process, abstraction denotes the extracting of the essential details about an item, or a group of items, ignoring the nonessential details.

Entity, abstraction denotes a model, a view, or some representation for an actual item which leaves out some of the details of the item.

Abstraction dictates that some information is more important than other information, but does not provide a specific mechanism for handling the unimportant information.

The aim of data abstraction is to identify which details of how data is stored and can be manipulated are important and which are not.

The aim of procedural abstraction is to identify which details of how a task is accomplished are important and which are not.

Abstraction leads to a top-down approach.

The Huffman encoding software uses priority queue to accomplish its tasks.

A priority queue supports the following operations: – INSERT(Q,x) inserts the element x into Q. – MIN(Q) returns the element of Q with minimal key. – EXTRACT-MIN(Q) removes and returns the element of Q with minimal key.

Information hiding tells us that we should actively keep all unimportant details secret from the user and try to prevent him from making use any unimportant details

The signature of a method is the combination of – method's name along with – number and types of the parameters (and their order).

As a process, encapsulation means the act of enclosing one or more items (data/functions) within a (physical or logical) container.

As an entity, encapsulation, refers to a package or an enclosure that holds (contains, encloses) one or more items (data/functions).

The separator between the inside and the outside of this enclosure is sometimes called wall or barrier

In object-oriented programming, encapsulation is the inclusion within an object of all the resources needed for the object to function – i.e., the methods and the data.

The object is said to publish its interfaces.

In communication, encapsulation is the inclusion of one data structure within another structure so that the first data structure is hidden.

Abstraction is a technique that helps us identify which specific information is important for the user of a module, which information is unimportant

Information hiding is the principle that all unimportant information should be hidden from a user

Encapsulation is then the technique for packaging the information in such a way as to hide what should be hidden, and make visible what is intended to be visible

Simpler, modular programs that are easier to design & understand

Side-effects from direct manipulation of data are eliminated or minimised

Localisation of errors (only methods defined on a class can operate on the class data), which allows localised testing

Program modules are easier to read, change, and maintain

speed of execution will be the number of accesses to data items stored in the data structure.

**Dynamic data** structures grow or shrink during run-time to fit current requirements e.g. a structure used in modelling traffic flow.

**Static data** structures are fixed at the time of creation • e.g. a structure used to store a postcode or credit card number (which has a fixed format).

Note that in the definition of a static data structure we have placed no constraints on when it is created, only that once it is created it will be fixed.

Static data structures

Advantages

Programming languages usually provide an easy way to create static data structures of almost arbitrary size no memory allocation overhead

Since static data structures are fixed in size, – there are no operations that can be used to extend static structures; – such operations would need to allocate additional memory for the structure (which takes time)

Disadvantages : must make sure there is enough capacity

more elements? (errors), fewer elements? (waste)

Dynamic data structures

Advantages

There is no requirement to know the exact number of data items since dynamic data structures can shrink and grow to fit exactly the right number of data items, there is no need to know how many data items we will need to store Efficient use of memory space

Disadvantages

Memory allocation/de-allocation overhead

Whenever a dynamic data structure grows or shrinks, then memory space allocated to the data structure has to be added or removed (which requires time)

Collection: bag, set, map, list, queue, stack, graph, tree

JAVA collections library

Interfaces: Classes:

Clooection: bag List classes: arraylist, linkedlist, vector

List: order collection Set classes: HashSet, Tree set

Set: unordered, duplicates Map classes: HashMap, Treemap

Queue: ordered collection, limited access (add at one end, remove from other)

A Java Interface corresponds to an Abstract Data Type

Specifies what methods can be called on objects of this type (specifies name, parameters and types, and type of return value) • Behaviour of methods is only given in comments (but cannot be enforced):

× No constructors - can’t make an instance: new Set()

× No fields - doesn’t say how to store the data

× No method bodies. - doesn’t say how to perform the operations

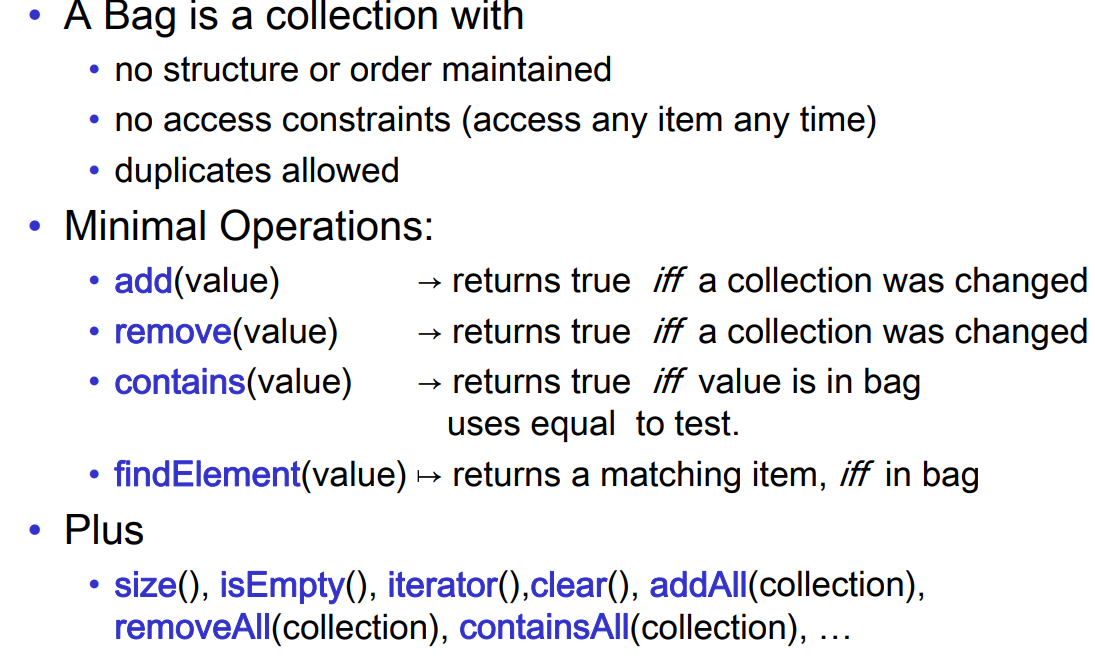
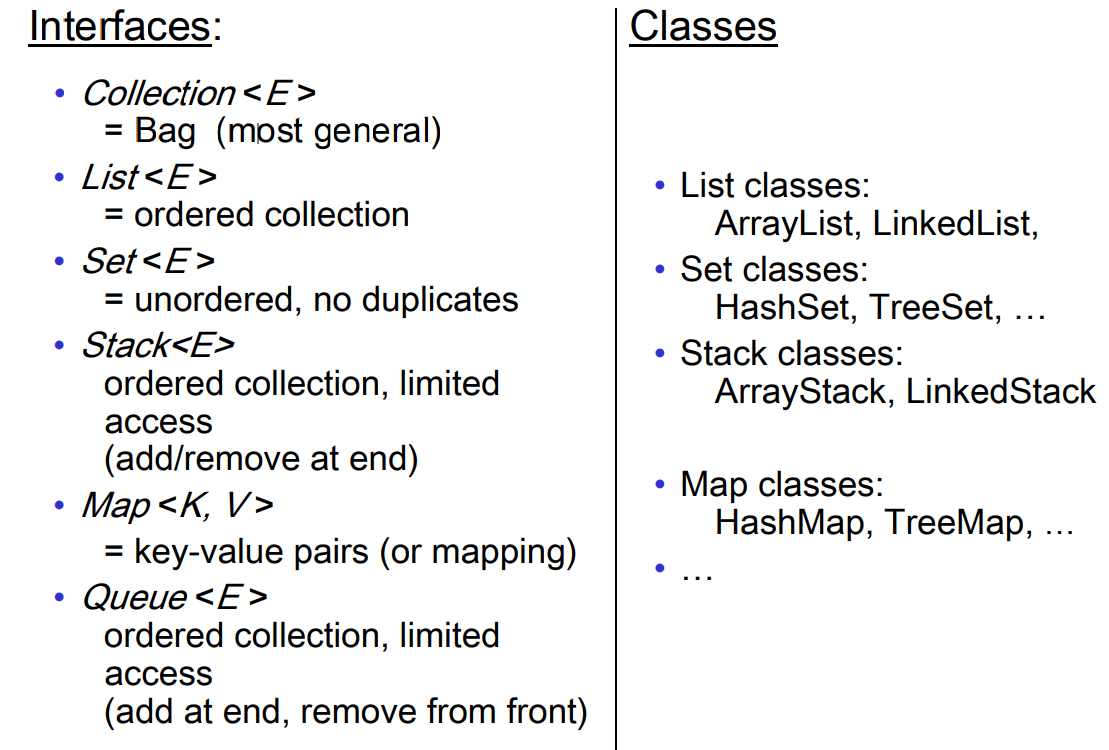
Classes in the Java Collection Library implement the interfaces

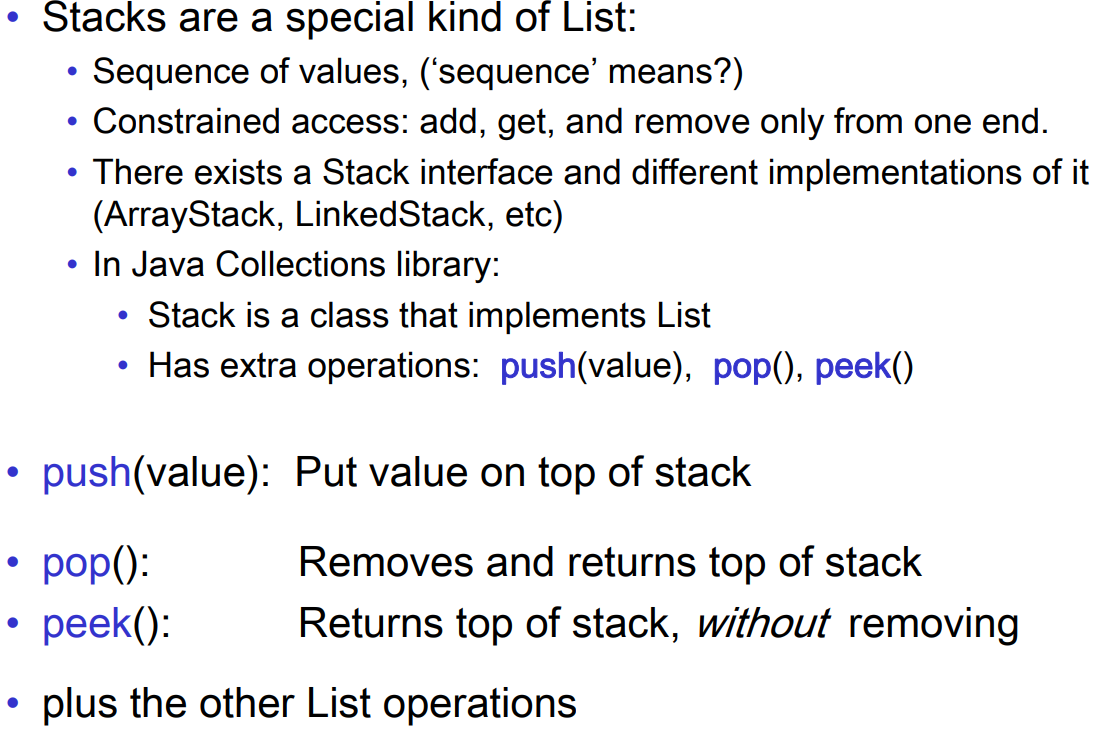
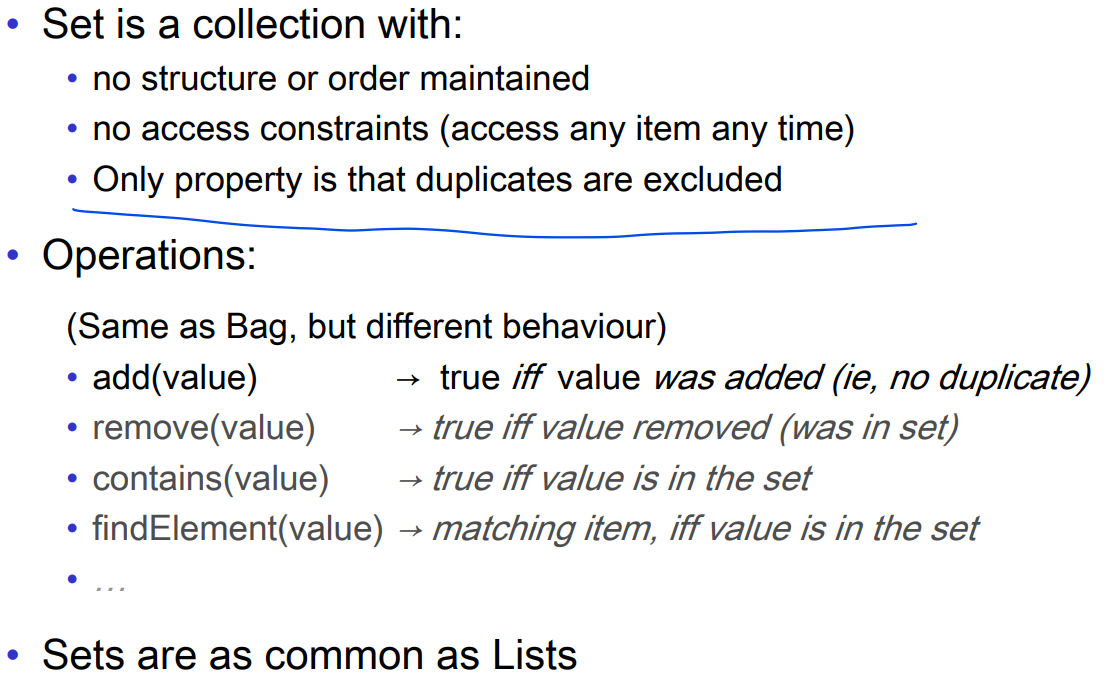
• Define constructors to construct new instances

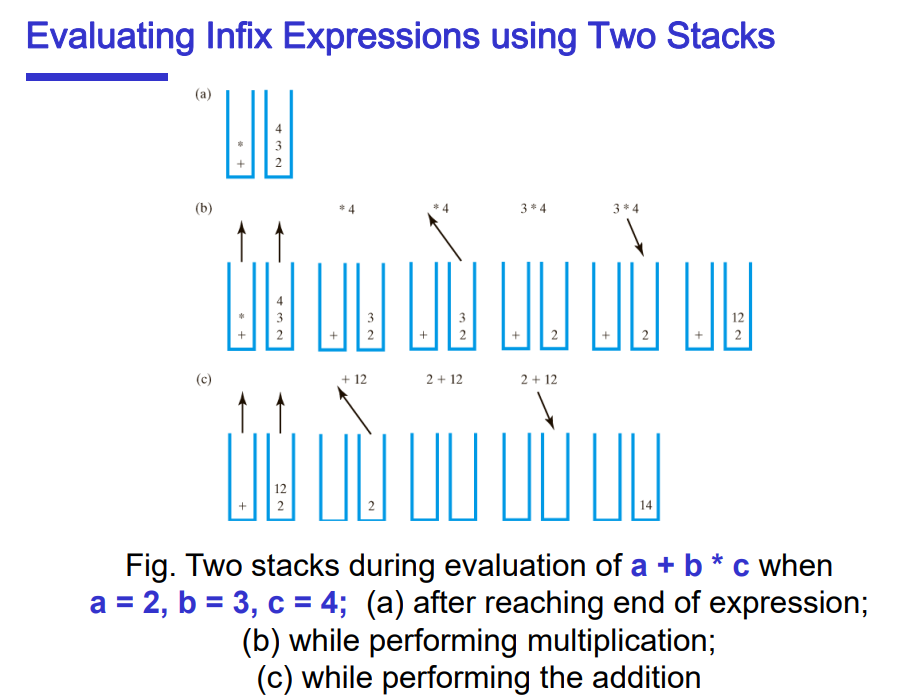
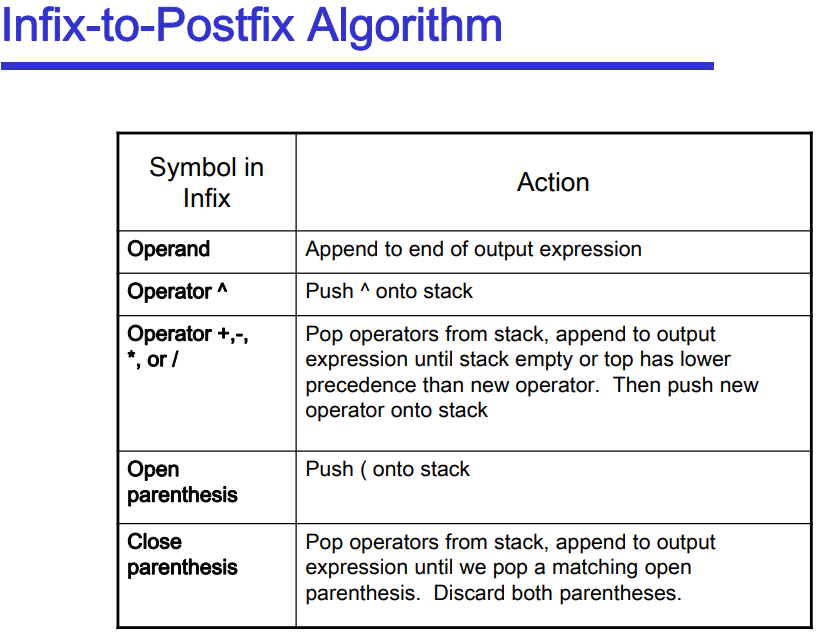
• Define method bodies for performing the operations

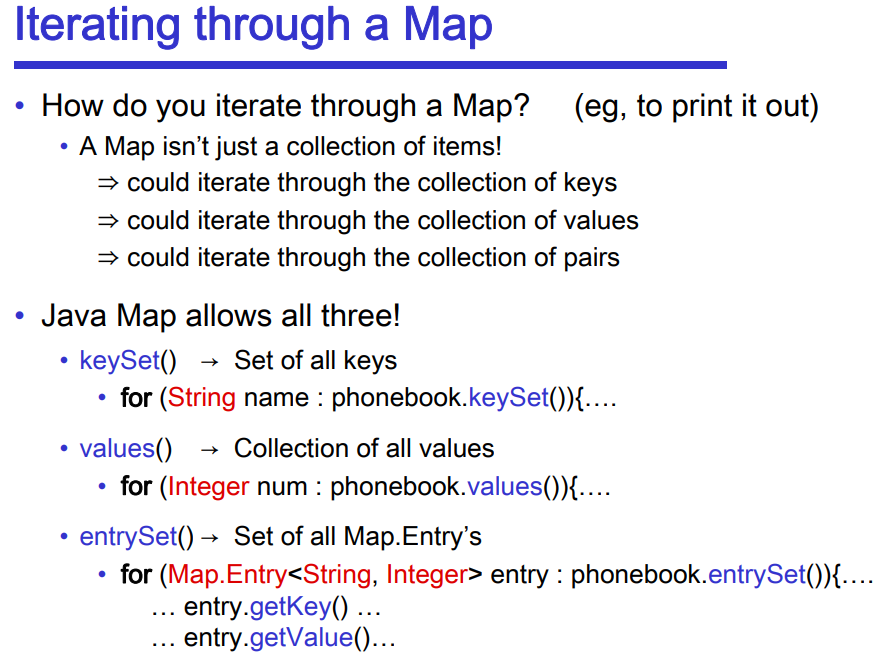
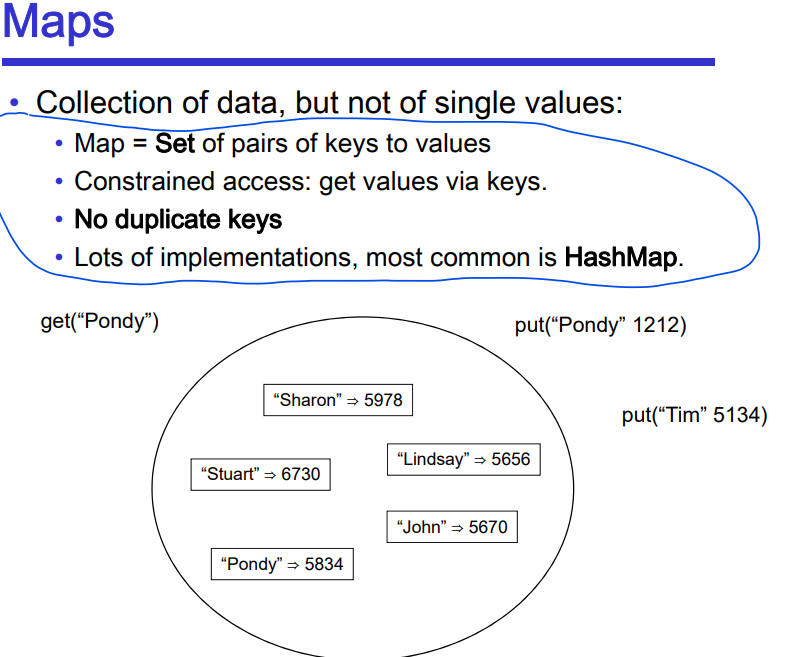
• Define fields to store the values

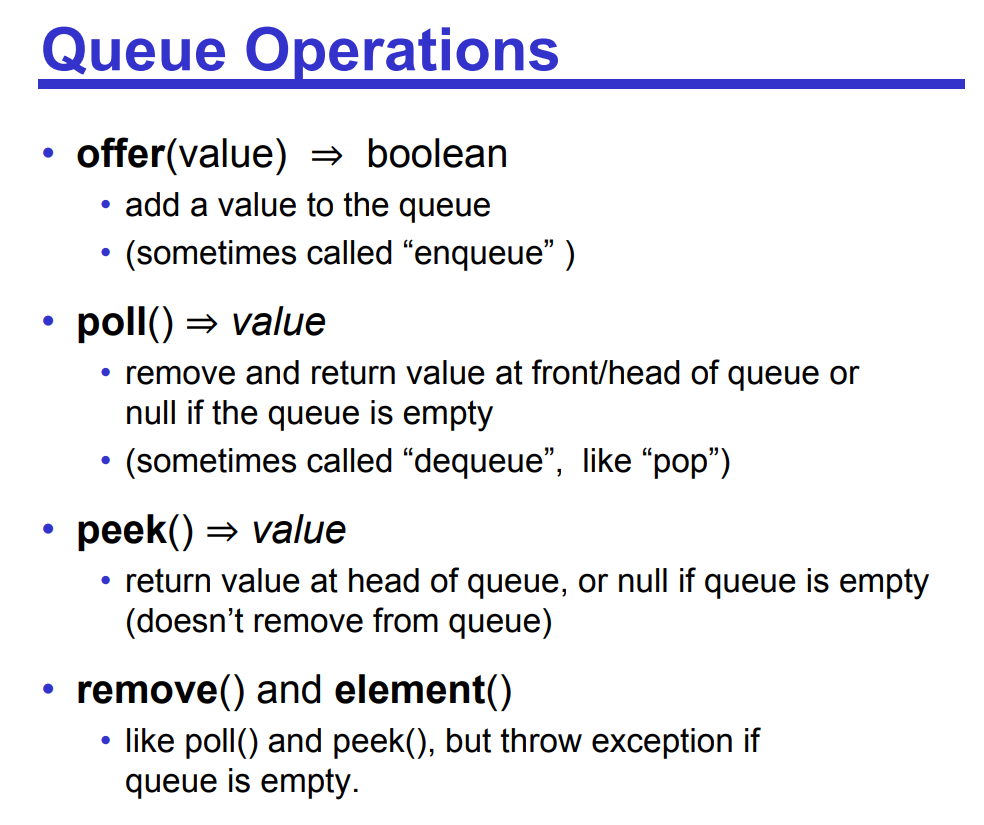
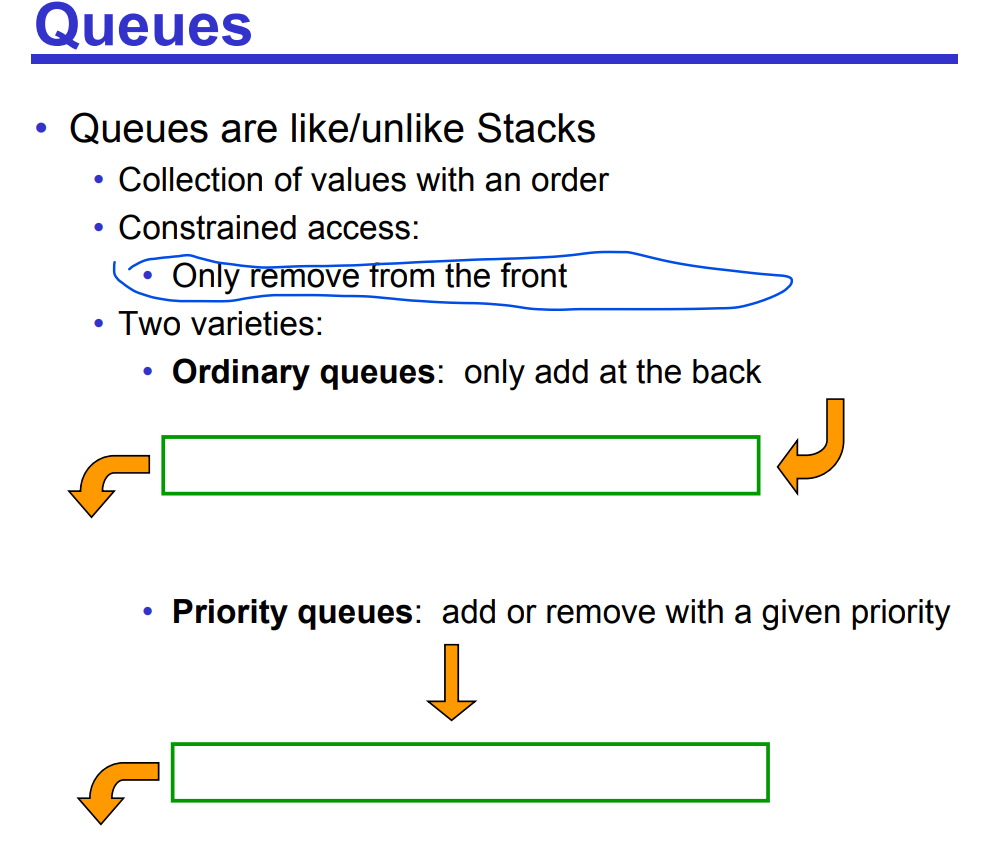
Interfaces can extend other interfaces: The sub interface has all the methods of the super interface plus its own methods

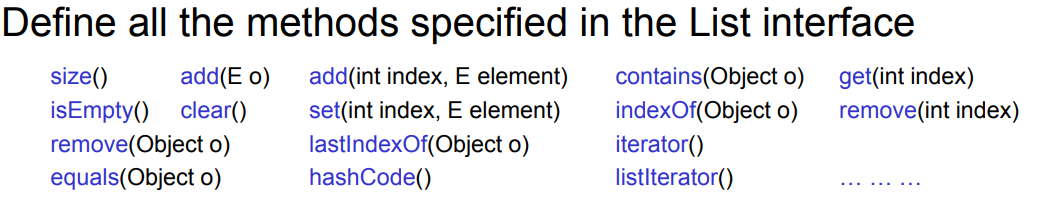
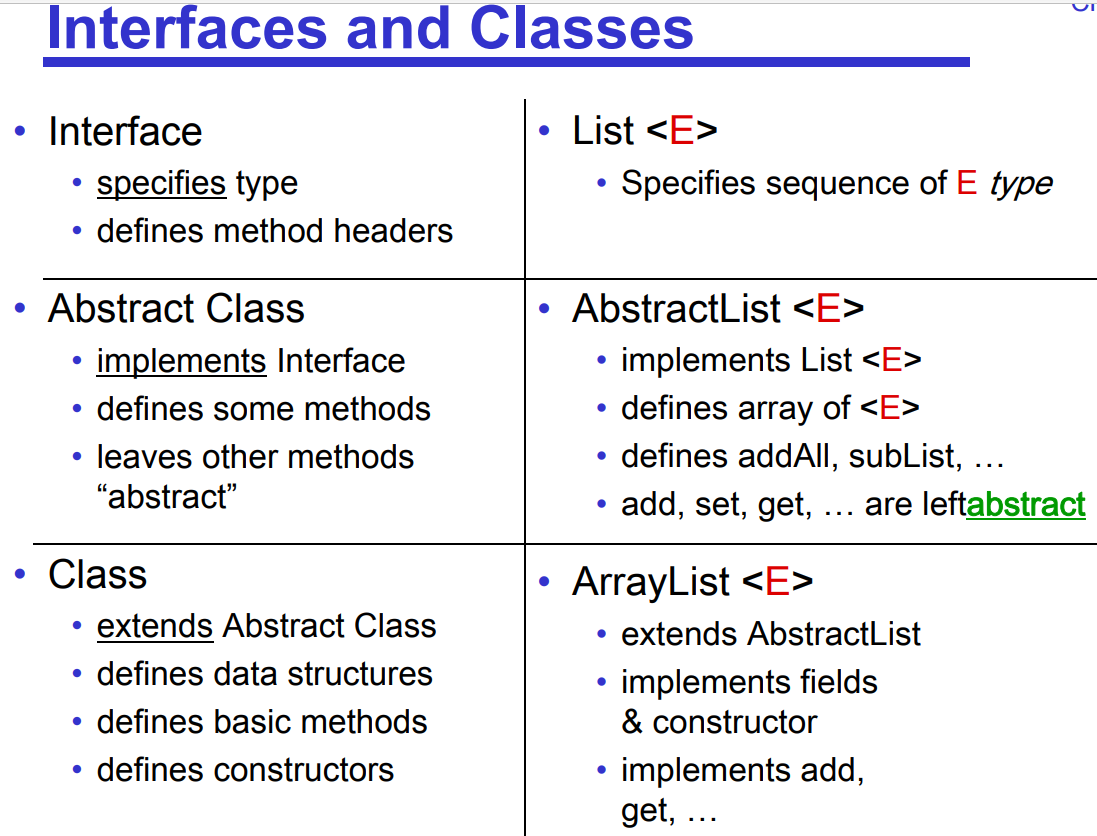
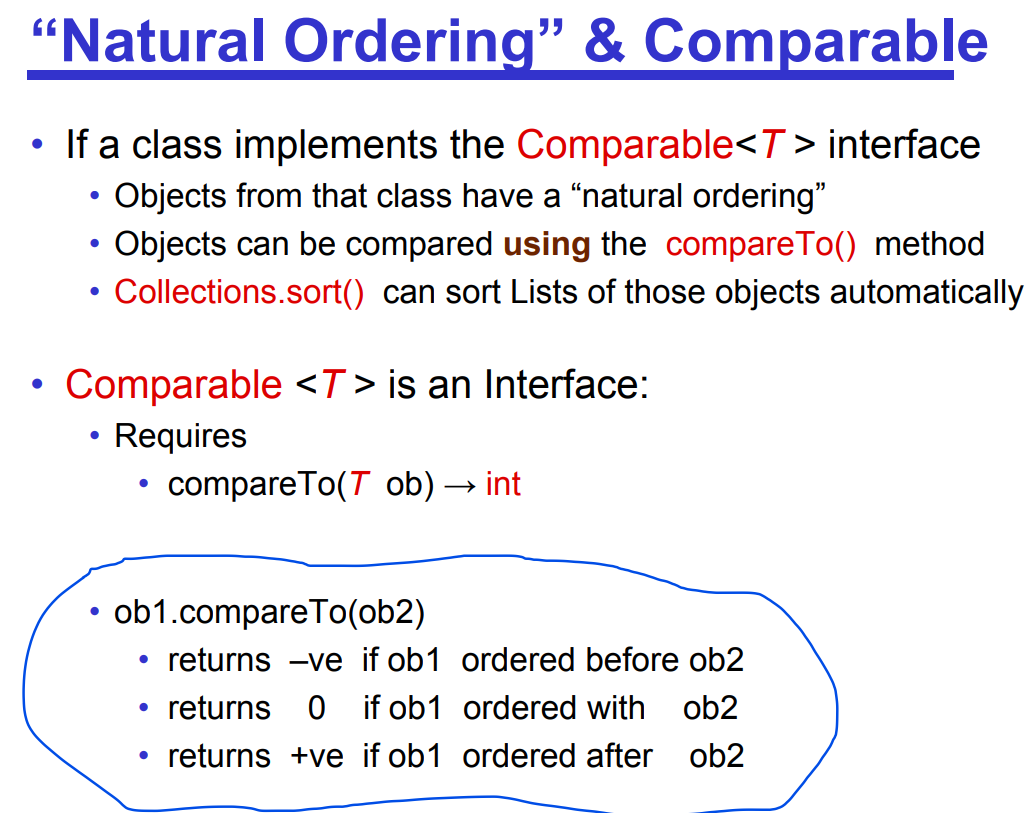


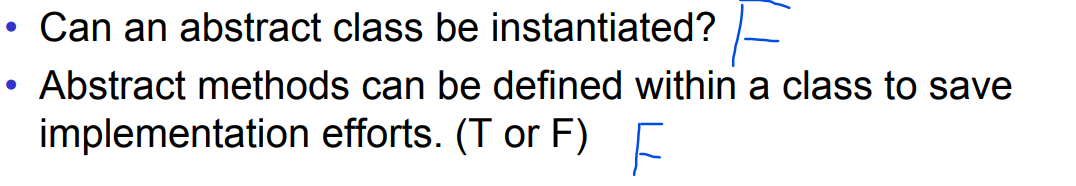
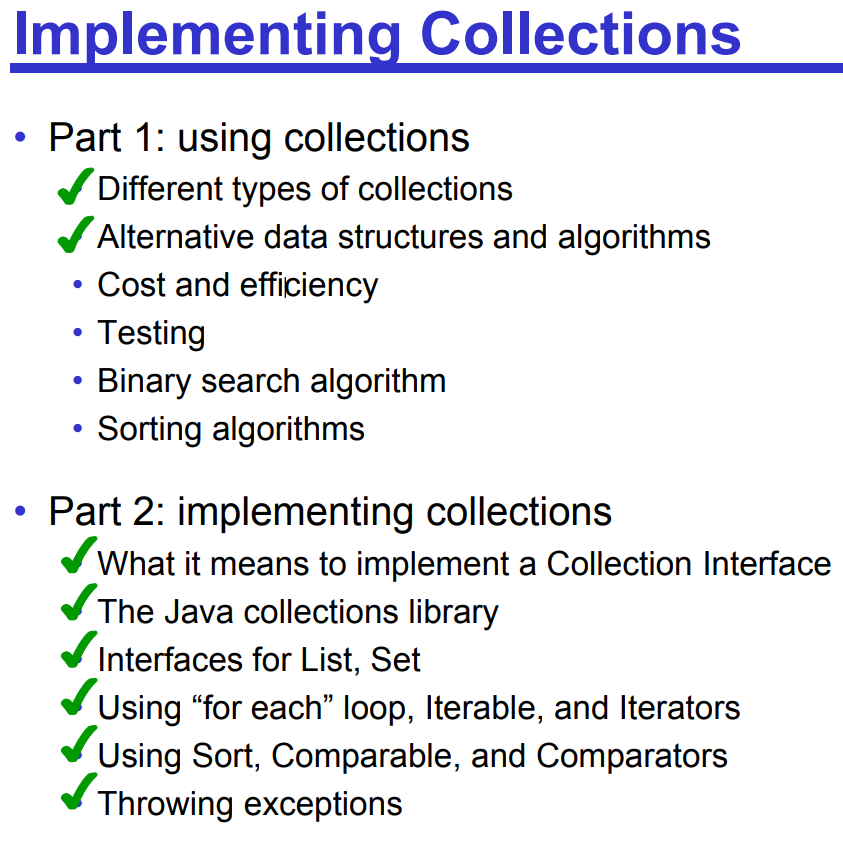












RuntimeExceptions don’t have to be handled:

• An uncaught RuntimeException will result in an error message

• You can catch them if you want.

• Other Exceptions must be handled: • eg IOException (which is why we always used a try…catch when opening files). • An exception object contains information: the state of the execution stack any additional information specific to the exception

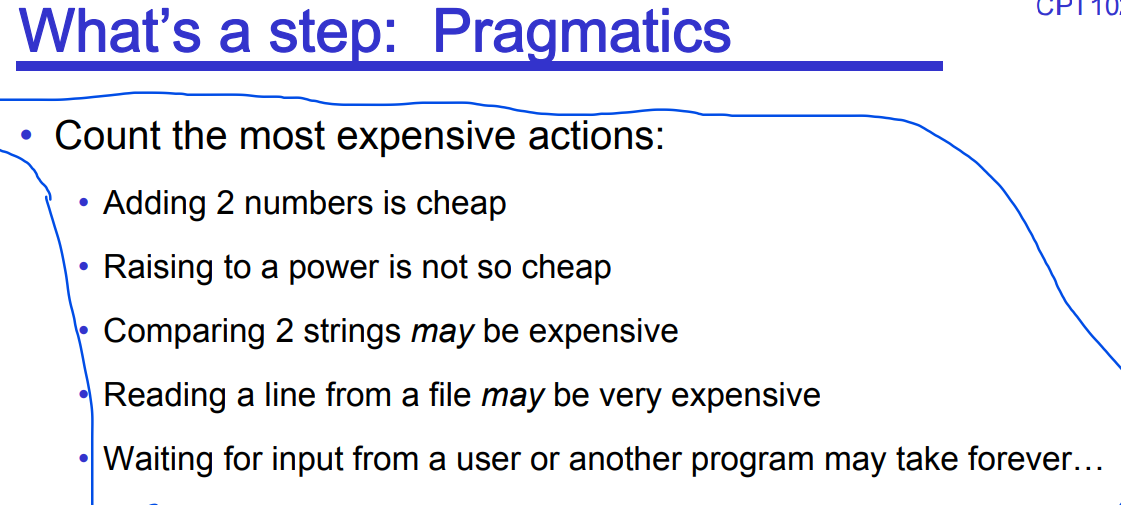
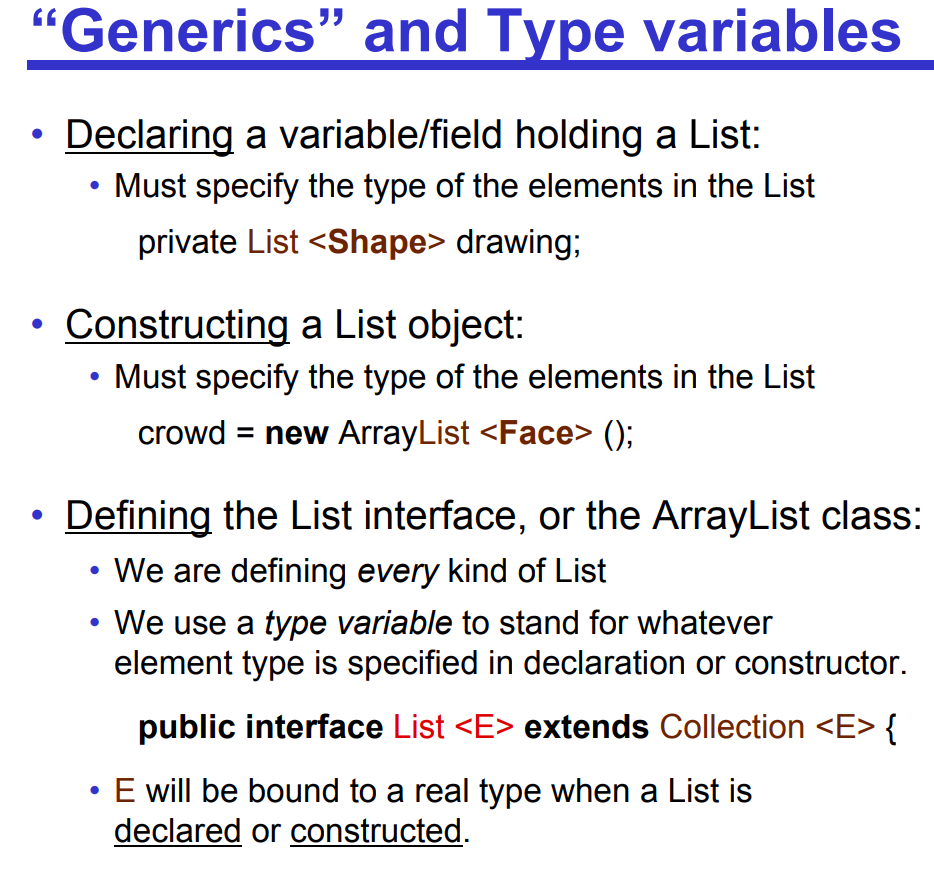
How can we determine the costs of a program?

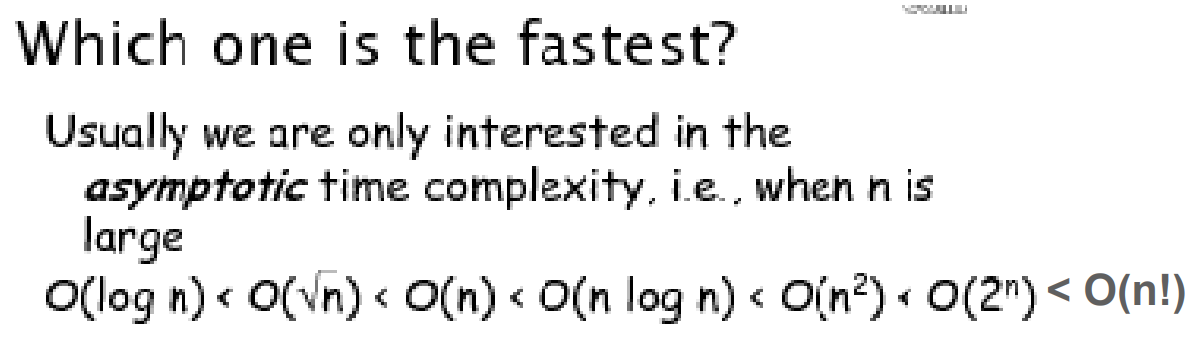
• Time: • Run the program and count the milliseconds/minutes/days. • Count the number of steps/operations the algorithm will take.

• Space: • Measure the amount of memory the program occupies • Count the number of elementary data items the algorithm stores.

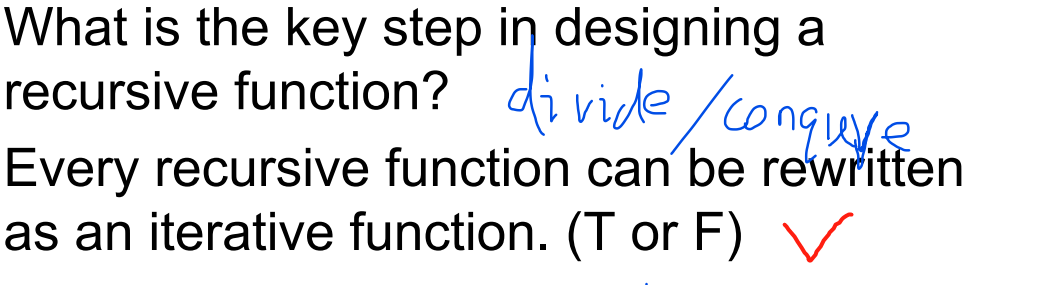
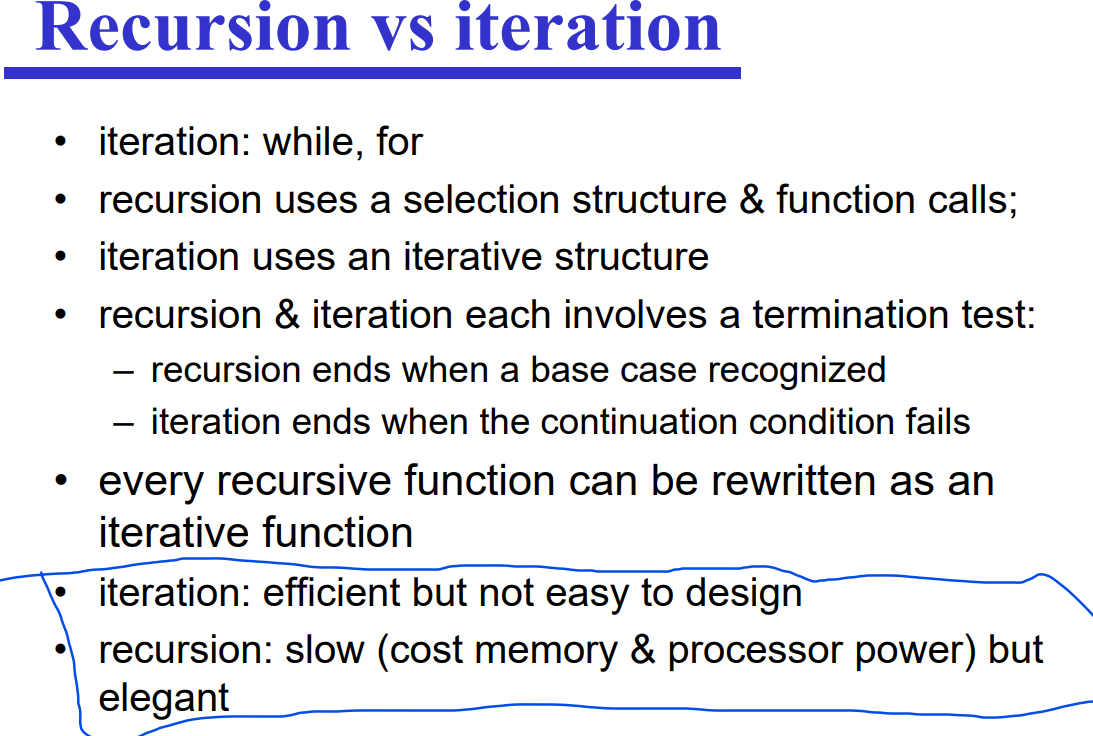
Measure number of “steps” as a function of the data size. • worst case (easier) • average case (harder) • best case (easy, but useless)

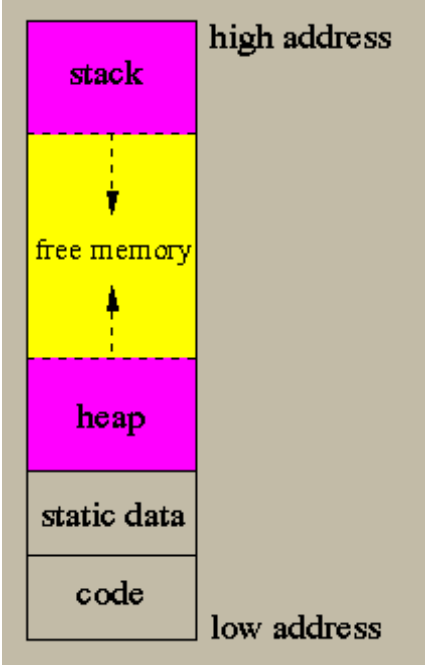
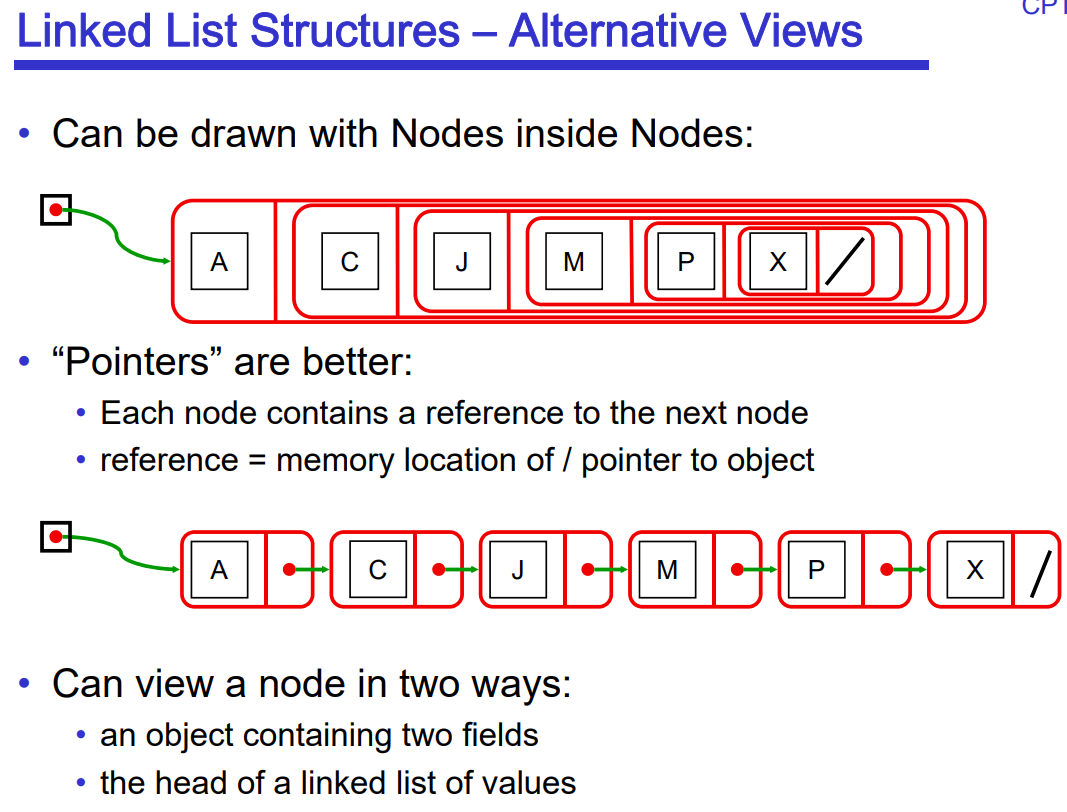
• Actual constant will depend on the hardware



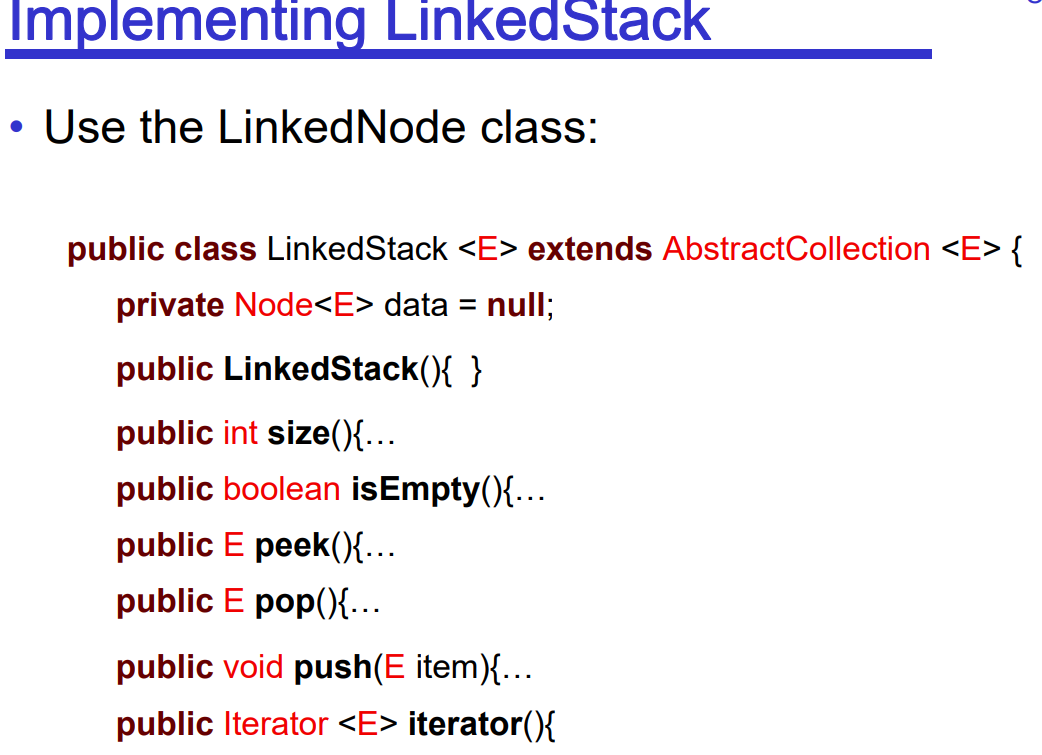


ArrayList costs: Summary • get O(1) • set O(1) • remove O(n) • add (at i) O(n) (worst and average) • add (at end) O(1) (average) O(n) (worst) O(1) (amortised average) • contains: O(n) • remove: O(1) O(n) • add: O(1) O(n)



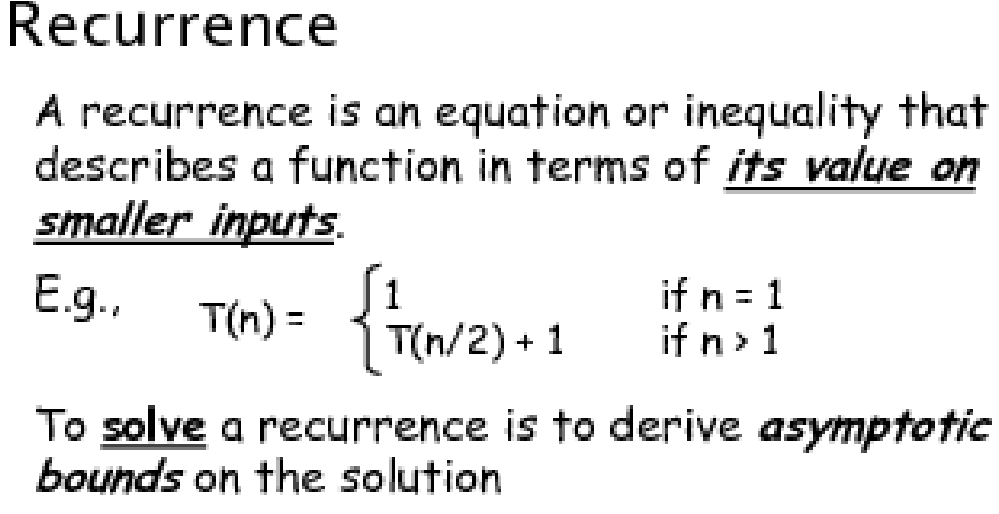
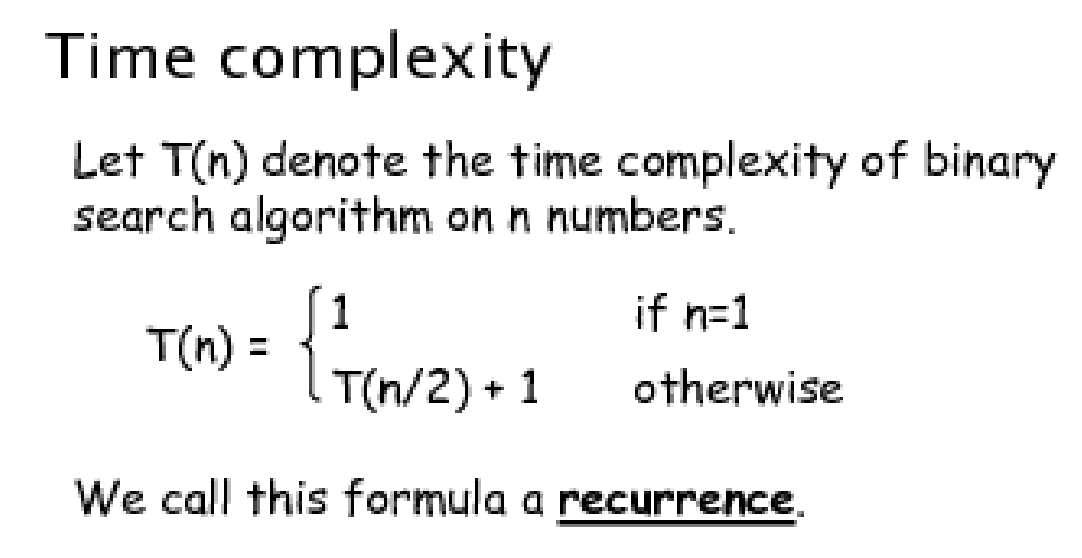






Linked Stack and Queue

• Uses a “header node” • contains link to head node, and maybe last node of linked list • Important to choose the right end. • easy to add or remove from head of a linked list • hard to add or remove from the last node of a linked list • easy to add to last node of linked list if have pointer to tail • Linked Stack and Queue: • all main operations are O(1) • Can combine Stack and Queue • addFirst, addLast, removeFirst • also need removeLast to make a “Deque” (double-ended queue) ⇒ need doubly linked list



ArraySet with Binary Search ArraySet: unordered • All cost in the searching: O(n) • contains: O(n ) • add: O(n ) • remove: O(n ) SortedArraySet: with Binary Search • Binary Search is fast: O(log( n )) • contains: O(log(n )) • add: O( n ) • remove: O( n )

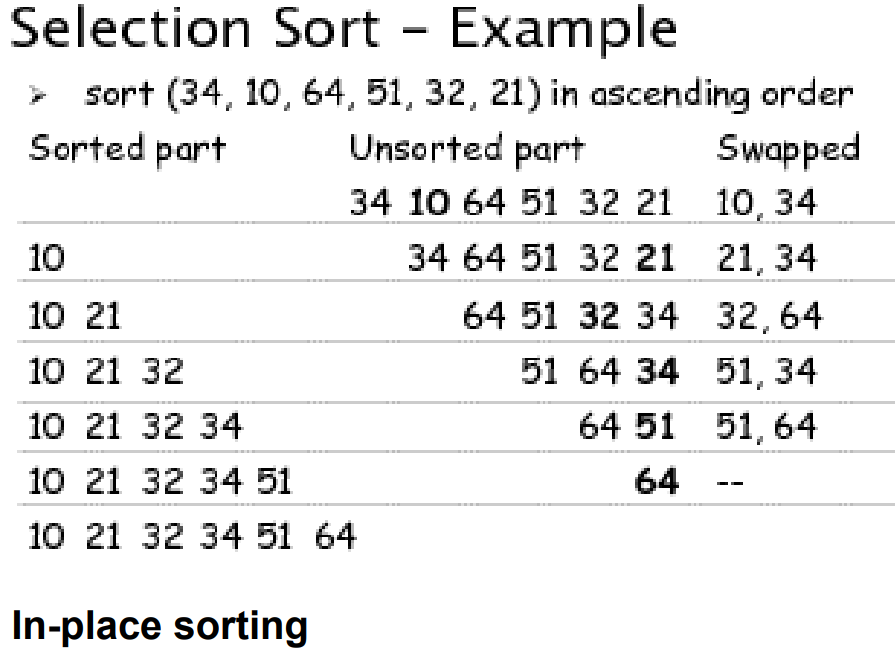
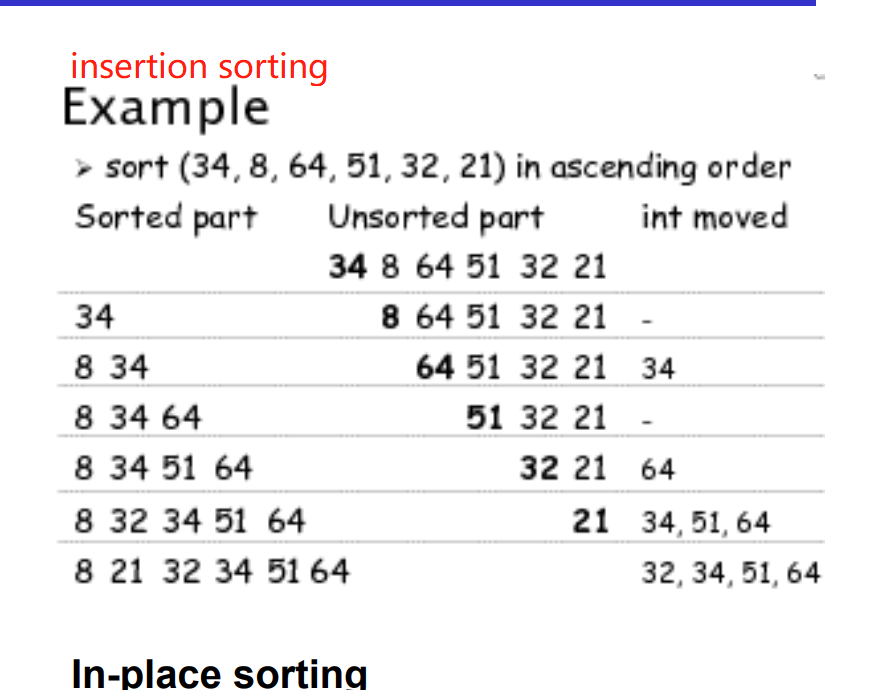
Ways of sorting

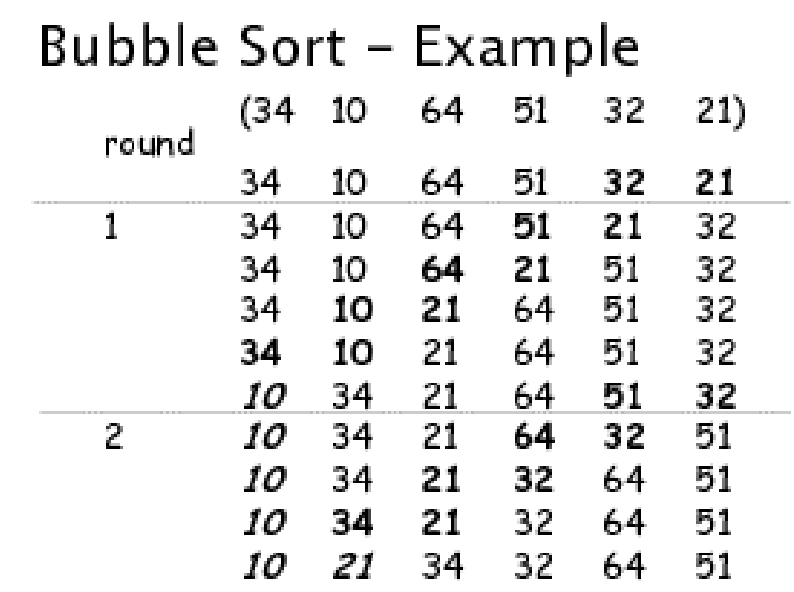
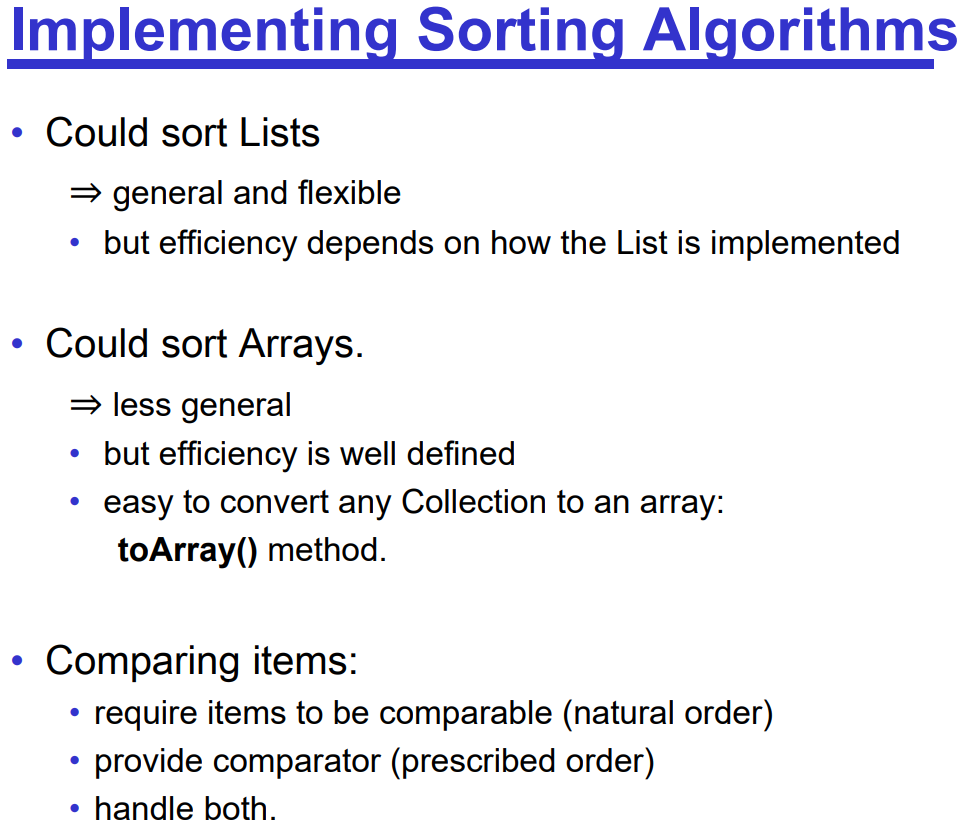
• Selecting sorts: • Find the next largest/smallest item and put in place • Builds the correct list in order

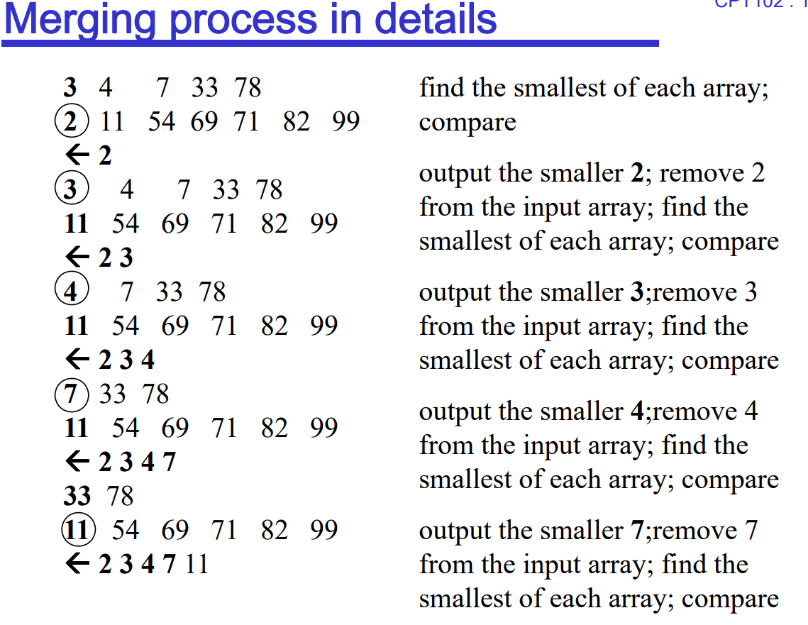
• Inserting Sorts: • For each item, insert it into an ordered sublist • Builds a sorted list, but keeps changing it

• Compare and Swap Sorts: • Find two items that are out of order, and swap them • Keeps “improving” the list

• Radix Sorts • Look at the item and work out where it should go. • Only works on some kinds of values

also called as sinking sort

• smaller values bubble their way up during the process

Slow sorts

Insertion sort, Selection Sort, Bubble Sort:

• All slow (except Insertion sort on almost sorted lists)

• Insertion sort is better than the others

• Problem:

• Insertion and Bubble

• only compare adjacent items

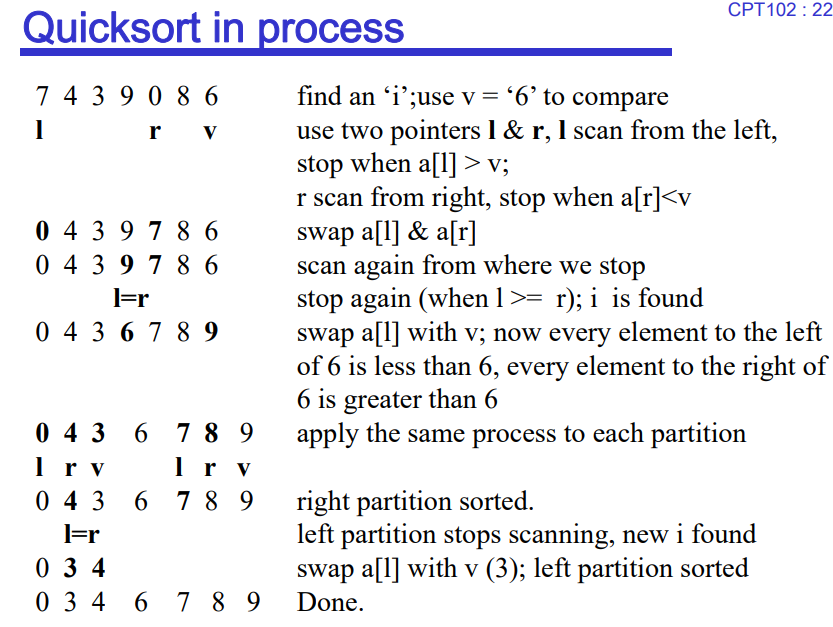
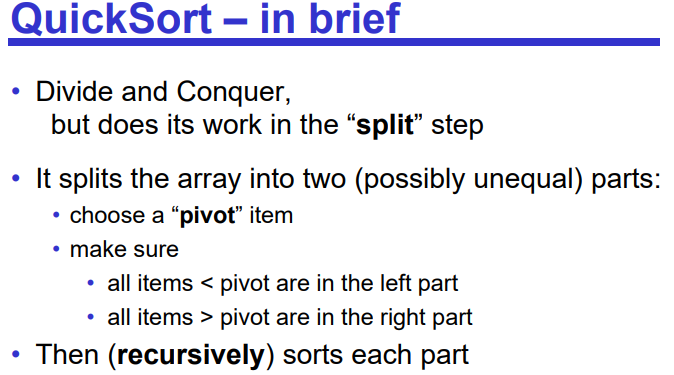
• only move items one step at a time

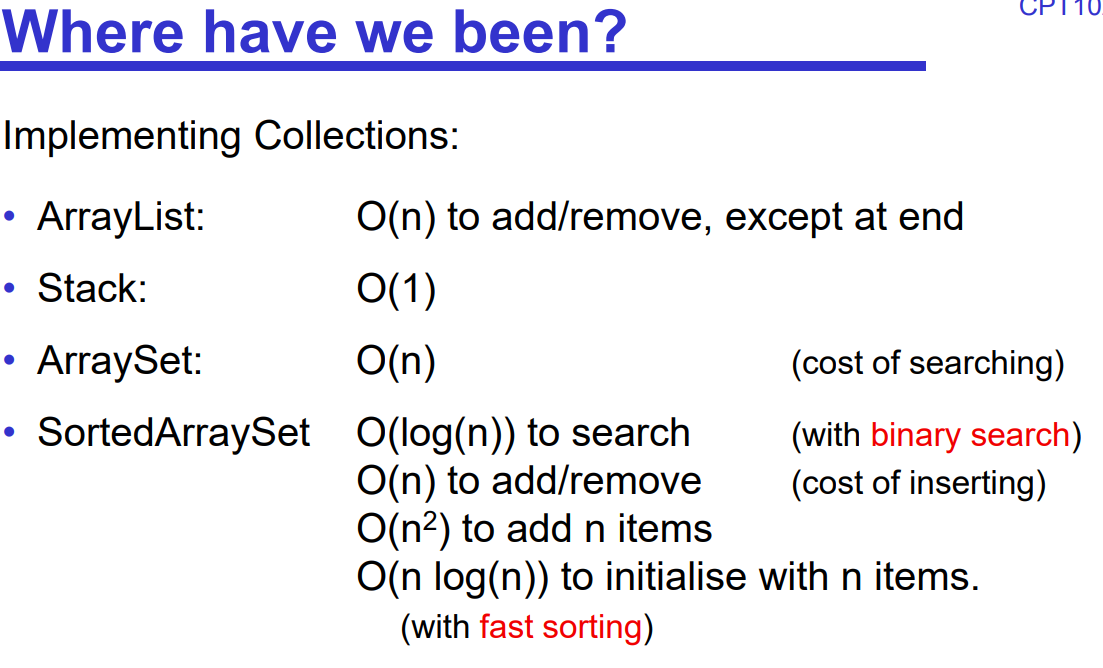
• Selection

• compares every pair of items –

• ignores results of previous comparisons.

• Solution: • Must be able to compare and swap items at a distance • Must not perform redundant comparisons



Trees

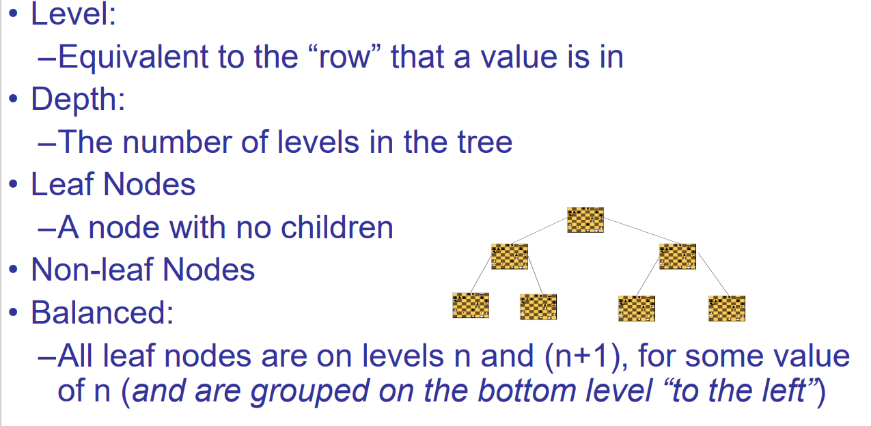
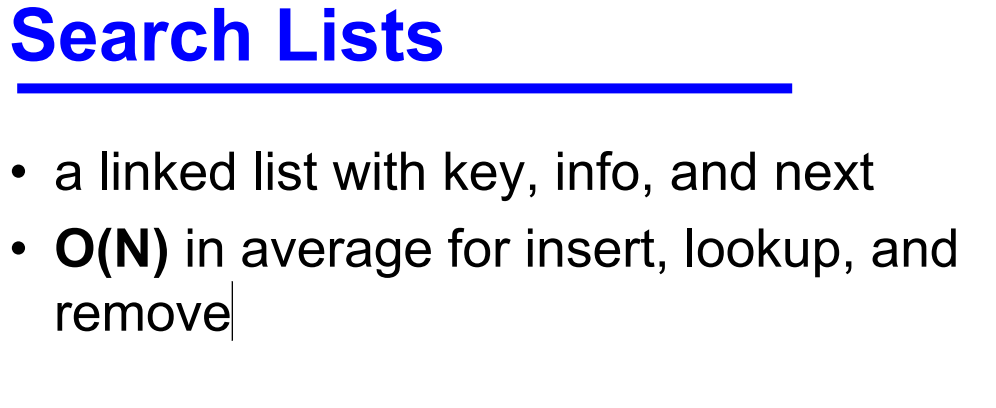
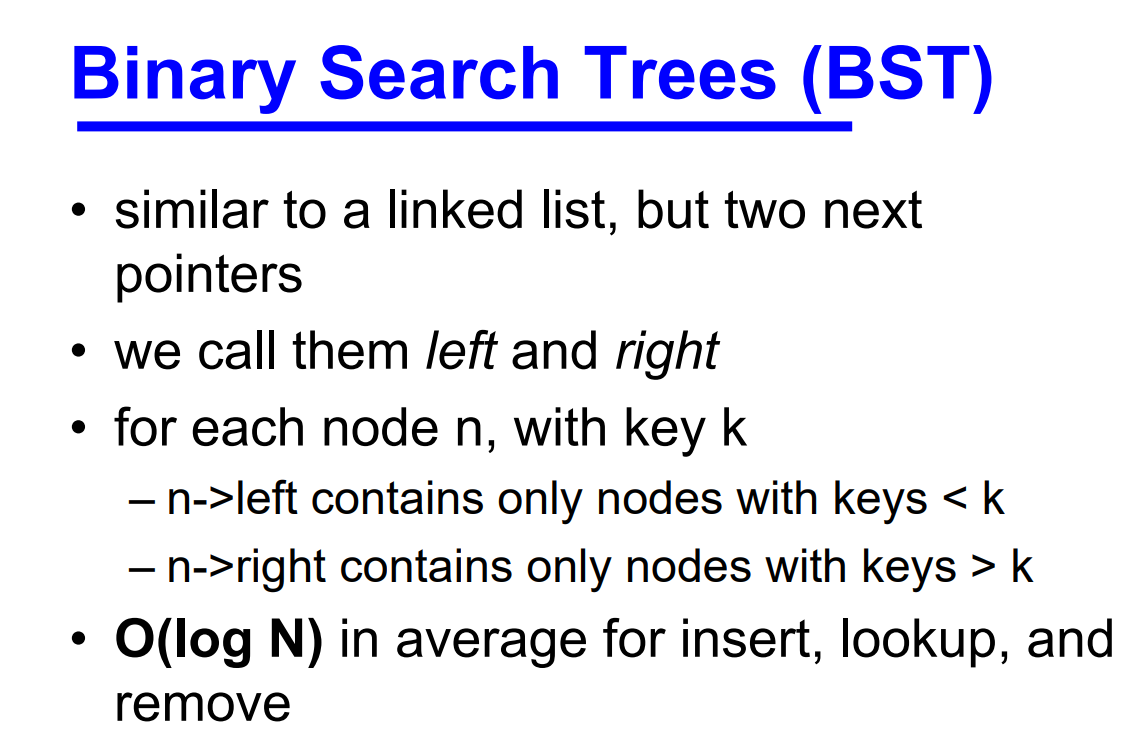
• linear data structures: lists, stacks, queues

**Binary Tree**

• Every node in the tree has a left child (or null) • Every node in the tree has a right child (or null) • The root of the tree is just another node• non-linear data structures: tree

• A binary tree is either: –empty or –a root node together with two binary trees - left subtree & right subtree of the root

What is a tree useful for?

• Artificial Intelligence – planning, navigating, games • Representing things: –Simple file systems –Class inheritance and composition –Classification, e.g. taxonomy (the is-a relationship) –HTML pages –Parse trees for languages –Essential in compilers like Java, C# etc. –3D graphics (e.g. BSP trees)   


**Binary Search Tree Definitions**

• A binary search tree is a binary tree where each node has a key

• The key in the left child (if exists) of a node is less than the key in the parent

• The key in the right child (if exists) of a node is greater than the key in the parent

• The left & right subtrees of the root are again binary search trees

**Binary Search Trees (BST)**

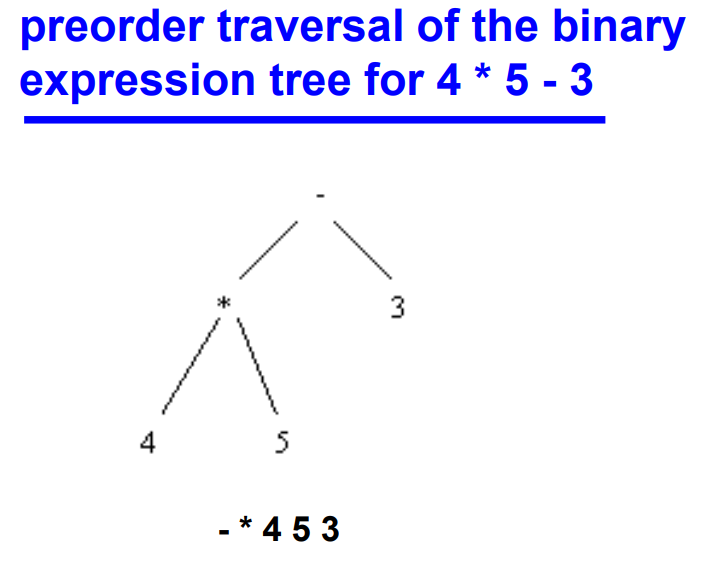
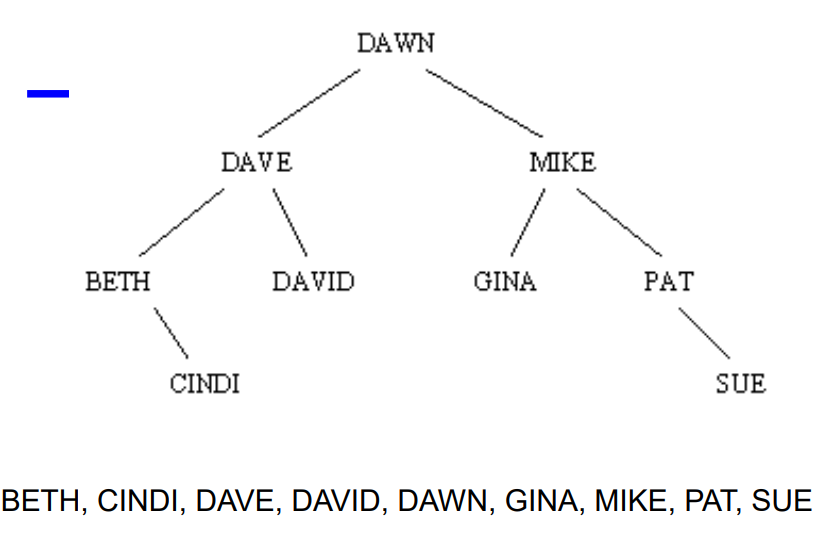
• similar to a linked list, but two next pointers

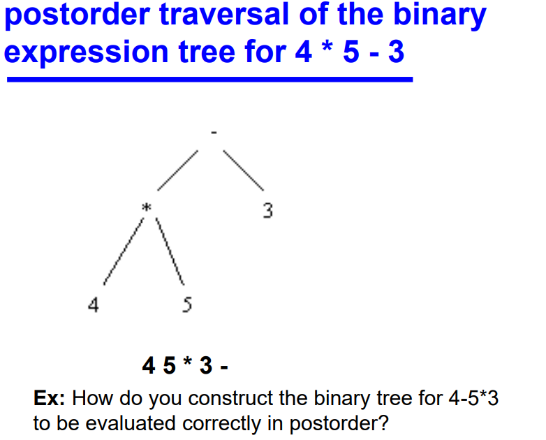
• we call them left and right

• for each node n, with key k – n->left contains only nodes with keys < k – n->right contains only nodes with keys > k

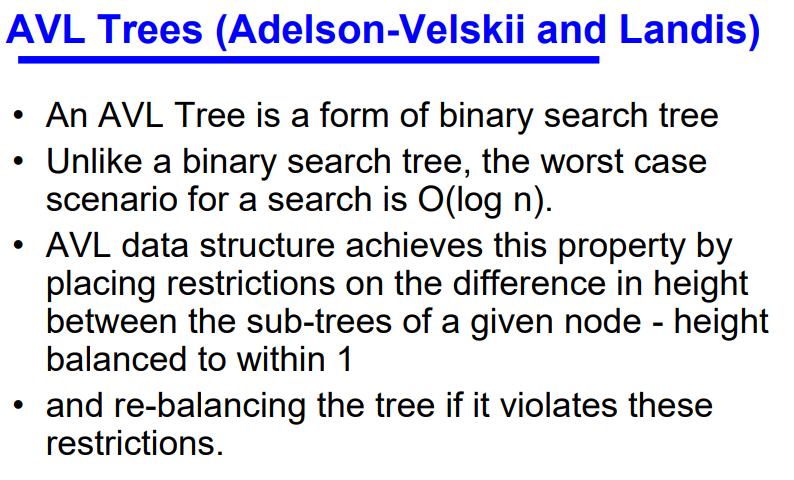
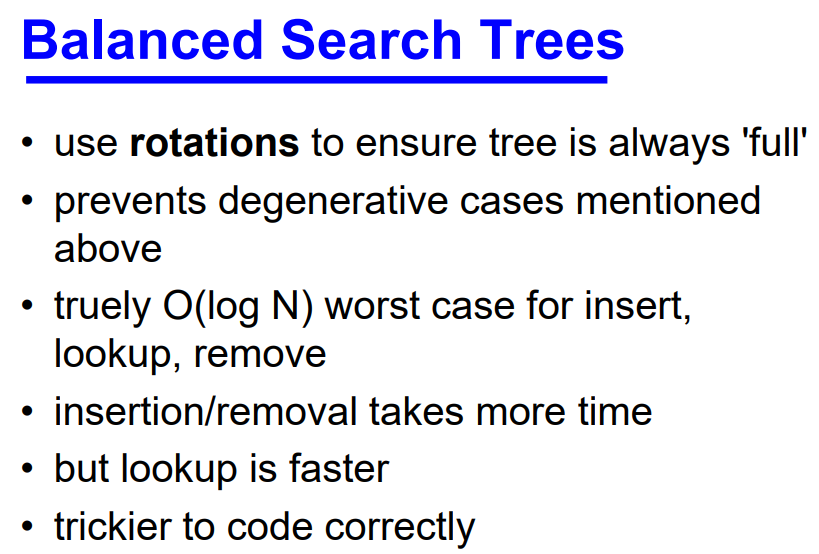
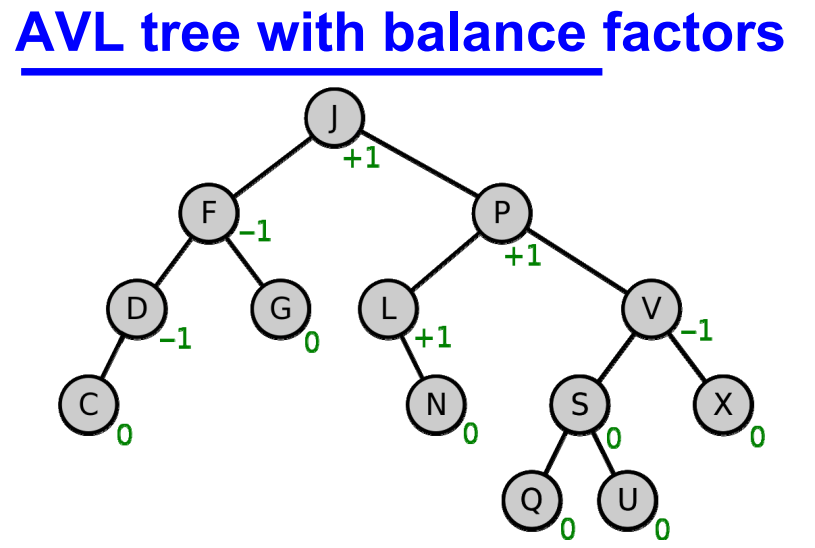
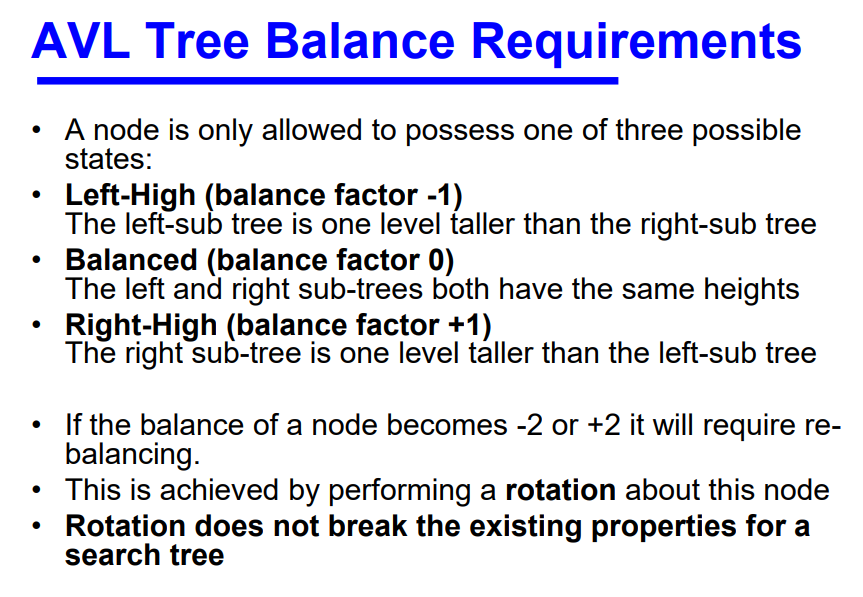
• O(log N) in average for insert, lookup, and remove

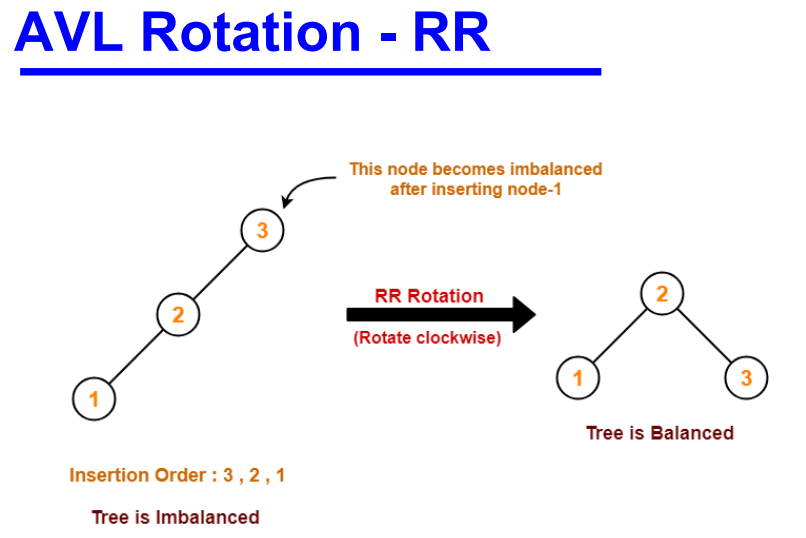
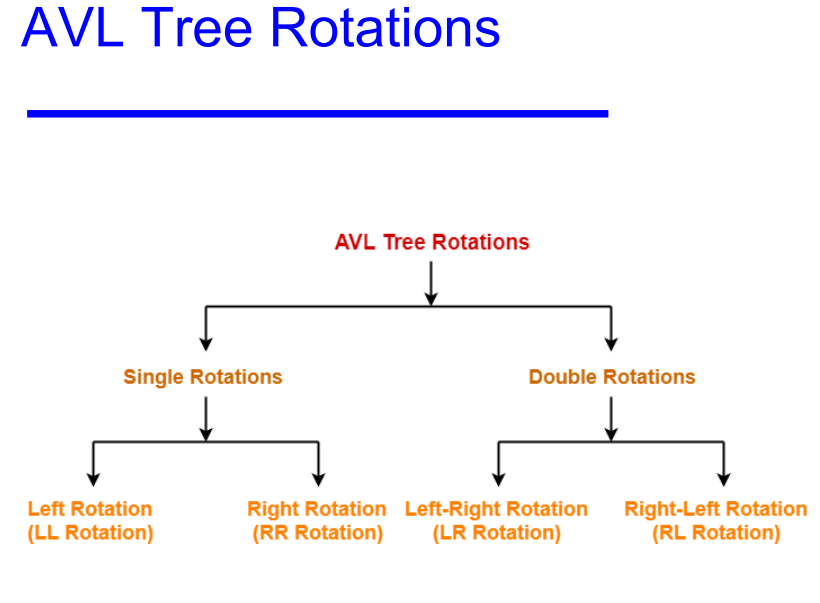
**Inorder: 左中右 Preorder:中左右**

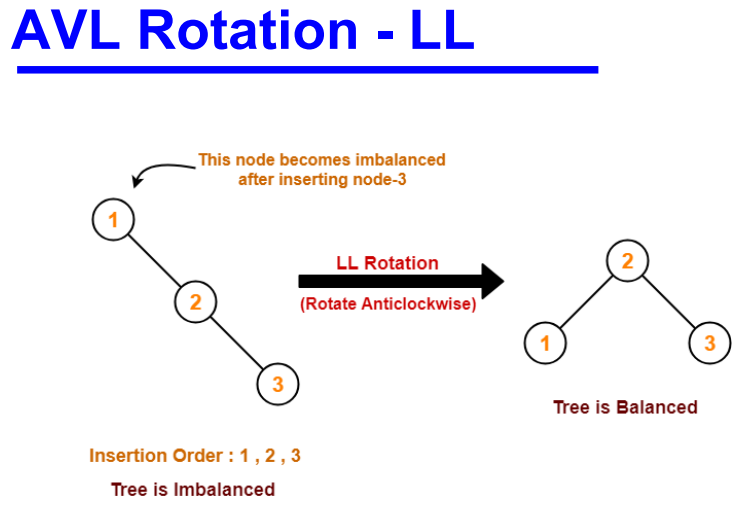
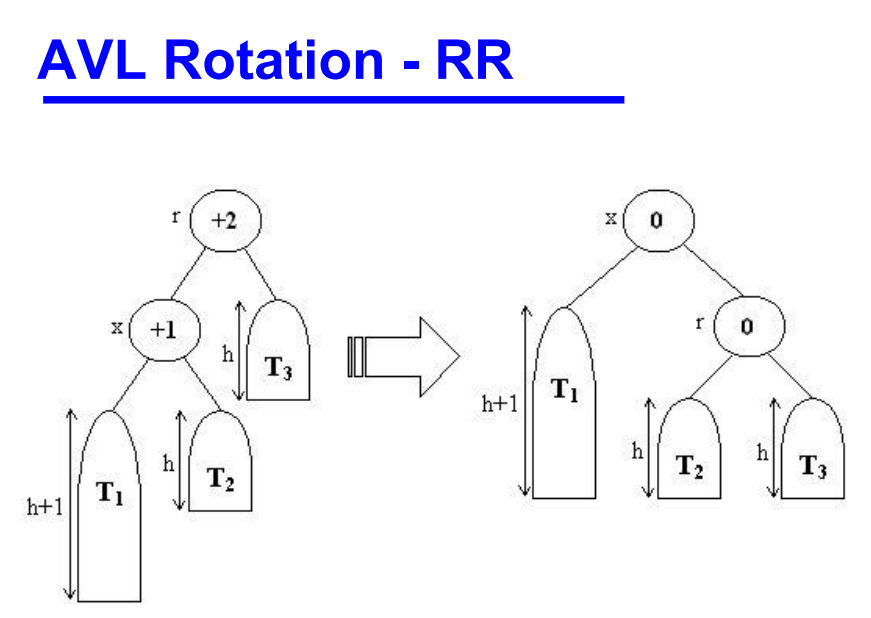


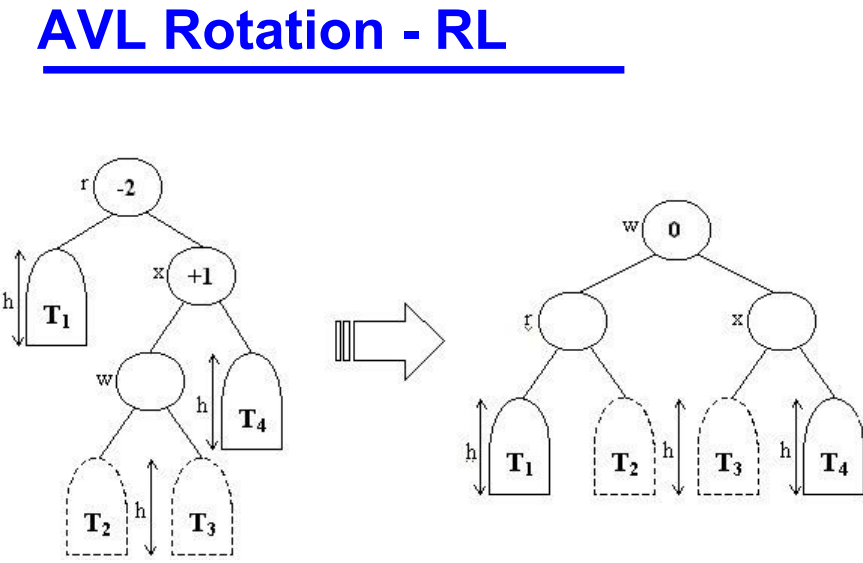
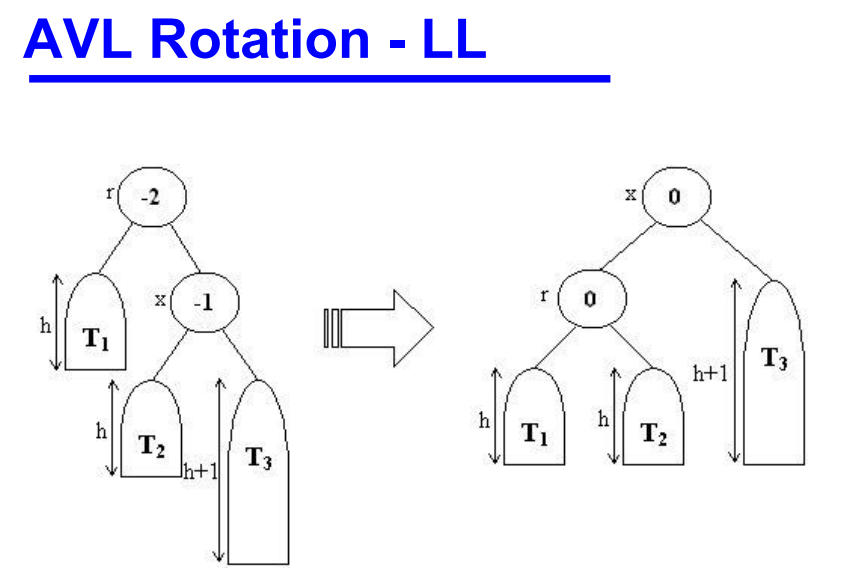


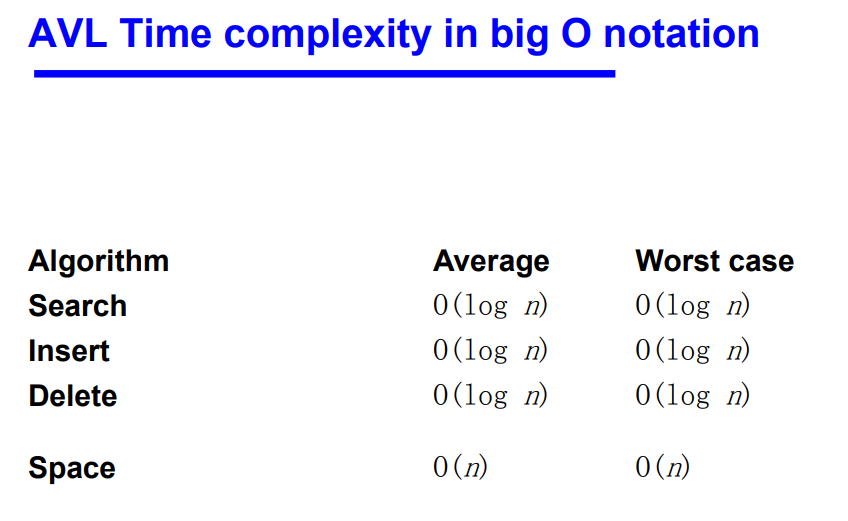
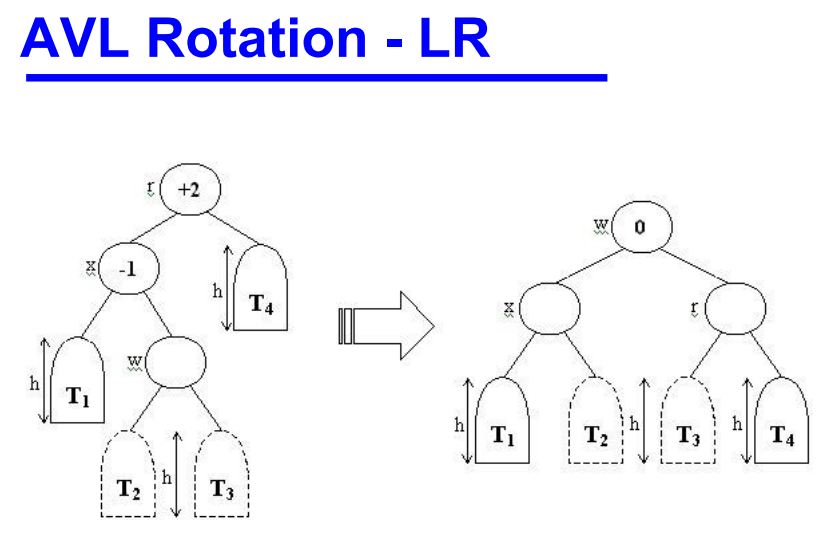
**Postorder：左右中**

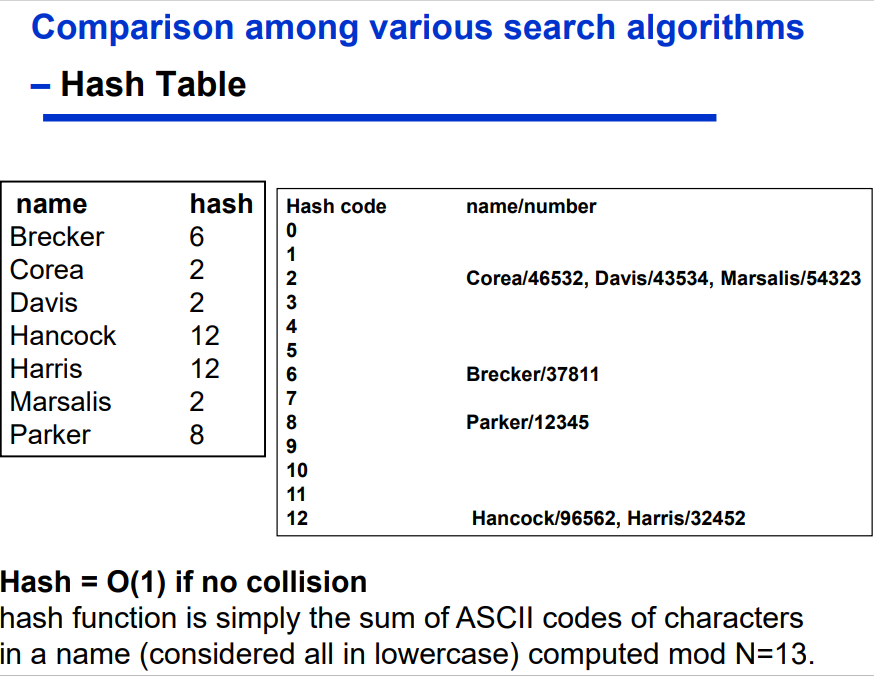
  










**Hash Tables**

• another kind of Table

• O(1) in average for insert, lookup, and remove

• use an array named T of capacity N

• define a hash function that returns an integer int H(string key)

• must return an integer between 0 and N-1

• store the key and info at T[H(key)]

• H() must always return the same integer for a given key

Linear Search = O(N)

Binary Search = O(log(N))

**A hash function is any well-defined procedure or mathematical function for turning data into an index into an array.**

• The values returned by a hash function are called hash values or simply hashes.

• A hash function H is a transformation that – takes a variable-size input k and – returns a fixed-size string (or int), which is called the hash value h (that is, h = H(k))

• In general, a hash function may map several different keys to the same hash value.

• inexpensive to compute - must be O(1)

**Hash Functions for Strings**

• must be careful to cover range from 0 through capacity-1

• some poor choices – summing all the ASCII codes – multiplying the ASCII codes

• important insight – letters and digits fall in range 0101 and 0172 octal – so all useful information is in lowest 6 bits

• hash(s) is O(1)

**Hash Functions for Integer Keys**

• Mid-square method

• Squaring the key value first, and then takes out the middle r bits of the result, giving a value in the range 0 to 2 r−1.

• This works well because most or all bits of the key value contribute to the result

**Hash Functions for String Keys**

• This function takes a string as input. It processes the string 4 bytes at a time, and interprets each of the four-byte chunks as a single long integer value.

• The integer values for the 4-byte chunks are added together. • The resulting sum is converted to the range 0 to M−1 using the modulus operator.

• There is nothing special about using 4 characters at a time. Other choices could be made.

**Dealing with Collisions**

• open addressing - collision resolution • key/value pairs are stored in array slots

• linear probing – hash(k, i) = (hash1(k) + i) mod N – increment hash value by a constant, 1, until free slot is found – simplest to implement – leads to primary clustering

• quadratic probing – hash(k, i) = (hash1(k) + c1 \*i + c2 \*i\*i) mod N – leads to secondary clustering

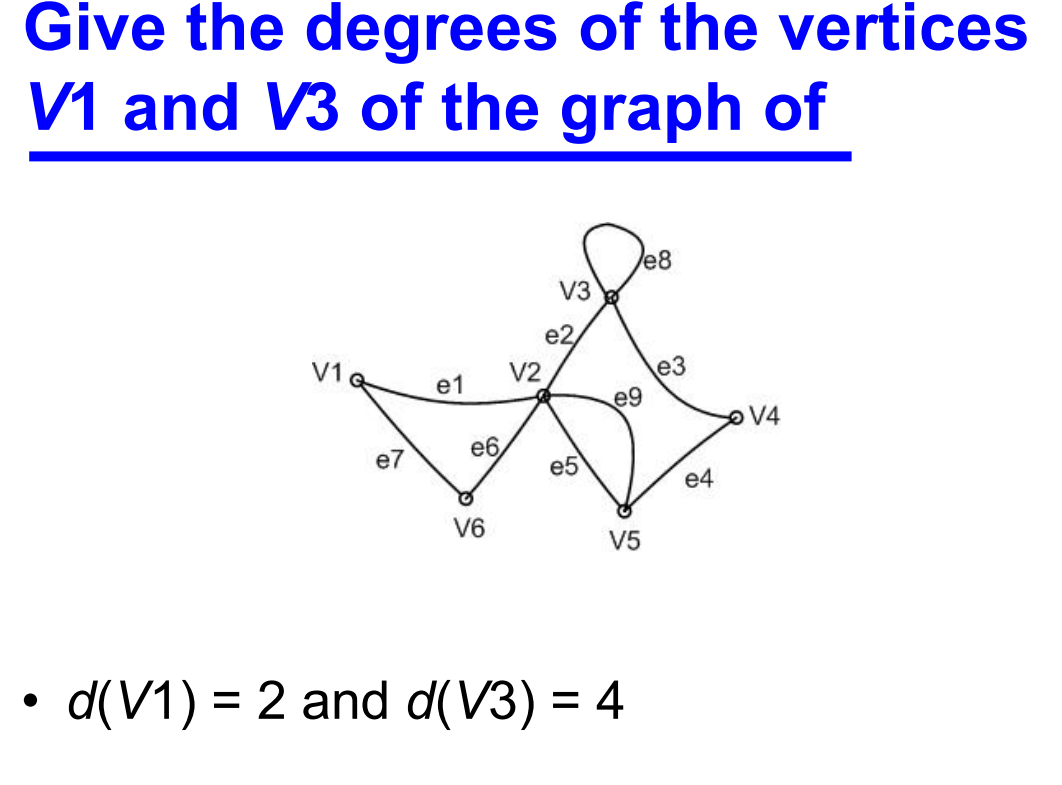
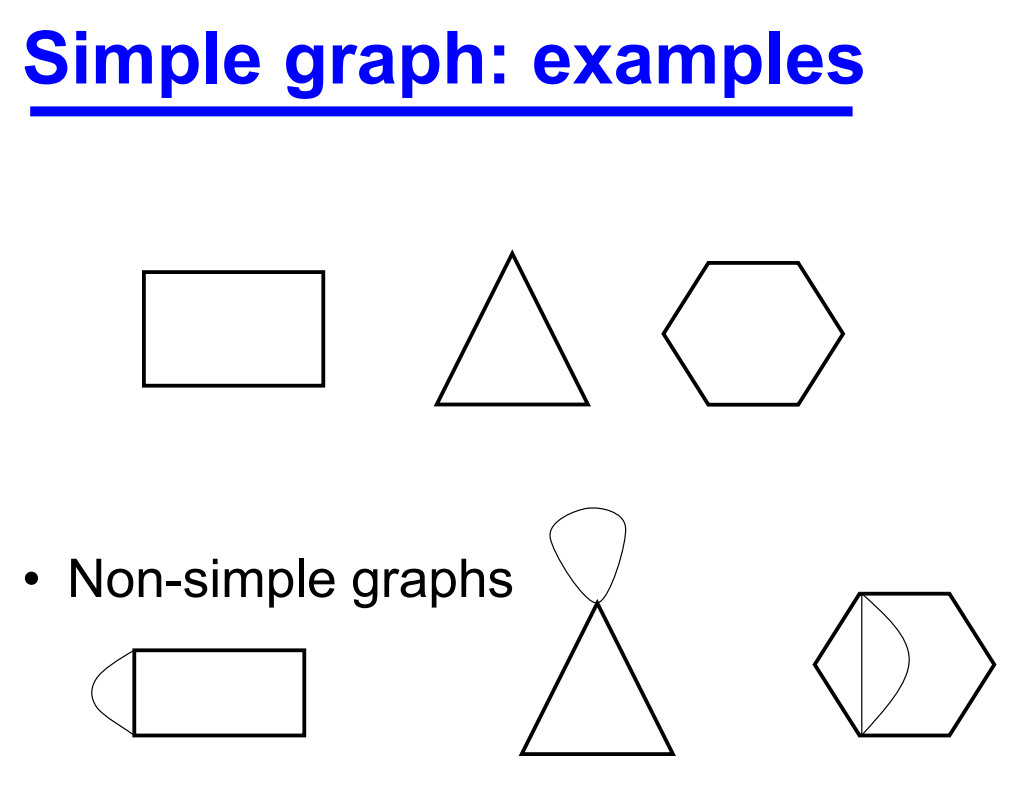
• double hashing – hash(k, i) = (hash1(k) + i\*hash2(k)) mod N • avoids clustering

**Dealing with Collisions**

• separate chaining

• each array slot is a SearchList

• never gets 'full' • deletions are not a problem



Degree • For a digraph we get • where |A| is the number of arcs.

**Subgraphs**

• A subgraph of G is a graph, H, whose vertex set is a subset of G’s vertex set, and whose edge set is a subset of the edge set of G.

• If a subgraph H of G spans all of the vertices of G, i.e. V(H) = V(G), then H is called a spanning subgraph of G.

**Walks, paths and circuits**

• A sequence of edges of the form VsVi , ViVj , VjVk , VkVl , VlVt is a walk from Vs to Vt .

• If these edges are distinct then the walk is called a trail, and

• if the vertices are also distinct then the walk is called a path.

• A walk or trail is closed if Vs = Vt .

• A closed walk in which all the vertices are distinct except Vs and Vt , is called a cycle or a circuit.

• The number of edges in a walk is called its length