Exam 2 Note Sheet

-**Hash Tables:**

**General:** generally O(1) lookup, insertion, and deletion

Rehashing: when things get bad, re-hash into a new table of twice the size;Expensive and inefficient; do when a large table gets about half full

**Cuckoo Hashing:** O(1) lookup time; Inserting may push an older key to a different location, but in a second table (there are exactly 2 tables, each w/unique hash functions); A key is always in its first or second choice

**Linear probing:** if desired location is full, look at next spot; Can make deletion complicated – must use lazy deletion;

Primary clustering: things that didn’t want to go to the same location compete for space

**Chaining:** a container at each location in the table (array, LL, vector, etc) to chain things that hash to the same value; Solves primary clustering, results in secondary clustering; Secondary clustering:  two records do only have the same collision chain if their initial position is the same

**Double Hashing:** if a collision occurs, each key jumps by a personalized increment, usually key % tablesize; Tries all keys before making it back to original key if tablesize and increment are prime relative to each other; Quadratic Probing: if hashed to H and there is a collision, try H + 12; H + 22, H + 32, etc; Can cause secondary clustering

-**Priority Queues:**

**Heap as an Array:** can be min heap or max heap; **Insertion:** Elements are inserted to the furthest but soonest left spot available so all but the last level is full; When inserting, look at parent and swap if needed; repeat until done or new node is at the root O(1) best and O(log n) worst; i = subscript of a node; 2i+1 and 2i+2 = i’s children; (i-1)/2 = i’s parents; always round down; **Deletion:** only ever delete the root – swap hole at the root w/ bottom-leftmost element and let heap sort itself as you would when inserting; O(log n)

**Leftist Heap:** keep track of null path length (npl) for each node; npl is dist. to nearest nullptr; npl of left child must be >= npl of right child; On merge O(log n) compare npl on the way back up and swap children as needed; Skew Heaps: always swap children after recursively merging; amortized O(log n); don’t keep track of npl

**Binomial Queues:** have the heap property; have B0, B1, B2 trees, etc. subscript denotes #nodes off root; B1 is 2 B0 trees, B2 is 2 B1­ ­trees, etc.; rules: may only have 1 of each kind of tree, but it is not req’d to have one of each; merge by attaching one tree to the root of another and keeping the heap property (only ever merge 2 of the same trees); have as few trees as possible and follow these rules; O(1) insert; O(log n) find min/max, delete min/max, merge

-**Sorting:** it is impossible to sort in O(logn) time or better

**Terms: Internal:** sort done in main memory; **External:** uses auxiliary storage (disk); **Stable:** retains original order in the case of duplicates; **Adaptive (non-oblivious):** takes advantage of existing order; **Sort-by-Address:** uses indirect addressing so the structure doesn’t need to be moved; **Indirect Sort:** make an array of pointers and sort those; used for very large records; **Inversion:** a pair of elements that is out of order; **In-Place:** uses at most O(1) aux space beyond initial array;

**Selection Sort:** 1) Find largest element, put it at the end and place what was there in the hole 2) find next largest element, put it at second to last 3) repeat; O(n2), can be stable by pushing back rather than swapping, not adaptive, in place

**Bubble Sort:** Compare 2 elements at a time and swap if necessary; repeat but exclude the last value from the prev run; repeat until sorted (stop when no swaps are done); O(n2), O(n) if nearly sorted, stable, adaptive if hasSwapped flag is included, in place

**Insertion Sort:** Take from an unsorted list and insert it in its proper place in a sorted list; uses 2 groups of keys, sorted and unsorted; O(n2), O(n) if nearly sorted, stable, adaptive, in place

**Shell Sort:** Sorts in stages: look at 2 things that are some gap away, sort (insertion), then repeat using a smaller gap, eventually w/gap of 1 (usually divide gap by 2); Worst: O(n2), avg bet: O(n1.17) and (n1.25), nearly sorted: O(nlogn); adaptive, unstable, in place

**Merge Sort**: divide into single elements; merge adjacent lists 1 at a time, sorting as you go; repeat until list is reconstructed in order; O(nlogn), stable, not adaptive, not in place

**Quick Sort:** pick a pivot; move it to 1st position, look for things smaller/larger than pivot; swap pivot into its proper place; recurse w/new partitions on either side of the old pivot; Worst: O(n2), Best: O(nlogn), not stable, not adaptive, not in place

**Heap Sort:** O(nlogn), not stable, not adaptive, in place;

**Bucket/Bin Sort:** all vals bet. 1 and k all are unique; place each element in its proper index(bin) of size k; Insert: O(n), print: O(n+k)

**TimSort:** Hybrid insertion & adaptive mergesort; looks for existing order, merge ordered sections 2 at a time until done; galloping; O(nlogn), not in place, stable, adaptive

-**Union – Find:** find: O(1), union: O(n)