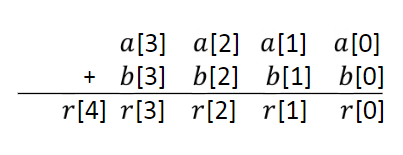
**COMP250 REVIEW**

**Lecture 1**

**Algorithm**: a sequence of instructions or operations for manipulating data to produce some result

Grade School Arithmetic:

1. Addition of 2 numbers:



- Use arrays to store each number

- For each column, add r[i] = a[i] + b[i] and carry value to next column

- The result array needs to have 1 more slot than max(a, b)

2. Multiplication of 2 numbers

- Slow: a\*b = add a times the number b

- Fast:

- Create 2D array and compute all rows in advance

- For each column, sum of the single digits in the row and add the carry

- Could instead use a running sum and add as we compute the rows

3. Division

- Slow: use num = q \* b + r. Subtract num by b, and add 1 to q, until r is smaller than b.

- Fast: long division

**Lecture 2**

Logarithms:

Modulo:

- Quotient remainder theorem: a = b\*q + r

- Remainder: a mod b = r

- Convention: find positive remainder

- However, in JAVA:

- In j=Java defined as:

- Addition:

- Multiplication:

Shift elements using mod:

- left shift n spots:

- right shift n spots:

**Lecture 3**

Base representation:

Base conversion: m, from decimal to base b

- divide m by b and prepend the remainder of division

- m = previous quotient

Binary Arithmetic:

- Carry when your sum is greater or equal to your base

- carry = # of times you can pull out base from sum

For all rows, and assign carry to r[n]

**Lecture 3**

Primitive type:

- predefined by the language

- named by a reserve keyword

In Java:

- Integer: byte, short, int, long

- Real num: float, double

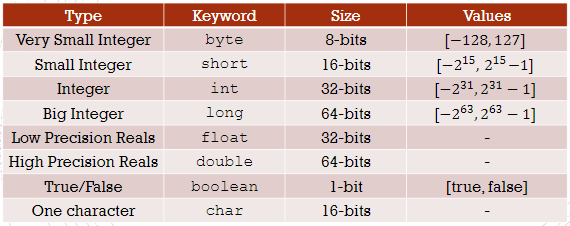
- Boolean

- Char

- One can represent values with bits

- One needs (round up) to represent different values

- One needs to represent a positive integer



Char

- single quotes

- single character

- A character set: an ordered list of character, **each character corresponds to a number** (i.e. Unicode)

Type Casting:

- Convert back and forth between variables of different types

- Explicit cast is not necessary when going from int -> double, but necessary in reverse!

- In general, widening (going to more bits) does not need explicit type conversion

- Char is more narrow than short, short can store negative values while char cannot, though they are of the same size

**Lecture 5**

**Packages**

- A group of classes: each class is a package member

- A class: a group of methods

- A method: an ordered group of commands

File and folder naming rules in Java:

1. Name of class must match name of file

2. Folder path must match exactly the package name. Each period in package name is a subfolder

Using a package member outside its package:

- write full path

- import the specific class of the package

- import entire package

Java automatically imports java.lang and current package

**Objects and Classes**

- class is a blueprint/template for a type of object

- object: an instance of some class

A class:

- Has attributes/fields

- Constructor (method to create an object)

- Other methods

If you write your own constructor, you **no longer have access** the default constructor!

Inheritance: Every class’ constructor implicitly calls the **super()** constructor. If the constructor is overwritten in the super class, the subclass must have a corresponding **super(with parameters)**

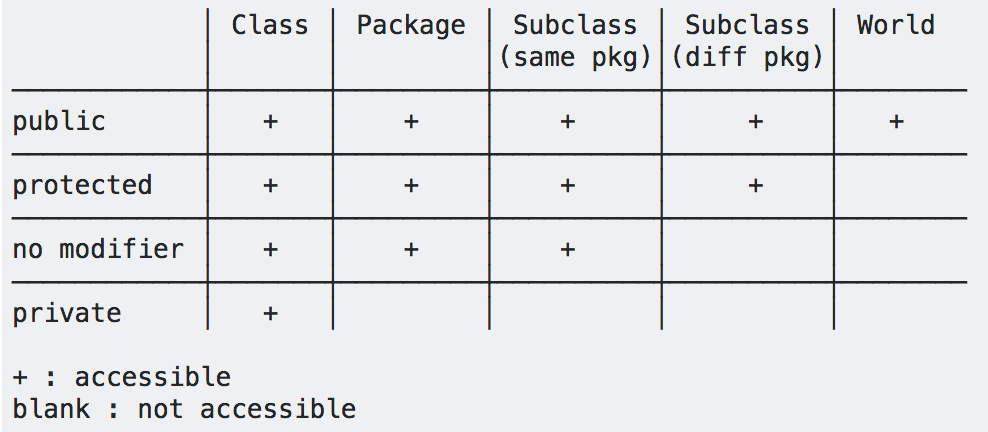
Nested Class

- Define class within another class (outer class)

- Benefits: organize, encapsulation (control over data), readable and maintainable code

**Modifiers**

- keywords that you add to class/method/variable’s definition to change their meaning



- Only when a member is visible, it can be inherited

- Outer classes can only be declared **public** or **package private**

Encapsulation:

- Wrapping data and code acting on that data in one unit. Better control data

- Make all fields private

- Provide getters and setters

Non-access modifiers:

- static

- final

- abstract

Static:

- fields, methods, and nested classes

- Associated with the entire class and not to a specific instance

- Independent from one specific instance of the class

**- Static fields: class variables**

- Non-static method belongs to an instance of the class

**- Non-static fields: instance variable**

Final:

- Variables, methods, and classes

- Value can never be changed after initial assignment

- Final fields must be initialized

Abstract:

- methods and classes

**UML Diagram**

3 sections:

1. Class name

2. Attributes/Fields

3. Methods

+ if attributes/fields or methods are public and – if they are private

Underline if method or variable is static

#: protected

Italics: abstract class

Local Variables and Fields

- local variables are declared inside a method or block

- fields (class and instance variables) are declared inside a class but outside a method

Differences:

- Scope: where they can be accessed

- local variables: only accessible within method or block of declaration

- class variables: accessible from any method or block within the class

- instance variables: requires existence of an object

Access:

- local variables: cannot have access modifiers. Cannot access local variables from other classes or methods

- Fields: access modifiers, can be accessed from methods within the class and from other classes if public

**Lecture 6**

Inheritance

- a subclass inherits all **public (or protected) fields and methods** from its superclass. Constructors are the only thing not inherited

Object Class

- the only class without a superclass, every class is implicitly a subclass of Object

In the subclass:  
- Declaring a same field: hiding the inherited field

- Writing a **non-static** method with same signature: **overriding**. The non-static method in the **subclass** is called.

- Writing a **static** method with the same signature: **hiding**. The static method in the **superclass** is called.

Overloading

- Two or more methods in the same class with same name but different parameters

Overriding

- Two (non-static) methods with same signature and return type

Constructor

- Default constructor

- Java automatically inserts a call to the no-argument constructor of superclass

Super

- Access members of superclass

- Access overridden methods in the superclass

Modifiers and Inheritance

- A final class cannot be extended!

- A final method cannot be overridden!

**Lecture 7**

toString()

- toString() for **Object** class returns a string consisting of name of class, @, and the hashcode (unsigned hexadecimal representation of hashcode)

- toString() in **String** class returns the object itself

Equals()

- In Object class, equals() is true if and only if they are the same object with the same address

- In String class, equals() is true if both String objects have the same sequence of characters

- In ArrayList class, equals() is true if both lists have same size and corresponding, identical pairs of elements

Clone()

- In ArrayList: clone() returns a shallow copy. Only 1 arraylist

**Type Conversion**

- Implicit upcasting is allowed

- Explicit downcasting: results in run-time error if type casted is of the wrong type

- **Casting does NOT change the object itself, it just labels it differently**

Instanceof

- test whether an object is an instance of the specified type

- returns: true or false

- if the class is a subclass of the specified class: returns true

- a method to prevent error when downcasting

**Lecture 8**

Polymorphism

- JVM calls the **appropriate** method for object that is referred to in each variable. It does not call the method that is defined by the variable’s type

Dog snoopy = new Beagle();

- At **compile-time**: the compiler uses bark() in Dog class to validate statement

- During **run-time**: JVM invokes bark() from Beagle class since snoopy is referring to a Beagle object

Abstract

- If you want a class to contain a particular method, but would like the implementation to be specified by the subclass

- **declared without implementation**

Abstract Methods

- the class containing it must be abstract

- every subclass of the current class must either override (implement) the abstract method or declare

Itself as abstract

Abstract Classes

- declared with abstract keyword

- can have abstract and non-abstract methods

- cannot be instantiated

- can have constructors (called when instance of subclasses are created) and static methods

- can have final methods, will force subclass not to change body of method

**Lecture 9**

Arrays

- O(1)

List (array list, linked list, etc.)

Array List:

1. get(i):

- check i range

- return a[i]

2. set(i,e):

- check i range

- a[i] = e

3. add(i,e):  
 - check if array is full, if full, create bigger array and copy all elements forward

- shift all elements after i one spot down the array

- insert element at i and increase size by 1

4. remove(i)

- shift all elements after i one spot up

- decrease size by 1

Note: add(i,e) allows adding to i == size, but set(i,e) does not allow this!

ArrayList class

- Grows the size of array by 50% when it is full and a new element is added

ArrayList object

- a private field for size of arraylist

- a private field that references an array object

**Lecture 10**

ArrayList: slots are in consecutive locations in memory, but objects can be anywhere

LinkedList: nodes and objects can be anywhere in memory

Singly Linked List:

- consists of a sequence of nodes, reference to the first (head) and last (tail) node

- each node object points to the next node, and points to the object

1. addFirst(e):

- create new node

- nextnode points to current head

- head points to the new node

- increase size by 1

2. removeFirst():

- head points to the next

- decrease size by 1

3. addLast(e):

- create new node

- tail.next points to the new node

- tail points to the new node

- increase size by 1

4. removeLast():

- create a tmp to point to head

- iterate tmp until it tmp.next is tail

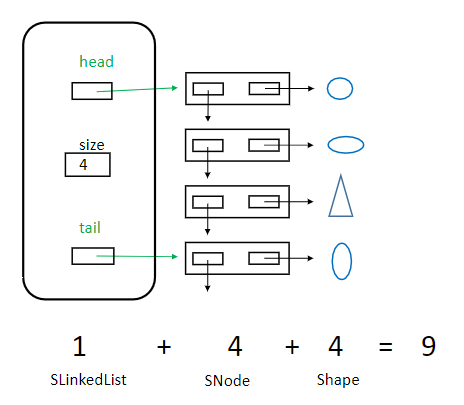
- tail point to what tmp is pointing

- decrease size by 1

ArrayList and LinkedList

- In linked list, addFirst(e) and removeFirst() does not depend on the number of elements

- In arraylist they do, because of shifting



**Lecture 11**

Doubly Linked List

- Each node has a reference to the next node and previous node

- Motivation: able to access both previous and next nodes

1. removeLast():

- tail points to previous

- size decreases by 1

2. remove(i):

- get the node by traversing either from bottom or top

- node.next.prev points to node.prev

- node.prev.next points to node.next

- edge case: null next field and null prev fields in singly and doubly linked lists

- Dummy Nodes: avoid edge cases by adding two nodes (dummyHead and dummyTail) and the beginning and end of DLL

Time Complexity

|  |  |  |  |
| --- | --- | --- | --- |
|  | ArrayList | LinkedList | DLL |
| addFirst | O(N) | O(1) | O(1) |
| removeFirst | O(N) | O(1) | O(1) |
| addLast | O(1)\* | O(1) | O(1) |
| removeLast | O(1) | O(N) | O(1) |
| remove(i) | O(N) | O(N) | O(N) |

\*if array is full

Java Enhanced for loop

- iterates through the list

**Lecture 12**

Sorting

- arranging items in a ordered list

- O(N^2): selection, bubble, insertion

- O(N\*logN): heap, merge, quick

Bubble Sort

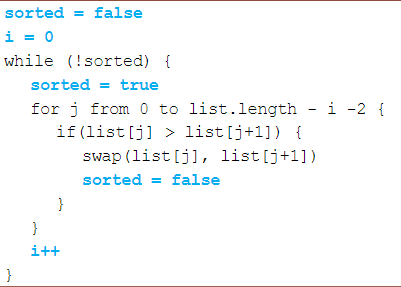
- simplest sorting algorithm

- Idea: repeatedly iterate through the list and swap adjacent elements if they are in the wrong order

- finishes when no sorting takes place in an iteration

- worst case: O(N^2)

- best case: O(N)



Selection Sort

- Idea: consider the list as if it was divided into two parts, one sorted and the other one unsorted

- Procedure:

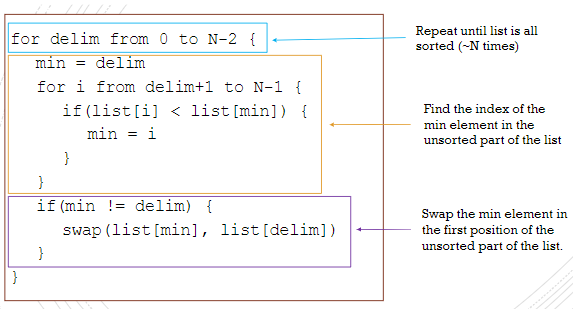
- select smallest element in unsorted part

- swap element to the beginning of list

- change where the sorted and unsorted parts are divided and repeat process

- worst case: O(N^2)

- best case: O(N^2)



Insertion Sort

- Idea: consider the list as if it was divided into two parts, one sorted and the other unsorted

- Procedure:

- select first element of unsorted part

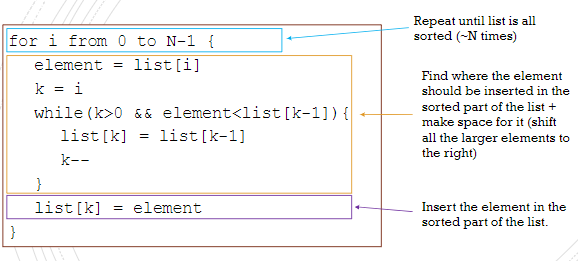
- insert the element at correct position

- change where the sorted and unsorted parts are divided

- Inserting: similar to adding element to array, shift elements ahead to make hole, and fill hole

- best case: O(N) list already ordered and no shifting is needed

- worst case: O(N^2) slowest when sorted in reverse order



**Lecture 13**

Abstract Data Type (ADT)

- defines a data type by the values of data and operation on the data

- ignores details of implementation

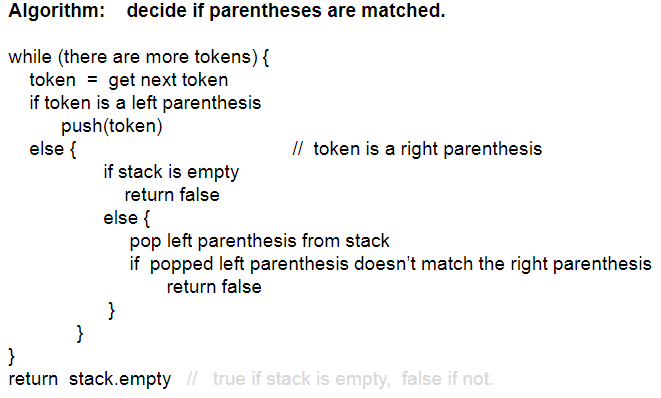
Stack ADT

- push and pop

- Last in first out (LIFO)

|  |  |  |
| --- | --- | --- |
|  | Push(e) | Pop() |
| Array list\* | addLast(e) | removeLast() |
| SLL | addFirst(e) | removeFirst() |
| DLL | Either row above | |

\*Java arraylist does not have addfirst or removelast methods



**Lecture 14**

Queue ADT

- enqueue(e), dequeuer

- first in first out (FIFO)

|  |  |  |
| --- | --- | --- |
|  | Enqueue(e) | dequeue() |
| ArrayList | addLast(e) | removeFirst() |
| SLL | addLast(e) | removeFirst() |
| DLL | Either row above | |

- with ArrayList it is slow due to shifting

- even with expanding array it is still bad

Circular Array

- Increasing array length:

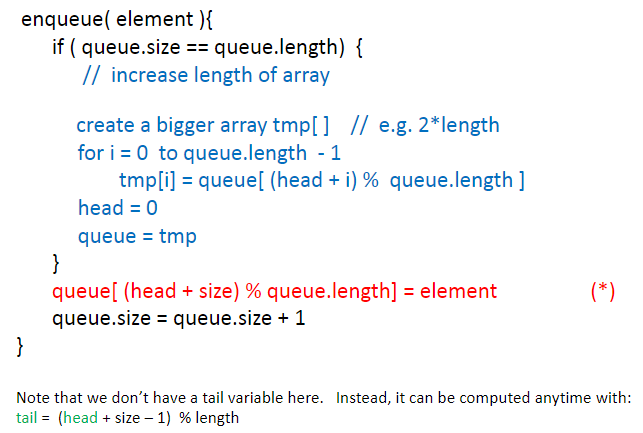
- Head stays at same index

- Head moves to slot 0

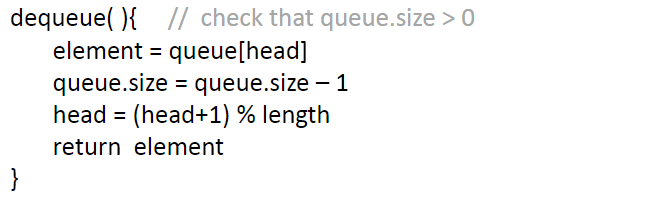
**Note:**

Since Java might return a negative number for mod operations, we could do:

To ensure tail is valid



- in the new array, head is slot 0 of new array



- Using two stacks, we can create a queue

Java API

- API = application program interface

Java interface

- reserved word

- like a class, but only method signatures are defined

**Lecture 15**

Interfaces

- public or package-private (default)

- all **methods** are by default public and abstract

- all **fields** are by default public, static, and final

- interfaces cannot be instantiated

- An interface extends another interface and cannot extend another class

Syntax of interface

- interface instead of class

- implicitly abstract, does not need abstract keyword

- all methods are implicitly abstract

**Inheritance**

- implements instead of extends

- specifies what a class must do, not how

- a class can **implement one or more interfaces**, to achieve multiple inheritance

- if a class implements the interface, but does not implement all methods specified in the interface, then the class must be abstract class

**Interface VS Abstract**

- Abstract:

- not all methods have to be abstract

- abstract keyword must be added to class declaration

- can contain implemented methods and instance variables

- useful when some general methods should be implemented and specified by subclasses

- Interface:

- all methods are abstract by default (no keyword)

- interfaces are implicitly abstract

- no methods can be implemented and only constants (final static fields) can be declared

- interfaces are useful in situation where all properties should be implemented

Generics in Java

- A generic type is a class or interface that is parametrized over types

**Usage of Interfaces:**

- Interfaces define new data types

- We can create variables of these data types and assign the variables any instance created from the classes that implemented the interface.

- In the case of a method parameter, whenever an object of the interface type is required, any instance of any of the classes that implemented the interface can be used

**Comparable**

- comparable defines a natural ordering

- used to define an ordering on objects of user-defined class

- contains only one method: compareTo(object)

**Lecture 16**

Iterable and Iterator

- Objects of type Iterable are representations of a series of elements that can be iterated over (e.g. a specific ArrayList)

- Objects of type Iterator allows you to iterate through objects that represent a collection (a series of elements)

Java Iterable Interface

- A class that implements Iterable needs to implement the iterator() method. The iterator() method returns: an object of type Iterator that can be used to iterate through elements of that class

- A class that implements the Iterator needs to implement the methods hasNext() and next()

- The iterator() method returns an iterator to the start of the collection. You can traverse the collection using hasNext() and next().

How to Implement Iterable Interface

- Generally, when we write a class that implements the interface Iterable, we also write a class implementing the interface Iterator. Often, such class is defined as an inner class.

- The reason is: to implement Iterable, we need to implement iterator(). Since iterator() needs to return a Iterator type object, we need a class to create such an object.

- However, iterators cannot reset and start over again. The only way to restart iteration is to call iterator() method to obtain a new iterator that points to the head of the provided list.

**Lecture 17**

**The Class class**

- the compiler translates the .java file to a .class file

- a “class descriptor”, created during runtime by JVM, is an instance of the class Class.

- instances represent classes and interfaces in a running Java application

getClass():

- returns the **run-time** class of the calling object

getSuperclass():

- method from class Class

- returns class representing the superclass of the calling class

**Memory Allocation**

- Heap: used by java runtime to allocate memory to Object and JRE classes. Objects are stored in Heap.

- Stack: used for execution of a thread. Threads contain method specific values and references to other objects in the heap that are getting referred from the method.

**Stack** (LIFO data structure)

- stores methods

- each method block has all the local values, as well as **references** to other objects that are being used by the method

- after a method terminates, its block will be erased

- the values stored in each block are accessible only from that particular method

- local variables and method parameters

**Heap**

- stores objects

- no specific order in reserving blocks

- objects created in heap space has global access and can be referenced from anywhere of the application

- Garbage Collection runs on heap memory to free memory used by objects that doesn’t have any reference

- object instances and fields

**Permanent Generation:**

- contains all data required by JVM to describe the classes and methods used at runtime

- methods and static fields

**Lecture 18**

Recursive Definition

- Base clause: basic element of the set

- Inductive clause: how to generate new elements of the set from old ones

- Final clause: states that no other element is part of set

(Weak) Mathematical Induction

- Base case: show all properties hold for initial elements of set

- Inductive: assume property holds for some element n, and show the property holds for any element generated from n

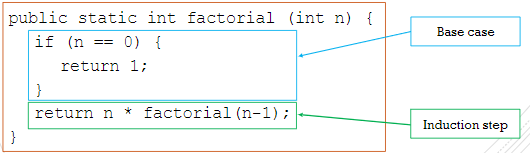
- Conclusion: property holds for all elements

(Strong) Mathematical Induction

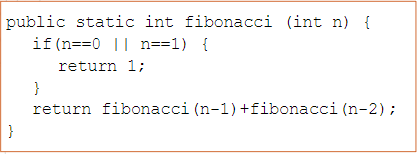
- Prove property holds for all n

**Lecture 19**

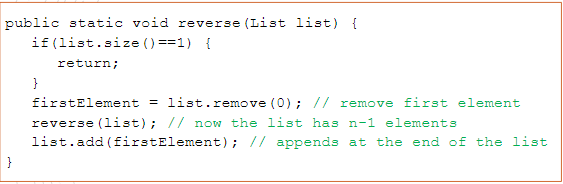
Factorial



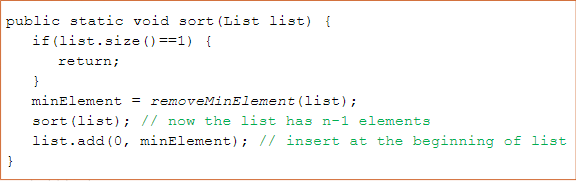
Fibonacci



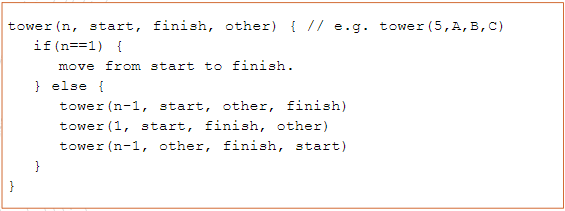
Reverse list



Sorting a list



Tower of Hanoi



- Anything recursion can do, iteration can do

- Anything iteration can do, recursion can do

**Lecture 20**

DEC to BIN

- Euclid’s algorithm

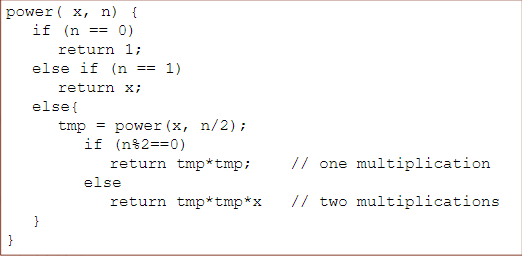
- recursively call method on

Power

- recursively call method on

- Complexity:

- Better implementation:



Binary Search

- Idea:

- first compare key with middle element

- if key is greater, search second half and discard first half

- if key is smaller, search first half and discard second half

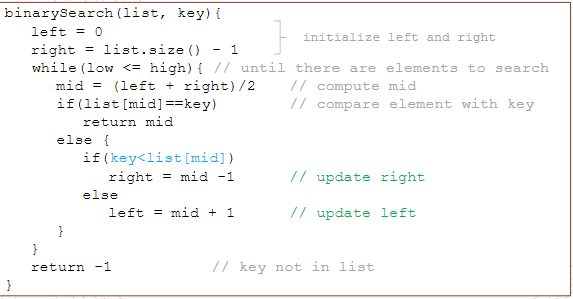
- if key equals middle element, return index

- Complexity:

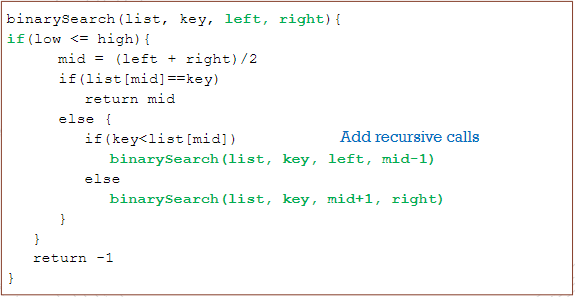
Implementation

- keep track of left and right indices denoting section of list needs to be searched

Iterative implementation:



Recursive implementation:



**Lecture 21**

**Merge Sort**

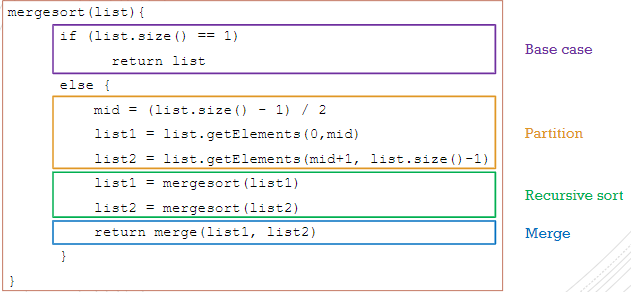
- divide and conquer

- Idea:

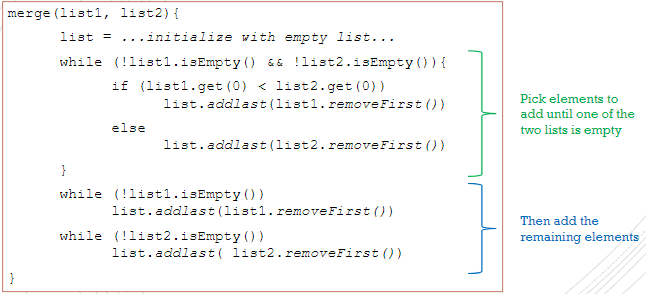
- partition list into two halves

- sort each half recursively

- merge the sorted half **maintaining order**



Where merge(list1, list2) is implemented as:



- Complexity:

**Quick Sort**

- divide and conquer

- Idea:

- pick an element as the pivot

- partition list by placing all elements smaller than the pivot to its left, and all elements larger than the pivot to the right

- sort left and right parts recursively

- repeat until there is nothing left to sort

Picking the pivot:

- always pick first element

- always pick last element

- pick random element

- pick median as pivot

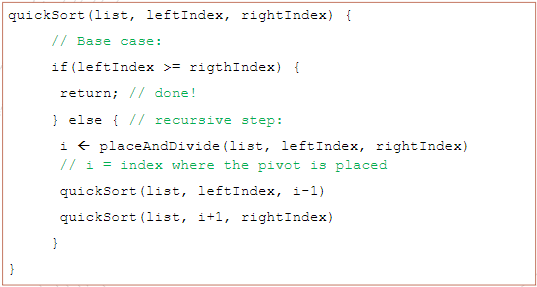
Implementation

- compare elements with pivot

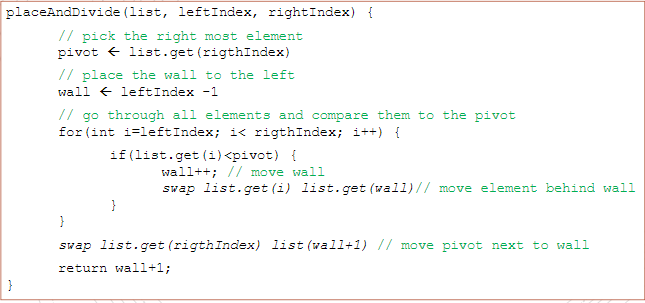
- if element is smaller then move wall and swap place with element left of wall

- stop when the right index is reached

- call quicksort on left and right parts until base cases are reached



- Implementation of placeAndDivide():



Merge Sort VS Quick Sort

- Merge sort typically uses extra lists, hurts performance for big lists

**Lecture 22**

Linear Data Structures:

- array

- linked list

Non-Linear Data Structures

- tree

- graph

Tree Terminology:

- root: highest node

- directed edge: ordered pair of nodes (from, to)

- sibling: has same parent

- internal nodes: non-empty file directories, points to some other node

- external nodes, leaves: files or empty directories

- path: a sequence of nodes

- length of a path: number of edges in the path (number of nodes – 1)

- ancestor: higher node in a path

- descendent: lower node in a path

- depth or level: length of the path from the root to the node

- height: the **maximum** length of a path from the node to a leaf

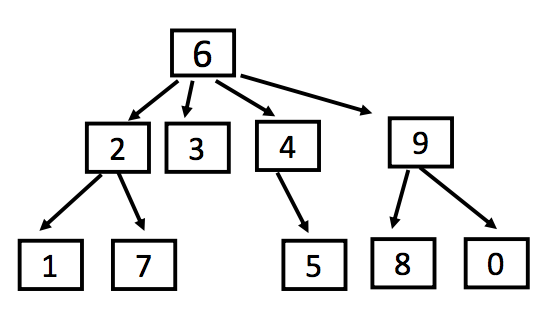
- Every node except root is a child, and has exactly 1 parent

- A tree with nodes has nodes

Recursive definition of a rooted tree:

- consists of subtrees

Represented using lists:



**Lecture 23**

Tree Traversal

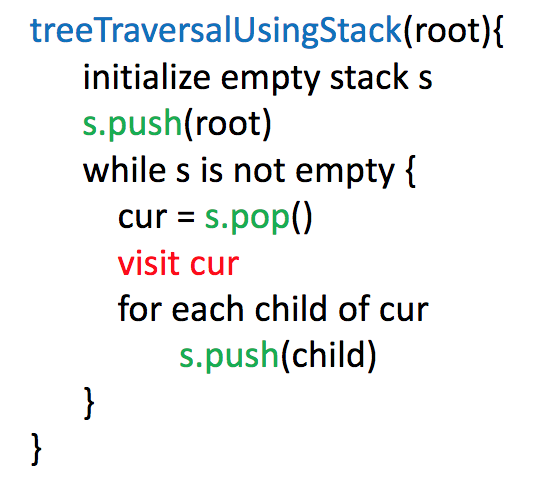
- Recursive: (Depth first)

- Preorder traversal: root, left, right

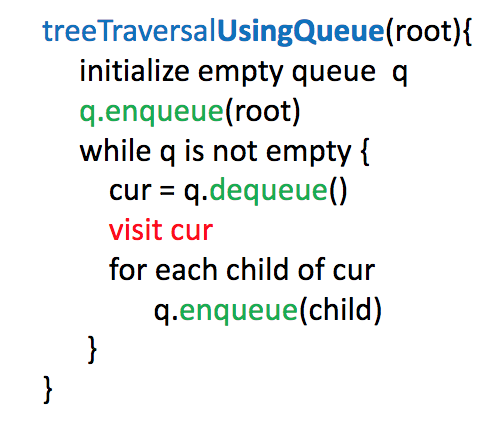
- Postorder traversal: left, right, root

- Non-Recursive:

- Stack: depth first with different order: root, right, left



- Queue: breadth first, root, left to right



**Lecture 24**

Binary Tree

- each node has at most two children

- Max number of nodes and height:

- Min number of nodes and height:

- Traversal:

- preorder: root, left, right

- postorder: right, left, root

- inorder: left, root, right

Prefix, infix, postfix expressions:

- prefix: root, left, right

- infix: left, root, right

- postfix: left, right, root

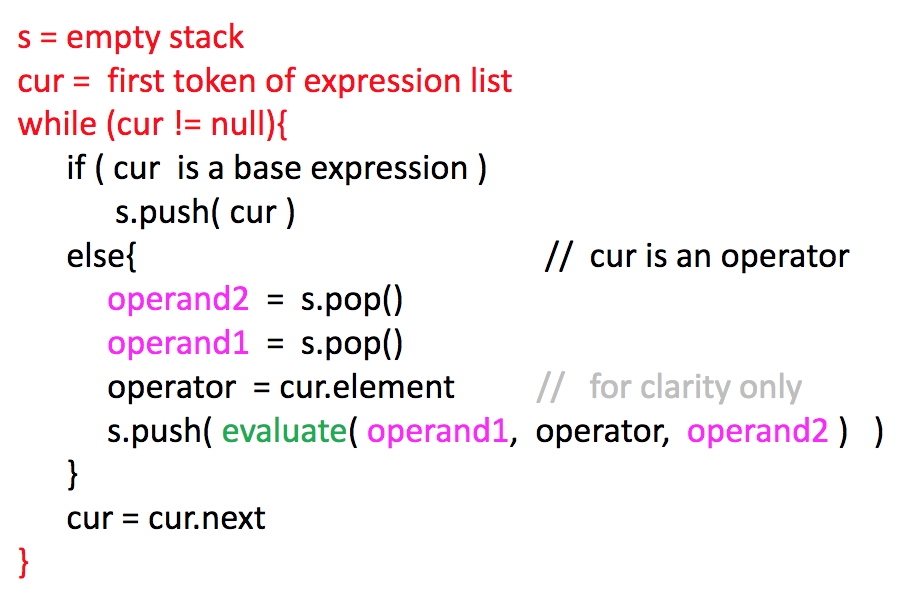
Prefix: polish notation

Postfix: reverse polish notation (RPN)

Algorithm: Use stack to evaluate postfix expression

- push when token is a base expression

- pop two elements and push in the evaluated integral element when token is an operator



**Lecture 25**

Binary Search Tree (BST)

- keys are comparable, unique (no duplicates)

- for each node, all descendents in left subtree are less than the node, and all descendents in the right subtree are more than the node

- an inorder (left, root, right) traversal on a BST visits the nodes in the natural order defined by the key

- a new key is always a leaf

1. find(root, key):

- recursive

- if key < root.key, call find(root.left, key)

- else, call find(root.right, key)

2. findMin(root):

- keep traversing left until leftmost node is reached

3. findMax(root):

- keep traversing right until a rightmost node is reached

4. add(root, key):

- compare at each node

- if smaller, call add(root.left, key)

- if larger, call add(root.right, key)

- when null reached, add

- does nothing when the same key is already present

5. remove(key):

- searches for the key by calling remove recursively until a match

- if left or right of the current node is null, set the left or right child to be the new root, return the root of this subtree

- if neither is not null, then the value of the root is the value of the **smallest element in the right subtree**

- the right subtree is then the subtree with the minimum element removed

|  |  |  |
| --- | --- | --- |
|  | Best case | Worst case |
| findMin() | O(1) | O(n) |
| findMax() | O(1) | O(n) |
| find(key) | O(1) | O(n) |
| add(key) | O(1) | O(n) |
| remove(key) | O(1) | O(n) |

**Lecture 26**

Priority Queue (ADT)

- assume a set of comparable elements

- heap is a good implementation

Complete Binary Tree

- Binary tree of height h such that every level less than h is full, and all nodes at level h are as fa to the left as possible

Min Heap

- Complete binary tree with unique comparable elements, such that **each node’s element is less than its children’s elements**

1. add(element): (“upheap”)

- create new node at next available leaf position

- while current element is smaller than direct parent, swap places

2. removeMin(): (“downheap”)

- replace root with last element

- compare with the **smaller** of two childs and swap if necessary

3. remove(element):

- remove element and replace slot with last element

- compare with parent:

- if smaller than parent: call upheap()

- if larger than parent: call downheap()

Parent child relations in **array implemented** heap structure:

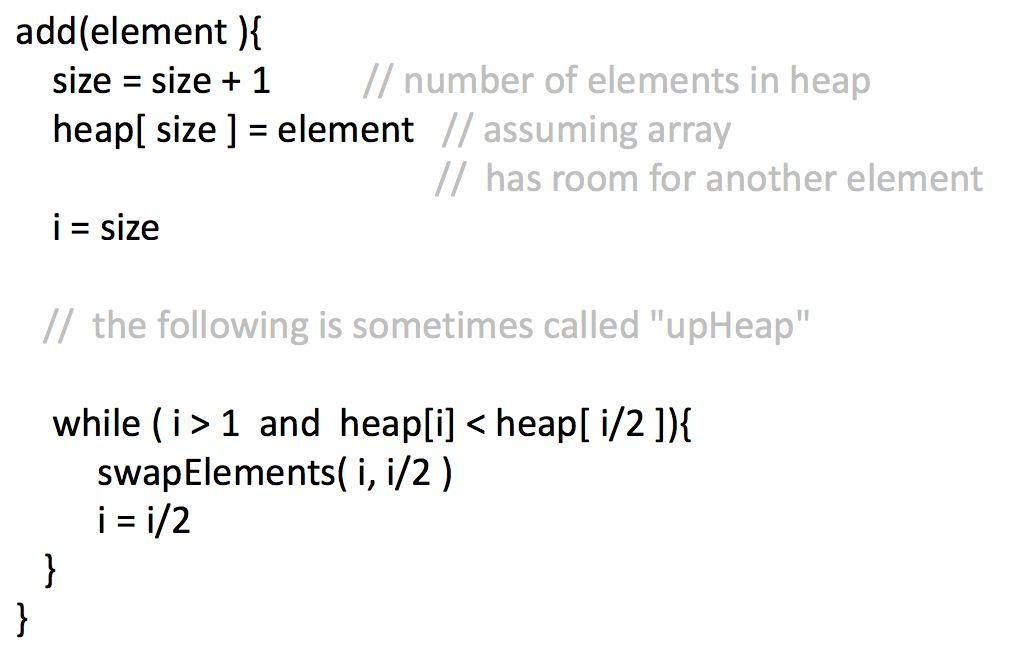
- 0th index is not used

-

-

-

**- Implementation of add(element):**



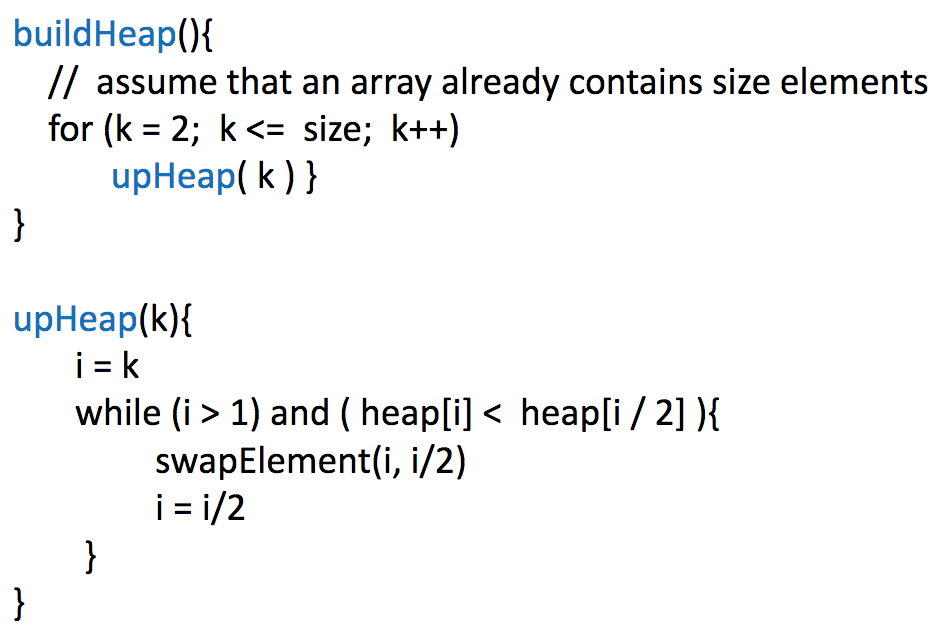
**Lecture 27**

**Building a Heap**

- upheap() elements in array, add element to last and check with parent to see if child is smaller than parent, if so, swap.

- Repeat until i = 1 (reached root)

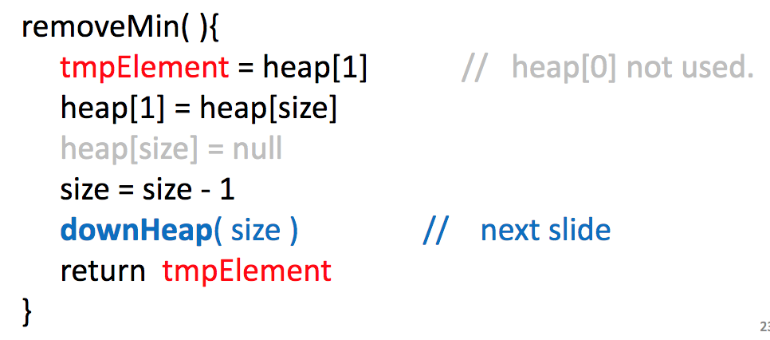
**- implementation of upheap():**



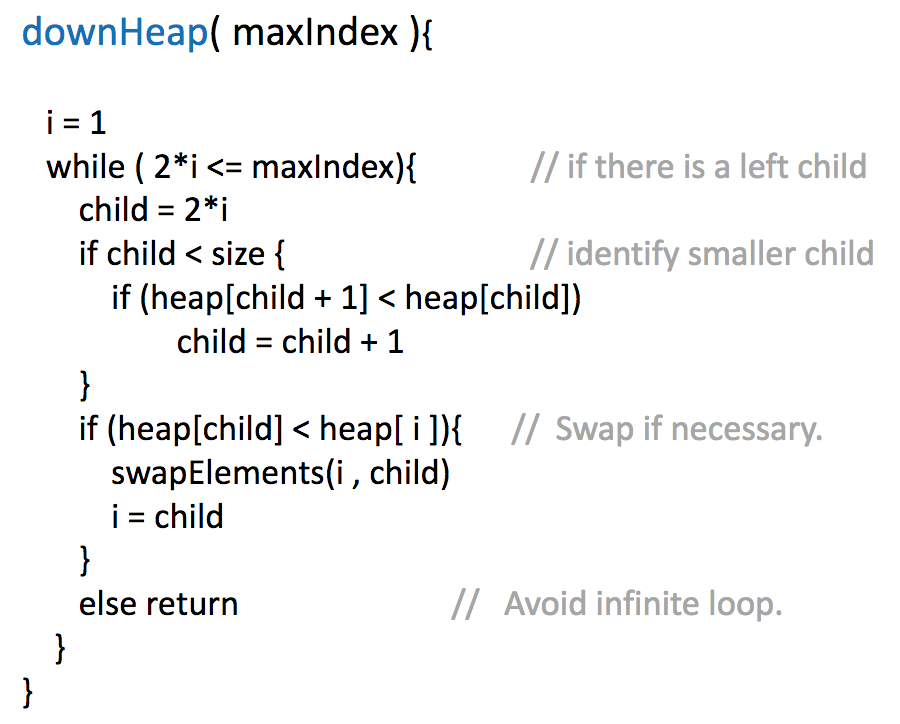
- Best Case: . Elements already satisfy the heap parent-child ordering constraint, no swaps

- Worst Case:. Original list is from large to small.

**- Implementation of removeMin():**



**- Implementation of downHeap:**

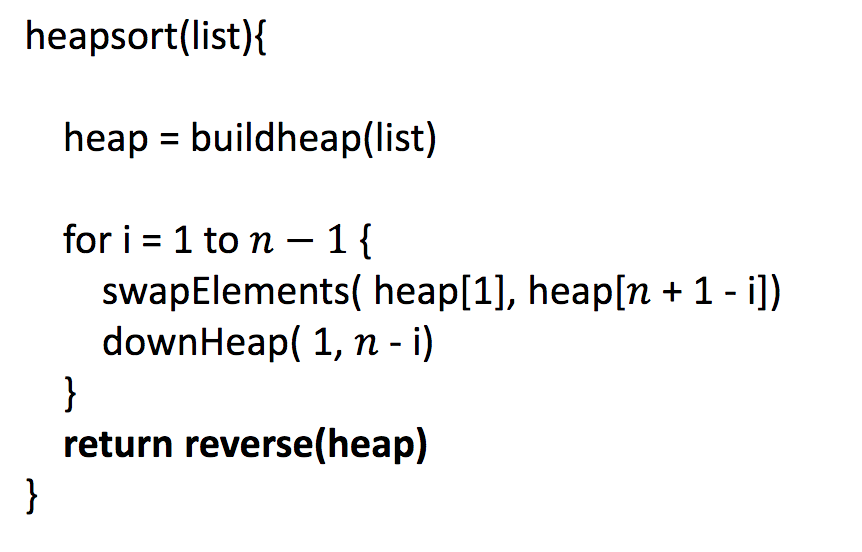


**Heapsort**

- build heap with n elements, then call down heap times

- each time swapping the first and last elements, then calling downheap considering a smaller heap

- finally reverse the heap array to obtain a sorted array



- we could build a “max heap” and remove the maximum element each time to avoid **reverse** at the end

- worst case:

**Lecture 28**

Map

- set of pairs {x, f(x)}

Map (ADT)

- set of (key, value) pairs

- for each key, there is at most one value

- each (key, value) pair is called an entry

1. put(key, value):

- map key and place value

- If map previously contained a mapping for the key, old value is replaced by the specified value, and the previous value is returned. Otherwise, return null.

2. get(key)

- returns the value corresponding to the mapping of the key. Returns null if no entry is found.

3. remove(key)

- removes the entry for the key, if it is present, and returns the corresponding value. Returns nul if the map contained no mapping for the key.

Data Structure for Maps:

1. ArrayLists, SLL, or DLL

2. Comparable keys:

ArrayList:

- get(key) :

- put(key, value):

- remove(key):

BST: (depends on tree, assume balanced here)

- put(key, value):

- get(key):

- remove(key):

minHeap:

- put(key, value):

- get(key):

- remove(key):

3. Keys are unique positive integers in small range:

Array:

- array of type V (value): O(1) access

4. General Case:

- keys might not be comparable

- define a mapping of keys to large range positive integers (i.e. hash code)

**Java Object.hashcode()**

- object’s (base) address in JVM memory (unique)

- default definition:

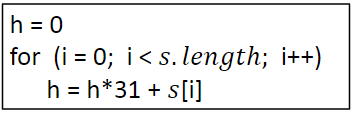
Is equivalent to:

**Java String.hashcode()**

- for each string, defines an integer

Where and

- Java uses **Horner’s Rule** for efficient polynomial evaluation



**Lecture 29**

**Hashing**

- Keys are hashed into hashcodes using hashcode()

- hashcodes are compressed into hashvalues using %

Compression:

- where is the length of the array

Hash Function: hashcode and compression

Collision:

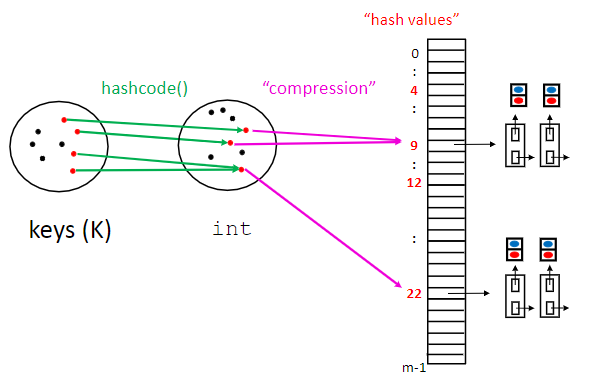
- when 2 or more **hashcodes** map to the same **hash value**

- when 2 or more **Keys K** map to the same **hashcode**, and mapping to the same **hash value**

Solution:

- Hash Table (or Hash Map): each array slot holds a SLL of entries

- each array slot + linked list is called a bucket, there are m buckets



Load Factor of Hash Table

- typically load factor below 1

- In Java HashTable and HashMap classes, default maximum load factor is 0.75

Good Hash: no collisions

Bad Hash: many collisions

Performance of Hash Maps

- If load factor is less than 1 and if hash function is good

1. put(key, value): in practice

2. get(key): in practice

3. remove(key): in practice

4. contains(value): need hash table traversal

5. getKeys(): need hash table traversal

6. getValues(): need hash table traversal

**Java HashMap <K,V> class**

- specify initial number of buckets and load factor in constructor

- Hash Function: Use key’s hashcode(), take absolute value, and compress it by taking mod of the number of buckets

**Java HashSet <E> class**

- similar to HashMap, but there are no values. Use it to store a set of objects of some type

- add(e), contains(e), remove(e), etc.

**Lecture 30**

**Graphs**

Terminology

- Directed graph: a set of vertices, and a set of **ordered** pairs of these vertices called edges

- Undirected graph: a set of vertices and a set of **unordered** pairs called edges

- In degree: # of incoming edges to the vertex

- Out degree: # of outgoing edges from the vertex  
- Path: a sequence of edges such that end vertex of one edge is the start vertex of the next edge. **No vertex may be repeated except first and last!**

- Cycle: a path such that the last vertex is the same as the first vertex

- Direced Acyclic Graph: directed graph with no cycles

**Graph ADT**

Implementation

- graphs are a generalization of trees, but does not have a root vertex

- outgoing edges: children of a vertex in a tree

- incoming edges: parent (s)

1. Adjacency List (for edges)

- shows the outgoing edges for the vertices

Implementation of a Graph class in Java:

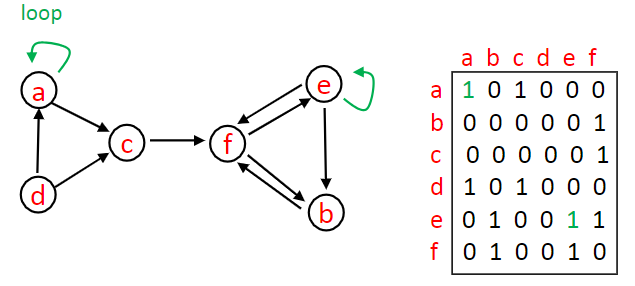
- **Vertex** class: adjacency list (of outgoing edges) using ArrayList and the element itself

- **Edge** class: Vertex the edge is pointing to and the weight, a double, of the edge

- **HashMap** object containing all Vertices as values with keys (could be a String, etc.)

2. Adjacency Matrix

- a square matrix



**Adjacency List VS Adjacency Matrix**

1. Sparse graph: 10000 vertices and 20000 edges. Save space

- Use adjacency list: matrix has slots, while list uses objects

2. Dense graph: 10000 vertices and 20000000 edges. Save space

- Use adjacency matrix: matrix has but list has objects

3. Check adjacency as quickly as possible.

- Use adjacency matrix. access

4. Insert new vertex

- Use adjacency list. For matrix, you need to add new row and new row in array, and copy everything over

**Lecture 31**

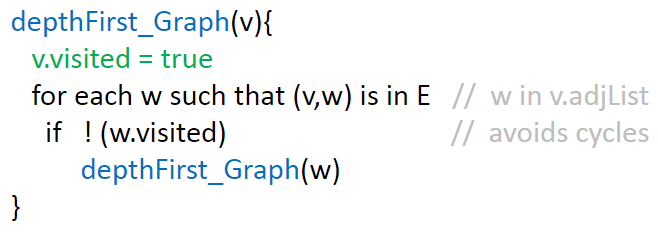
**Graph Traversal (Recursive)**

- need to specify starting vertex

- visits all nodes that are “reachable” by a path from starting vertex

**Depth-First:**

- order of nodes visited depends on order of nodes in the adjacency list



**Breadth-First:** (Queue)

- implement using a queue

- for each child vertex, if not visited, visit and add to queue

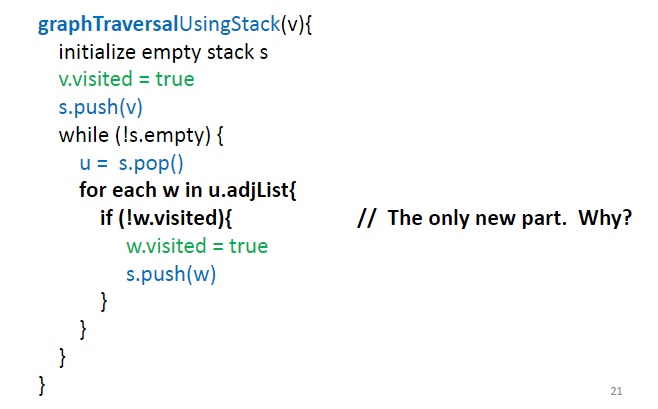
- for the first node, add to queue and set state to visited

**Graph Traversal (Non-Recursive)**

- Use stack or queue

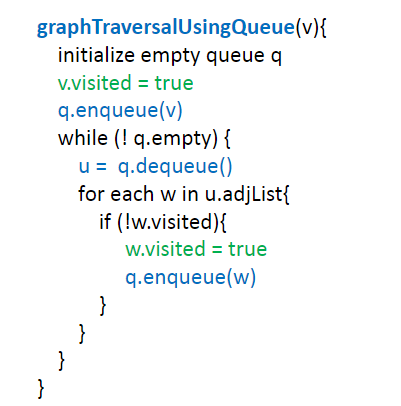
**Depth-First:** (Stack)

- Last pushed in vertex is popped and visited



**Breadth-First:** (Queue)

- Vertices are visited in enqueuer order



**Lecture 32**

**Garbage Collection**

- Object is not referenced = garbage

- Live object: referenced either from call stack variable or from an instance variable in a live object

- JVM maintains a linked list of all objects, stores Object.hashCode() of each object

**Garbage Collection: Mark and Sweep**

1. build a graph and identify live objects

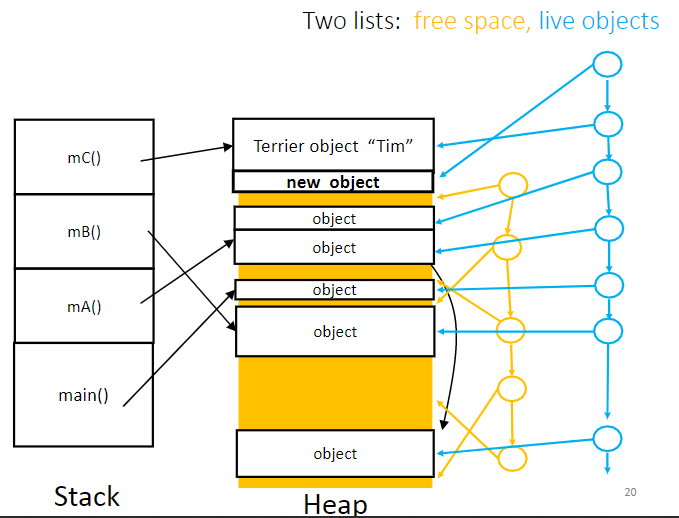
- vertices are reference variables in call stack and the objects in heap

- edges are references (arrows)

- for each reference variable on call stack, traverse graph and mark each object reached “visited”

2. remove garbage

- for objects not visited, JVM has another list (free space list) to keep track, and marks these objects as available space



- new objects can be added, where there is a big enough gap in free space

- garbage collection is needed again when there is no gap big enough for the new object. Objects can also be moved around if heap space is too fragmented

- program temporarily stops to do garbage collection, not good for real time applications

**Graph Traversal Applications**

- the set of web pages on the world wide web define a graph. Webpage are vertices and hyperlinks are edges.

- Google traverses this graph by following the links and retrieves as many web pages as it can find

- Google then builds a graph data structure:

- vertices are web pages

- edges are hyperlinks within the web pages

- keys for the vertex map are the URL’s

**Google PageRank: Importance of V**

- which set of pages link to v, and how important are these pages

- how many other pages does each w point to

**Naïve PageRank:**

- Rank of v is determined by summing up all the in/out degree ratios of all the w pointing to v.

- Since importance is relative, initialize R(w) and calculate R(v), then set the new R(v) as the initial rank. R(v) is a vector containing ranks of all web pages

**Google Search Engine:**

- web crawler downloads all reachable web pages

- build/updates the graph

- compute page rank for each web page

- build a map: maps keywords (keys) to URLs of web pages containing keyword (values)

**User:**

- enter keywords

**Google Search Engine:**

- for each key, get value

- for multiple keys, compute intersection of lists of web pages

- output resulting list, in order of PageRank

**Lecture 33**

**Recurrence Relation**

- a recurrence relation is a sequence of numbers where the n-th term depends on previous terms

- : time to execute a recursive algorithm as a function of the input size

**Reverse a list:**

- is the base case

**Tower of Hanoi**

-

**Binary search**

- assume

Geometric Series Sums:

**Lecture 34**

**Merge Sort**

- cn from copying array

-

- Total calls to mergesort is

- this is also the case for adding n elements to an empty arraylist and resizing times

- also the case for putting n elements into empty hash table and rehasing times

**Quick Sort**

- best case: same as merge sort

- cn from partition based on pivot

- best case when two sublists have same size

- worst case: one sublist has only 1 element

Reduce chance of unbalanced partition for quicksort:

- Median of three: take median of the first, last, and median of the list and set as pivot

**Lecture 35**

**Formal Definition of Limit:**

- A sequence has a limit if, for any , there exists an such that for any ,

**Bounded Definition:**

- Let and be two functions, where . Then is asymptotically bounded above by if there exists such that, for all ,

**Definition of Big O:** (upper bound)

- Let and be two functions, where . is if there exist two positive constants and such that, for all ,

**Lecture 36**

**Definition of Big Omega:** (asymptotic lower bound)

- Let and be two functions, where . We say is , if there exist two positive constants and such that, for all .

- The two statements are equivalent:

Definition of Big Theta :

- Let and be two functions of . We say is if is both and

**Lecture 37**

- Some does not have a “simple” such that is

The time for an algorithm to run depends on:

- constant factors, implementation dependent

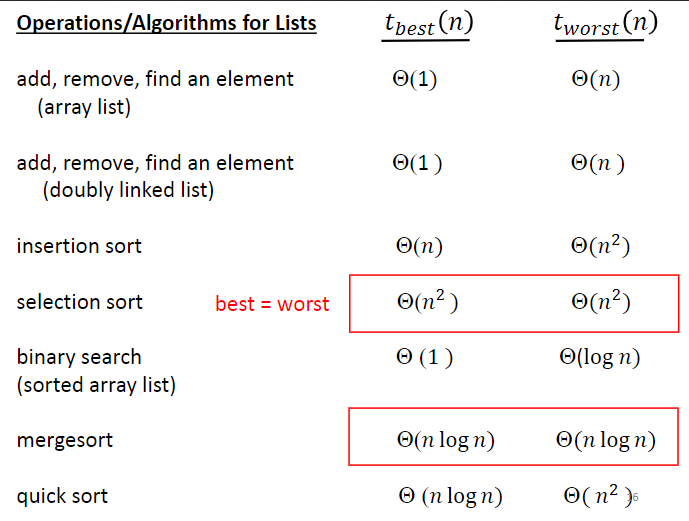
- size of input

- values of input

For any algorithm:

- is the runtime on the best case input(s)

- is the runtime on the worst case input(s)



- is an asymptotic upper bound

- is an asymptotic lower bound