Data Structures and Algorithms

Chapter 2 : Linear Data Structures

Fundamental data structures

Linear Data Structures

- list
  - array
  - linked list
  - string
- stack
- queue
  - priority queue/heap

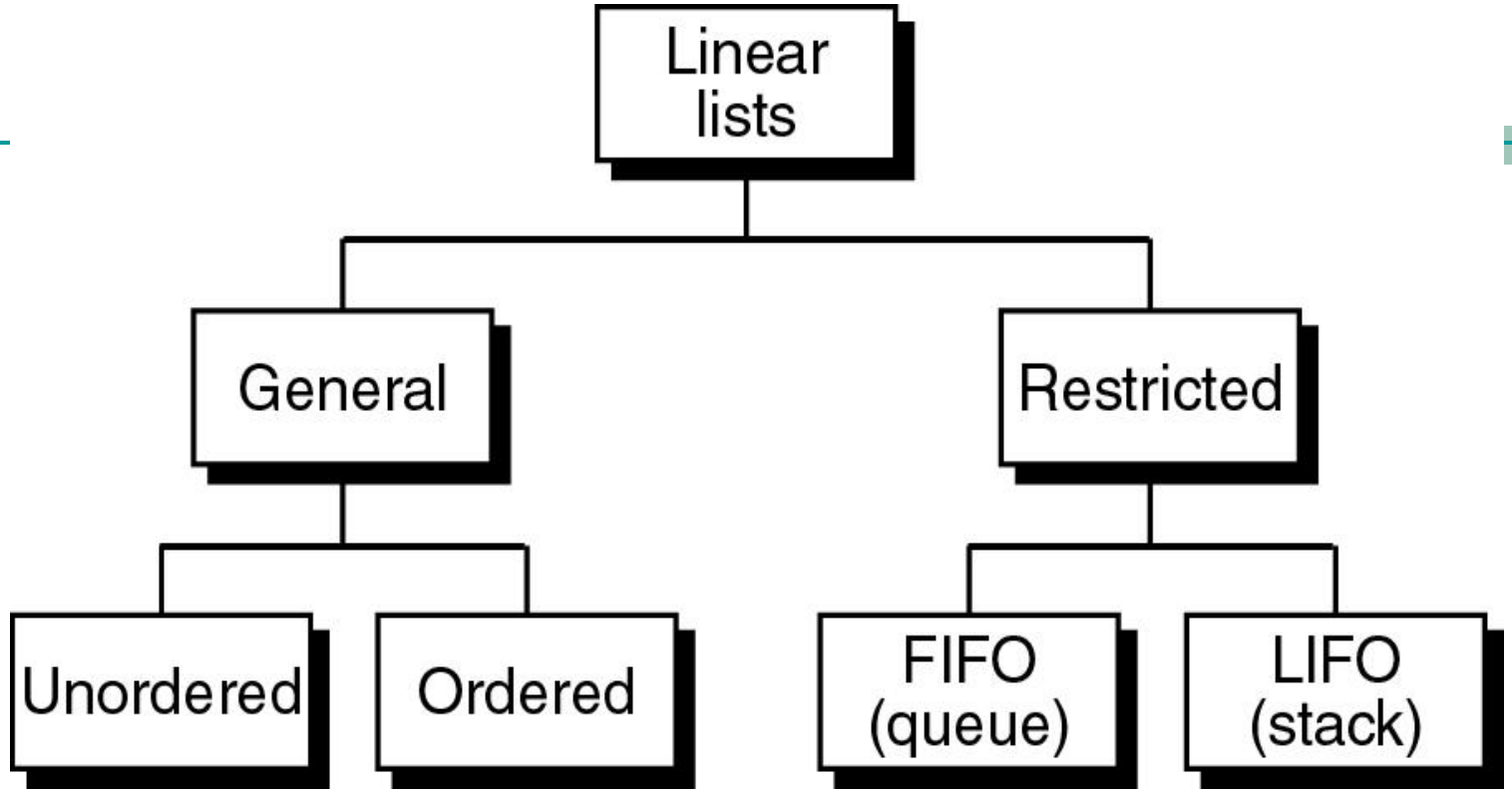
Non-Linear Data Structures

- graph
- tree and binary tree
- set and dictionary

Lists

- A *list* is a **sequence of zero or more data items**.
- The total number of items is said to be the ***length*** of the list.
- The length of a given list **can grow and shrink** on demand.
- The items can be **accessed**, **inserted**, or **deleted** at any position in a list.

Linear Lists



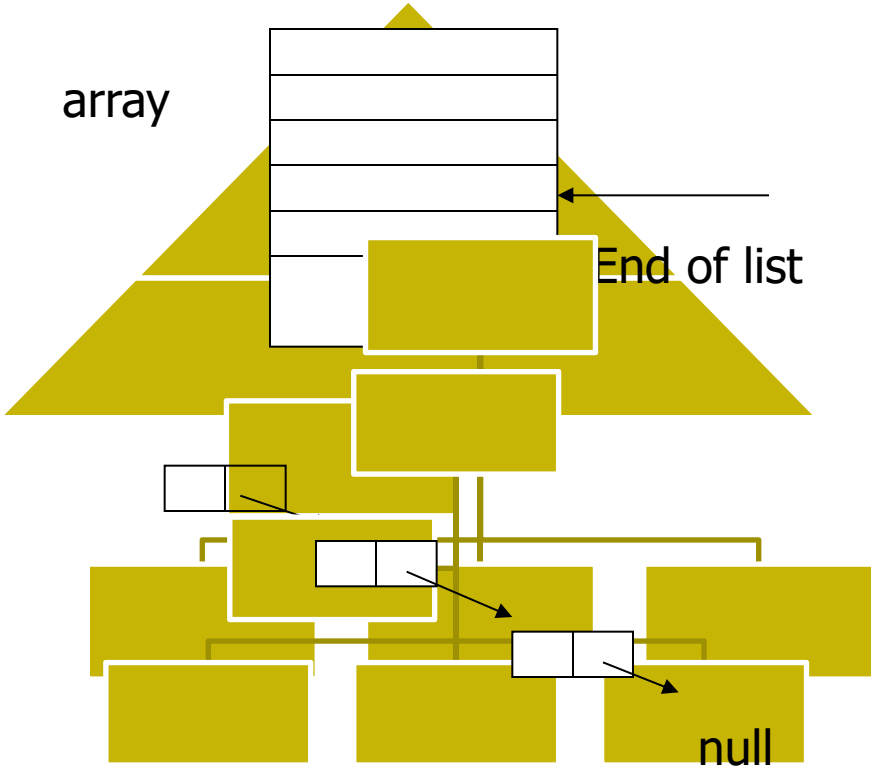
Operations are;

1. Insertion
2. Deletion
3. Retrieval
4. Traversal (exception for restristed lists).

Two ways to implement lists

There are two basic structures we can use to implement an ADT list:

- Array implementation
- Linked list implementation



Array

- **Collection of data elements**
  - A **fixed-size sequence** of elements, all of the **same type**
  - fixed length (need preliminary reservation of memory)
  - contiguous memory locations
  - direct access
  - Insert/delete
- **Basic Operations**
  - **Direct access** to each element in the array by **specifying its position** so that values can be retrieved from or stored in that position

Array Implementations

- In an array, the **sequentiality of a list is maintained by the order structure** of elements in the array (indexes).
- **Searching** an array for an individual element can be **very efficient**
- However, **insertion and deletion of elements are complex and inefficient** processes.

Different types of Array

- **One-dimensional array**: only one index is used
- **Multi-dimensional array**: array involving more than one index

- **Static array**: the compiler determines how memory will be allocated for the array
- **Dynamic array**: memory allocation takes place during execution

Example of array

- Consider arrays a,b,c,d to store collection of 10 integers declared by:

```
int capacity=10
int       a[capacity],
          b[capacity]={1,2,3,4,5,6,7,8,9,10},
          c[capacity]={1,2,3},
          d[capacity]={0};

char name[capacity]="John Doe";
```

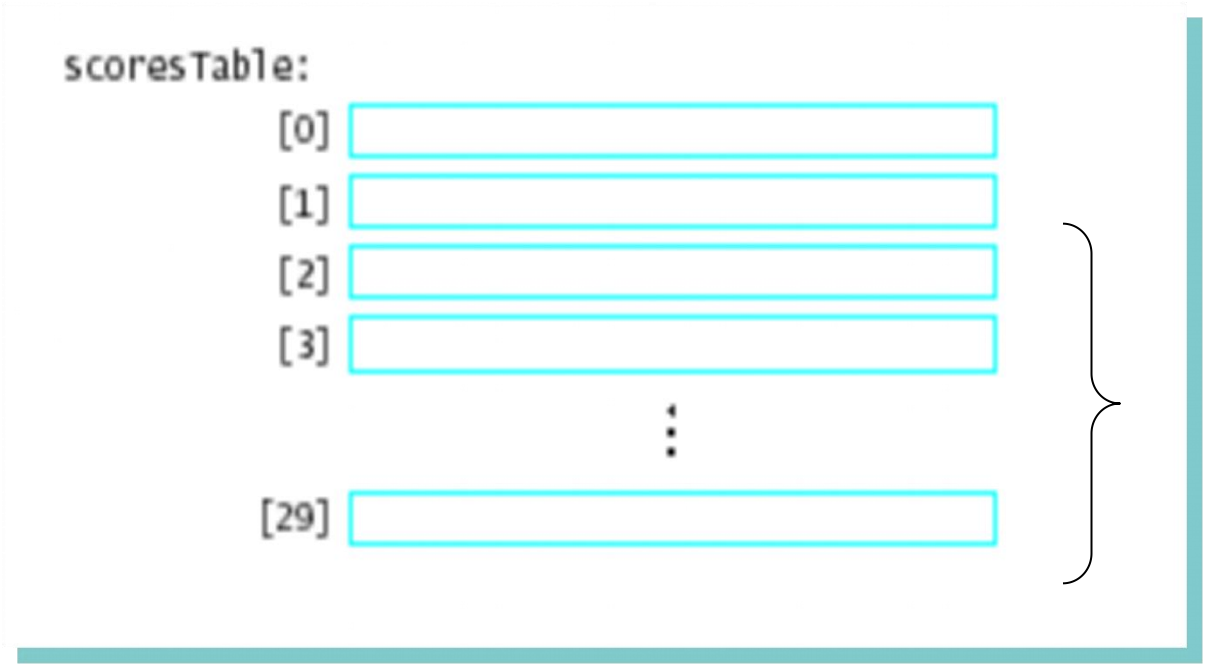
Multidimensional Arrays

- Consider a table of test scores for several different students

	Test 1	Test 2	Test 3	Test 4	Test 5
Student 1	99.0	93.5	89.0	91.0	97.5
Student 2	66.0	68.0	84.5	82.0	87.0
Student 3	88.5	78.5	70.0	65.0	66.5
⋮	⋮	⋮	⋮	⋮	⋮
Student 30	100.0	99.5	100.0	99.0	98.0

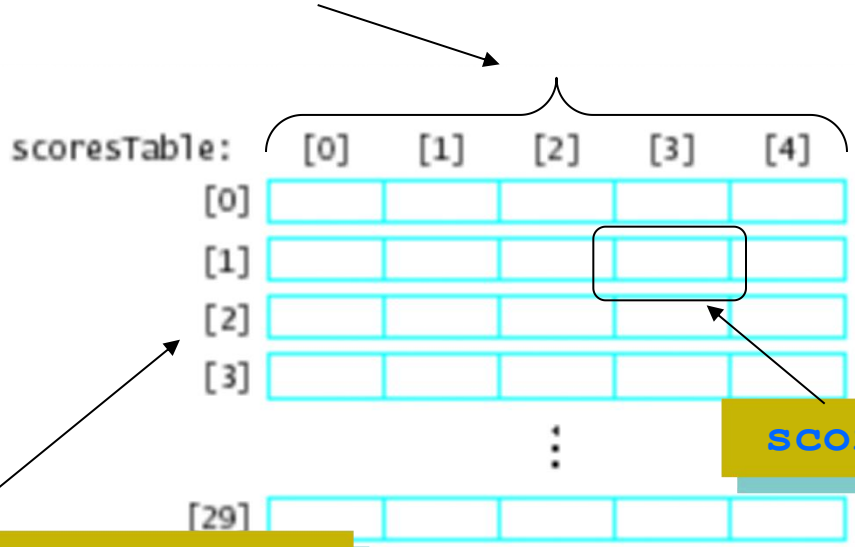
Array of Arrays

- An array of arrays
  - An **array whose elements are other arrays**



Array of Array Declarations

- Each of the rows is itself a one dimensional array of values



`scoresTable [1][3]`

`scoresTable[2]`

is the whole row numbered 2

Multidimensional Arrays

- Syntax:

```
ElementType arrayName [num_rows][num_columns];
```

int scoretable [num_students][num_tests];

int scoretable[2][5]={{80,80,80,80,80},{60,60,60,60,60}};

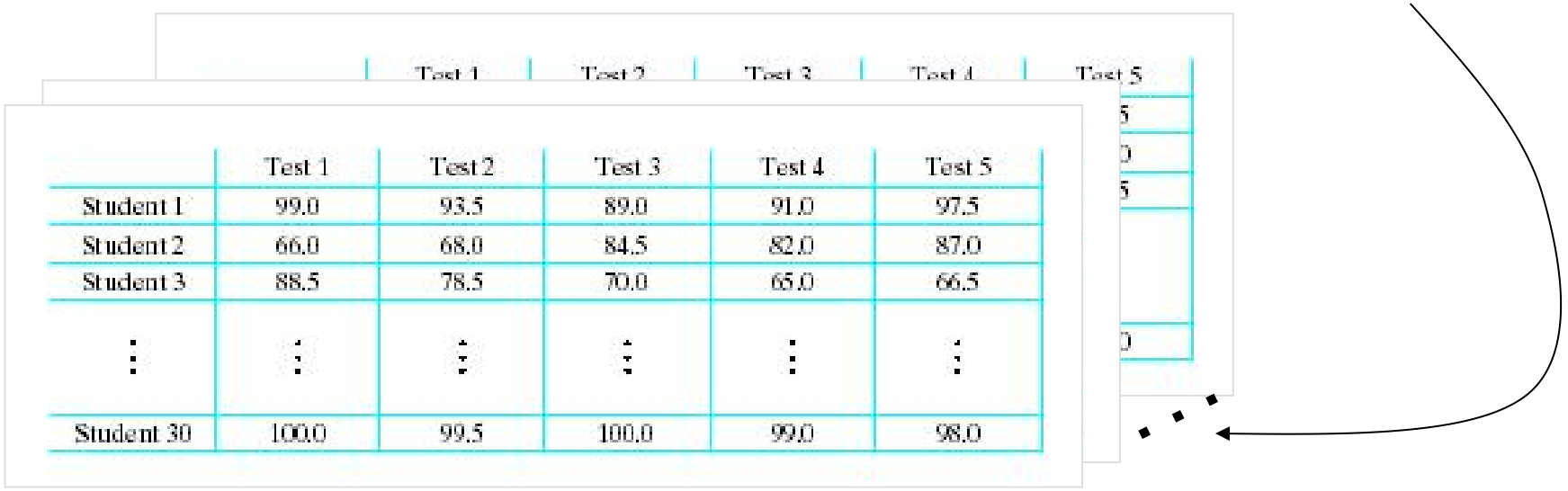
If you want to change the score of first student's 3rd test score to 100, you just need to do:

Scoretable[0][2]=100

Multidimensional Arrays

- Consider multiple pages of the student grade book

```
typedef double
   ThreeDimArray[NUM_ROWS][NUM_COLS][NUM_RANKS];
```



	Test 1	Test 2	Test 3	Test 4	Test 5
Student 1	99.0	93.5	89.0	91.0	97.5
Student 2	66.0	68.0	84.5	82.0	87.0
Student 3	88.5	78.5	70.0	65.0	66.5
⋮	⋮	⋮	⋮	⋮	⋮
Student 30	100.0	99.5	100.0	99.0	98.0

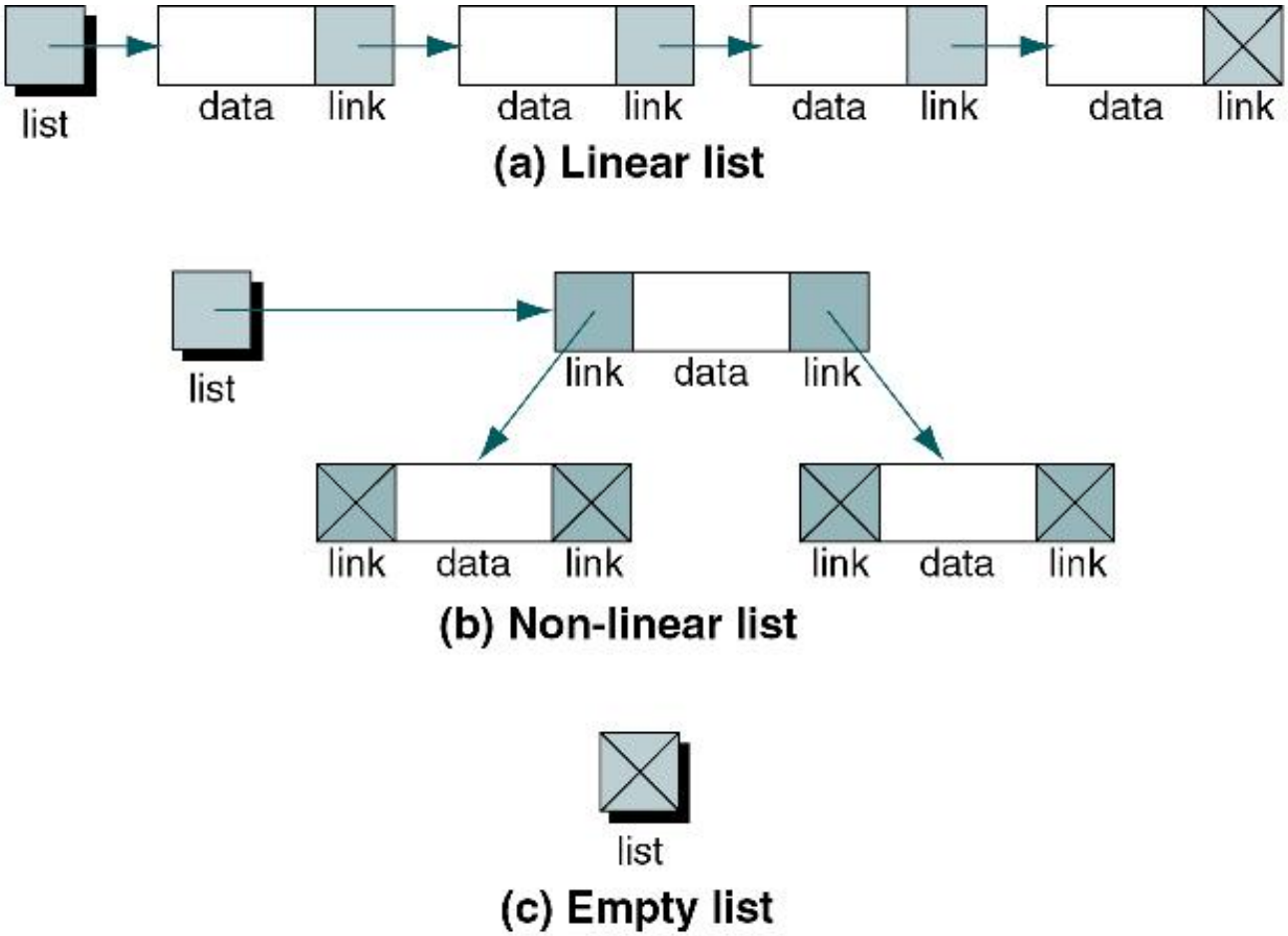
Linked Lists

- A **Linked List** is an ordered collection of data in which **each element contains the location of the next element or elements**.
- In a linked list, each element contains two parts:
  - data and one or more links.
- The data part holds the application data – the data to be processed.
- Links are used to chain the data together. They contain **pointers** that identify the next element or elements in the list.

Linear and non-linear Linked Lists

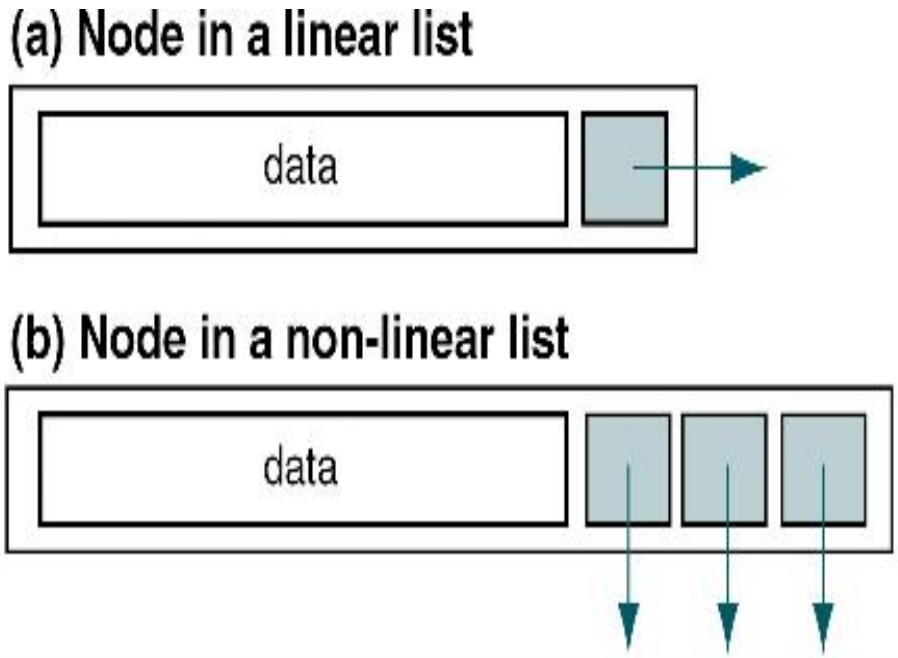
- In linear linked lists, each element has only zero or one successor.
- In non-linear linked lists, each element can have zero, one or more successors.

Linked Lists examples



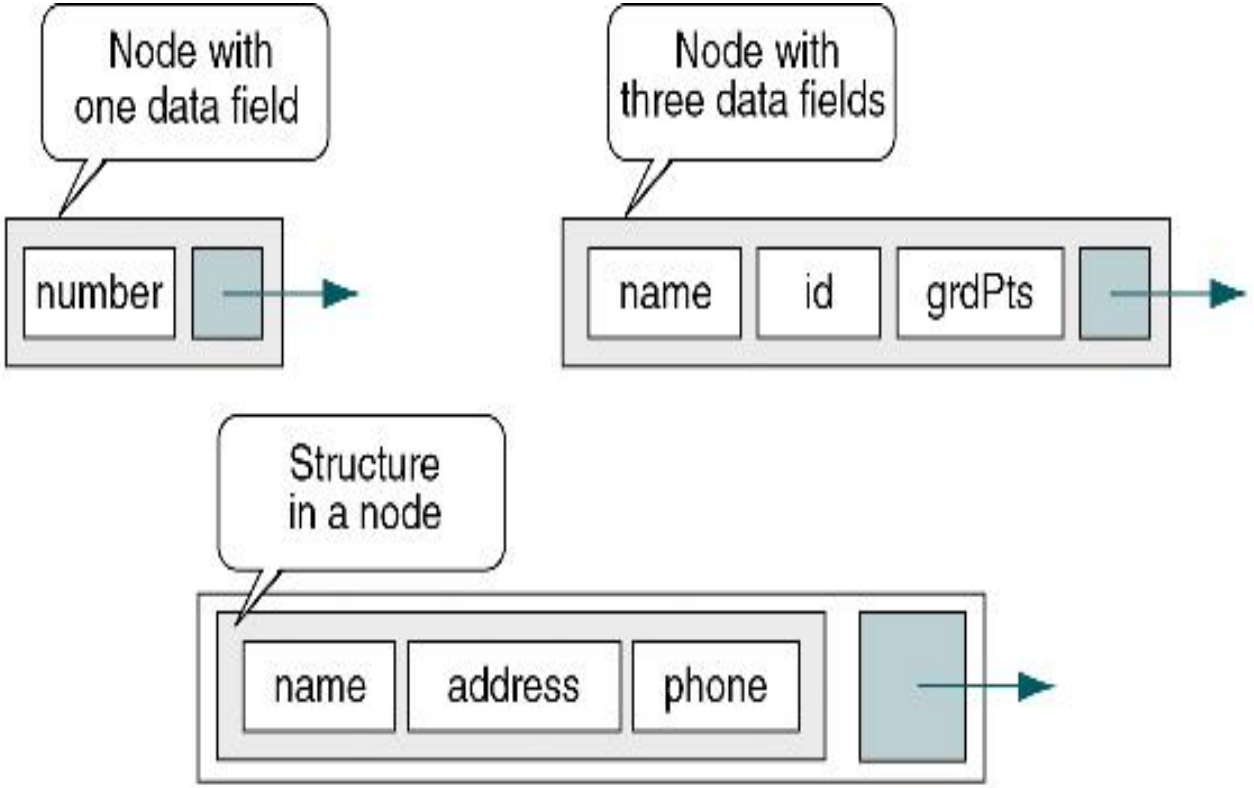
Nodes

- A **node** is a structure that has **two parts**: the **data and one or more links.**
- The nodes in a linked list are called **self-referential structures**.
- In such a structure, each **instance of the structure contains one or more pointers to other instances** of the same structural type.



Nodes

- The **data part** in a node can be a **single field, multiple fields, or a structure** that contains several fields, but it always acts as a single field.



Linked Lists vs. Arrays

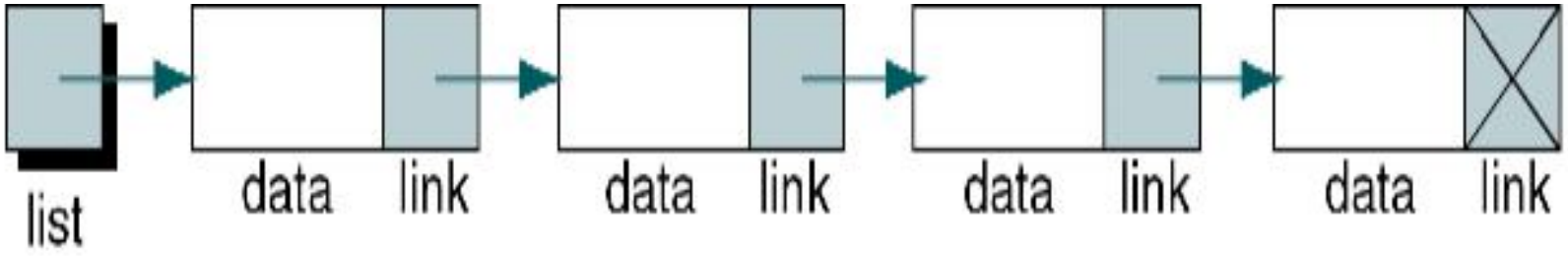
- The major advantage of the linked list over the array is that **data are easily inserted and deleted**.
- It is **not necessary to shift elements of a linked list** to make room for a new elements or to delete an element.
- However, **because the elements are no longer physically sequenced** in a linked list, we are **limited to sequential searches**.

Linked Lists vs. Arrays

- Features
  - dynamic length
  - arbitrary memory locations
  - access by following links
  - Insert/delete
- Use a linked list instead of an array when
  - You have an **unpredictable number of data elements**
  - Your list needs to be **sorted quickly**

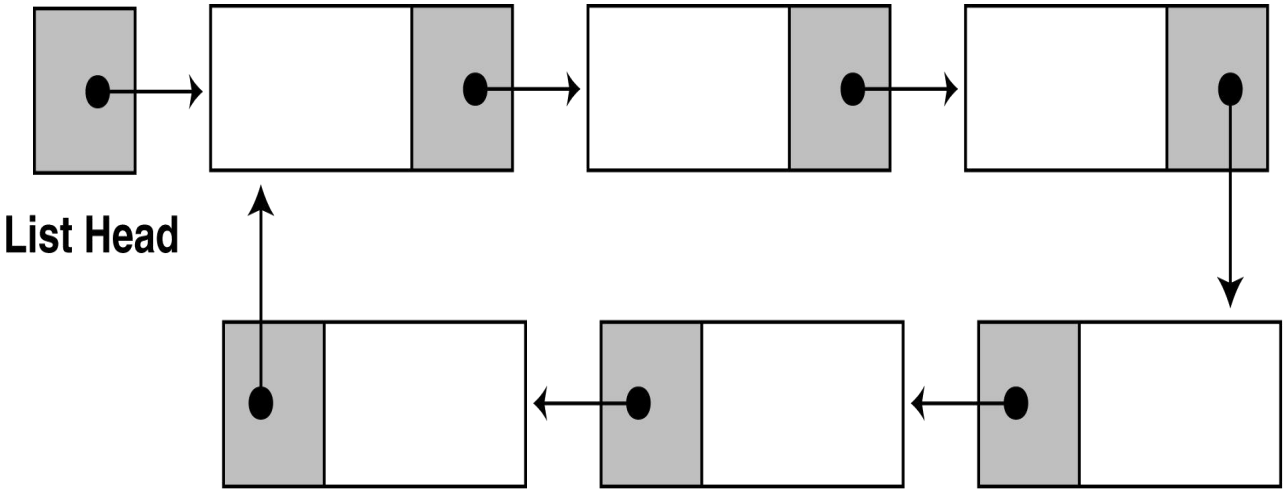
Types of Linked Lists

- **Singly linked** list
  - Begins with a pointer to the first node
  - Terminates with a null pointer
  - Only **traversed in one direction**



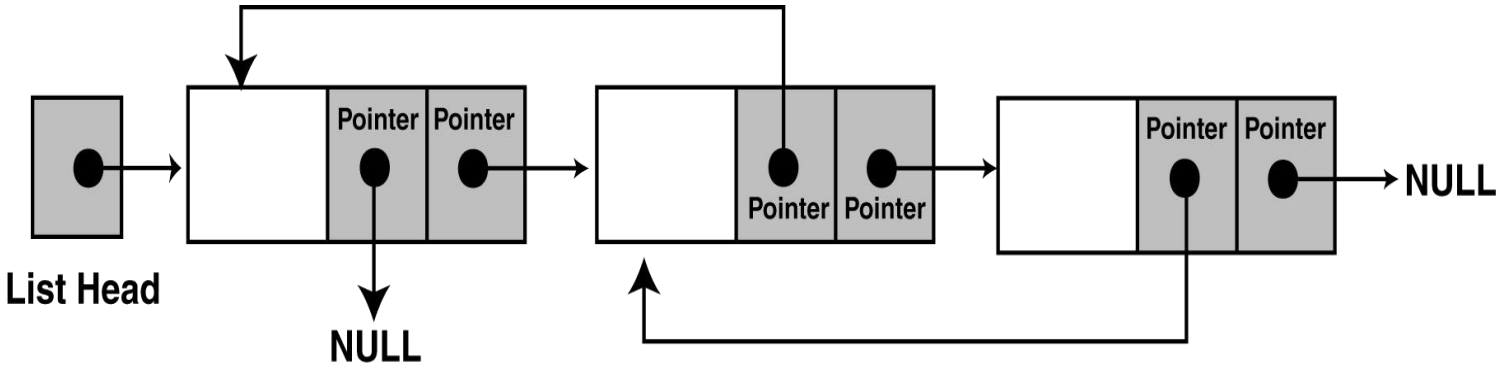
Types of Linked Lists

- **Circular**, **singly** linked
  - **Pointer in the last node points back to the first node**



Types of Linked Lists

- **Doubly linked** list
  - **Two "start pointers"** – first element and last element
  - Each node has a **forward pointer and a backward pointer**
  - Allows **traversals both forwards and backwards**



Types of Linked Lists

- **Circular, doubly** linked list
  - **Forward pointer of the last node points to the first node** and **backward pointer of the first node points to the last node**

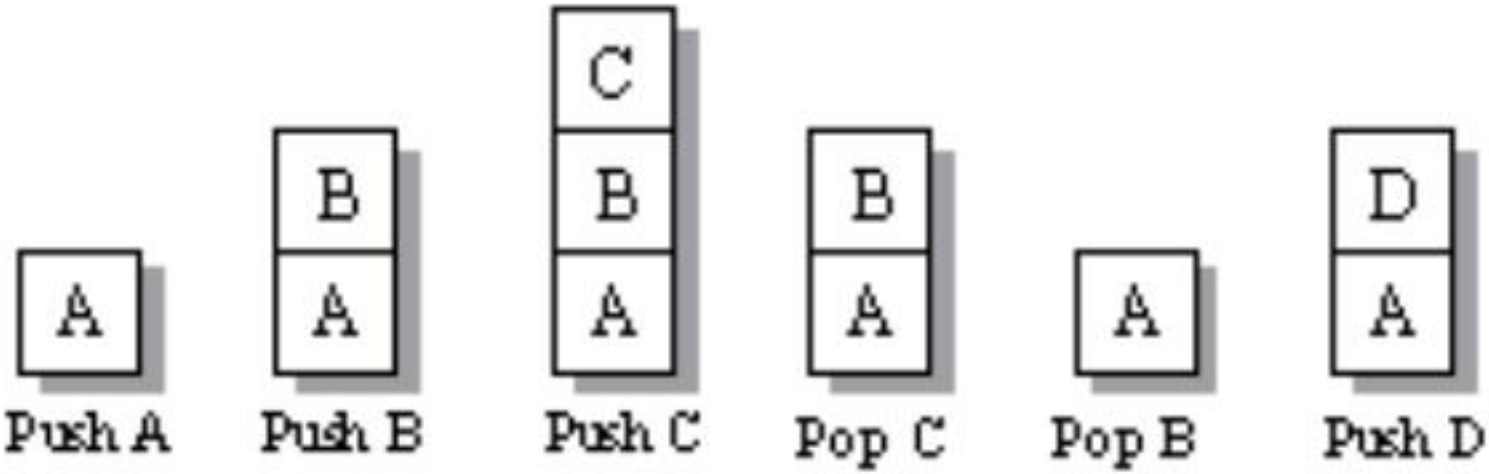
Stacks

- What is a Stack?
  - A stack is a data structure of ordered items such that **items can be inserted and removed only at one end**.
- A stack is a LIFO (Last-In/First-Out) data structure
- A stack is sometimes also called a **pushdown store**

Designing and Building a Stack class

- The basic functions are:
  - **Constructor**: construct an empty stack
  - **Empty**(): Examines whether the stack is empty or not
  - **Push**(): Add a value at the top of the stack
  - **Top**(): Read the value at the top of the stack
  - **Pop**(): Remove the value at the top of the stack
  - **Display**(): Displays all the elements in the stack

Stacks Example



Stack Applications

- There are many applications of stack:
  - Reversing data
  - Parsing Arithmetic Operations
  - Evaluating postfix expressions
  - Pairing data
  - Postponing data usage
  - Backtracking steps
  - Program execution

Parsing Arithmetic Expressions

- **Infix Notation** – operator between operands
  - 2 + 3
  - 4*(3+2) – change of meaning if brackets not there
  - Has brackets
- **Prefix Notation** – operator infront of the operands
  - + 2 3
  - * 4 + 3 2
  - Has no brackets
- **Postfix Notation** – operator after the operands
  - 2 3 +
  - 4 3 2 + *
  - Has no brackets

Converting Infix to Postfix

- Analysis:
  - *Operands* are in same order in infix and postfix
  - *Operators* occur later in postfix, and before in prefix
- Strategy:
  - Send **operands straight to output**
  - Send **higher precedence operators first**
  - If **same precedence, send in left to right order**
  - **Hold pending operators on a stack**

Converting Infix to Postfix - Example

INFIX	POSTFIX
x – y * z	x

INFIX	POSTFIX
x – y * z	x

THE OPERANDS FOR ‘-’ ARE NOT YET

IN POSTFIX, SO ‘-’ MUST BE

TEMPORARILY SAVED SOMEWHERE. (STACK)

INFIX

x – y * z

POSTFIX

xy

INFIX	POSTFIX
x – y * z	xy

THE OPERANDS FOR ‘*’ ARE NOT YET IN POSTFIX, SO ‘*’ MUST BE TEMPORARILY SAVED SOMEWHERE, *AND RESTORED BEFORE ‘-’*.

INFIX

x – y * z

POSTFIX

xyz

INFIX

x – y * z

POSTFIX

xyz* –

Suppose, instead, we started with

x*y-z

After moving ‘x’ to postfix, ‘*’ is temporarily saved, and then ‘y’ is appended to postfix. What happens when ‘-’ is accessed?

INFIX	**POSTFIX**
x * y – z	**xy**

The ‘*’ must be moved to postfix now, because both of the operands for ‘*’ are on postfix.

Then the ‘-’ must be saved temporarily. After ‘z’ is moved to postfix, ‘-’ is moved to postfix, and we are done.

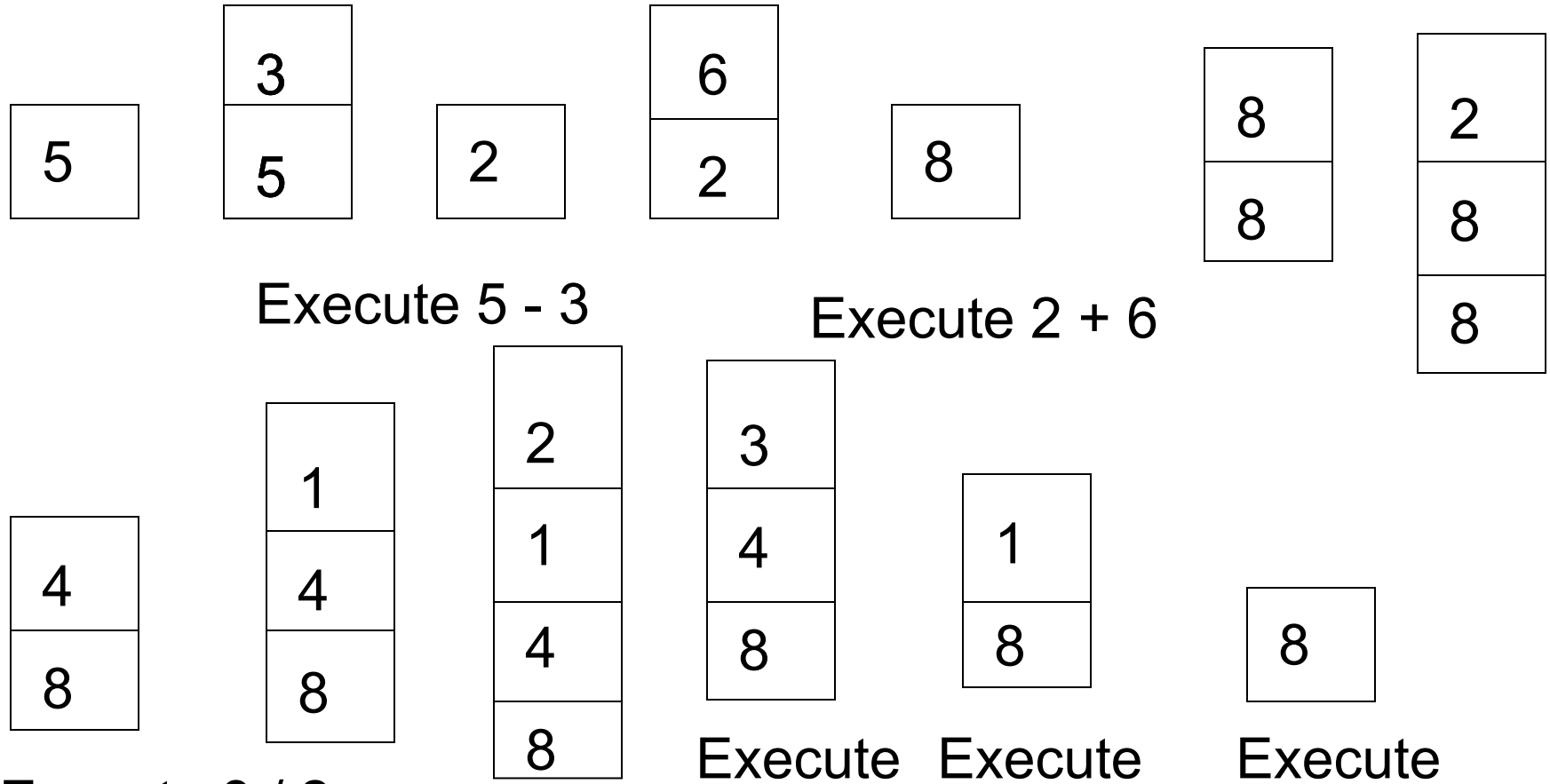
INFIX	POSTFIX
x * y – z	xy*z–

Evaluating expressions

- Whenever an operand is encountered, push onto stack
- Whenever operator is encountered, pop required number of arguments from operand stack and evaluate
- Push result back onto stack

Evaluating expressions - Example

Expression: 5 3 - 6 + 8 2 / 1 2 + - *



Implementing a Stack

- There are two ways we can implement a stack:
  - Using an array
  - Using a linked list

Implementation of the Operations

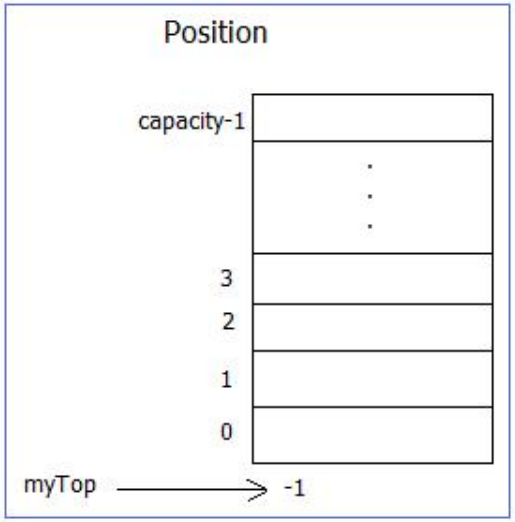
- **Constructor**:

  Create an array: (int) array[capacity]

  Set myTop = -1

- **Empty**():

  check if myTop == -1



Implementation of the Operations

- **Push(int** x):

  if array is not FULL (myTop < capacity-1)

  myTop++

  store the value x in array[myTop]

  else

  output “out of space”

Implementation of the Operations

- **Top**():

  If the stack is not empty

  return the value in array[myTop]

  else:

  output “no elements in the stack”

Implementation of the Operations

- **Pop**():

  If the stack is not empty

  myTop -= 1

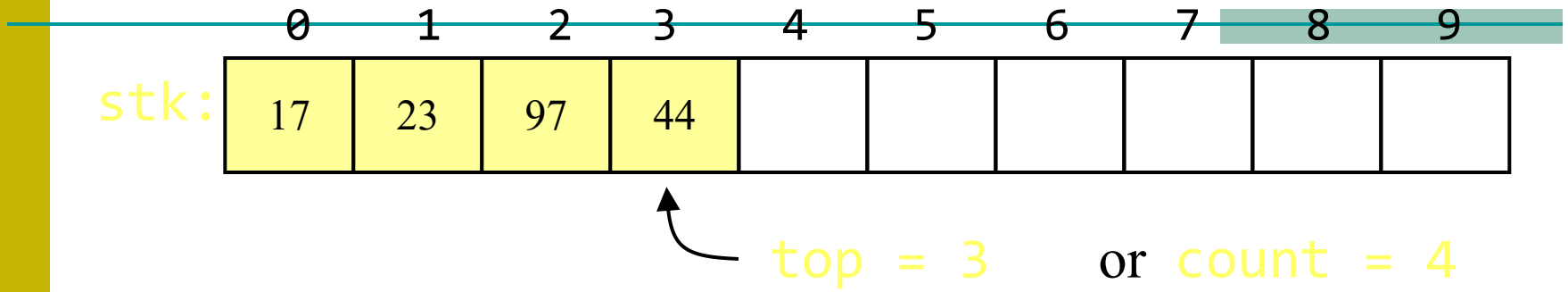
  else:

  output “no elements in the stack”

Array implementation of stacks

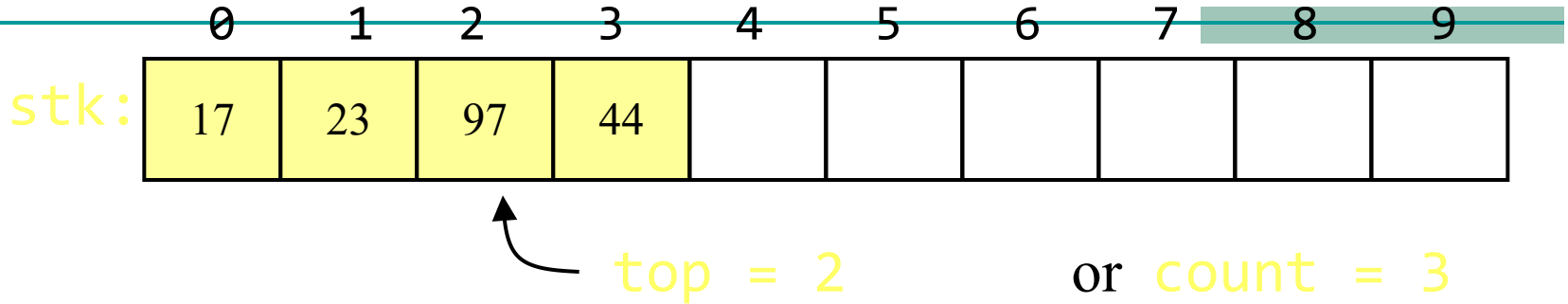
- To implement a stack, items are inserted and removed at the same end (called the top)
- Efficient array implementation requires that the top of the stack be towards the center of the array, not fixed at one end
- To use an array to implement a stack, you need both the array itself and an integer
  - The integer tells you either:
    - Which location is currently the top of the stack, or
    - How many elements are in the stack

Pushing and popping



- If the bottom of the stack is at location `0`, then an empty stack is represented by `top = -1` or `count = 0`
- To add (push) an element, either:
  - Increment `top` and store the element in `stk[top]`, or
  - Store the element in `stk[count]` and increment `count`
- To remove (pop) an element, either:
  - Get the element from `stk[top]` and decrement `top`, or
  - Decrement `count` and get the element in `stk[count]`

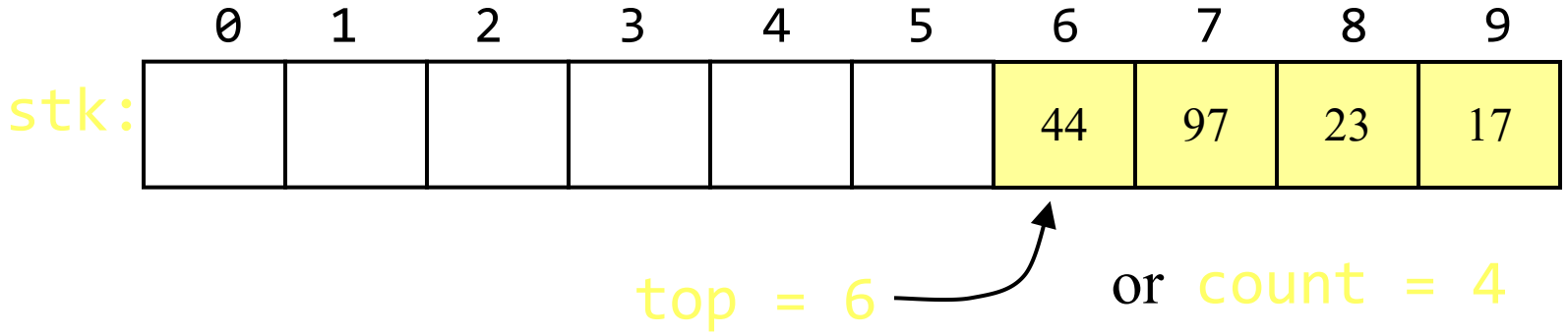
After popping



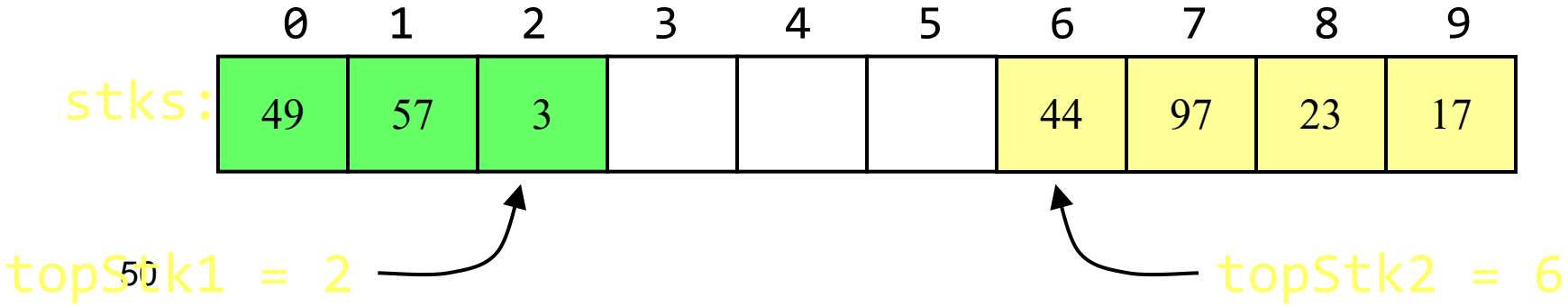
- When you pop an element, do you just leave the "deleted" element sitting in the array?
- The surprising answer is, *"it depends"*
  - If this is an array of primitives, *or* if you are programming in C or C++, *then* doing anything more is just a waste of time
  - If you are programming in Java, and the array contains objects, you should set the "deleted" array element to `null`
  - Why? To allow it to be garbage collected!

Sharing space

- Of course, the bottom of the stack could be at the *other* end



- Sometimes this is done to allow two stacks to share the *same storage area*

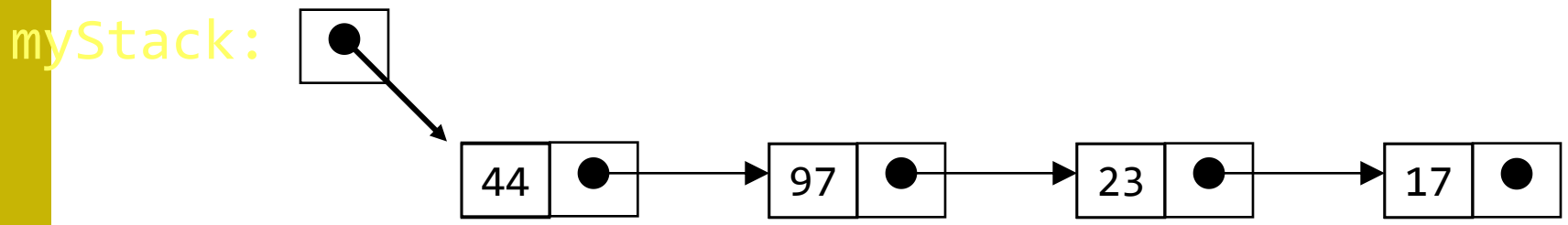


Error checking

- There are two stack errors that can occur:
  - Underflow: trying to pop (or peek at) an empty stack
  - Overflow: trying to push onto an already full stack
- For underflow, you should throw an exception
  - If you don't catch it yourself, Java will throw an `ArrayIndexOutOfBounds` exception
  - You could create your own, more informative exception
- For overflow, you could do the same things
  - Or, you could check for the problem, and copy everything into a new, larger array

Linked-list implementation of stacks

- Since all the action happens at the top of a stack, a singly-linked list (SLL) is a fine way to implement it
- The header of the list points to the top of the stack



- Pushing is inserting an element at the front of the list
- Popping is removing an element from the front of the list

Linked-list implementation details

- With a linked-list representation, overflow will not happen (unless you exhaust memory, which is another kind of problem)
- Underflow can happen, and should be handled the same way as for an array implementation
- When a node is popped from a list, and the node references an object, the reference (the pointer in the node) does *not* need to be set to `null`
  - Unlike an array implementation, it really *is* removed--you can no longer get to it from the

Queues

- What is a queue?
  - A data structure of ordered items such that **items can be inserted only at one end and removed at the other end**.
- Example
  - A line at the supermarket

Queues

- A queue is called a **FIFO** (First in-First out) data structure.
- What are some applications of queues?
  - Round-robin scheduling in processors
  - Input/Output processing
  - Queueing of packets for delivery in networks

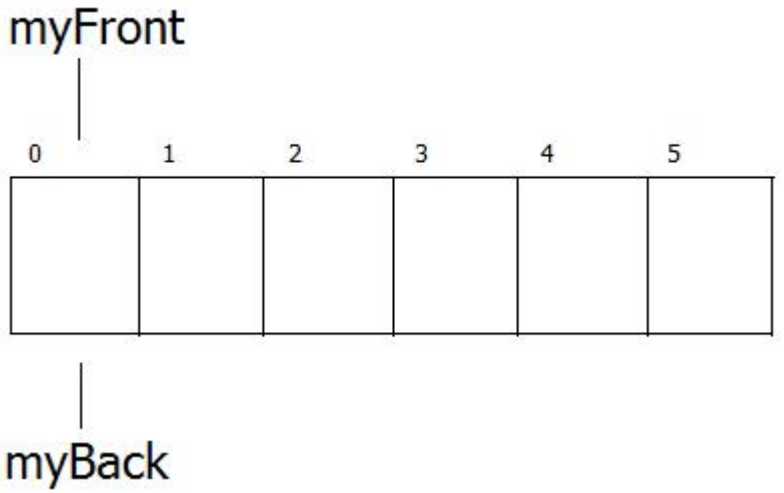
Queue Operations

- There are four basic queue operations.
- Data can be inserted at the rear and processed from the front.

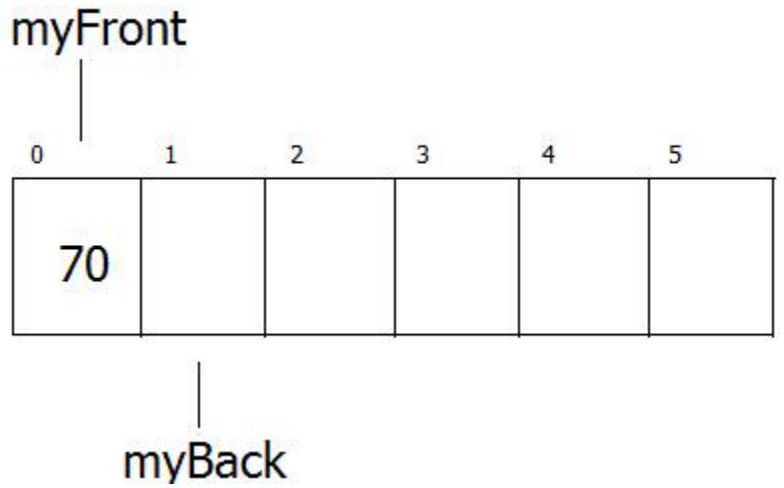
1. Construct a queue
2. Check if empty
3. **Enqueue** - inserts an element at the rear of the queue.
4. **Dequeue** - deletes an element at the front of the queue.
5. **Queue Front** - examines the element at the front of the queue.
6. **Queue Rear** - examines the element at the rear of the queue.

Queue Operation

- Empty Queue

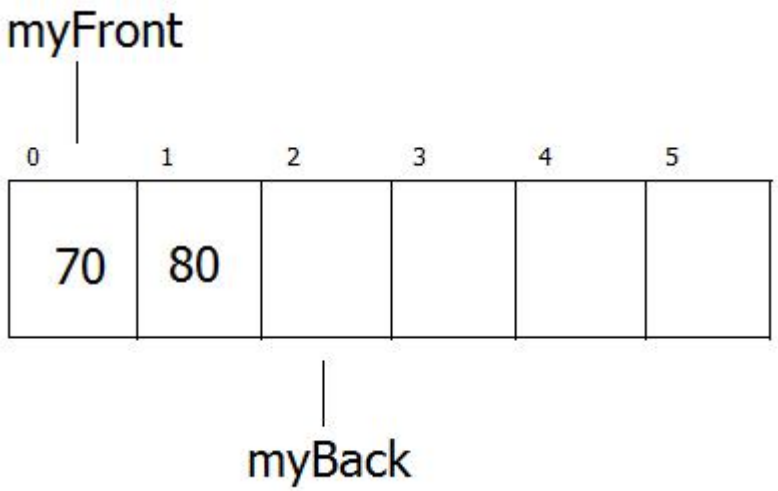


Enqueue(70)

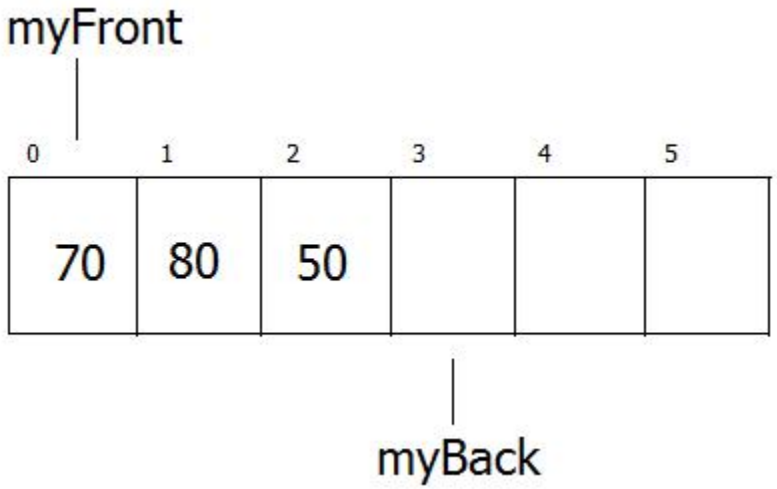


Queue Operation

- Enqueue(80)

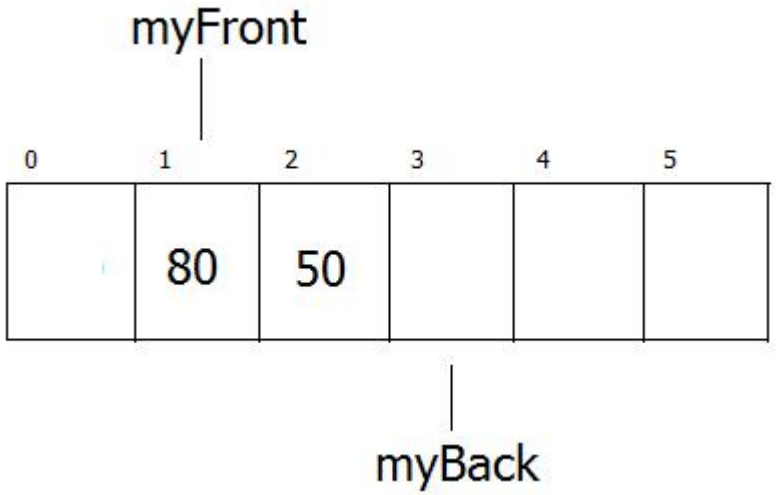


- Enqueue(50)

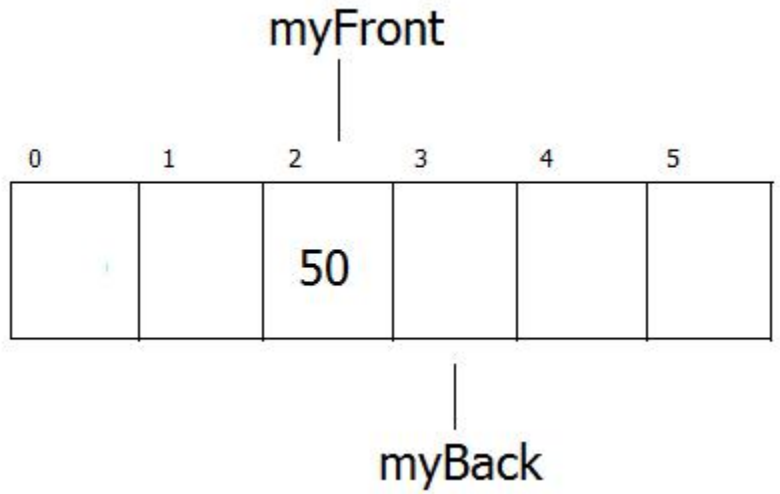


Queue Operation

- Dequeue()

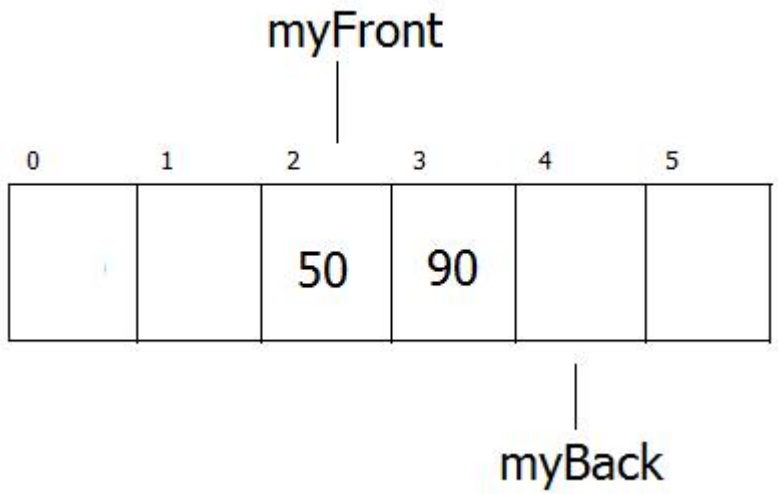


- Dequeue()

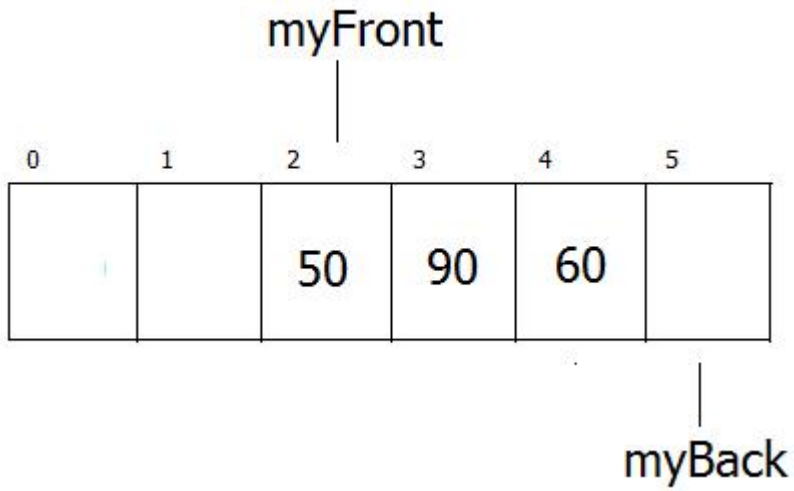


Queue Operation

- Enqueue(90)



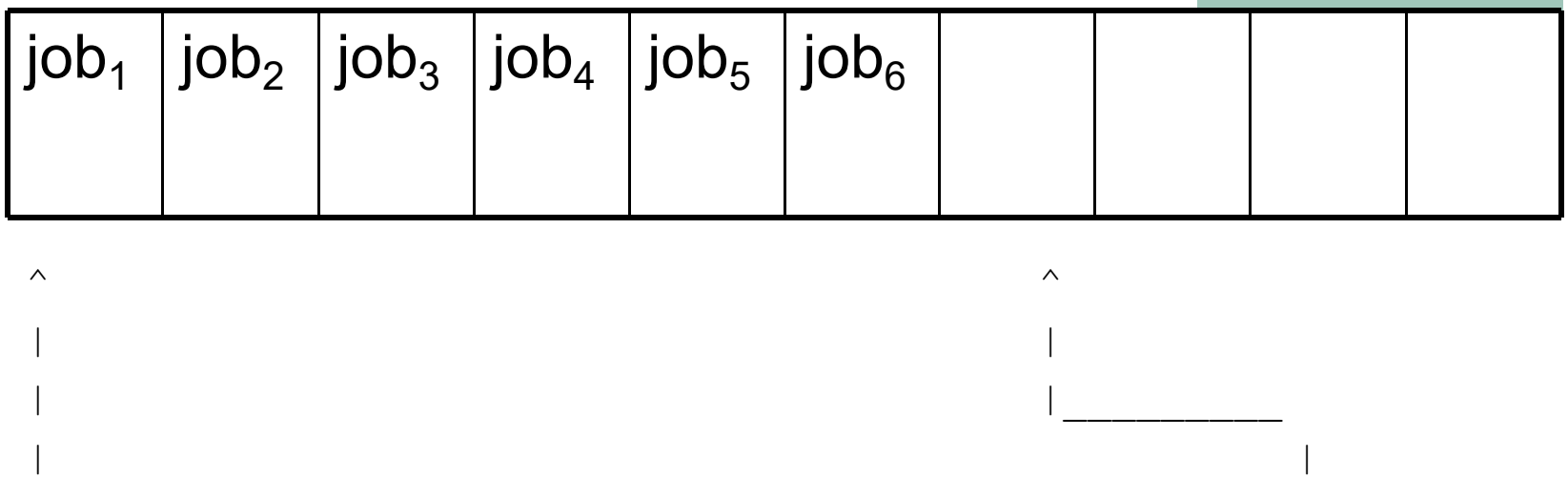
- Enqueue(60)



Queues in computer science

- Operating systems:
  - queue of print jobs to send to the printer
  - queue of programs / processes to be run
  - queue of network data packets to send
- Programming:
  - modeling a line of customers or clients
  - storing a queue of computations to be performed in order
- Real world examples:
  - people on an escalator or waiting in a line
  - cars at a gas station (or on an assembly line)

Application - Printing task queue



jobs removed from here (dequeue)

new jobs added here (enqueue)

Various Queues

- Normal queue (FIFO)
- Circular Queue (Normal Queue)
- Double-ended Queue (Deque)
- Priority Queue

Deque

- It is a **double-ended queue**.
- **Items can be inserted and deleted from either ends**.
- **More versatile data structure than stack or queue**.
- E.g. policy-based application (e.g. **low priority go to the end, high go to the front**)
- In a case where you want to sort the queue once in a while, What sorting algorithm will you use?

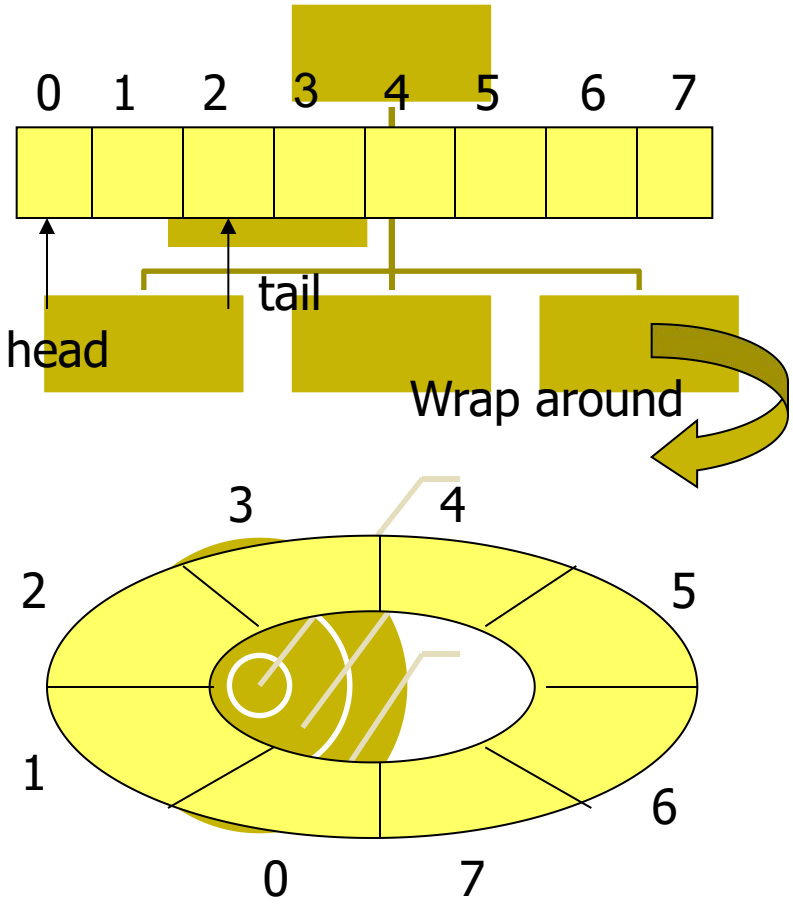
Circular Queue

- 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.

Circular Queue

- To solve this problem, queues implement **wrapping around**. Such queues are called Circular Queues.
- 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.

Circular queue - implement queues



- How to make both enqueue & dequeue operations efficient ? (**avoid shifting items**)
- enqueue

  tail=

  (tail+1) mod size
- dequeue

  head=

  (head+1) mod size

Implementing a Queue

- Just like a stack, we can implementing a queue in two ways:
  - Using an array
  - Using a linked list

Implementing a Queue

- **Using an array to implement a queue is significantly harder than using an array to implement a stack**. Why?
  - Unlike a stack, where we add and remove at the same end, **in a queue we add to one end and remove from the other**.

Implementing a Queue

- There are two options for implementing a queue using an array:
- Option 1:
  - *Enqueue* at data[0] and shift all of the rest of the items in the array down to make room.
  - *Dequeue* from data[numItems-1]

Implementing a Queue

- Option 2
  - *Enqueue* at data[rear+1]
  - *Dequeue* at data[front]
  - The **rear variable always contains the index of the last item** in the queue.
  - The **front variable always contains the index of the first item** in the queue.
  - When we reach the end of the array, **wrap around** to the front again.

Implementing a Queue

- Implementing a queue using a linked list is still easy:
  - **Front of the queue is stored as the head node** of the linked list, **rear of the queue is stored as the tail node**.
  - *Enqueue* by adding to the end of the list
  - *Dequeue* by removing from the front of the list.

Palindromes

- We can determine whether or not a word is a palindrome using a stack and a queue.
- How?

Palindromes

- Read each letter in the phrase. Enqueue the letter into the queue, and push the letter onto the stack.
- After we have read all of the letters in the phrase:
  - Until the stack is empty, dequeue a letter from the queue and pop a letter from the stack.
  - If the letters are not the same, the phrase is not a palindrome

Priority Queues

- In a priority queue, **each item stored in the queue has a priority** associated with it.
- **Items are ordered by key value so that the item with the lowest key (or highest) is always at the front**.
- When we call enqueue, we pass the item to be enqueued and the priority associated with that item.

Implementing a PQ

- There are several ways in which we might implement a priority queue:
  - Use an **array of ordinary queues, one for each priority**.
    - Queues[0] is the queue for priority 0, queues[1] is the queue for priority 1
  - Use a sorted linked list
    - **The list should be sorted according the priorities** of the items contained
- Which approach is better?

An example of using queue

- One printer is connected to several computers
- Printing a file takes much longer time than transmitting the data; a **queue of printing jobs** is needed
- When **new job P arrives, do *enqueue(P)***
- When a **job is finished, do *dequeue(P)***