# Introduction to Data Structures and Algorithms

Effective:

1. Time Complexity
2. Space Complexity

## Best/Worst/Average Case

1. **Worst Case is usually used**. It is an **upper bound** and in certain application domains(Eg: Air Traffic control, Surgery) knowing the worst case time complexity is of crucial importance.
2. For some algorithms, worst case occurs fairly often.
3. Average Case is often as bad as the worst case.
4. Finding average case can be very difficult.

## Asymptotic Analysis

**Goal:** To simplify analysis of running time by getting rid of details which may be affected by specific implementation and hardware.

Example: rounding 1, 000,001 to 1, 000, 000

3n2 = n2

**Capturing the essence:**

How the running time of an algorithm increases with the size of the input.

## Asymptotic Notation

1. The “big-Oh” O-notation
2. Asymptotic upper bound
3. Used for worst case analysis.

**Simple Rule**:

Drop lower order terms and constant factors.

50nlogn = O(nlogn)

7n – 3 = O(n)

8n2logn + 5n2 + n = O(n2logn)

Even though 50nlogn is O(n5), it is expected that such an approximation be of as small an order as possible.

## Asymptotic Analysis of Running Time

1. Use O-notation to express number of primitive operations executed as function of input size.
2. Comparing asymptotic running times

* An algorithm that runs in O(n) time is better than one that runs in O(n2) time.
* Similarly, O(logn) is better than O(n)
* **Logn < n < n^2 < n^3 < 2^n**

1. Caution: Beware of very large constant factors.

An algorithm running in time 1,000,000 n is still O(n) but might be less efficient than one running in time 2n^2 which is O(n2).

## Asymptotic Notation

Special classes of algorithms:

1. **Logarithmic**: O(logn)
2. **Linear**: O(n)
3. **Quadratic**: O(n2)
4. **Polynomial**: O(n^k), k >=1
5. **Exponential**: O(a^n), a > 1

## Relatives of the Big-Oh

1. Ω(f(n)): Big Omega – Asymptotic lower bound
2. O(f(n)): Big Theta – Asymptotic tight bound
3. Little-Oh
4. Little-omega

## Analogy with real numbers

F(n) = O(g(n)) f <= g – Big Oh

F(n) = Ω(g(n)) f >= g – Big Omega

F(n) = theta(g(n)) f = g – Big Theta

F(n) = o(g(n)) f < g – Little Oh

F(n) = ꙍ(g(n)) f > g – Little Omega

# 2: Stacks

## Abstract Data Types (ADTs)

What is Abstract Data Type?

Abstract Data Type defines a set of it’s instances with:

1. **Specific Interface** – a collection of signatures of operations that can be invoked on an instance.
2. **Set of Axioms** – (Precondition and Postcondition) that defines the semantics of operations (What operations instances of ADT do, but not how)

## Types of Operations:

1. Constructors – Create instance of that particular data type – New()
2. Access functions – Access elements of data type - IsIn()
3. Manipulation Procedures – Modify the elements of that data type. – insert() and delete()

## Simple ADTs

1. Queue
2. Deque
3. Stack

## Stack

1. A stack is a container of objects that are inserted and removed according to the Last-In-First-Out(**LIFO**) principle.
2. Objects can be inserted at any time, but only the last (the most-recently inserted) object can be removed.
3. Inserting an item is known as “**Pushing**”

Removing an item is known as “**Popping**”

## Stack Methods

1. Empty() – Tests if this stack is empty
2. Peek() – Looks at the object at the top of this stack without removing it from the stack. Throws EmptyStackException if the stack is empty.
3. Pop() – Removes the object at the top of this stack and returns that object as the value of this function. Throws EmptyStackException if the stack is empty.
4. Push() – pushes an item on to the top of this stack
5. Search(Object o) – Returns the 1-based position where an object is on this stack.

The topmost item on the stack is considered to be at distance 1.

## Methods inherited from java.util.Vector

Size

isEmpty

addAll

clear

iterator

removeAll

remove

add etc

## Interfaces:

1. Interface can have methods and variables, but the methods declared in interface are by default abstract (only method signature and no body).
2. Blueprint of class.
3. If a class implements an interface and does not provide method bodies for all function specified in the interface, then class must be declared abstract.

## Array-Based Stack Implementation

1. Create a stack using an array.
2. Integer variable t – the index of top element in array.
3. Array indices start at 0, so we initialize t to -1.
4. Array-Based Stack implementation is simple and efficient (methods performed in O(1) time)
5. There is an upper bound N, on the size of the stack. The arbitrary value N may be too small for a given application or waste of memory.

## A Growable Array-Based Stack

1. When array gets full, create new array and copy elements
2. **Tight Strategy** (add a constant) f(N) = N + c
3. **Growth Strategy** (double up) f(N) = 2N

## Tight versus Growth Strategy

1. Push Operation = 1 unit
2. Create array and copy elements and add one element

Cost = f(N) + N + 1 units

1. Total cost of phase I =

C\*i = The cost of creating array

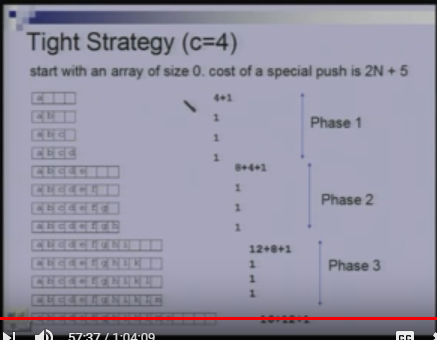
C \* (i – 1) = The cost of copying elements into new array

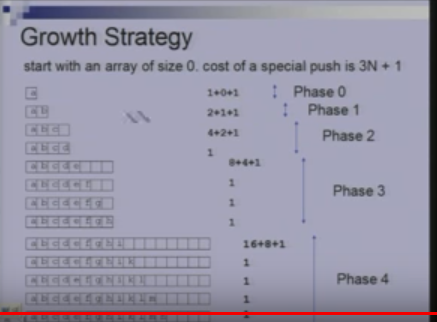
C = The cost of c pushes

Cost of phase I = 2ci

In each phase we do c pushes. Hence for n pushes we need n/c phases.

Total cost of n/c phases = 2c(1 + 2 + 3 …n/c) = O(n^2/c)





1. In phase i, the array has size 2^i
2. Total cost of phase i

2^I = The cost of creating array

2 ^ i-1 = The cost of copying elements into new array

2 ^ I – 1 = The cost of 2^i-1 pushes done in this phase.

Cost of phase I = 2 ^ i+1

1. If we do n pushes then we have logn pushes.
2. Total cost of n pushes = 2 + 4 + 8 + 2 ^ logn + 1 = 4n – 1

The Growth Strategy wins.

# 3: Queues and Linked List

## Queues

1. Queue follows First in First out (FIFO) principle.
2. Elements are inserted at the rear (enqueued) and removed from the front(dequeued).

## Queue Methods

1. Add(Element e) – Inserts the specified element into this queue.

Returns true upon success and throws IllegalStateException if no space is available.

1. Element() – Retrieves, but does not remove, the head of this queue.
2. Offer(Element e) – Inserts the specified element into the queue.
3. Peek() – Retrieves but does not remove, the head of this queue and returns null if this queue is empty.
4. Poll() – Retrieves and removes the head of this queue, and returns null if this queue is empty.
5. Remove() – Retrieves and removes the head of this queue.

## Implementing Queue using Linked List

1. Linked List has nodes which has data and next pointer.
2. The head of the Linked List is the front of the queue and the tail of the Linked List is the rear of the queue.
3. We cannot make tail as the front because for removing we will have to traverse the Linked List.

## Double-ended queue

1. A Double-ended queue or deque supports insertion and deletion from front and back.
2. Deque methods:

addFirst

addLast

removeFirst

removeLast

pollFirst

pollLast

1. Implement deque using Doubly Linked Lists.
2. Doubly Linked List has next and previous pointer.
3. All methods of deque have constant running time O(1).

## Circular Linked List

1. Circular Linked List is implementation of stack or a queue.

## Vector

A sequence with rank.

Methods

1. elementAtRank(r)
2. replaceAtRank(r, e)
3. insertAtRank(r, e)
4. removeAtRank(r)

## Array-Based Vector Implementation

1. Shift the array elements to the right when doing insert by rank.
2. Shift the array elements to the left when doing remove by rank

## Time Complexity for Array-Based Vector Implementation

1. ElementAtRank(r) – O(1)
2. replaceAtRank(r, e) – O(1)
3. insertAtRank(r, e) – O(n)
4. removeAtRank(r) – O(n)

## Using Doubly Linked List to implement Vector

1. It takes O(n) time because we must traverse the linked list.
2. Inserting is in constant time.
3. Linked List support the efficient execution of node-based operations:

insertAtRank and removeAtRank in O(1) provided we have access to node.

1. When I give you access to particular node, you also know the implementation details like how I have implemented it whether it’s singly linked list or doubly linked list. Suppose I want to hide those information.

# Dictionaries ADT (Abstract Data Type)

1. Dictionaries store elements so that they can be located quickly using **keys**.
2. A dictionary may hold bank accounts
3. Each account is an object that is identified by an account number.
4. Each account stores a wealth of additional information

* Including the current balance
* The name and address of the account holder and
* The history of deposits and withdrawals performed.

1. An application wishing to operate on an account would have to provide the account number as search key.
2. A dictionary is an abstract model of a database
3. A dictionary stores key-element pairs
4. The main operation supported by a dictionary is searching by key.
5. Simple Container Methods: size(), isElement(), elements()
6. Query Methods – findElement(k), findAllElements(k)
7. Update Methods – insertItem(k, e), removeElement(k), removeAllElements(k)
8. Special element: NIL, returned by an unsuccessful search.

Given two keys, they can be compared for equality but not for supporting order(min, max, successor, predecessor).

## Different data structures implementing the Dictionaries

1. Arrays, Linked Lists (in-efficient)
2. HashTable
3. Binary Trees
4. Red/Black Trees
5. AVL trees
6. B-trees

In java,

Java.util.Dictionary – abstract class

Java.util.Map – interface

## Searching

**Input**:

Sequence of numbers

A single number(query)

**Output**:

Index if found or NIL

2, 5, 4, 10, 7/ 5 = 2

2, 5, 4, 10, 7 / 9 = NIL

## Binary Search

Divide and Conquer

Narrow down the search range in stages.

findElement(22)

