COMPSCI 201 MIDTERM 1 REFERENCE SHEET

Arrays: stores fixed number of entries of a single type

- Type[] name = new Type[length]
- contains primitives or objects (vs objects)
- no pretty print (vs yes for collections)
- .asList(), .toArray(), List to Array with loop

Strings: hold an Array of characters

- += concatenation O(n) because immutable
- must be copied over, unchanged after creation
- StringBuilder.append() = O(1) per char; large capacity until overflow, geometric growth

methods

- dynamic methods: called on a created object, has access to object data and arguments
- static methods: called on the class, only has access to arguments

class: blueprint for objects

- specifies data and operations for a type of object
- . operator accesses instance variable or method of this object (on what it's called)
- public (can be accessed by code outside of class) vs private (only inside)

objects: an instance of a class

- must allocate memory and create
- constructor: specifies how to create a new object
- Object: default class from which all inherit

equals

- equals() Object default → checks if two objects are the same object in memory
    - should be overridden to check value
- == → checks if two objects are the same object in memory

interfaces

- ADT (Abstract data type): specifies what a ds does (functionality) but not how (implementation)
- interface: specify a set of abstract methods that an implementing class must override and define (ADT)
- implement → public class NewList implements List
    - must have at least same methods

collections

- interface, group of objects
- need implementing class at creation
    - List: ordered sequence of values
    - Set: unordered collection of unique values
    - Map: collection that associates keys and values

ArrayList

- ArrayList implements List with Array, by keeping an array with extra space at the end
    - Array out of space → make one twice as large and copies old over
- get() = O(1)
- contains() = O(n), calls .equals() on every element
- size() = O(1)
- add()
    - O(1) at end of the list (amortized bc copying)
    - O(n) at beginning of list (has to shift over)
- remove() = O(N), caused by having to shift all elements

HashSet

- unordered but very efficient; internally creates a HashMap to store elements of set
- add(), remove(), contains(), size() = constant time O(1)
- loop over with enhanced for loop (or iterator)

HashMaps

- calculates hash (int) of a key with .hashCode() to see where to store values
- puts it in a "bucket" of k,v pairs with that hash index
- different keys can have same hash because we round (%) on value of .hashCode()
- put() adds to hash index bucket, updates if present = O(1)
- get() gets value at computed location to get hash bucket iterates over hashBucket keys to find .equals() key = O(1)
- containsKey() checks if location empty = O(1)

TreeSet/TreeMap

- ordered set/map, slightly less efficient than HashSet/HashMap
- keeps values (for Set) and keys (for Map) sorted by "natural ordering," uses special kind of binary tree
- most methods are runtime complexity O(log n)

Big O (asymptotic) analysis

- describes the runtime (# of operations performed) or memory (space for storing variables)
- constant time O(1): doesn't depend on size of input (arithmetic, comparison, object attributes)
- take biggest term

Big O	Name	Example
$O(2^N)$	Exponential	Calculate all subsets of a set
$O(N^3)$	Cubic	Multiply NxN matrices
$O(N^2)$	Quadratic	Loop over all *pairs* from N things
O(N log(N))	Nearly-linear	Sorting algorithms
O(N)	Linear	Loop over N things
O(log(N))	Logarithmic	Binary search a sorted list
O(1)	Constant	Addition, array access, etc.

- complexity of the line, multiply by # of times the line is executed, add complexity over all lines
    - for loop when i increments by 1 to n is generally O(n)
    - nested for loops where both i and j increment by 1 to n is O(n^2)
    - for loop when i is doubled from 1 to n is O(log(n))
- methods: go from the inside to the outside, e.g. for outer(inner(n))
    - calculate some value returned by inner(n) and runtime of inner(n)
    - calculate the runtime of outer() on that value return by inner(n)
    - add both runtimes to get runtime complexity of outer(inner(n))

SUHA (Simple Uniform Hashing Assumption)

- suppose we hash N pairs to M buckets
- SUHA: any given key $k \in K$ is equally likely to be hashed to one of the M possible values, independently of what values other keys might hash to
- assuming uniform hashing, probability two random keys hash to same bucket is 1/M
- expected number of pairs per bucket is N/M (very likely true)
- time to get the hash is constant, but time to search over hash index "bucket" by calling .equals() on everything will take N/M runtime
- we face memory/runtime tradeoff with N pairs and M buckets, where…
    - if N >> M – too many pairs in too few buckets, runtime N/M is NOT constant
    - if M >> N – too many buckets, constant runtime but NOT memory efficient
    - usually "load factor", or maximum N/M ratio allowed in Java is 0.75, where M is slightly bigger than N, overall constant runtime but reasonable memory usage
    - HashMap/Set resizes if load factor is exceeded