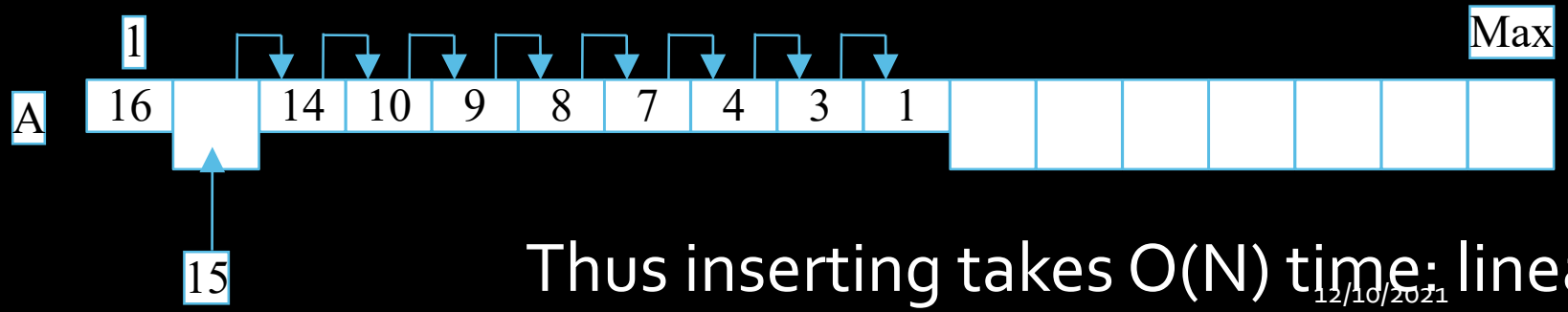
Array Implementation of Priority Queues

Suppose items with priorities 16, 14, 10, 9, 8, 7, 4, 3, 1 are to be stored in a priority queue.

One implementation:



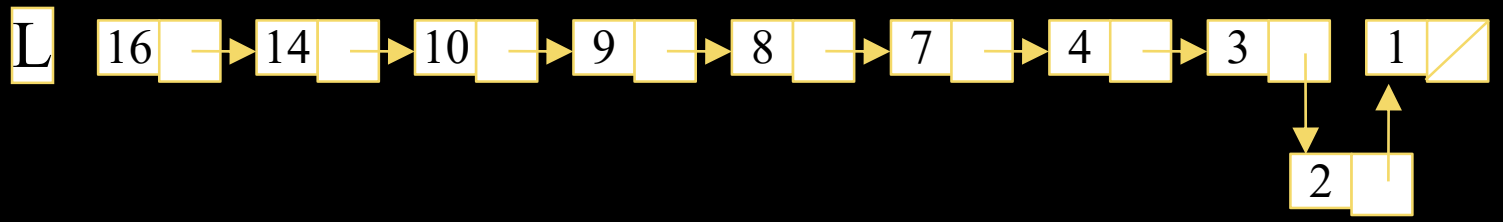
Suppose an item with priority 15 is added:



Thus inserting takes O(N) time: linear

Linked List Implementation of Priority Queues

Suppose an item with priority 2 is to be added:



Only O(1) (constant) pointer changes required, but it takes O(N) pointer traversals to find the location for insertion.

Wanted: a data structure for PQs that can be both searched and updated in better than O(N) time.

Demerging and Merging Queues

- **Demerging Queues**
  - is the process of creating two or more queues from a single queue.
  - used to give priority for some groups of data
    - *Example:*
      - *The following two queues can be created from the below priority queue.*

Abebe	Alemu	Aster	Belay	Kedir	Meron	Yonas
Male	Mal	Female	Male	Male	Female	Male

Aster	Meron
Female	Female

Abebe	Alemu	Belay	Kedir	Yonas
Male	Mal	Male	Male	Male

Demerging Algorithm

```
create empty females and males queue
while (PriorityQueue is not empty)
{
    // delete data at the front
    Data=DequeuePriorityQueue();
    if(gender of Data is Female)
        EnqueueFemale(Data);
    else
        EnqueueMale(Data);
}
```

Merging Queues

- is the process of creating a priority queue from two or more queues.
- The ordinary dequeue implementation can be used to delete data in the newly created priority queue.
  - *Example: The following two queues (females queue has higher priority than the males queue) can be merged to create a priority queue.*

Aster	Meron
Female	Female

Abebe	Alemu	Belay	Kedir	Yonas
Male	Mal	Male	Male	Male

Aster	Meron	Abebe	Alemu	Belay	Kedr	Yonas
Female	Female	Male	Mal	Male	Male	Male

Merging Algorithm

```
create an empty priority queue
while(FemalesQueue is not empty)
{
  EnqueuePriorityQueue(DequeueFemalesQueue());
}
while(MalesQueue is not empty)
{
    EnqueuePriorityQueue(DequeueMalesQueue());
}
```

Next Time!!

Ch7 - Trees