Introduction, ADT, API, Bags, Lists, Stacks

Neerja Mhaskar

Dept. of Computing and Software, McMaster University, Canada

Acknowledgments: Material mainly based on the textbook Algorithms by Robert Sedgewick and Kevin Wayne (Chapters 1.2-1.3) and Prof. Janicki's course slides

Course Outline

Instructor: Dr. Neerja Mhaskar
Email: pophlin@mcmaster.ca
Office Hours: Thurs., 1:30 – 2:30pm (starts on Jan 14th, 2021)
Course URL: http://avenue.mcmaster.ca/
All announcements and course related communications will be posted on the course website; that is, on Avenue. It is your responsibility to check the course website (in particular the Announcements section) on a regular basis.

Lectures and Tutorials:
Lectures on Ms Team:
Wed., 3:30pm – 5:20pm (1 hour 50 minutes)
Fri., 3:30pm – 4:20pm (50 minutes)
Tutorials on Ms Team:
T01, T02: Mon., 11:30am - 12:20pm, headed by Morteza/Mehdi
T03: Mon., 1:30am – 2:20pm, headed by Morteza/Mehdi

Teaching Assistants:
[-] Morteza Alipour Langouri: alipoum@mcmaster.ca
[-] Mehdi Jafarizadeh: jafarizm@mcmaster.ca
[-] Victor Chen: chenv5@mcmaster.ca
[-] Tianyi Zhang: zhangt73@mcmaster.ca

[-] Office hours by Morteza Alipour Langouri: Fridays, 11-12
[-] Office hours by Mehdi Jafarizadeh: Wednesdays, 1:30 -2:30
[-] Office hours Victor Chen: Tuesdays, 1:20 - 2:20
[-] Office hours Tianyi Zhang: Thursday, 11:30 to 12:30

Calendar Description:
Basic data structures: stacks, queues, hash tables, and binary trees; searching and sorting; graph representations and algorithms, including minimum spanning trees, traversals, shortest paths; introduction to algorithmic design strategies; correctness and performance analysis.

Prerequisites: COMPSCI 1DM3 or 2DM3; COMPSCI 1XC3 or 1XD3 or 1MD3.

In addition to these prerequisites students should have basic knowledge of discrete mathematics and basic programming skills (especially in JAVA) to only understand the textbook examples.

If you are interested in the experimental aspect of this course, consider taking Comp Sci 2XB3 - Computer Science Practice and Experience: Binding Theory to Practice.

Antirequisites: SFWRENG 2C03

Learning Objectives: Students should know and understand

- Worst case analysis of algorithms
- Basic searching algorithms (elementary sorts, quicksort, mergesort, heapsort)
- Basic sorting algorithms (binary search, search trees, hashing)
- Elementary data structures (stacks, queues, priority queues, search trees, heaps, hash tables, tries, graph representations)
- Graph algorithms (topological sort, breadth/depth-first-search, strongly connected components, minimum spanning trees, shortest paths)
- Basic string algorithms
- FSA's and Regular expressions

Students should be able to:

- Analyze the running time of algorithms
- Identify the time/space trade-offs in designing data structures and algorithms
- Given a problem such as searching, sorting, graph and string problems, select from a range of possible algorithms, provide justification for that selection
- Understand implementation issues for the algorithms studied
- Reduce a given application to (or decompose it into) problems already studied

Textbook:

- R. Sedgewick, K. Wayne, Algorithms, 4th Ed., Addison-Wesley 2011.
    - Textbook Website – https://algs4.cs.princeton.edu/home/
    - The website has JAVA code which you can execute, demos, slides etc. for your reference.

Tentative course schedule on Avenue.

The course may not always follow the text-book closely.

Other texts that might be useful:

- T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 3rd Ed., McGraw Hill, 2001 – this is an encyclopedia of algorithms, preferred reference book,

Other texts that might be useful:

- J. Kleinberg, E. Tardos, Algorithm design, Addison-Wesley 2005 – more devoted to algorithms, assume knowledge of basic data structures,
- M. Soltys, An Introduction to the Analysis of Algorithms, 2nd Ed., World Scientific – excellent compact book that concentrates on the analysis of algorithms.

Marking Scheme:

- Mid-terms: 40% (2 mid-terms worth 20% each)
- Assignments: 30% (3 assignments worth 10% each)
- Final Exam: 30%

Assignments:
Assignments are 30% of your grade. You will have three assignments worth 10% each of your grade.

Assignments may be done individually or in groups of two. However, if you choose to do in a group of two, the groups will be decided at the beginning of your term and cannot change later during the term. You may choose your own partner.

No late assignments are accepted.

You may discuss the general ideas and concepts of the course material with your classmates However, your assignments/labs must be your individual effort. You may consult other sources, such as textbooks, but all such sources must be documented. Failure to do so will result in academic dishonesty charges.

Exams:
Mid-term constitutes 40% of your grade. You will have two mid-terms worth 20% each of your grade. The mid-terms will be 60 minutes exams each. The final exam constitutes 30% of your grade, and will be for 2 hours. Since, the term is conducted in an on-line format, extra time will be given for all exams.

Important:

- Missed work will be given a mark of zero, unless an MSAF is provided.
- If you MSAF an assignment, its weight will be moved towards your final exam. If you MSAF Mid-term I, its weight will be moved to Mid-term 2. If you MSAF mid-term II, its weight will be moved to the final exam. If you MSAF both the mid-terms their weight will be moved to the final exam.
- Any issues with your marks/grade for labs/assignments/mid-term/final exam must be reported and discussed within one week of the distribution of marks. Any re-grading request after this period will not be considered.

Introduction

Algorithms - methods for solving problems that are suited for computer implementation.
In computer science an algorithm is used to describe a finite, deterministic, and effective problem solving method suitable for implementation as a computer program.

- Space and time complexity; that is, the space and time required by the algorithm from start to finish is the most important aspect of algorithm design.
- A clean and good algorithm design can possibly reduce the processing of millions of objects a million times faster. In contract hardware can improve the performance only by 10 or 100 times.

Algorithms can be defined/represented by describing a procedure for solving a problem in a natural language or by writing a computer program or by using a pseudocode.

Pseudocode: is an artificial and informal language that helps programmers to articulate algorithms in a human readable format. In other words, Pseudocode is a "text-based" algorithmic design tool.

We will follow the Pseudocode style adopted by T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 3rd Ed., McGraw Hill, 2001.

Data structures

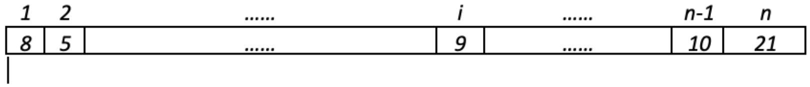
Data structures - schemes for organizing data that leave them amenable to efficient processing by an algorithm.

- More precisely, a data structure is a set of values and a set of operations on those values.
- Data structures serve as the basis for abstract data types (ADT).
- Algorithms and data structures go hand in hand, and are central objects of study in computer science.

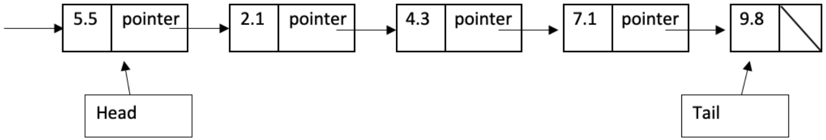
Basic Data Structures

An **array** is a data structure consisting of finite number of elements in a specific order, usually all of the same type.

- Elements are accessed using an integer index to specify which element is required.
- Typical implementations allocate contiguous memory words for the elements of arrays (but this is not always a necessity).
- Arrays are fixed-length or resizable (dynamic arrays).



A **singly linked list** (also just called a **list**) is a linear collection of data elements of any type, called nodes, where each node has a value, and pointer (*next*) to the next node in the linked list. The $head$ is the starting node of the list, and the $tail$ is the last node of the list. The pointer in the tail is NIL or $NULL$ (represented by a slash).

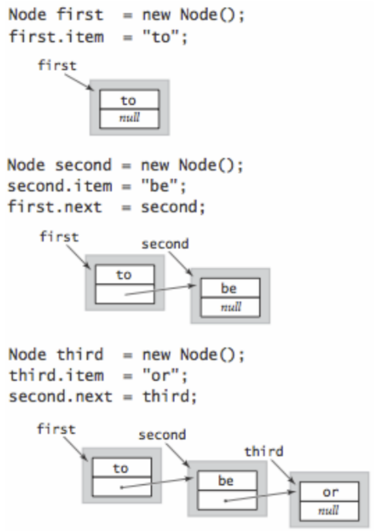


Below is the node abstraction.

private class Node {
Item item;
Node next;
}

- Linked list operations
    - Search - Search the list for an item
    - Add - Add an item to the list
    - Delete - Delete or remove the item from the list
- Linked lists provide a simple, flexible representation for dynamic sets.
- Unlike an array in which the linear order is determined by the array indices, the order in a linked list is determined by a pointer in each node.
- The principal advantage of a linked list over an array, is that values can always be inserted and removed without relocating the rest of the list.
- Certain other operations, such as random access to a certain element, are however slower on lists than on arrays.

To build a linked list that contains the items to, be, and or, we create a Node for each item, set the item field in each of the nodes to the desired value, and set the next fields to build the linked list.

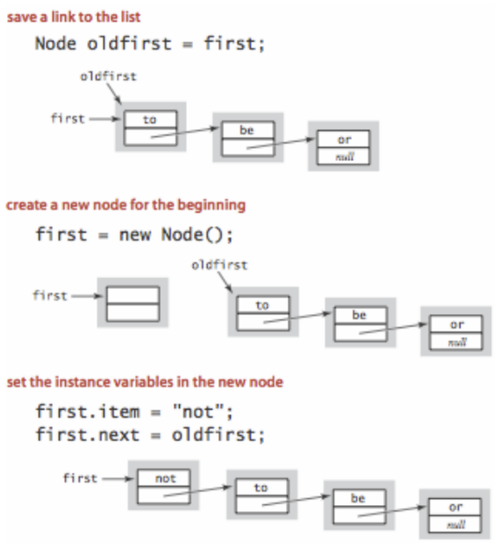


```
Node first  = new Node();
first.item  = "to";
```

```
Node second = new Node();
second.item = "be";
first.next  = second;
```

```
Node third  = new Node();
third.item  = "or";
second.next = third;
```

Insert Node with value ‘not’ at the beginning. The easiest place to insert a new node in a linked list is at the beginning.



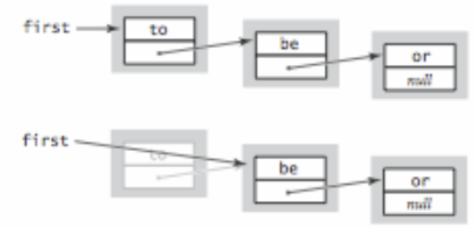
```
Node oldfirst = first;
```

```
first = new Node();
```

```
first.item = "not";
first.next = oldfirst;
```

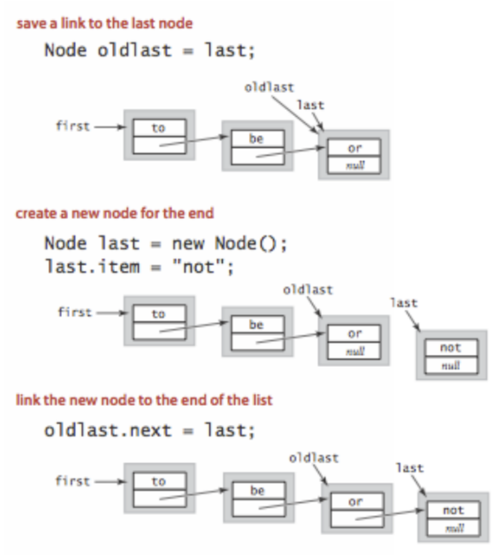
Remove from the beginning. Removing the first node in a linked list is also easy.

```
first = first.next;
```



Removing the first node in a linked list

Insert a Node with data value ‘not’ at the end of the list. To insert a node at the end of a linked list, we maintain a link to the last node in the list.



save a link to the last node

```
Node oldlast = last;
```

create a new node for the end

```
Node last = new Node();
last.item = "not";
```

link the new node to the end of the list

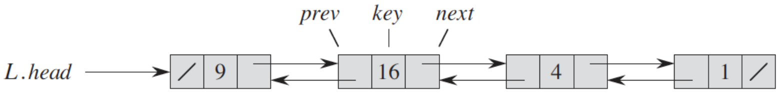
```
oldlast.next = last;
```

A **doubly linked list**, is a linked list where each node contains a pointer to its previous node in addition to the pointer to the next node.

Below is the node abstraction.

private class Node {
Item item (or key);
Node next;
Node prev; }

Example of a Doubly Linked list:



L is the linked list. We search a node with the data value k in the list.

```
LIST-SEARCH($L, k$)
$x = L.head$
while $x \neq$ NIL and $x.key \neq k$
    $x = x.next$
return $x$
```

We insert new elements at the front of the list.

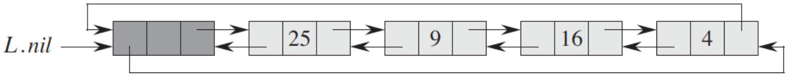
```
LIST-INSERT$(L, x)$
$x.next = L.head$
if $L.head \neq$ NIL
    $L.head.prev = x$
$L.head = x$
$x.prev =$ NIL
```

Delete a node x from the list L.

```
LIST-DELETE$(L, x)$
if $x.prev \neq$ NIL
    $x.prev.next = x.next$
else $L.head = x.next$
if $x.next \neq$ NIL
    $x.next.prev = x.prev$
```

A **circular list** is a linked list where $prev$ of the $head$ ($head.prev$) points to the $tail$, and the $next$ pointer of the $tail$ ($tail.next$) points to the $head$. We can think of a circular list as a ring of elements/nodes.

Example of a Circular Linked list:



Abstract Data Types

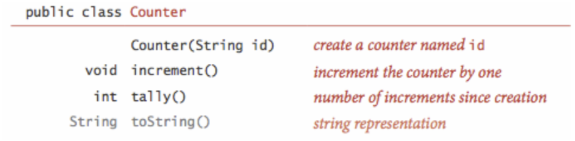
An **abstract data type (ADT)** is a mathematical model for data types, where a data type is defined by its behaviour (semantics) from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type, and the behaviour of these operations.

- This contrasts with data structures, which are concrete representations of data, and are the point of view of an implementer, not a user.
- The ADT defines the logical form of the data type.
- Formally, an ADT may be defined as a "class of objects whose logical behaviour is defined by a set of values and a set of operations".

Application Programming Interface (API)

An **Application Programming Interface (API)** is an interface or communication protocol between different parts of a computer program intended to simplify the implementation and maintenance of software.

- An API may be for a web-based system, operating system, database system, computer hardware, or software library.
- To specify the behaviour of an Abstract Data Type, we use an Application Programming Interface.
- In the latter case, API is just a list of constructors and instance methods (operations), with an informal descriptions, as shown in the API for Counter below:
- In this course we will often consider 'ADT $\equiv$ API'.



```
public class Counter
-----------------------------------------------------------------
                Counter(String id)     create a counter named id
           void increment()            increment the counter by one
            int tally()                number of increments since creation
         String toString()             string representation
```

Bags, Queues, and Stacks

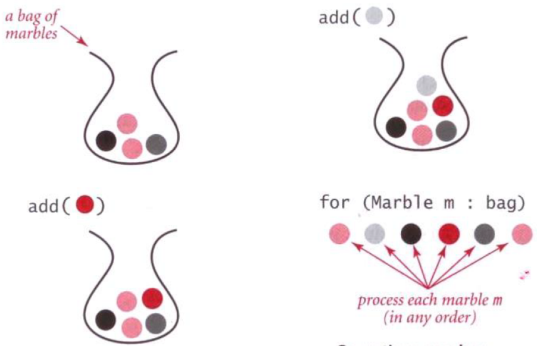
Several data structures involve a collection of objects (set of values) and the operations revolve around adding, removing and examining the objects in the collection. Here we discuss three such data structures call the Bag, Queue, and Stack.

A **bag** is a collection of items where removing an item is NOT supported. The only operations supported on it are

- Add - adds an item to the Bag
- Iterate - iterate though collected items (order is not specified)
- Is Empty - Boolean function which returns true if the Bag is empty; otherwise returns false
- Size - Number of items in the bag

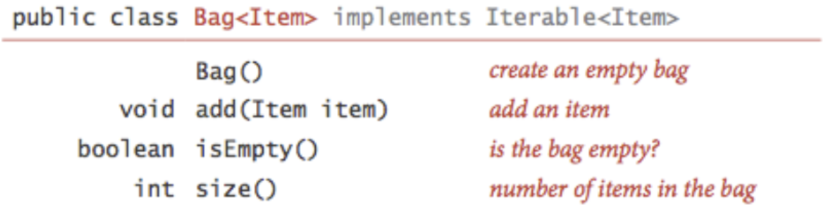
USAGE: Computing Maximum/Minimum.

Bag Example



Operations on a bag

Bag API



`public class Bag<Item> implements Iterable<Item>`

	`Bag()`	*create an empty bag*
`void`	`add(Item item)`	*add an item*
`boolean`	`isEmpty()`	*is the bag empty?*
`int`	`size()`	*number of items in the bag*

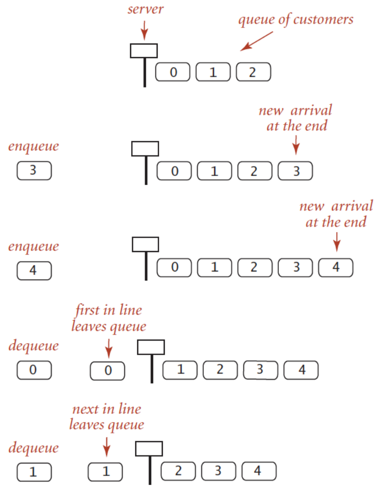
Queue

A **FIFO queue** (or just a queue) is a collection based on the First-in-first-out (FIFO) policy; that is, the policy of doing tasks in the same order in which they arrive. Operations supported on a queue are:

- Enqueue - adds an item to the end of the queue (where the relative order of the items processed is preserved)
- Dequeue - removes an item from the front of the queue
- Is Empty - Boolean function which returns true if the Queue is empty; otherwise returns false
- Count/Size - Number of items in the queue

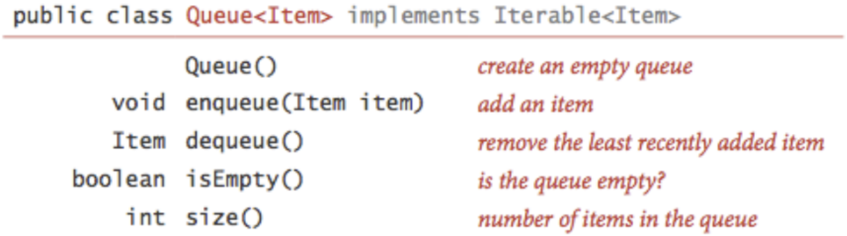
USAGE: Waiting lines, cars at toll both, tasks waiting to be serviced by an application on a computer.

Queue Example



A typical FIFO queue

Queue API



`public class Queue<Item> implements Iterable<Item>`

	`Queue()`	*create an empty queue*
`void`	`enqueue(Item item)`	*add an item*
`Item`	`dequeue()`	*remove the least recently added item*
`boolean`	`isEmpty()`	*is the queue empty?*
`int`	`size()`	*number of items in the queue*

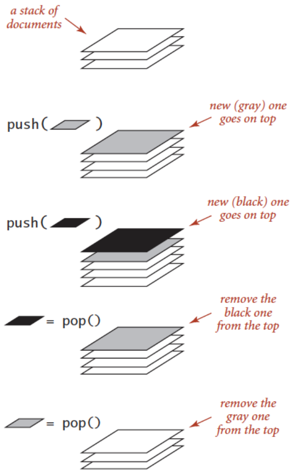
Stack

A **pushdown stack** (or just a stack) is a collection based on the Last-in-First-out (LIFO) policy. Operations supported on a stack are:

- Push - adds an item on the top of the stack(where the relative order of the items processed is preserved, but in reverse)
- Pop - removes an item from the top of the stack
- Is Empty - Boolean function which returns true if the Stack is empty; otherwise returns false
- Count/Size - Number of items in the stack

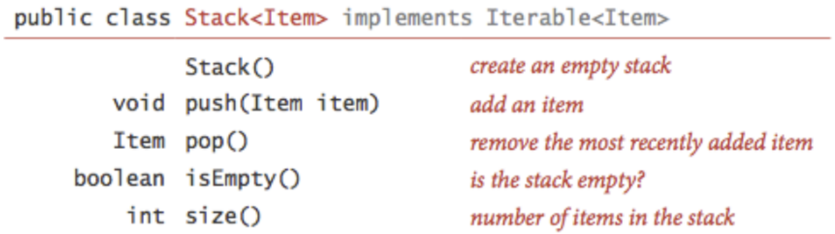
USAGE: Mail on the desk, Email clients, browser back button.

Stack Example


a stack of documents
push( )
new (gray) one goes on top
push( )
new (black) one goes on top
= pop()
remove the black one from the top
= pop()
remove the gray one from the top

Operations on a pushdown stack

Stack API



`public class Stack<Item> implements Iterable<Item>`

	`Stack()`	*create an empty stack*
`void`	`push(Item item)`	*add an item*
`Item`	`pop()`	*remove the most recently added item*
`boolean`	`isEmpty()`	*is the stack empty?*
`int`	`size()`	*number of items in the stack*

Implementations for Bag, Queue, Stack

We can consider two ways to implement Bag, Queue, Stack

[-] For fixed sized Bag, Queue, Stack – Arrays.

[-] For dynamic Bag, Queue, Stack - Resizing arrays or Linked Lists.

Resizing arrays:

[-] If no space, then double the array size.

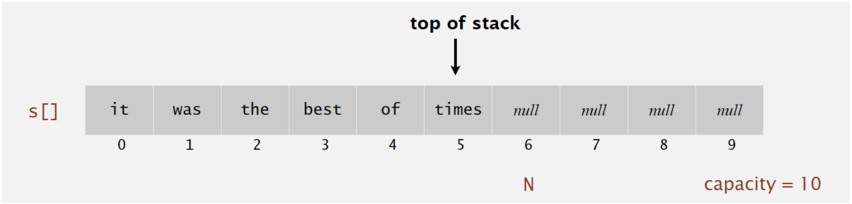
[-] If the number of elements in the array is $= \frac{array\ size}{4}$, then halve the array size.

[-] Arrays are mostly implemented as fixed data structures. Therefore, resizing arrays will involve copying items from the existing array to the new resized array.

Linked Lists: Linked Lists when implemented using the Node abstraction are easily resizable/dynamic. Therefore elements can easily be added to the linked lists without the need for resizing.

Stack implementation with an Array (fixed)

- Use array $s[]$ to store the number of items $=$ capacity, on the stack.
- $N =$ number of is elements in the stack.
- $Push()$: add new item at $s[N]$.
- $Pop()$: remove item from $s[N-1]$.



Stack Operations using an Array

A stack typically supports the *top* operation. It returns either the value on top of the stack, or (as in the below case) returns the index position of the element in the stack. The below code assumes that the array starts from 1 and not 0 (as in the example on the previous slide).

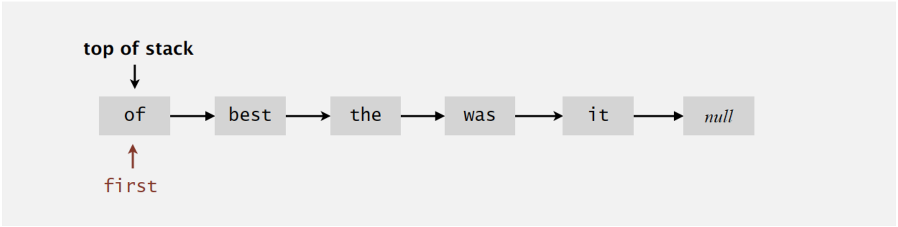
```
STACK-EMPTY(S)
if $S.top == 0$
    return TRUE
else return FALSE

PUSH(S, x)
$S.top = S.top + 1$
$S[S.top] = x$
```

```
POP(S)
if STACK-EMPTY(S)
    error "underflow"
else $S.top = S.top - 1$
    return $S[S.top + 1]$
```

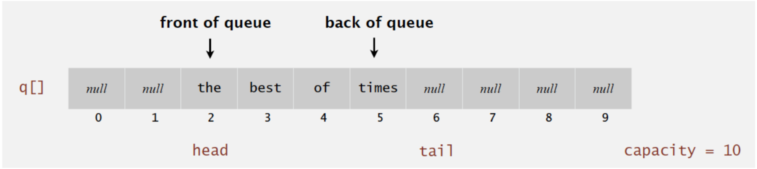
Stack implementation with a Linked List

- Maintain a pointer $first$ to first node ($head$) of the singly-linked list.
- Push new item before $first$.
- Pop item from $first$.

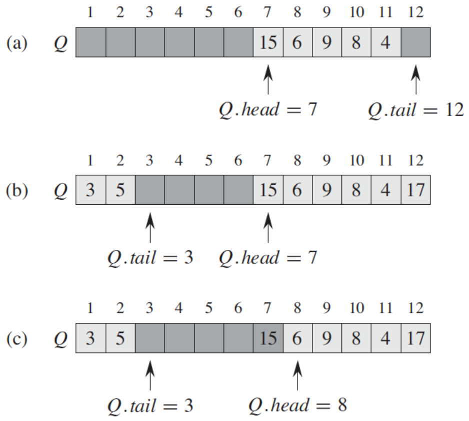


Queue implementation with an Array (fixed)

- Use array $q[]$ to store items in queue.
- Maintain pointers $head$ and $tail$, where $head$ points to front of the queue and $tail$ either points to the next empty position at back of the queue.
- $enqueue()$: add new item at $q[tail]$.
- $dequeue()$: remove item from $q[head]$.

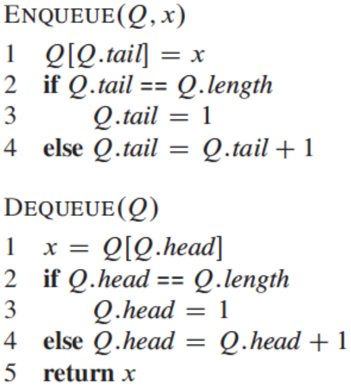


Queue implementation with an Array (fixed)



Queue Operations using Array (fixed)

Depending on the programming language you can either pass the queue data structure Q or not to your methods/opertions (as shown in the figure below).

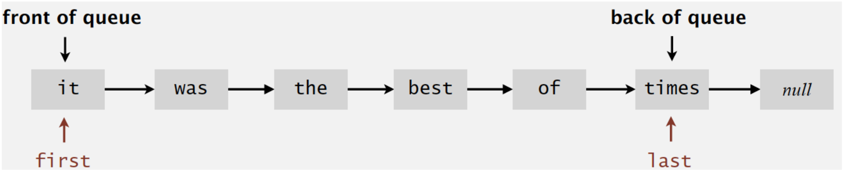


```
ENQUEUE($Q, x$)
$Q[Q.tail] = x$
if $Q.tail == Q.length$
    $Q.tail = 1$
else $Q.tail = Q.tail + 1$

DEQUEUE($Q$)
$x = Q[Q.head]$
if $Q.head == Q.length$
    $Q.head = 1$
else $Q.head = Q.head + 1$
return $x$
```

Queue implementation with a Linked List

- Maintain one pointer $first$ pointing to the first node or $head$ of the singly-linked list.
- Maintain another pointer $last$ pointing to the last node or $tail$ of the singly-linked list.
- Dequeue from $first = head$.
- Enqueue after $last = tail$.


front of queue
back of queue
it
was
the
best
of
times
null
first
last