Data structures sheet

hashmap

- java: load factor = .75, default init capacity: 16, uses buckets
- string hash function: s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1] where n is length mod (table_size)

arrays and strings

- start by checking for null, length 0

linkedlist

```
class Node {
	Node next;
	int val;
	public Node(int d) { val = d; }
}
```

- finding a loop is tricky, use visited

stack

```
class Stack {
	Node top;
	Node pop() {
		if (top != null) {
			Object item = top.data;
			top = top.next;
			return item;
		}
		return null;
	}
	void push(Object item) {
		Node t = new Node(item);
		t.next = top;
		top = t;
	}
}
```

- sort a stack with 2 stacks
	- make a new stack called ans
	- pop from old
	- while old element is > ans.peek(), old.push(ans.pop())
	- then new.push(old element)
- stack with min - each el stores min of things below it
- queue with 2 stacks - keep popping everything off of one and putting them on the other
- sort with 2 stacks

trees

- to go through *bst (without recursion) in order*, use stacks
	- push and go left
	- if can't go left, pop and go right
- *breadth-first tree*
	- recursively print only at a particular level each time
	- create pointers to nodes on the right
- *balanced tree* = any 2 nodes differ in height by more than 1
	- (maxDepth - minDepth) <=1
- *trie* is an infix of the word “retrieval” because the trie can find a single word in a dictionary with only a prefix of the word
	- root is empty string
	- each node stores a character in the word
	- if ends, full word
		- need a way to tell if prefix is a word -> each node stores a boolean isWord
- *AVL tree*
	Guarantees log(n)
	balance factor := The height of the right subtree minus the height of the left subtree - always should be between -1 and 1
- *red-black tree*
	- Every simple path from a node to any descendant leaf contains the same number of black nodes - The height of the right and left subtree can differ by a factor of n
- *splay tree* - a self-balancing tree that keeps "recently" used nodes close to the top

heaps

- used for *priority queue*
- peek(): just look at the root node
- add(val): put it at correct spot, percolate up
	- percolate - Repeatedly exchange node with its parent if needed
	- expected run time: ∑i=1..n 1/2^n*n=2
- pop(): put last leaf at root, percolate down
	- Remove root (that is always the min!)
	- Put "last" leaf node at root
	- Repeatedly find smallest child and swap node with smallest child if needed.