

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:

$$\begin{array}{r} a[3] \ a[2] \ a[1] \ a[0] \\ + \ b[3] \ b[2] \ b[1] \ b[0] \\ \hline r[4] \ r[3] \ r[2] \ r[1] \ r[0] \end{array}$$

- 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: $a * b = \text{add } a \text{ times the number } b$
- 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 $\text{num} = q * b + r$. Subtract num by b, and add 1 to q, until r is smaller than b.
- Fast: long division

Lecture 2

Logarithms:

$$\log_b a = \log_b c * \log_c a$$
$$a^{\log_b c} = c^{\log_b a}$$

Modulo:

- Quotient remainder theorem: $a = b * q + r$
- Remainder: $a \bmod b = r$
- Convention: find positive remainder
$$-3 \bmod 5 = 2$$
$$-3 = (-1) * 5 + 2$$
- However, in JAVA:
$$-3 \bmod 5 = -3$$
- In Java defined as:

$$a = \frac{a}{b} * b + a \% b$$

- Addition:

$$(a + b) \bmod c = ((a \bmod c) + (b \bmod c)) \bmod c$$

- Multiplication:

$$(a * b) \bmod c = ((a \bmod c) * (b \bmod c)) \bmod c$$

Shift elements using mod:

- left shift n spots:

$$\text{index} = (i - 2 + \text{arr.length}) \% \text{arr.length}$$

- right shift n spots:

$$\text{index} = (i + 2) \% \text{arr.length}$$

Lecture 3

Base representation:

$$(132)_5 = 1 * 5^2 + 3 * 5^1 + 2 * 5^0 = 42$$

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

$$\text{carry} = (a_i + b_i + \text{carry}) \% \text{base}$$

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 2^n values with n bits

- One needs $\log_2 x$ (round up) to represent x different values

- One needs $\text{floor}((\log_2 m) + 1)$ to represent a positive integer M

Type	Keyword	Size	Values
Very Small Integer	byte	8-bits	$[-128, 127]$
Small Integer	short	16-bits	$[-2^{15}, 2^{15} - 1]$
Integer	int	32-bits	$[-2^{31}, 2^{31} - 1]$
Big Integer	long	64-bits	$[-2^{63}, 2^{63} - 1]$
Low Precision Reals	float	32-bits	-
High Precision Reals	double	64-bits	-
True/False	boolean	1-bit	$[\text{true}, \text{false}]$
One character	char	16-bits	-

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

	Class	Package	Subclass (same pkg)	Subclass (diff pkg)	World
public	+	+	+	+	+
protected	+	+	+	+	
no modifier	+	+	+		
private	+				

+ : accessible
blank : not accessible

- 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)`:

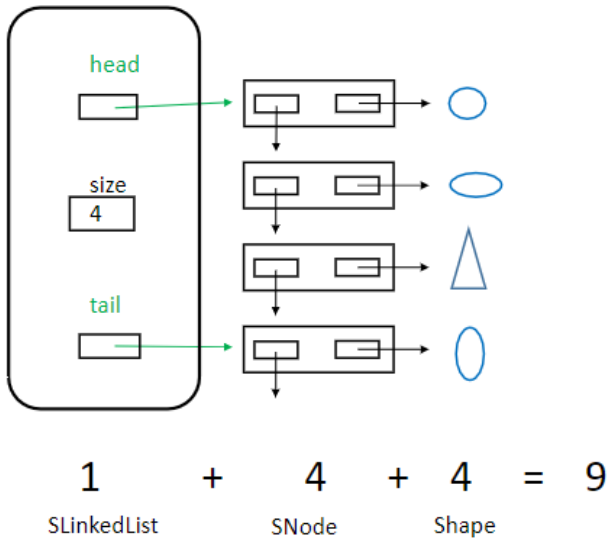
- create new node
- `tail.next` points to the new node
- tail points to the new node
- increase size by 1

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
- $O(N \log N)$: heap, merge, quick

Bubble Sort

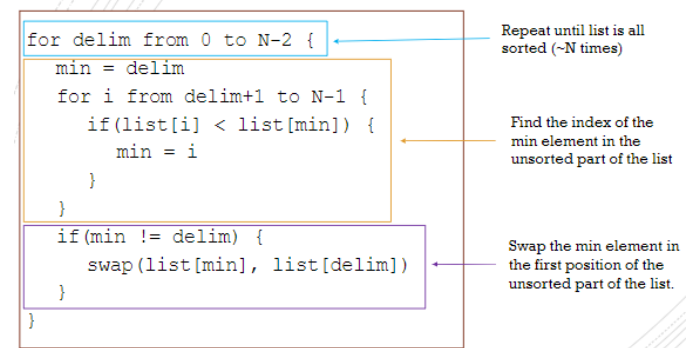
- simplest sorting algorithm
- Idea: repeatedly iterate through the list and swap adjacent elements if they are in the wrong order
- finishes when no sorting takes place in an iteration
- worst case: $O(N^2)$
- best case: $O(N)$

```

sorted = false
i = 0
while (!sorted) {
    sorted = true
    for j from 0 to list.length - i - 2 {
        if(list[j] > list[j+1]) {
            swap(list[j], list[j+1])
            sorted = false
        }
    }
    i++
}
  
```

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

```

for i from 0 to N-1 {
    element = list[i]
    k = i
    while (k > 0 && element < list[k-1]) {
        list[k] = list[k-1]
        k--
    }
    list[k] = element
}

```

Repeat until list is all sorted (~N times)

Find where the element should be inserted in the sorted part of the list + make space for it (shift all the larger elements to the right)

Insert the element in the sorted part of the list.

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

Algorithm: decide if parentheses are matched.

```

while (there are more tokens) {
    token = get next token
    if token is a left parenthesis
        push(token)
    else {
        // token is a right parenthesis
        if stack is empty
            return false
        else {
            pop left parenthesis from stack
            if popped left parenthesis doesn't match the right parenthesis
                return false
        }
    }
}
return stack.empty // true if stack is empty, false if not.

```

Lecture 14

Queue ADT

- enqueue(e), dequeue()
- 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

$$tail = (head + size - 1) \% length$$

- 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:

$$tail = (head + size - 1 + length) \% length$$

To ensure tail is valid

```

enqueue( element ){
    if ( queue.size == queue.length ) {
        // increase length of array

        create a bigger array tmp[] // e.g. 2*length
        for i = 0 to queue.length - 1
            tmp[i] = queue[ (head + i) % queue.length ]
        head = 0
        queue = tmp
    }
    queue[ (head + size) % queue.length ] = element (*)
    queue.size = queue.size + 1
}

```

Note that we don't have a tail variable here. Instead, it can be computed anytime with:
 $tail = (head + size - 1) \% length$

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

```

dequeue() { // check that queue.size > 0
    element = queue[head]
    queue.size = queue.size - 1
    head = (head+1) % length
    return element
}

```

- 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

```
public static int factorial (int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n-1);
}
```

Fibonacci

```
public static int fibonacci (int n) {
    if(n==0 || n==1) {
        return 1;
    }
    return fibonacci(n-1)+fibonacci(n-2);
}
```

Reverse list

```
public static void reverse(List list) {
    if(list.size()==1) {
        return;
    }
    firstElement = list.remove(0); // remove first element
    reverse(list); // now the list has n-1 elements
    list.add(firstElement); // appends at the end of the list
}
```

Sorting a list

```
public static void sort(List list) {
    if(list.size()==1) {
        return;
    }
    minElement = removeMinElement(list);
    sort(list); // now the list has n-1 elements
    list.add(0, minElement); // insert at the beginning of list
}
```

Tower of Hanoi

```
tower(n, start, finish, other) { // e.g. tower(5,A,B,C)
    if(n==1) {
        move from start to finish.
    } else {
        tower(n-1, start, other, finish)
        tower(1, start, finish, other)
        tower(n-1, other, finish, start)
    }
}
```

- 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 $\frac{n}{2}$

Power

- recursively call method on $n - 1$
- Complexity: $O(\log_2 n)$
- Better implementation:

```
power(x, n) {
    if (n == 0)
        return 1;
    else if (n == 1)
        return x;
    else {
        tmp = power(x, n/2);
        if (n%2==0)
            return tmp*tmp;    // one multiplication
        else
            return tmp*tmp*x    // two multiplications
    }
}
```

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: $O(\log_2 n)$

Implementation

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

Iterative implementation:

```
binarySearch(list, key) {
    left = 0
    right = list.size() - 1
    while (low <= high) { // until there are elements to search
        mid = (left + right) / 2 // compute mid
        if (list[mid] == key) // compare element with key
            return mid
        else {
            if (key < list[mid])
                right = mid - 1 // update right
            else
                left = mid + 1 // update left
        }
    }
    return -1 // key not in list
}
```

Recursive implementation:

```
binarySearch(list, key, left, right) {
    if (low <= high) {
        mid = (left + right) / 2
        if (list[mid] == key)
            return mid
        else {
            if (key < list[mid])
                binarySearch(list, key, left, mid - 1)
            else
                binarySearch(list, key, mid + 1, right)
        }
    }
    return -1
}
```

Lecture 21

Merge Sort

- divide and conquer
- Idea:
 - partition list into two halves
 - sort each half recursively
 - merge the sorted half **maintaining order**

```
mergesort(list) {
    if (list.size() == 1)
        return list
    else {
        mid = (list.size() - 1) / 2
        list1 = list.getElements(0, mid)
        list2 = list.getElements(mid + 1, list.size() - 1)
        list1 = mergesort(list1)
        list2 = mergesort(list2)
        return merge(list1, list2)
    }
}
```

Base case

Partition

Recursive sort

Merge

Where merge(list1, list2) is implemented as:

```
merge(list1, list2) {
    list = ...initialize with empty list...
    while (!list1.isEmpty() && !list2.isEmpty()) {
        if (list1.get(0) < list2.get(0))
            list.addLast(list1.removeFirst())
        else
            list.addLast(list2.removeFirst())
    }
    while (!list1.isEmpty())
        list.addLast(list1.removeFirst())
    while (!list2.isEmpty())
        list.addLast(list2.removeFirst())
}
```

Pick elements to add until one of the two lists is empty

Then add the remaining elements

- Complexity: $O(n \log_2 n)$

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

```
quickSort(list, leftIndex, rightIndex) {
    // Base case:
    if(leftIndex >= rightIndex) {
        return; // done!
    } else { // recursive step:
        i ← placeAndDivide(list, leftIndex, rightIndex)
        // i = index where the pivot is placed
        quickSort(list, leftIndex, i-1)
        quickSort(list, i+1, rightIndex)
    }
}
```

- Implementation of placeAndDivide():

```
placeAndDivide(list, leftIndex, rightIndex) {
    // pick the right most element
    pivot ← list.get(rightIndex)
    // place the wall to the left
    wall ← leftIndex - 1
    // go through all elements and compare them to the pivot
    for(int i=leftIndex; i < rightIndex; i++) {
        if(list.get(i) < pivot) {
            wall++; // move wall
            swap list.get(i) list.get(wall) // move element behind wall
        }
    }
    swap list.get(rightIndex) list.get(wall+1) // move pivot next to wall
    return wall+1;
}
```

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

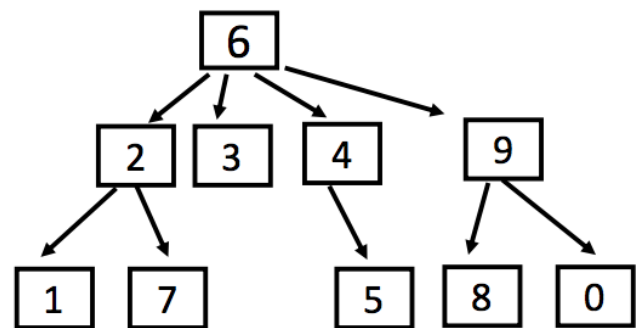
- path: a sequence of nodes
- length of a path: number of edges in the path (number of nodes – 1)
- ancestor: higher node in a path
- descendent: lower node in a path
- depth or level: length of the path from the root to the node
- height: the **maximum** length of a path from the node to a leaf

- Every node except root is a child, and has exactly 1 parent
- A tree with n nodes has $n - 1$ edges

Recursive definition of a rooted tree:

- consists of subtrees

Represented using lists:



(6(217)3(45)(980))

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

```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```

- Queue: breadth first, root, left to right

```

treeTraversalUsingQueue(root){
    initialize empty queue q
    q.enqueue(root)
    while q is not empty {
        cur = q.dequeue()
        visit cur
        for each child of cur
            q.enqueue(child)
    }
}

```

Lecture 24

Binary Tree

- each node has at most two children
- Max number of nodes and height: $n = 2^{h+1} - 1$
- Min number of nodes and height: $n = h + 1$
- 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

```

s = empty stack
cur = first token of expression list
while (cur != null){
    if ( cur is a base expression )
        s.push( cur )
    else{
        // cur is an operator
        operand2 = s.pop()
        operand1 = s.pop()
        operator = cur.element // for clarity only
        s.push( evaluate( operand1, operator, operand2 ) )
    }
    cur = cur.next
}

```

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 far 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
- $parent = \frac{child}{2}$
- $left = 2 * parent$
- $right = 2 * parent + 1$

- Implementation of add(element):

```
add(element ){
    size = size + 1    // number of elements in heap
    heap[ size ] = element // assuming array
                        // has room for another element

    i = size

    // the following is sometimes called "upHeap"

    while ( i > 1 and heap[i] < heap[ i/2 ] ){
        swapElements( i, i/2 )
        i = i/2
    }
}
```

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():**

```
buildHeap(){
    // assume that an array already contains size elements
    for (k = 2; k <= size; k++){
        upHeap( k )
    }

    upHeap(k){
        i = k
        while (i > 1) and ( heap[i] < heap[ i / 2 ] ){
            swapElement(i, i/2)
            i = i/2
        }
    }
}
```

- Best Case: $O(n)$. Elements already satisfy the heap parent-child ordering constraint, no swaps
- Worst Case:. Original list is from large to small.

- Implementation of removeMin():

```
removeMin( ){
    tmpElement = heap[1]    // heap[0] not used.
    heap[1] = heap[size]
    heap[size] = null
    size = size - 1
    downHeap( size )        // next slide
    return tmpElement
}
```

21

- Implementation of downHeap:

```
downHeap( maxIndex ){

    i = 1
    while ( 2*i <= maxIndex ){    // if there is a left child
        child = 2*i
        if child < size {        // identify smaller child
            if (heap[child + 1] < heap[child])
                child = child + 1
        }
        if (heap[child] < heap[ i ]){ // Swap if necessary.
            swapElements(i , child)
            i = child
        }
        else return              // Avoid infinite loop.
    }
}
```

Heapsort

- build heap with n elements, then call down heap n times
- each time swapping the first and last elements, then calling downheap considering a smaller heap ($n - 1$)
- finally reverse the heap array to obtain a sorted array

```

heapsort(list){
    heap = buildheap(list)

    for i = 1 to n - 1 {
        swapElements( heap[1], heap[n + 1 - i])
        downHeap( 1, n - i)
    }
    return reverse(heap)
}

```

- we could build a “max heap” and remove the maximum element each time to avoid **reverse** at the end
- worst case: $O(n \log_2 n)$

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 null if the map contained no mapping for the key.

Data Structure for Maps:

1. ArrayLists, SLL, or DLL
2. Comparable keys:

ArrayList:

- get(key) : $O(n \log_2 n)$
- put(key, value): $O(n)$
- remove(key): $O(n)$

BST: (depends on tree, assume balanced here)

- put(key, value): $O(\log_2 n)$
- get(key): $O(\log_2 n)$
- remove(key): $O(\log_2 n)$

minHeap:

- put(key, value): $O(\log_2 n)$

- get(key): $O(n)$
- remove(key): $O(\log_2 n)$

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:

obj1.hashCode() == obj2.hashCode()

Is equivalent to:

obj1 == obj2

Java String.hashCode()

- for each string, defines an integer

$$s.hashCode() = \sum_{i=0}^{n-1} s[i] * x^{n-1-i}$$

Where $n = s.length$ and $x = 31$

- Java uses **Horner's Rule** for efficient polynomial evaluation $O(n)$

```

h = 0
for (i = 0; i < s.length; i++)
    h = h*31 + s[i]

```

Lecture 29

Hashing

- Keys are hashed into hashcodes using hashCode()
- hashcodes are compressed into hashvalues using %

Compression: $i \rightarrow |i| \bmod m$

- where m is the length of the array

Hash Function: hashCode and compression

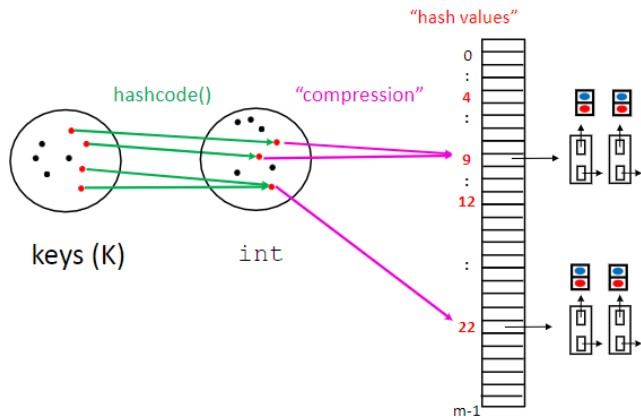
Collision:

- 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

$$= \frac{\text{number of entries}}{\text{number of buckets, } m}$$

- 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): $O(1)$ in practice
2. get(key): $O(1)$ in practice
3. remove(key): $O(1)$ in practice
4. contains(value): need hash table traversal $O(n + m)$
5. getKeys(): need hash table traversal $O(n + m)$
6. getValues(): need hash table traversal $O(n + m)$

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
- Directed 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

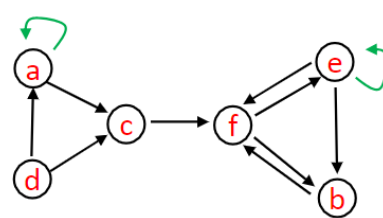
Implementation of a Graph class in Java:

- **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

loop



	a	b	c	d	e	f
a	1	0	1	0	0	0
b	0	0	0	0	0	1
c	0	0	0	0	0	1
d	1	0	1	0	0	0
e	0	1	0	0	1	1
f	0	1	0	0	1	0

Adjacency List VS Adjacency Matrix

1. Sparse graph: 10000 vertices and 20000 edges. Save space
 - Use adjacency list: matrix has 10000^2 slots, while list uses 20000 objects
2. Dense graph: 10000 vertices and 20000000 edges. Save space
 - Use adjacency matrix: matrix has 10000^2 but list has 20000000 objects
3. Check adjacency as quickly as possible.
 - Use adjacency matrix. $O(1)$ 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

```
depthFirst_Graph(v){
    v.visited = true
    for each w such that (v,w) is in E // w in v.adjList
        if !(w.visited) // avoids cycles
            depthFirst_Graph(w)
}
```

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

```
graphTraversalUsingStack(v){
    initialize empty stack s
    v.visited = true
    s.push(v)
    while (!s.empty) {
        u = s.pop()
        for each w in u.adjList{
            if (!w.visited){ // The only new part. Why?
                w.visited = true
                s.push(w)
            }
        }
    }
}
```

Breadth-First: (Queue)

- Vertices are visited in enqueue order

```
graphTraversalUsingQueue(v){
    initialize empty queue q
    v.visited = true
    q.enqueue(v)
    while (!q.empty) {
        u = q.dequeue()
        for each w in u.adjList{
            if (!w.visited){
                w.visited = true
                q.enqueue(w)
            }
        }
    }
}
```

Lecture 32

Garbage Collection

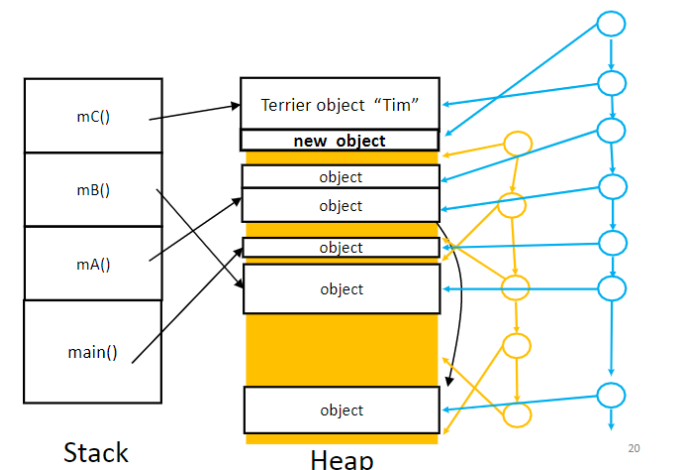
- Object is not referenced = garbage
- Live object: referenced either from call stack variable or from an instance variable in a live object
- JVM maintains a linked list of all objects, stores Object.hashCode() of each object

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

Two lists: free space, live objects



- 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:

$$R(v) = \sum_{\text{incoming edges } (w,v) \text{ to } v} \frac{R(w)}{N_{\text{out}}(w)}$$

- 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
- $t(n)$: time to execute a recursive algorithm as a function of the input size n

Reverse a list:

$$\begin{aligned} t(n) &= c + t(n-1) \\ t(n) &= c + c + t(n-2) \\ t(n) &= c + c + c + t(n-3) \end{aligned}$$

$$\begin{aligned} &\dots \\ t(n) &= c(n-1) + t(1) \\ t(n) &= cn + t(0) \end{aligned}$$

- $t(0)$ is the base case

Tower of Hanoi

$$\begin{aligned} t(n) &= c + 2t(n-1) \\ t(n) &= c(2^n - 1) + 2^n t(0) \end{aligned}$$

- $n = k$

Binary search

$$\begin{aligned} t(n) &= c + t\left(\frac{n}{2}\right) \\ t(n) &= c \log_2 n + t(1) \end{aligned}$$

- assume $n = 2^k$

Geometric Series Sums:

$$1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2}$$

$$1 + 2 + 4 + 8 + \dots + 2^{n-1} = \frac{1}{1-2} - \frac{2^n}{1-2} = 2^n - 1$$

$$1 + x + x^2 + x^3 + \dots + x^{n-1} = \frac{x^n - 1}{x - 1}$$

Lecture 34

Merge Sort

$$t(n) = cn + 2t\left(\frac{n}{2}\right)$$

- cn from copying array
- $t(n) = c n \log_2 n + n t(1)$
- $n = 2^k$
- Total calls $f(n)$ to mergesort is $2n - 1$

$$1 + 2 + 4 + \dots + n = 2^{k+1} - 1 = 2n - 1$$
- this is also the case for adding n elements to an empty arraylist and resizing $\log_2 n$ times
- also the case for putting n elements into empty hash table and rehashing $\log_2 n$ times

Quick Sort

- best case: same as merge sort

$$t(n) = cn + 2t\left(\frac{n}{2}\right)$$

- cn from partition based on pivot
- best case when two sublists have same size

$$t(n) = c n \log_2 n + n t(1)$$

- worst case: one sublist has only 1 element

$$t(n) = cn + t(n-1)$$

$$t(n) = c \frac{n(n+1)}{2}$$

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 $t(n)$ has a limit if, for any $\epsilon > 0$, there exists an n_0 such that for any $n \geq n_0$, $|t(n) - t_\infty| < \epsilon$

Bounded Definition:

- Let $t(n)$ and $g(n)$ be two functions, where $n \geq 0$. Then $t(n)$ is asymptotically bounded above by $g(n)$ if there exists n_0 such that, for all $n \geq n_0$, $t(n) \leq g(n)$

Definition of Big O: (upper bound)

- Let $t(n)$ and $g(n)$ be two functions, where $n \geq 0$. $t(n)$ is $O(g(n))$ if there exist two positive constants n_0 and c such that, for all $n \geq n_0$, $t(n) \leq c g(n)$

Lecture 36

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

- Let $t(n)$ and $g(n)$ be two functions, where $n \geq 0$. We say $t(n)$ is $\Omega(g(n))$, if there exist two positive constants n_0 and c such that, for all $n \geq n_0$, $t(n) \geq c g(n)$

- The two statements are equivalent:

$$f(n) \text{ is } O(g(n))$$

$$g(n) \text{ is } \Omega(f(n))$$

Definition of Big Theta (Θ):

- Let $t(n)$ and $g(n)$ be two functions of $n \geq 0$. We say $t(n)$ is $\Theta(g(n))$ if $t(n)$ is both $O(g(n))$ and $\Omega(g(n))$

Lecture 37

- Some $t(n)$ does not have a "simple" $g(n)$ such that $t(n)$ is $\Theta(g(n))$

The time for an algorithm to run depends on:

- constant factors, implementation dependent
- size of input
- values of input

For any algorithm:

- $t_{best}(n)$ is the runtime on the best case input(s)
- $t_{worst}(n)$ is the runtime on the worst case input(s)

Operations/Algorithms for Lists	$t_{best}(n)$	$t_{worst}(n)$
add, remove, find an element (array list)	$\Theta(1)$	$\Theta(n)$
add, remove, find an element (doubly linked list)	$\Theta(1)$	$\Theta(n)$
insertion sort	$\Theta(n)$	$\Theta(n^2)$
selection sort	best = worst $\Theta(n^2)$	$\Theta(n^2)$
binary search (sorted array list)	$\Theta(1)$	$\Theta(\log n)$
mergesort	$\Theta(n \log n)$	$\Theta(n \log n)$
quick sort	$\Theta(n \log n)$	$\Theta(n^2)$

- $O()$ is an asymptotic upper bound
- $\Omega()$ is an asymptotic lower bound