# 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

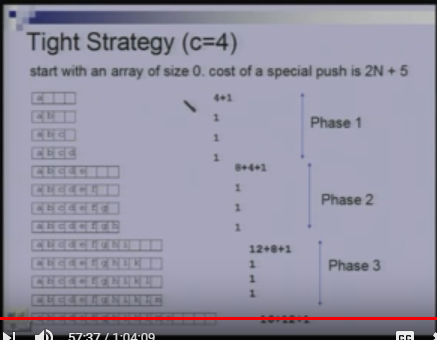
C \* (i – 1) = The cost of copying elements into new array

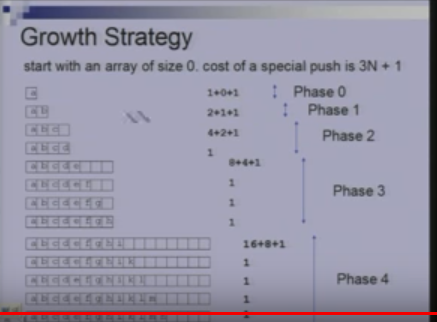
C = The cost of c pushes

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

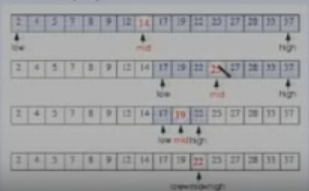
## Binary Search

Sorted Order

Divide and Conquer

Narrow down the search range in stages.

findElement(22)

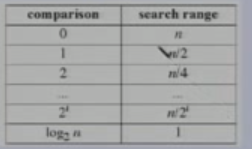


Recursive Approach

Iterative Approach

Time Complexity: O(logn)

The range of candidate items to be searched is halved after each comparison.



## Not Sorted order

1. Do the comparisons one by one.
2. Worst case running time: O(n)
3. Average case running time: O(n)

## The Dictionary Problem

T&T is a large phone company, and they want to provide Caller ID capability:

1. Given a phone number, return the caller’s name
2. Phone numbers range from 0 to r = 10^8 – 1
3. There are n phone numbers, n << r
4. Want to do this as efficiently as possible.

### Using an unordered sequence

Unordered sequence

34 🡪 14 🡪 12 🡪 22 🡪 18

1. Searching and removing takes O(n) time
2. Inserting takes O(1) time
3. Applications to log files (frequent insertions, rare searches and removals)

### Using an ordered sequence

Array-based ordered sequence (Assumes keys can be ordered)

12 🡪 14 🡪 18 🡪 22 🡪 34

1. Searching takes O(logn) time (Binary Search)
2. Inserting and Removing takes O(n) time because we have to first find the location for inserting and then shift all elements to the right and for removal we have to first find the location to remove and then shift all elements to the left.
3. Application to look-up tables (frequent searches, rare insertions and removals).

### Other Suboptimal ways

Direct Addressing: An array indexed by key

(null) (Ankur) (null)

0000-0000 9635-8904 0000-0000

1. Takes O(1) time for all insert, delete and search operations.
2. But O(r) space where r is the range of numbers – (10^8)
3. Huge amount of wasted space.

## Hashing – Hash Table

1. Hash table

O(1) expected time and O(n+m) space where m is the table size

1. Hashing function: To map broad range into one which we can manage.
2. For Example: Take the original key modulo size of array and use that as an index.

Insert (96358904, Ankur) into a hashed array with five slots. 96358904 % 5 = 4

null 🡪 null 🡪 null 🡪 null 🡪 Ankur

0 🡪 1 🡪 2 🡪 3 🡪 4

1. Let keys be entry no’s of students in CSL201 eg: 2004CS10110

There are 100 students in the class.

We create hash table of size say 100

Hash function is say last two digits

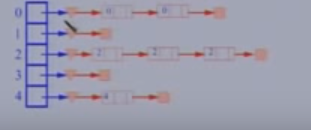
Then 2004CS10110 goes to location 10

Where do 2004CS50310 go?

## Hash Collision and Resolution

1. How to deal with two keys which hash to the same spot in the array?
2. **Use chaining**:

* Set up an array of Linked Lists indexed by the keys to the list of items with same key



1. In worst case, all keys gets mapped to one location which will have time complexity(of linked lists) of O(n) for delete and search. Insert is still O(1).
2. To find/insert/delete an element

Using hash function h, look up it’s position in the table T

Search/insert/delete the element in the linked list of the hashed slot

1. If you keep the linked list in sorted order then insertion is going to take more than constant amount of time.
2. If you keep the linked list unordered then insertion will take constant time.
3. The time to compute hash function is h(k) is O(1)

## Analysis of Hashing

1. A good hash function is the one which distributes keys evenly amongst the slots.
2. An ideal hash function would pick a slot, uniformly at random and hash the key to it.
3. However, this is not a hash function since we would not know which slot to look into when searching for a key.
4. For our analysis, we will use this simple uniform hash function.
5. Given a hash Table T with m slots holding n elements, the **load factor** is defined as n/m

### Unsuccessful search:

1. Element is not in the linked list
2. Simple uniform hashing yields an average list length x = n/m
3. Expected number of elements to be examined x.
4. Search time O(1+x) (including computing hash value)

### Successful Search

1. Somewhere in the middle of the linked list you might find the element.
2. O(1 + x/2) = O(1+x)

### What is good value for x

X is the load factor of the table

1. Assume that the number of hash table slots is proportional to the number of elements in the table.
2. n = O(m)
3. x = n/m = O(m)/m = O(1)
4. Searching takes constant time on average.
5. Insertion takes O(1) worst case time
6. Deletion takes O(1) worst case time when the lists is doubly linked list.

What if we don’t know the number of elements – Growable stack and change hash function to hash to larger hash table.

Start with small hash table and if there’s a need then grow it rather than starting with a very large hash table and then there is a waste of space.

## Hashing

Hash functions

1. Hash-code maps – Key 🡪 integer
2. Compression maps – integer 🡪 [0, N – 1]

Collision techniques/ Open addressing

1. Linear Probing
2. Double hashing

### What is the need to choose a good hash function

1. Quick to compute
2. Distribute keys uniformly throughout the table
3. Good hash functions are very rare.

### How to deal with hashing non-integer keys

1. Find some way of converting keys into integers

For example: remove hyphen in 9635-8904 to get 96358904

For string, add up ASCII values of characters of your string

Example: java.lang.String.hashCode()

1. Then use standard hash function on the integers.

### From keys to indices

1. The mapping of keys to indices of a hash table is called a hash function.
2. A hash function is usually the composition of two maps,

a hash code map

compression map

1. An essential requirement of the hash function is to map equal keys to equal indices.
2. A “good” hash function minimizes the probability of collisions.
3. Hash Code map: Key 🡪 integer

Compression map: integer 🡪 [0, N – 1]

### Popular Hash-Code Maps

1. **Integer cast**: For numeric types with 32 bits or less, we can reinterpret the bits of the number as an int.
2. **Component sum**: For numeric types with more than 32 bits, we can add 32-bit components

### Why component-sum hash code bad for strings?

Because if we have string whose characters are at different place their ASCII addition will be the same. So there will be more collision.

For Example: “GOD” and “DOG”

If we have string: “az” and “by” their ASCII sum will be the same.

Because of all these reasons it’s not a good idea to have component sum for strings.

### Polynomial Accumulation

1. For strings of a natural language, combine the character values (ASCII or Unicode) a0, a1, …an-1 by viewing them as a coefficient of polynomial.

A0 + a1x + … xn -1 an-1

1. The polynomial is computed with Horner’s rule ignoring overflows, at fixed value x:

A0 + x(a1 + x(a2 + …x(an-1 + xan-1) …))

1. The choice of x = 33, 37, 39, and 41 gives at most 6 collisions on a vocabulary of 50000 English words.

### Compression Maps

### Mod with size of table

1. Use remainder

H(k) = k mod m where k is the key and m is the size of table

1. If m is power of 2, all the keys with the same ending go to the same place
2. **If m is prime then it’s good.**

It helps to ensure uniform distribution

Primes not too close to exact powers of 2.

1. Example: Hash table for n = 2000 character strings

We don’t mind examining 3 elements – 2000/701 which is roughly 3.

M = 701

A prime near 2000/3

But not near to any power of 2

### Technique 2

Mod with number between 0 to 1, then take the fraction part and multiply by m to get a number between 0 to m – 1.

1. H(k) = Lm ( K A mod 1)] – talking floor
2. K = key, m is the size of table and A is a constant 0 < A < 1
3. The steps involved

Map 0 ….Kmax into 0 … Kmax A

Take fractional park and multiply with m to map it into 0 … m – 1

### Choice of m and A

1. Value of m is not critical, typically use m = 2 ^ p
2. Optimal choice of A depends on the characteristics of the data
3. Knuth says use A = (root(5) – 1)/2 (conjugate of golden ratio) – Fibonacci hashing.

### Technique 3 MAD – Multiply, Add and Divide

1. H(k) = |ak + b| mod N

Where a and b are fixed numbers and N is the size of the table.

1. Eliminates patterns provided a is not multiple of N.
2. Same formula is also used in linear congruential (pseudo) random number generators.

### Universal Hashing

1. For any choice of hash function, there exists a bad set of identifiers.
2. A malicious adversary could choose keys to be hashed such that all go into the same slot (bucket).
3. Average retrieval time in that case is O(n)
4. Solution: Pick a random hash function every time.

Every time when a program runs choose the hash function randomly from a set of hash functions.

1. For one run of the hash table use the same hash function for insert, delete and search.
2. So malicious adversary can never come up with the set of bad keys

### Collisions

1. A key is mapped to an already occupied table location.
2. Use collision Resolution Technique
3. One technique is Chaining
4. We can also use **Open Addressing**

Linear Probing

Double Hashing

### Open Addressing

1. All elements are stored in the hash table

N <= m where n = Number of elements

M = table size

1. Each table entry contains either an element or null.
2. When searching for an element, systematically probe table slots.

### Linear Probing

1. Hash function determines the sequence of slots examined for a given key.
2. Probe sequence for a given key k given by

{h(k,0), h(k,1) …..h(k,m)}

1. If the current location is used, try the next table location

LinearProbingInsert(k)

If (table is full) error

Probe = h(k)

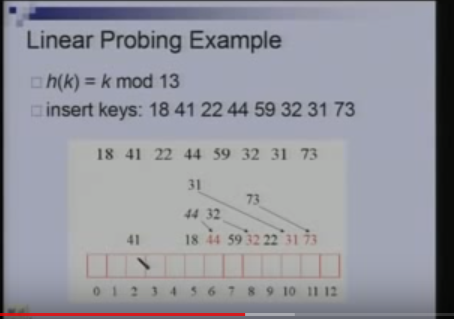
While(table[probe] occupied) {

Probe = (probe + 1) mod m; //If you reach end of the table, then start from the beginning – mod m

Table[probe] = k;

1. **Uses less memory** than chaining as one does not have to store all those links.
2. **Slower** than chaining since one might have to walk along the table for a long time.

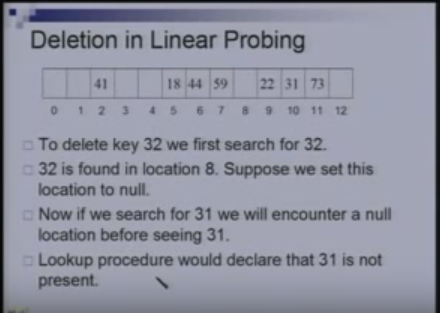
### Linear Probing Example

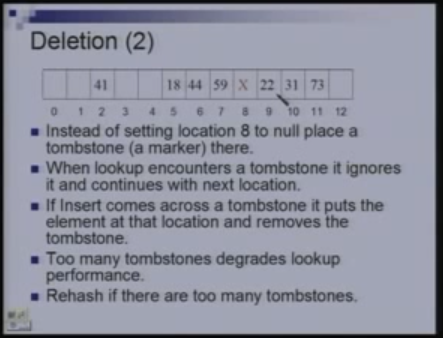


### Lookup in Linear Probing

1. To search for a key k, we do k mod 13 and continue looking at successive locations till we find k or encounter empty location.
2. Successful search: To search for 31, we do 31mod13 = 5 and continue onto 6, 7, 8… till we find 31 at location 10.
3. Unsuccessful search: To search for 33, we do 33mod13 = 7 and continue till we encounter empty location 12.

### Deletion in Linear Probing





### Double Hashing

1. Uses two hash functions, h1 and h2.
2. H1(k) is the position in the table where we first check for key k.
3. H2(k) determines the offset we use when searching for k.
4. In linear probing, h2(k) is always 1.

DoubleHashingInsert(k)

If (table is full) error

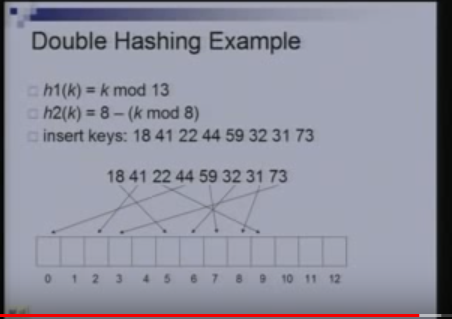
Probe = h1(k); offset = h2(k)

While(table[probe] occupied)

Probe = (probe+offset) mod m

Table[probe] = k

1. If m is prime, we will eventually examine every position in the table.
2. Many of the same disadvantages as linear probing
3. Distributes keys more uniformly than linear probing.



73 mod 13 = 8 occupied

8 – (73 mod 8) = 8 – 1 = 7

8 + 7 = 15 mod 13 = 2

2 + 7 = 9 occupied

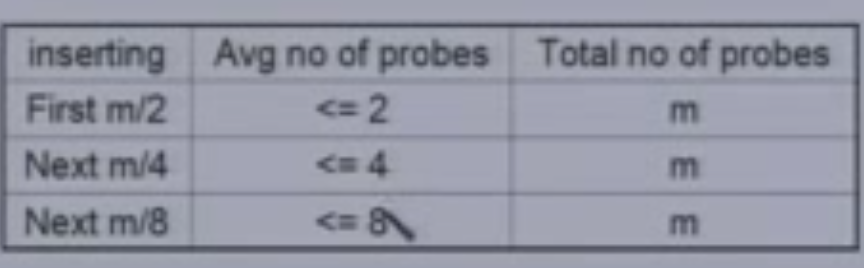
9 + 7 = 16 mod 13 = 3 empty

### Analysis of Double Hashing

1. The load factor = number of elements/size of table = x is less than 1.
2. We assume that every probe looks at a random location in the table.
3. 1 – x fraction of table is empty
4. Expected number of probes required for an unsuccessful search (before I hit empty location) is 1/(1-x)

#### For successful search

1. Average no of probes for a successful search = average no of probes required to insert all the elements.
2. To insert an element we need to find an empty location.
3. Suppose the table is empty, and I want to insert m/2 elements, m = 100, already inserted 48 or 49 elements but for 50th element I need 2 probes. So total number of probes required is <= m.
4. Already inserted m/2 and now trying to insert m/4, when trying to insert last element m/4 – 1, how much table is full m/2 + m/4 = 3m/4, so I need 4 probes for that element. m/4 x 4 = m = total number of probes.
5. To insert m/8 elements, required between 4 and 8 probes, so the total number of probes = m/8 \*8 = m



1. Number of probes required to insert m/2 + m/4 + m/8 + … + m/2^i elements = m x i
2. After inserting m/2, ½ table is empty, after inserting m/4, ¼ is empty, after inserting m/2^I,

1/ 2^I is empty = 2^-I empty

2^-I = - log(2^i) = I so we need m x i

1. Number of probes required to leave 1 – x fraction of table empty = -m log(1-x)
2. Average number of probes required to insert n elements =

-(m/n)log(1-x) = (1/x)log(1-x)

