Elementary Data Structures

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. Revised by Dr. N. Moore

Introduction

Algorithms - 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** are the most important aspects of algorithm design.
  - We are generally concerned with how quickly the required space and time increase as the size of the input increases.
- A good algorithm design can process millions of objects millions of times faster than a bad one.
- Hardware, on the other hand, yields only a 10-100 times improvement.

Algorithms are, generally, procedures for solving problems. They can be encoded in:

- Natural Language (boo!)
- **Pseudo-code** (better!)
- Computer Languages (Best!)

Pseudocode is a middle ground between Natural Language and Computer Language, combining elements of both.

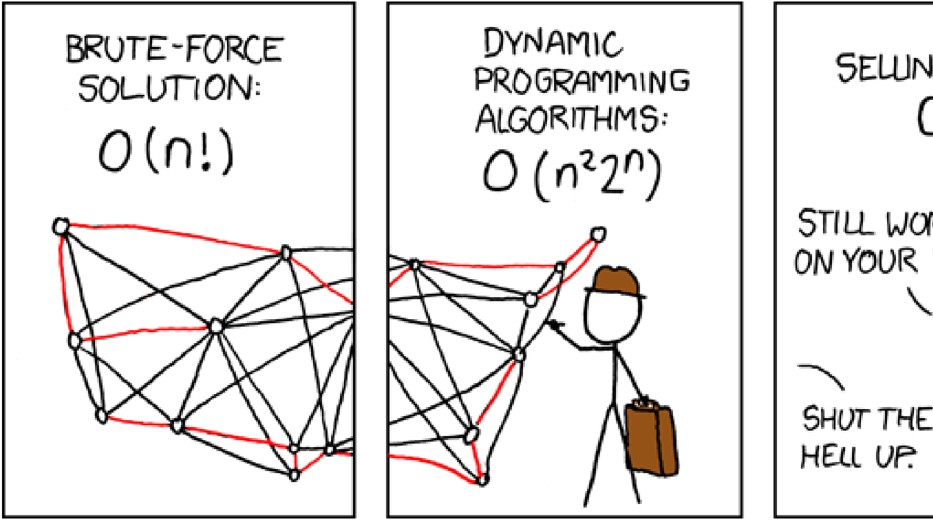
- Loops and if statements (i.e. program structure) is communicated.
- Language specific details and syntax are omitted.
- Called subroutines need not be explicitly defined.

We will use the Pseudocode style in Cormen, Leiserson, Rivest, 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.
  - Algorithms often require **preconditions**, such as binary search requiring data be sorted.
  - Data structures are designed to *maintain such preconditions*.


BRUTE-FORCE SOLUTION:
O(n!)
DYNAMIC PROGRAMMING ALGORITHMS:
O(n²2ⁿ)
SELLIN
STILL WO
ON YOUR
SHUT THE
HELL UP.

Basic Data Structures - Arrays

An **array** is a *linear* data structure, containing a finite number of elements with a specified order.

- Elements are stored/retrieved using a numerical index.
  - You can also think of it as a mapping between $\mathbb{N}$ and the set of contained elements.
- Typically (but not always) stored in **contiguous memory**.
- Arrays can be **static** (fixed size) or **dynamic** (variable size).
- Typically all elements are of the same datatype.

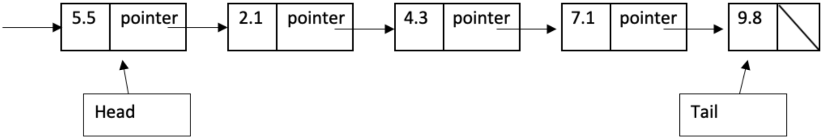
1	*2*	......	*i*	......	*n-1*	*n*
8	*5*	......	*9*	......	*10*	*21*

Basic Data Structures - (Singly) Linked Lists

A **singly linked list** (also just called a **list**) is a linear collection of nodes.

- Each node has a value, and pointer (*next*) to the next node in the list.
- The $head$ is the starting node.
- The $tail$ is the last node.
- The $tail$ points to $NULL$ (represented by a slash).

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

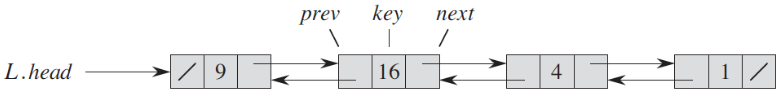


Basic Data Structures - Doubly Linked Lists

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.

- A key flaw with singly linked lists is that they can only be traversed in one direction.
- Doubly linked lists are one of many examples where we spend a little memory to save a large amount of time.

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



Linked list operations

Let's take a look at some basic operations over linked lists:

- **Search** - Search the list for a node with data value k
- **Insert** - Given a pointer to a node not currently in the list, add (or insert) a node to the list
  - In our case, we will examine **prepending**, or adding the node to the beginning of the list.
- **Delete** - Given a pointer to a node in a list, delete (or remove) it!

(FYI: Pointers are assumed knowledge in this class!)

List Search

We linearly search a node with the data value k by **traversing** the list L.

- **Traversing** or *walking* a data structure means to follow the node pointers from node-to-node to accomplish some goal, such as finding a specific element.

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

List Insert

This algorithm inserts a new node x at the head of the list L (i.e., *prepends the list*).

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

In this pseudocode:

- `L.head` is the address of the first element of L.
- `x`, when assigned to something, is the *address of* x.
- `NIL` = `NULL`

List Delete

Deletes a node x from the list L.

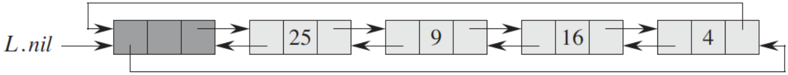
```
LIST-DELETE(L, x)
if x.prev ≠ NIL
    x.prev.next = x.next
else L.head = x.next
if x.next ≠ NIL
    x.next.prev = x.prev
```

- The algorithm assumes that the node we wish to delete has already been found.
- Deleting a node by it's value would require invoking both LIST-SEARCH and LIST-DELETE

Basic Data Structures - Circular Linked List

A **circular list** is a linked list where $head.prev$ points to the tail node, and $tail.next$ points to the head node.

We can think of a circular list as a ring of elements/nodes.



Arrays Vs. Linked Lists

Linked Lists

- Better for dynamic data sets.
- Indexing requires node traversal ($O(n)$).
- Expansion only requires allocating new nodes.
- Deletion and Insertion are faster, requiring only pointer reassignment.

Arrays

- Better for static data sets.
- Indexing requires calculating a memory offset ($O(1)$).
- Expansion may require copying entire array to a new memory chunk.
- Deletion and Insertion are more expensive, requiring all data to be shifted up/down.

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.

- 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 between different parts of a computer program, intended to simplify the implementation and maintenance of software.

- To specify the behaviour of an Abstract Data Type, we use an Application Programming Interface.
- In this case, APIs are a list of constructors and instance methods (operations), with an informal descriptions,

`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, Stacks, & Queues

Several data structures involve a collection of objects (set of values), with operations for adding, removing and examining the objects in the collection.

Here we discuss three such data structures:

Bags!

Stacks!

Queues!

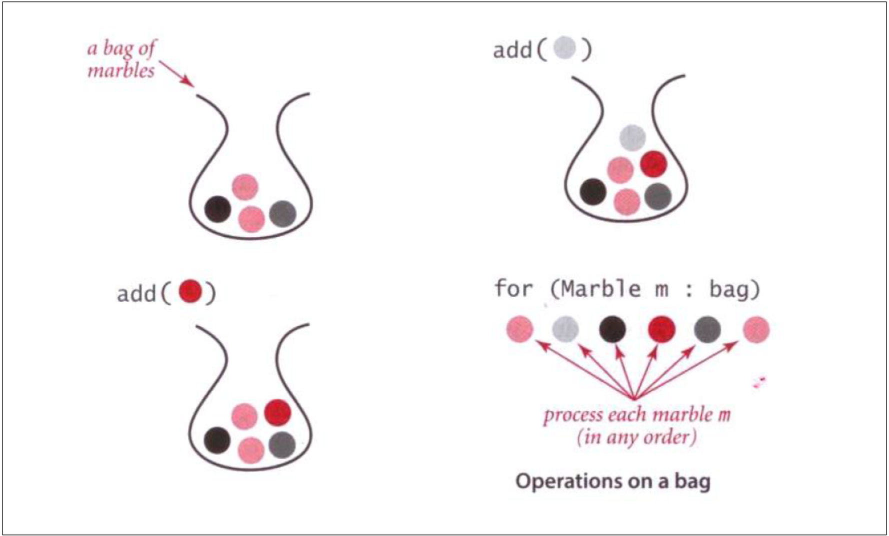
Bags

A **bag** is a collection of items with the following operations:

- ADD - adds an item to the Bag
- ITERATE - iterate though collected items (order is not specified. It could be random, it could also be "arbitrary but deterministic").
- ISEMPTY - Returns true if the Bag contains no items, false otherwise.
- SIZE - Counts the number of items in the bag

Note that there is no way to remove an item from a bag once added. Bags are useful for computing maximum/minimum values.

Bags Visualized



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*

A Brief Aside on Java Generics

The previous example used a notation that might be new to you: **Generics**. Consider the following Java code:

```
class Test<T> {
    T obj;                 // An object of type T is declared

    Test(T obj) {          // constructor
        this.obj = obj;
    }

    public T getObject() { // fetches the contained object
        return this.obj;
    }
}
```

Angle Braces Are Real Braces!

On the previous slide, `Test<T>` indicates that the class `Test` is **parameterized by** the *type* `T`.

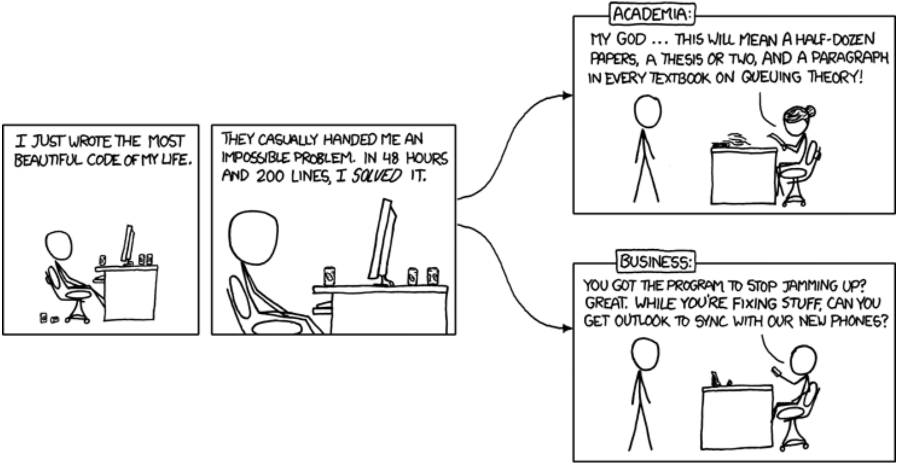
- This is similar to the idea of passing data to a function.
- Throughout the class, we can use `T` to represent a type provided during class declaration.
- Essentially, generics are **type variables**.
- As on the previous slide, the type variable `T` can be used anywhere a type is called for.

Isn't This a Bit Generic?

```
class Main {
    public static void main(String[] args) {
        // instance of Integer type
        Test<Integer> iObj = new Test<Integer>(15);
        System.out.println(iObj.getObject());

        // instance of String type
        Test<String> sObj = new Test<String>("Hello, World!");
        System.out.println(sObj.getObject());
    }
}
```

Generics allow us to build classes (like Bags) that can hold any other declared type. They get much fancier than this, but this is good enough for now.


I JUST WROTE THE MOST BEAUTIFUL CODE OF MY LIFE.
THEY CASUALLY HANDED ME AN IMPOSSIBLE PROBLEM. IN 48 HOURS AND 200 LINES, I SOLVED IT.
ACADEMIA:
MY GOD ... THIS WILL MEAN A HALF-DOZEN PAPERS, A THESIS OR TWO, AND A PARAGRAPH IN EVERY TEXTBOOK ON QUEUING THEORY!
BUSINESS:
YOU GOT THE PROGRAM TO STOP JAMMING UP? GREAT. WHILE YOU'RE FIXING STUFF, CAN YOU GET OUTLOOK TO SYNC WITH OUR NEW PHONES?

Queue

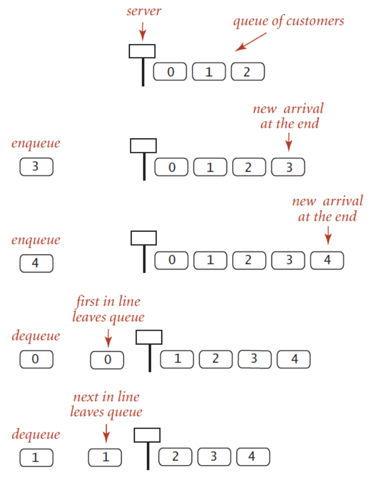
A FIFO **queue** is a collection where elements are drawn in the same order they are added.

- Queues have many important applications. `stdin` is a FIFO queue!

Supported Operations:

- ENQUEUE - adds an item to the queue.
- DEQUEUE - removes the next item from the queue.
- ISEMPTY - Boolean function which returns true if the Queue is empty; otherwise returns false
- SIZE or COUNT - Number of items in the queue

Queue Example



A typical FIFO queue

USAGE:

- Generally accepted social norm, useful when multiple humans want access to something and can't all be served at once.
- Tasks waiting to be serviced by an application on a computer.

Queue API

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

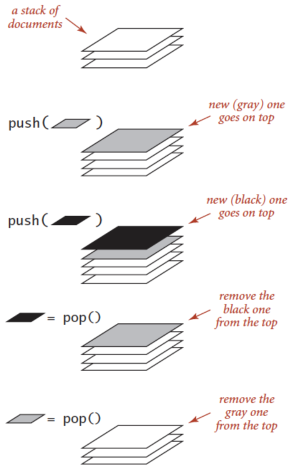
	`Queue()`	*create an empty queue*
`void`	`enqueue(Item item)`	*add an item*
`Item`	`dequeue()`	*remove the least recently added item*
`boolean`	`isEmpty()`	*is the queue empty?*
`int`	`size()`	*number of items in the queue*

Stack

A **pushdown stack** or just **stack** is a LIFO (Last-in-First-out) queue. Items are retrieved in the reverse of the order added!
Supported Operations:

- PUSH - adds an item on the top of the stack
- POP - removes an item from the top of the stack
- ISEMPTY - Boolean function which returns true if the stack is empty; otherwise returns false
- SIZE or COUNT - Number of items in the stack

Stack Example



Operations on a pushdown stack

USAGE:

- The function call stack.
- "Undo" functions in programs.
- Browser back button.
- Plates in your kitchen cupboard

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

How would one implement a Bag, Queue, or Stack?

- Fixed Size? use an array!
- Dynamic Size? Linked List, or a Resizing Array.

Array Resizing Procedure:

- If the data exceeds the currently allocated size, double the size of the array through reallocation.
- If only 25% of the array is currently being used, cut the allocated memory in half.

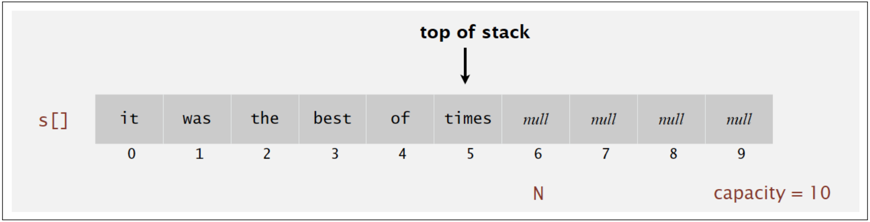
In most languages this will involve calling `realloc()` (in the C standard library), which can involve copying the entire array to a new memory chunk.

The purpose of the above algorithm is to provide thresholds which minimize the number of resize operations.

A Stack Using a Static Array

Using an array `s[]`:

- With $N =$ `stack.size()`
- $Push()$: add new item at $s[N]$.
- $Pop()$: remove item from $s[N-1]$.



Stack Operations Using an Array

In this example, `stack.top` returns the index of the top element in the stack. Further, this pseudocode assumes array indexes start at 1, rather than 0.

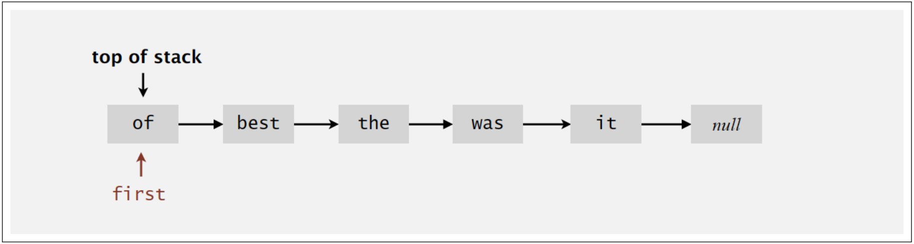
```
PUSH(S, x)
    S.top = S.top + 1
    S[S.top] = x
```

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

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

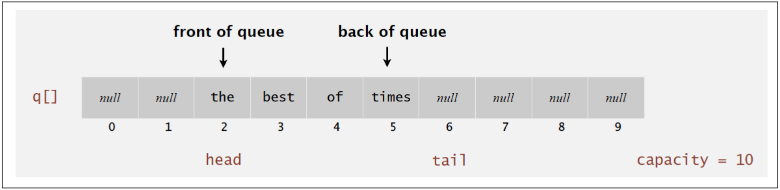
Stack Operations Using a Linked List

- Maintain a pointer, `first`, to head of a singly-linked list (which you would normally do anyways).
- Pushed items are *prepended* to the list.
  - This way, the head of the list is always the top of the stack!
- Popped items are removed and returned from the head of the list.

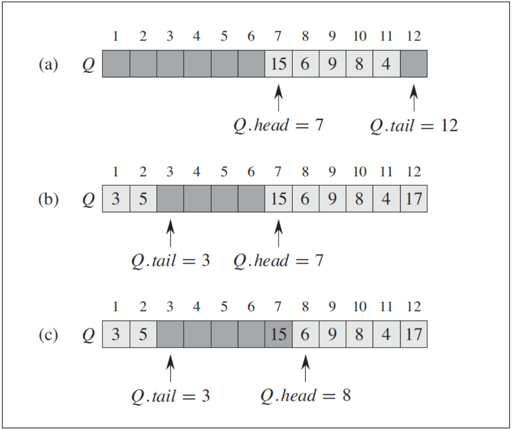


Queue Operations Using a Static Array

- Use array `q[]` to store items in queue.
- Maintain pointers:
  - `head` ⇒ points to front of the queue
  - `tail` ⇒ points to the next empty position at back of the queue.
- ENQUEUE adds a new item at `q[tail]`.
- DEQUEUE: remove item from `q[head]`.



Static Array Wrap-Around



When `tail` reaches the end of allocated space, the pointer wraps-around to the empty side of the array.

- Therefore, we can not assume that `head` $>$ `tail`!
- If `head` $=$ `tail`, the queue could be *empty* or *full*!

Queue Operations using Array (fixed)

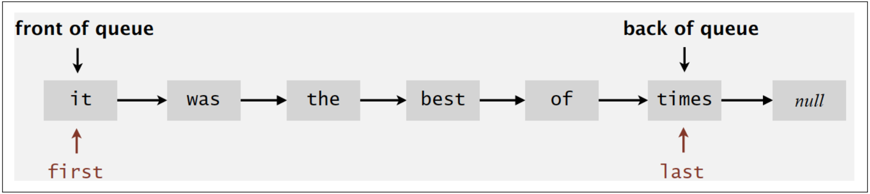
- Here, Q is the queue, and x is the item we are enqueueing.
- Once again, 1-indexing is used, rather than 0-indexing.
- The reason pseudocode uses 1-indexing is that 0-indexing is an artifact of how array indexing uses pointer offsets. This makes it an implementation detail, and pseudocode is supposed to lack implementation details.

```
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 head.
- Maintain another pointer, `last`, pointing to the tail.
- Items are dequeued from `first`.
- Items are enqueued from `last`.



In Summary


YOUR COUNTRY NEEDS YOU
TO STRUCTURE THAT DATA
imgflip.com


PICTURE
UNRELATED


Disjoint Tree Collection Path Compression in Java
Forestpresso