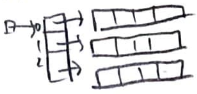
Midterm 1

8 primitive
1) boolean
2) char
3) byte
4) short
5) int
6) long
7) float
8) double

Reference Types
1) String
2) Array
3) Classes

2D Arrays
int [][] a = new int [row][column]



ArrayCopy (src, strt, target, strt, #)
src = source array
strt = start position in source
target = target array
start = start position in target
# = # to copy

TGROP
b=a copies all bits from a into b

Static variable: only 1 instance exists and it belongs to the class instead of instance

Static method: belongs to the class rather than an object of the class
- can be invoked w/o need for creating an instance
- can only access static variables and methods

Interface Inheritance
- when a subclass inherits through inherits keyword, it inherits the method signatures

ex) List Interface
SLList | AList
DLList

Implementation Inheritance
- subclass extends superclass
- inherits all instance variables, methods, nested classes,
DN inherit constructor, call super();

Overload: same method name, diff method signature

Overriding: when you have same everything

ERRORS
public int x (int y)
public char x (int y)
- Doesn't compile bc. Java sees this as same method signature.

Dog d = (Dog) new Animal();
- Runtime Error

calling this.function() on a static func
- Compile Error

4) if static type doesn't have a field (variable) or a method
A thingC = new C();
syso (thingC.valC); ERROR compile

5) accessing a private variable from a diff. class
- compile ERROR

6) Tree t = new Animal();
- compile ERROR

7) Caste ERRORS
- compile ERROR: Cat C = (cat) new Dog();
- compile ERROR: Animal a = (animal) new Tree();
- Runtime ERROR: Dog d = (Dog) new Animal();

Static vs Dynamic Type
List B = new SLList();
static — dynamic

- compiler treats it as the outer box
- dynamic type doesn't matter for passed in values.

static ≥ dynamic

steps: dynamic method lookup
1) check if static type has method
2) check if dynamic type has method
3) if it does, use it, otherwise use static
4) error if doesn't have in static

Rules
- if methods are overloaded, use static
- if methods are overridden use dynamic

Method Signature
- name of method
- parameter type
- # of parameters

Caste
(type1) var_name
↑ what you want to change to static type.

Test Taking
- Use .equals to compare actual equality in value.

Making a comparator class

```
public class ReverseComparator implements Comparator<Item> {
    public int compare(Item i1, Item i2) {
        return -i1.compareTo(i2);
    }
```

SORT Tricks
Mergesort good for big data that can't fit in memory.
Quicksort good for randomly shuffled list of items.

Bottom Up Dynamic Programming
1 cent 2 cent 4 cent 8 cent ...

```
public int countChange(int n) {
    int[] ways = new int[n+1];
    ways[0] = 1
    for (int coin = 1; coin <= n; coin *= 2) {
        for (int i = coin; i <= n; i++) {
            ways[i] += ways[i-coin]
    } }
    return ways[n]
public static int quicklyconstStairWay(int n) {
    if n == 1 return 1
    int[] ways = new int[n+1];
    ways[1] = 1
    ways[2] = 2
    for (int i = 3; i < n; i++) {
        ways[i] = ways[i-1] + ways[i-2]
    return ways[n]
```

(recursion of f(n-1) + f(n-2))

Selection sort: $\Theta(N^2)$ time worst case

Graph Tricks ✱
9) Edge weights are in range 1 to W. sorting and then set manage: $O(E + W + E\cdot\alpha(V))$

1) checking whether G has a cycle containing edge e. (s,t)
- remove e from graph, run DFS/BFS to see if path exists from s to t. If yes then cycle.

2) set that when removed, no cycles and the minimum weight of the set
- negate edges, run MST: Prims/Kruskals; return set of edges not in MST.

3) shortest path from any start vertex to end vertex from set of start S, and set of end e
- add dummy source connecting all start and dummy node connecting all end. Run dijkstra from source to end source.

4) If a DAG has a path that traverses every vertex, then its topological sort = unique that visits vertex 1 time

5) To run faster than Dijkstras, and want SPT, and weights are bounded 1-9 convert graph into new graph where the links are separated to w-1 edges and just run BFS which is |V+E|

6) Let G be an undirected connected graph. Give O(V+E) time algorithm to compute a path in G that traverses each edge in E exactly once in each direction
- Modify DFS such that, you mark each edge direction taken, if both directions taken, then remove edge. Prioritize going to unexplored edge and if you're stuck backtrack and try again.

7) directed graph G, check if all vertices are having one at most simple path.
- ∀ visited vertices:
visit vertex:
run DFS from vertex,
if vertex visited twice
false

8) Graph G where the weight of its edges are in range 1 to V.
- use counting sort, O(V+E) then add to the tree and manage the set
$O(E \cdot \text{Alpha}(V))$

Midterm 2

Iterators

- to make an Object Iterable
- implement Iterable<T>
- Boolean hasNext
- T next()
- optionally remove()

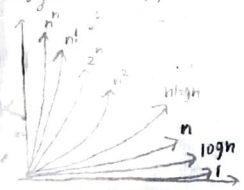
Generics

```
public class Node {
  Object first;
  Node rest;
  public Node(Object first, Node rest){
    this.first = first;
    this.rest = rest;
  }
}

public class Node<T>{
  T first;
  Node<T> rest;
  public Node(T first, Node<T> rest){
    this.first = first,
    this.rest = rest;
  }
}
```

Asymptotics

- Big Theta: exact order of growth
- Big O: upper bound, less than or equal to
- Ω Big Ω: lower bound, greater than or equal to



Searching Data Structures

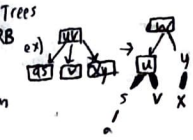
- Set
  - Chaining HT
  - Linear Probing HT
  - LinkedList
  - ResizingArray
  - BST (vanilla)
  - RedBlack - implemented like an binary Tree
  - B-Trees 2-3-4 - node can have 4 children
  - Heap (BinaryTree)
- Map
  - Heap
  - Balanced Tree
  - Ordered LinkedList
  - Chaining HT
- PQ
  - LinkedList
  - ResizingArray
- List
- DisjointSets
  - QuickFind - array to store the element
  - QuickUnion - array to store parent of
  - weighted QU - QU but add smaller to larger.
  - WQUPC - can have multiple branches, only increase in height when we add same length. best: Θ(1) worst: Θ(log N)

CS61B

- ArrayMap
- BSTMap
- HashMap
- Disjoint sets
  - QuickFind
  - QuickUnion
  - weightedQuickUnion

SearchTrees

- Binary Search Tree
- 2-3-4 trees
- B-Trees
- LLRB



Collection

Tree Traversals

- Level Order: top to bottom, left to right
- PreOrder: if x = null, return,
  print(x.key)
  preOrder(x.left)
  preOrder(x.right)
  DBACFEG
- InOrder: if x = null, return;
  InOrder(x.left)
  print(x.key)
  InOrder(x.right)
  ABCDEFG
- PostOrder: if x = null return;
  postOrder(x.left)
  postOrder(x.right)
  print(x.key)
  ACBEGFD

Array Sorting Algorithms

algorithm	best	Average	Worst
Quicksort	Ω(nlogn)	Θ(nlogn)	O(n^2)
MergeSort	Ω(nlogn)	Θ(nlogn)	O(nlogn)
Selection Sort	n^2	n^2	n^2
Bucket Sort	n+k	n+k	n^2

Summations

- 1+2+...+N = Θ(N^2)
- 1+2+4+...+N = Θ(N)
- 1+2+4+...+2^N = Θ(2^N)
- Σ geo = $a_1\frac{(1-r^n)}{(1-r)}$
- $a_n = a_1 r^{n-1}$
- $N^2 * (\frac{1}{4} + \frac{1}{16} + \frac{1}{64} + \ldots + \frac{1}{4^n}) = O(N^2)$

Hashing

- consistency; when hashcode() is called on the same object, same value
- Equality: if o1.equals(o2) is true then hashcode(o1) == hashcode(o2)
- Load factor L(x) = N/M N = # of items M = # of buckets (array.length)

Operations & Runtimes

1. add
   - takes Θ(1) time
   - when the LF is exceeded, resize will occur, takes Θ(N) time
2. contains
   - if elements are distributed evenly Θ(1)
   - if all elements are in the same bucket Θ(N)

Heaps

- tree based data structure that allows retrieval of min/max quickly
  - Each node has up to 2 children. Every level but last is filled. Last level as far left as possible.
  - Node's value is (less) (greater) than both its children.
- O(LogN) levels
- Represented using array. Element at level c are placed in indices $[2^c, 2^{c+1}-1]$
- index = k, leftChildIndex = 2k, rightChildIndex = 2k+1, parent = k/2
- Operations
  1. getMin: Θ(1)
  2. insert: O(LogN) add element at end, swim up
  3. removeMin: O(LogN), swap root w/ rightmost element on bottom level
     - remove it, swim down, swap with smaller of 2 children

Tree

Binary Search Tree: every key on left is less; every key on right is greater

Balanced Search Tree:

2-3 Tree:
- 1 value, 0-2 children
- 2 value, 0-3 children

2-4 Tree:
- Node has at most 3 value
- non leaf, 2-3, 4 children

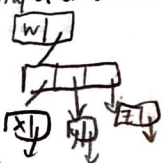
Operations

- deletion in BST's w/ 2 children (Hibbard)
  - delete a node by replacing it with its successor: the leftmost node of right subtree or rightmost node of left tree
- insertion into B-Trees:
  - insert element X into appropriate Node, if overstuffed, push left middle up to parent by one level. Repeat until no nodes are overstuffed.

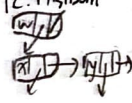
Approaches for Tree Representations

1a) Fixed number of links

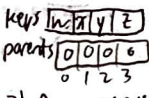
1b: array of child links



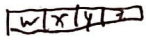
1c: FirstBorn



2: array of keys, array of structs



3) Array of keys



Disjoint Sets: connect, isConnected

- QuickFind (constructor Θ(N), connect C(N), isConnected C(1))
  - int[] each index corresponds to each item. Value represents the set.
- QuickUnion (find O(N), connect O(N), isConnect N))
  - int[] value represents the parent of that item
- WQU (LogN)

Math

$\log(N!) \in \Theta(N \log N)$

$N \log N \in \Omega(\log N!)$

$\in \Theta(\log N!)$

Proof of $N \log N \in \Omega \log N!$

$\log N! = \log N + \ldots \log(1)$

$N \log N = \log N + \log N + \log N + \ldots$

Thus, $N \log N \in \Omega(\log N!)$

Operators

- >>>: shifts bit to the right
- &: Compares bitwise: 1 and 0 = 0, 1 and 1 = 1
- |:
- >>:
- <<:

- Integer.MIN_VALUE+1 = returns itself.

! INT, are 32 bits long.

Java can't handle deep recursion

-so don't use recursive MSD for long similar strings

Equals

.equals checks same values
== checks same reference

Data Structure Operations

data structure	Average/Worst			
	Access	Search	Insertion	Deletion
Array	Θ(1) O(1)	Θ(n) O(n)	Θ(n) O(n)	Θ(n) O(n)
Stack	O(n) O(n)	O(n) O(n)	Θ(1) O(1)	Θ(1) O(1)
Queue	O(n) O(n)	O(n) O(n)	Θ(1) O(1)	O(1) O(1)
HashTable	N/A	Θ(1) O(N)	Θ(1) O(N)	O(1) O(N)
BST	Θ(logN) O(N)	Θ(logN) O(N)	Θ(logN) O(N)	Θ(logN) O(N)
B-Tree (2-3) (2-4)	Θ(LogN)			self balancing
LLRB Tree	Θ(LogN)			