ADVANCED DATA STRUCTURES USING C++

(MT-CSE-110)

Unit - 2

By:

Gurpreet Singh

Dean Academics & H.O.D. (C.S.E. / I.T.)

Yamuna Institute of Engineering & Technology, Gadholi

What is a Stack?

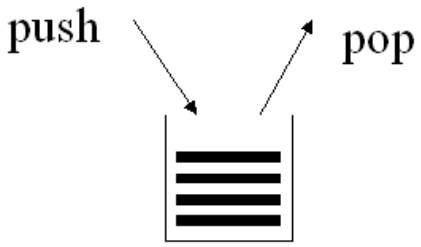
- A stack is a data structure in which data is added and removed at only one end called the top.

- To add (push) an item to the stack, it must be placed on the top of the stack.

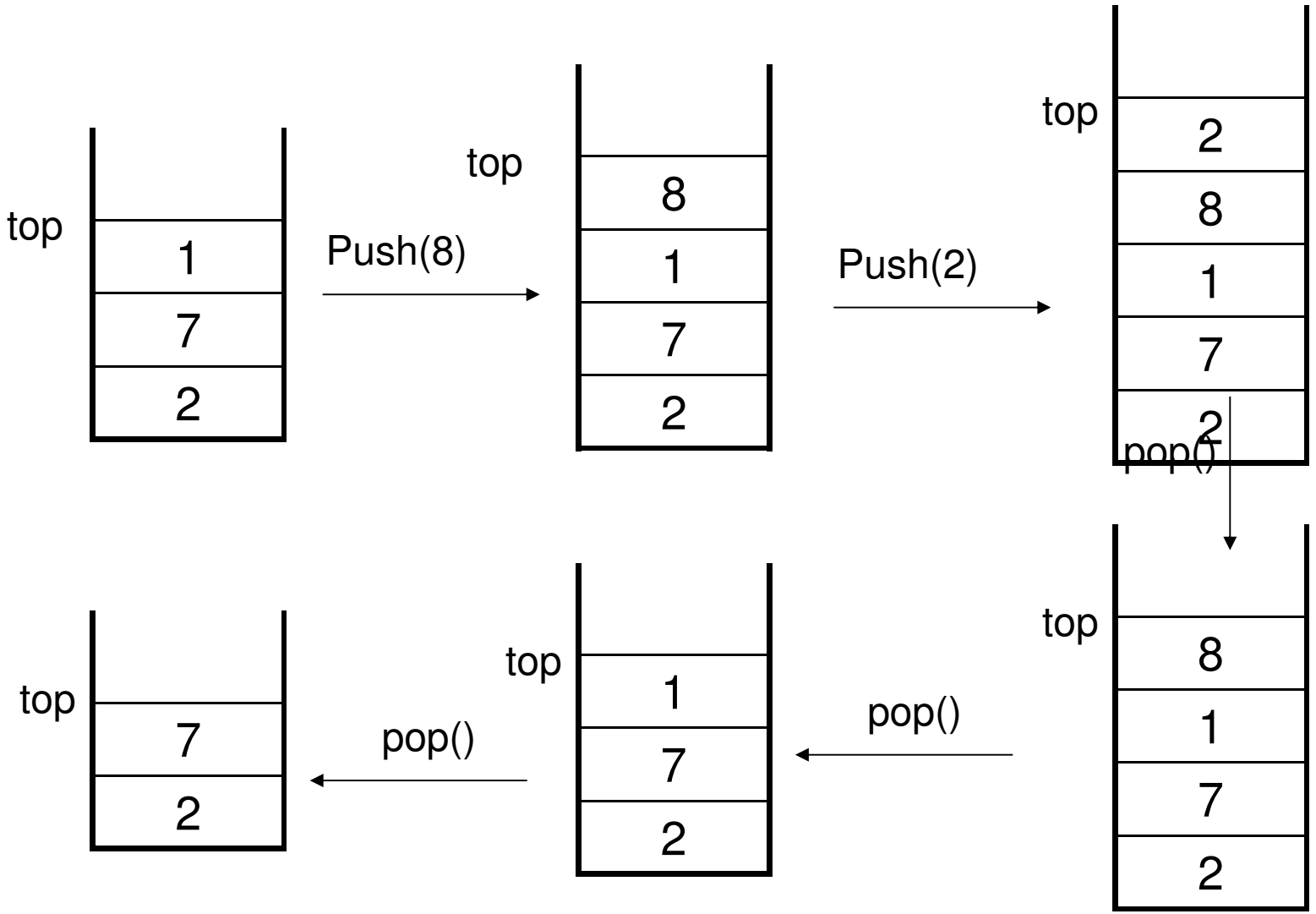
- To remove (pop) an item from the stack, it must be removed from the top of the stack too.

- Thus, the last element that is pushed into the stack, is the first element to be popped out of the stack.

  i.e., A stack is a Last In First Out (LIFO) data structure



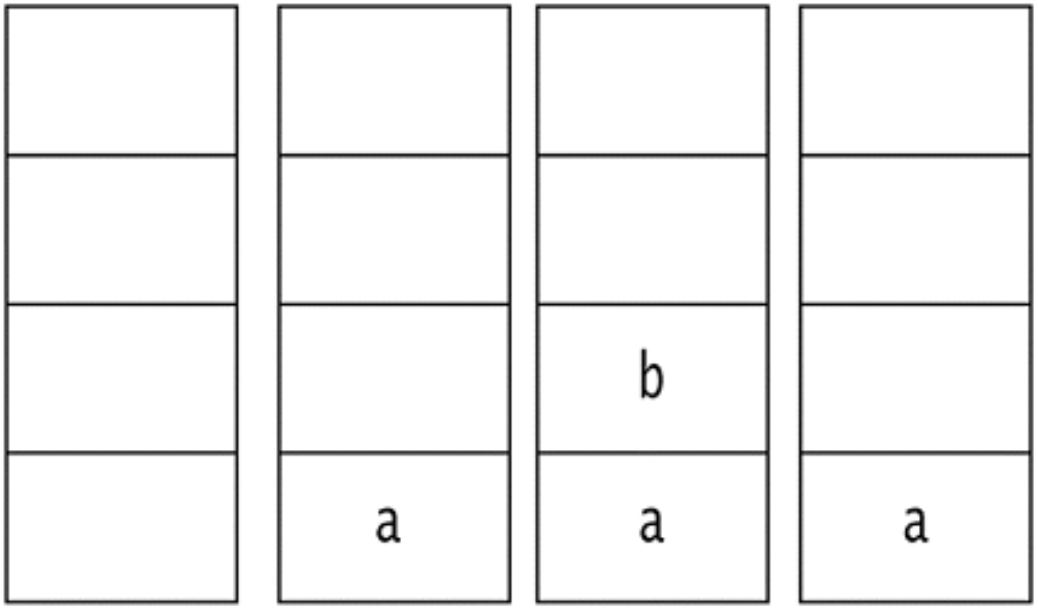
An Example of a Stack



Operations on a stack

- **push**: add an element to the top
- **pop**: remove and return the element at the top
- **Peek (or top)** : return (but not remove) top element, OR retrieves the top item without removing it
  - pop or peek on an empty stack causes an exception
- other operations: isEmpty, size

push(a)
push(b)
pop()



Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Saving local variables when one function calls another, and this one calls another, and so on.
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Array-based Stack

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element

```
Algorithm size()
  return t + 1

Algorithm pop()
  if isEmpty() then
    throw EmptyStackException
   else
    t ← t − 1
    return S[t + 1]
```

Array-based Stack (cont.)

- The array storing the stack elements may become full
- A push operation will then throw a FullStackException
  - Limitation of the array-based implementation
  - Not intrinsic to the Stack ADT

```
Algorithm push(o)
  if t = S.length – 1 then
    throw FullStackException
   else
    t ← t + 1
    S[t] ← o
```

An Array Implementation

- Create a stack using an array by specifying a maximum size *N* for our stack.
- The stack consists of an N-element array *S* and an integer variable *t*, the index of the top element in array S.

- Array indices start at 0, so we initialize *t* to -1

An Array Implementation (2)

- **Pseudo code**

```
Algorithm size()
return t+1

Algorithm isEmpty()
return (t<0)

Algorithm top()
if isEmpty() then
    return Error
return S[t]
```

```
Algorithm push(o)
if size()==N then
    return Error
t=t+1
S[t]=o

Algorithm pop()
if isEmpty() then
    return Error
S[t]=null
t=t-1
```

Performance and Limitations
- array-based implementation of stack ADT

- Performance
  - Let n be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$
- Limitations
  - The maximum size of the stack must be defined *a priori* , and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific exception

Stack Applications

- Postponement: Evaluating arithmetic expressions.

- Prefix: + a b
- Infix: a + b (what we use in grammar school)
- Postfix: a b +

- In high level languages, infix notation cannot be used to evaluate expressions. We must analyze the expression to determine the order in which we evaluate it. A common technique is to convert a infix notation into postfix notation, then evaluating it.

Infix, Postfix and Prefix Expressions

- **INFIX:** From our schools times we have been familiar with the expressions in which operands surround the operator, e.g. x+y, 6*3 etc this way of writing the Expressions is called infix notation.

- **POSTFIX**: Postfix notation are also Known as Reverse Polish Notation (RPN). They are different from the infix and prefix notations in the sense that in the postfix notation, operator comes after the operands, e.g. xy+, xyz+* etc.

- **PREFIX**: Prefix notation also Known as Polish notation. In the prefix notation, as the name only suggests, operator comes before the operands, e.g. +xy, *+xyz etc.

Algorithm for Infix to Postfix Conversion

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
   i) If stack is empty, push operator on stack.
   ii) If the top of stack is opening parenthesis, push operator on stack
   iii) If it has higher priority than the top of stack, push operator on stack.
   iv) Else pop the operator from the stack and output it, repeat step 4
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
7) If there is no more input, pop the remaining operators to output.

Converting from Infix to Postfix

TABLE 5.7
Conversion of `x1 + 2.5 * count / 3`

Next Token	Action	Effect on operatorStack	Effect on postfix
`x1`	Append `x1` to postfix.		`x1`
`+`	The stack is empty Push + onto the stack	+	`x1`
`2.5`	Append 2.5 to postfix	+	`x1 2.5`
`*`	precedence(*) > precedence(+), Push * onto the stack	* +	`x1 2.5`
`count`	Append `count` to postfix	* +	`x1 2.5 count`
`/`	precedence(/) equals precedence(*) Pop * off of stack and append to postfix	+	`x1 2.5 count *`

Converting from Infix to Postfix (Cont…)

TABLE 5.7
Conversion of `x1 + 2.5 * count / 3` *(continued)*

Next Token	Action	Effect on operatorStack	Effect on postfix
/	precedence(/) > precedence(+), Push / onto the stack	/ +	x1 2.5 count *
3	Append 3 to postfix	/ +	x1 2.5 count * 3
End of input	Stack is not empty, Pop / off the stack and append to postfix	+	x1 2.5 count * 3 /
End of input	Stack is not empty, Pop + off the stack and append to postfix		x1 2.5 count * 3 / +

Infix to Postfix Example # 2

A + B * C – D / E

Infix	**Stack(bot->top)**	**Postfix**
a) A + B * C – D / E		
b) + B * C – D / E	A	
c) B * C – D / E	+	A
d) * C – D / E	+	A B
e) C – D / E	+ *	A B
f) – D / E	+ *	A B C
g) D / E	–	A B C * +
h) / E	–	A B C * + D
i) E	– /	A B C * + D
j) – /	A B C * + D E	
k)	A B C * + D E / –	

Infix to Postfix Example # 3

A * B – (C + D) + E

	Infix	**Stack (bot->top)**	**Postfix**
a)	A * B – (C – D) + E	empty	empty
b)	* B – (C + D) + E	empty	A
c)	B – (C + D) + E	*	A
d)	– (C + D) + E	*	A B
e)	– (C + D) + E	empty	A B *
f)	(C + D) + E	–	A B *
g)	C + D) + E	– (	A B *
h)	+ D) + E	– (	A B * C
i)	D) + E	– (+	A B * C
j)	) + E	– (+	A B * C D
k)	+ E	–	A B * C D +
l)	+ E	empty	A B * C D + –
m)	E	+	A B * C D + –
n)		+	A B * C D + – E
o)		empty	A B * C D + – E +

Infix to Postfix Example # 4

- Consider the following infix expression into postfix expression
- A + (B * C - (D / E $ F) * G) * H

A	(	A
+	(+	A
(	(+ (	A
B	(+ (	A B
*	(+ (*	A B
C	(+ (*	A B C
-	(+ (-	A B C *
(	(+ (- (	A B C *
D	(+ (- (	A B C * D
/	(+ (- (/	A B C * D
E	(+ (- (/	A B C * D E
$	(+ (- (/$	A B C * D E
F	(+ (- (/ $	A B C * D E F
)	(+ (-	A B C * D E F $ /
*	(+ (- *	A B C * D E F $ / G
G	(+ (- *	A B C * D E F $ / G * -
)	(+	A B C * D E F $ / G * -
*	(+ *	A B C * D E F $ / G * - H
H	(+ *	A B C * D E F $ / G * - H * +

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*+

Evaluation a postfix expression

- Each operator in a postfix string refers to the previous two operands in the string.
- Suppose that each time we read an operand we **push** it into a stack.
- When we reach an operator, its operands will then be top two elements on the stack. We can then **pop** these two elements,
- **Pop** the top element in B, then **pop** the next element in A and perform the indicated operation on them **(A operator B)**, and **push** the result on the stack.
- So that it will be available for use as an operand of the next operator.

Evaluating Postfix Expressions (Repeat)

Algorithm for method eval

```
1.  Create an empty stack of integers.
2.  while there are more tokens
3.      Get the next token.
4.      if the first character of the token is a digit
5.          Push the integer onto the stack.
6.      else if the token is an operator
7.          Pop the right operand off the stack
8.          Pop the left operand off the stack.
9.          Evaluate the operation.
10.         Push the result onto the stack.
11. Pop the stack and return the result.
```

Evaluating Postfix Expressions (continued)

TABLE 5.6
Evaluating a Postfix Expression

Expression	Action	Stack
`5 6 * 10 -` `↑`	Push 5	5
`5 6 * 10 -` `  ↑`	Push 6	6 5
`5 6 * 10 -` `    ↑`	Pop 6 and 5 Evaluate 5 * 6 Push 30	30
`5 6 * 10 -` `      ↑`	Push 10	10 30
`5 6 * 10 -` `        ↑`	Pop 10 and 30 Evaluate 30 – 10 Push 20	20
`5 6 * 10 -` `          ↑`	Pop 20 Stack is empty Result is 20	

EXERCISE

- 6 2 3 + - 3 8 2 / + * 2 $ 3 +

symb	Op1	Op2	value	Stk value
6				6
2				6 2
3				6 2 3
+	2	3	5	6 5
-	6	5	1	1
3	6	5	1	1 3
8	6	5	1	1 3 8
2	6	5	1	1 3 8 2
/	8	2	4	1 3 4
+	3	4	7	1 7
*	1	7	7	7
2	1	7	7	7 2
$	7	2	49	49
3	7	2	49	49 3
+	49	3	52	52

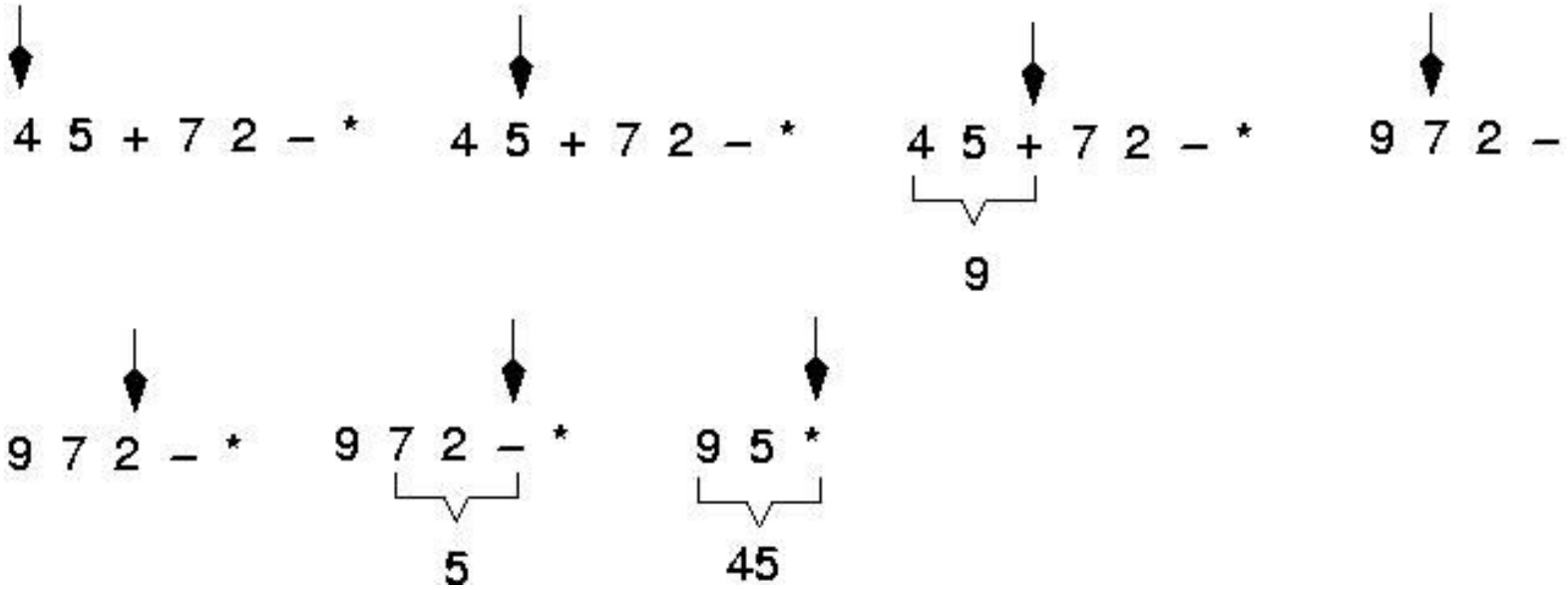
Postfix Evaulation

```
Operand: push
Operator: pop 2 operands, do the math, pop result
          back onto stack
```

1 2 3 + *

Postfix	Stack(bot -> top)
a) 1 2 3 + *	
b) 2 3 + *	1
c) 3 + *	1 2
d) + *	1 2 3
e) *	1 5 // 5 from 2 + 3
f)	5 // 5 from 1 * 5

Example: postfix expressions (cont.)

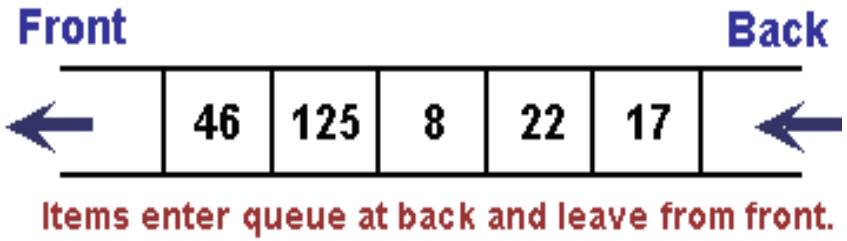


DEFINITION OF QUEUE

- A **Queue** is an ordered collection of items from which items may be deleted at one end (called the ***front*** of the queue) and into which items may be inserted at the other end (the ***rear*** of the queue).
- The first element inserted into the queue is the first element to be removed. For this reason a queue is sometimes called a **FIFO** (first-in first-out) list as opposed to the stack, which is a ***LIFO*** (last-in first-out).

Queue - ADT operations

- Queues are linear data structures in which we add elements to one end and remove them from the other end.



- The first item to be en-queued is the first to be de-queued. Queue is therefore called a First In First Out (FIFO) structure.
- Queue operations: (ADT Methods)

  Enqueue (Insertion) : It adds a new item to the tail / rear of the queue

  Dequeue (Deletion): It deletes the head / front item of the queue, and returns to the caller. If the queue is empty, this operation returns NULL

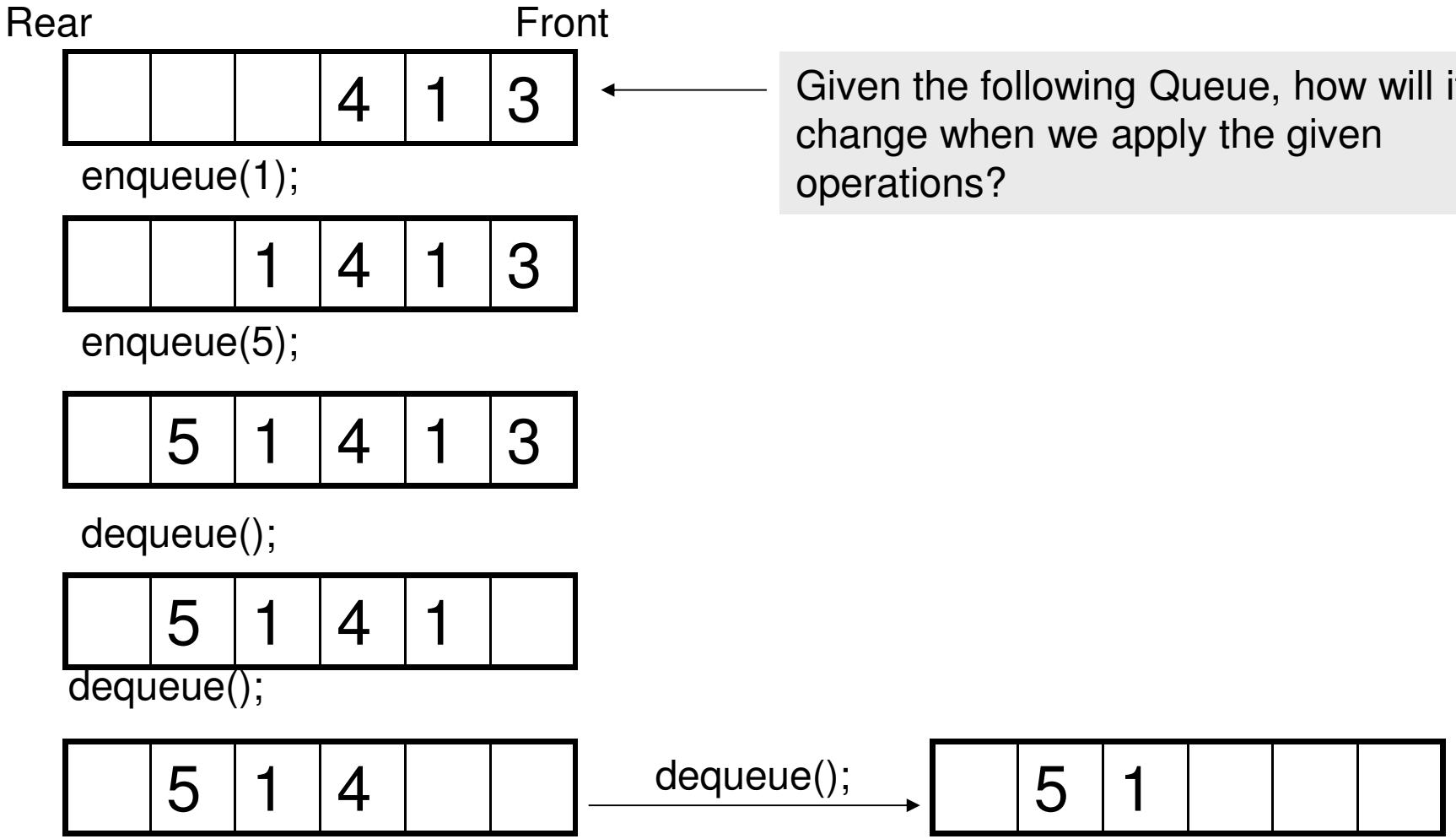
  getHead( ): Returns the value in the front / head element of the queue

  getTail( ): Returns the value in the rear / tail element of the queue

  isEmpty( ): Returns true if the queue has no items

  size( ): Returns the number of items in the queue

What is a queue?



Given the following Queue, how will it change when we apply the given operations?

An Array Implementation

- Pseudo code

```
Algorithm size()
return (r−f)

Algorithm isEmpty()
return (f=NULL)

Algorithm front()
if isEmpty() then
    return Error
return Q[f]

Algorithm isFull()
return (r=MaxSize)
```

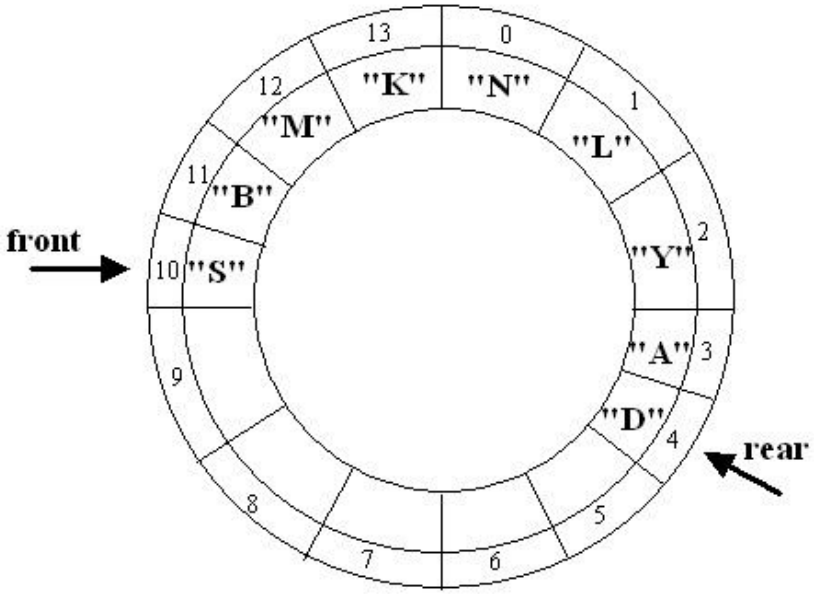
```
Algorithm dequeue()
if isEmpty() then
    return Error
Q[f]=null
f=(f+1)

Algorithm enqueue(o)
if size = N − 1 then
    return Error
Q[r]=o
r=(r +1)
```

Simple array implementation

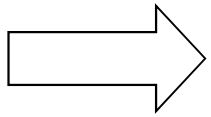
```
#define front 0   //assume front is always at location 0 of data
int  back = 0; int size = 0;
int  data[MAXSIZE] = {0};
void  enqueue(int a) { data[back] = a ; back++; size++ };
int dequeue(void) { int temp = data[front] ;
                    for (i=0 ; i < back ; i++) data[i] = data[i+i] ;
                    back--;
                    return temp;
                   };
int  isEmpty() { return back == 0; };
int  isFull() { back == MAXSIZE; };
```

QueueAsCircularArray Implementation

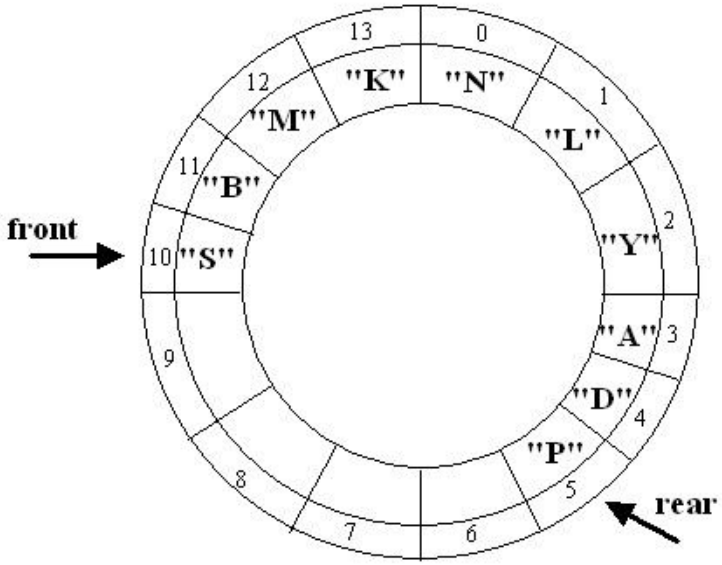
- By using modulo arithmetic for computing array indexes, we can have a queue implementation in which each of the operations enqueue and dequeue has complexity O(1)



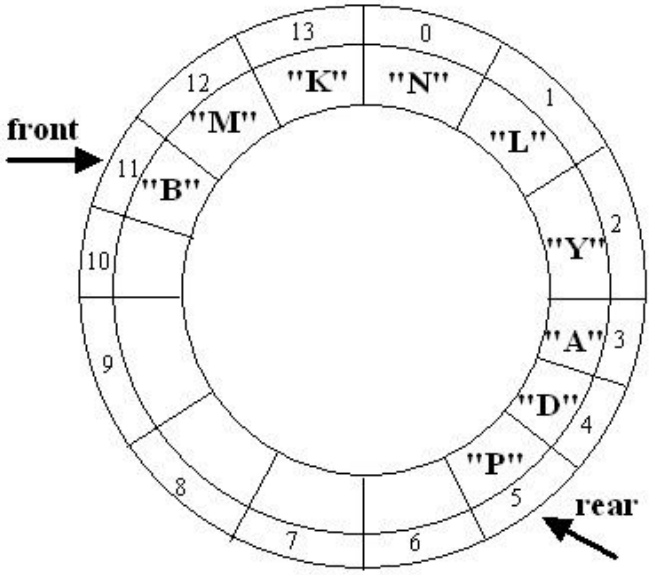
Enqueue("P") will result in …



QueueAsCircularArray Implementation (Cont.)

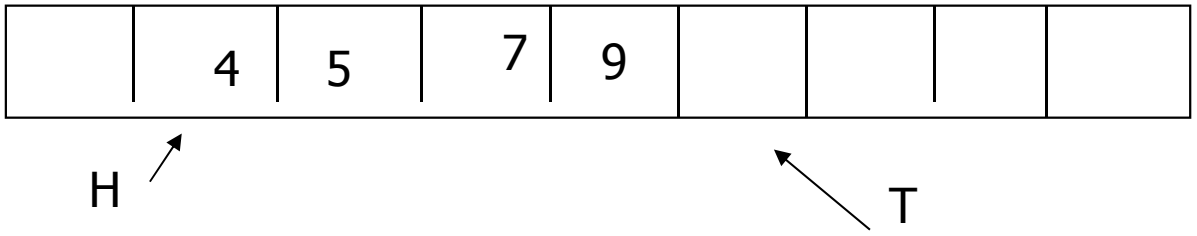


Dequeue() will result in



Array Implementation: Circular Queue

- When a new item is inserted at the rear, the pointer to rear moves upwards.
- Similarly, when an item is deleted from the queue the front arrow moves downwards.
- After a few insert and delete operations the rear might reach the end of the queue and no more items can be inserted although the items from the front of the queue have been deleted and there is space in the queue.



Array Implementation: Circular Queue

- To solve this problem, queues implement wrapping around. Such queues are called Circular Queues.
- Def: An implementation of a *bounded queue* *(A queue limited to a fixed number of items).* using an *array*.
- Both the front and the rear pointers wrap around to the beginning of the array.
- It is also called as “Ring buffer”.
- Items can inserted and deleted from a queue in O(1) time.

Insertion of Circular Queue

1. If (FRONT = 1 and REAR = N) or (FRONT = REAR + 1) then
   Write Overflow and Return
2. If (FRONT=NULL) then
   Set FRONT =1 and REAR = 1
   Else if (REAR = N) then
   Set REAR = 1
   Else
   Set REAR = REAR + 1
3. Set Queue[REAR] = item
4. Return

Deletion of Circular Queue

1. If (FRONT = NULL) then
   Write : UNDERFLOW & Return().
2. Set Item = Queue[FRONT]
3. If (FRONT = REAR) then
   Set FRONT = NULL and REAR = NULL
   Else if FRONT = N then
   Set FRONT = 1
   Else Set FRONT = FRONT + 1
4. Return

Applications

- discrete event simulation -
- digital delay line
- task scheduling in an operation system
- staging area for data acquisition systems
- I/O buffers
- print queues
- sorting

Application of Queues

- Direct applications
  - Waiting lines: Queues are commonly used in systems where waiting line has to be maintained for obtaining access to a resource. For example, an operating system may keep a queue of processes that are waiting to run on the CPU.
  - Access to shared resources (e.g., printer)
  - Multiprogramming

- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Priority Queues

- In a normal queue the enqueue operation add an item at the back of the queue, and the dequeue operation removes an item at the front of the queue.

- A priority queue is a queue in which the dequeue operation removes an item at the front of the queue; but the enqueue operation insert items according to their priorities.

- A higher priority item is always enqueued before a lower priority element.

- An element that has the same priority as one or more elements in the queue is enqueued after all the elements with that priority.

Double-Ended Queue

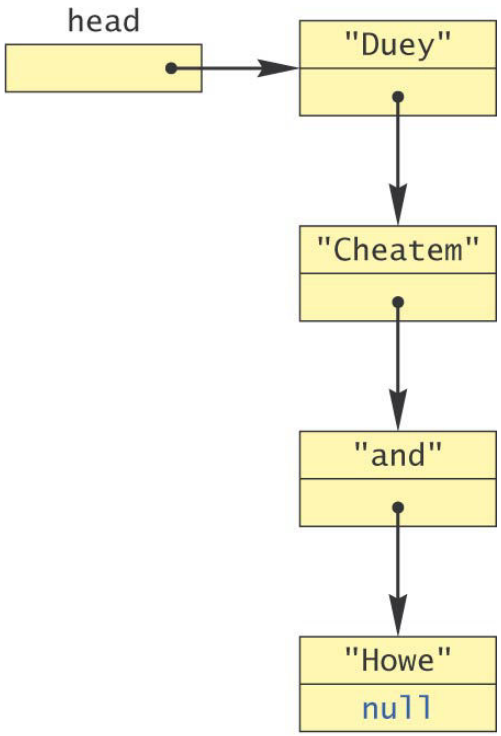
- **A double-ended queue**, or **deque**, supports insertion and deletion from the front and back
- The deque supports following basic methods
  - **InsertFirst** - Inserts *e* at the beginning of deque
  - **InsertLast** - Inserts *e* at end of deque
  - **RemoveFirst** – Removes the first element
  - **RemoveLast** – Removes the last element

Linked Lists

- Options for implementing an ADT List
  - Array has a fixed size
    - Data must be shifted during insertions and deletions
  - Linked list is able to grow in size as needed
    - Does not require the shifting of items during insertions and deletions

- A *linked data structure* is a collection of objects (called *nodes),* each containing data and a (potential) reference to (at least) one other node.

A Linked List



Display 10.3

A Linked List

- *Links*, shown as arrows are implemented as references and are instance variables of the node type.
- The reference marked head is a variable of the node type which provides access to the first node in the linked list
- Each node is an object of a class that has (at least) two instance variables:
  - the data
  - the link.

- A link instance variable with the value `null` indicates the last node.

Preliminaries

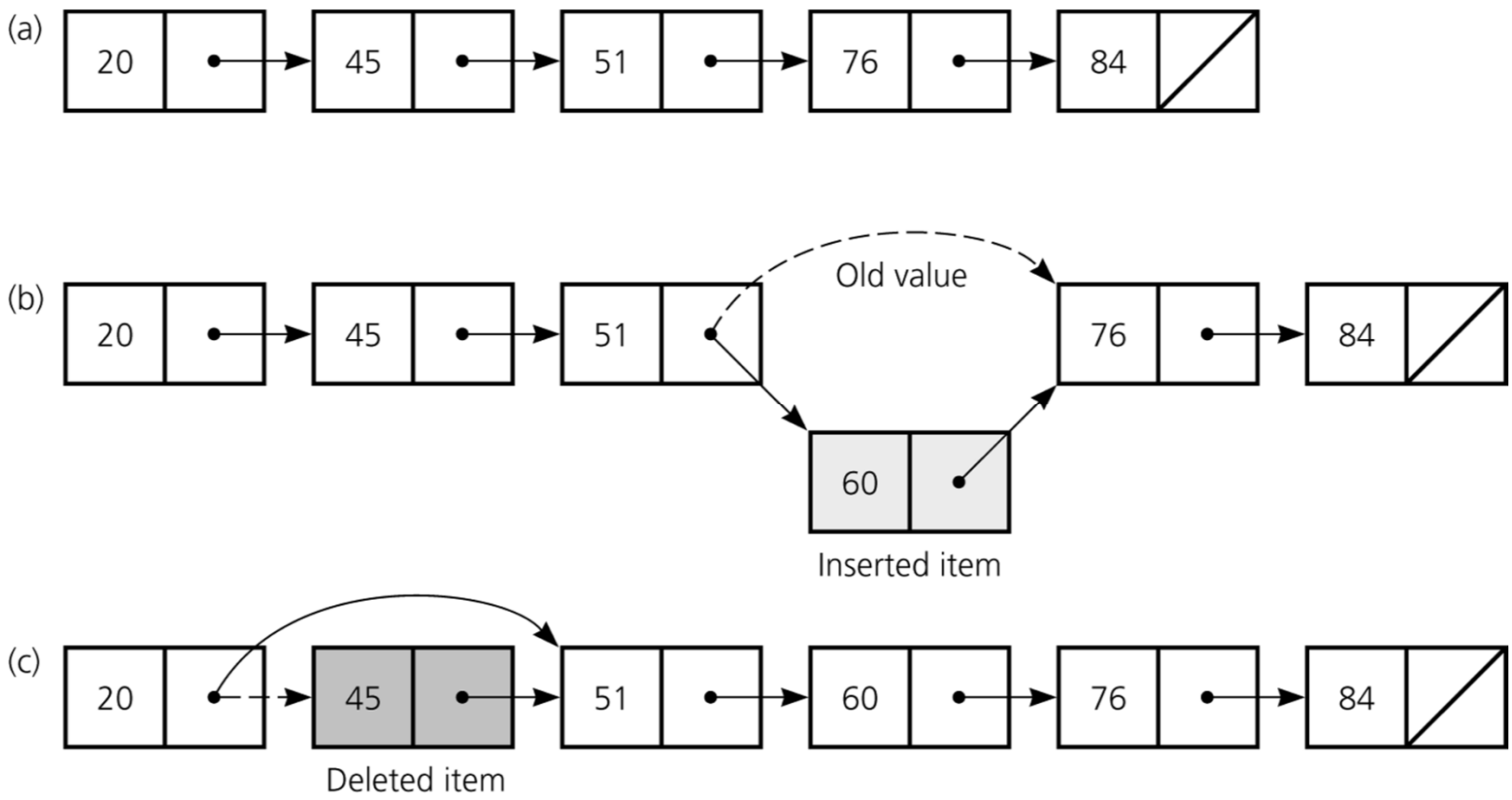


Figure a) A linked list of integers; b) insertion; c) deletion

Thanks