**Priority Queues**

Binary Heap

Parent – k/2

Child – k\*2, k\*2+1

Add – O(log n)

Remove – O(log n)

Code

Swim – compare to parent and swap

Sink – compare children to each other, compare to largest child and swap

Ternary Heap

Parent – (k+1)/3

Child – (3\*k), 3\*k+1, 3\*k-1

**Comparison Sorts**

Selection Sort

Compare – O(n^2)

Swap – O(n)

Steps

Find minimum key in list and swap

Heap Sort

Compare – O(n log n)

Swap – O(n log n)

Steps

Sink all non-leaf nodes to construct a heap

Swap the rightmost leaf with the root, and sink

Insertion Sort

Compare – O(n^2)

Swap – O(n^2)

Steps

Swap with adjacent key if less

Shell Sort

Compare – O(n^1.5)

Swap – O(n^1.5)

Knuth: (3t - 1) / 2 ⇒ 1, 4, 14, 40, 121, …

Prefer Knuth because larger gaps between sequences

Steps

Create sequence (max is n/3)

Compare keys that align to sequence

Insertion Sort those keys

Use smaller sequence

Divide & Conquer Sort

Merge

O(n log n)

Requires temp array

Steps

Keep dividing into smaller pieces

Merge smallest (size of 1)

Sort merged by comparing one element from one piece to element of another

Merge again, etc

\*Recursive

Quick

Worst – O(n^2) chooses worst key every time

Average – O(n log n)

Steps

Swap keys that are on left / right side of pivot

Move pivot into place

Intro

Reduces quick sort worst to O(n log n)

Gets max depth using 2\*log(n)

When max depth is 0 (decreases on every new pivot), uses Shell Sort

**Distribution Sort**

Bucket

Need to know min & max

Place into buckets for each int, or each range (sort the buckets)

Worst case - O(n^2)

Radix

Place into buckets by LSD (always 10 buckets)

Then go to next digit, repeat

O(n)

**BST**

Unbalance Worst Case – O(n) (for search, insert, delete)

Balanced Worst Case – O(log n) (for all)

AVL

Using heights of node in tree to get balance factor

Whenever you rotate, must reset heights to get correct balance factors

Steps to Balance

BF = 2, then left heavy

If BF of child is -1, then only right child, rotate left

Else only left child, rotate right

BF = -2 then right heavy

If BF of child is 1, then only left child, rotate right

Else only left child, rotate left

Else balance parent

Red & Black

Balancing is not perfect

The root and all leaves are black

Every red node must have two black children

Any path from any node to its descend must contain the same number of black nodes

Steps to Balance

Always set parent to black and grandparent to red

If red uncle, set red uncle to black, do the above, balance the grandparent

If black uncle,

If left right, rotate left

If right left, rotate right

Do the above

If left left, rotate right

If right right, rotate left

**Trie**

Put

Start at root

Add nodes from char array

Set endstate as true

Find

Start at root

Check children for matches from char array

Return null or last found node

Remove

If can’t find key, return false

If has children, set endstate to false and return

Else remove node

Check parent for endstate or children

If so, stop

Else set node to parent

Get alphabetical list

Do BFS

Remove from queue and check if it is a word

Add its children to queue

Queue is FIFO, so checks other parent’s children before its children

**Dynamic Programming**

Hanoi

Get steps to move from source to intermediate

If exist, add to list, otherwise solve for source to intermediate through destination

Record the step

Get steps to move from intermediate to destination

If exist, add to list, otherwise solve for intermediate to destination through source

If steps do not exist, add them to map

LCS

**solve**

if (i < 0 || j < 0)

return "";

//Found common sequence

if (c[i] == d[j])

return solve(c, d, i-1, j-1, table) + c[i];

//If search on String c has exhausted

if (i == 0) return solve(c, d, i, j-1, table);

//If search on String d has exhausted

if (j == 0) return solve(c, d, i-1, j, table);

//Recursively search for the table[i][j] with the largest element

return (table[i-1][j] > table[i][j-1]) ? solve(c, d, i-1, j, table) : solve(c, d, i, j-1, table);

**create table**

final int N = c.length, M = d.length;

int[][] table = new int[N][M];

for (int i=1; i<N; i++)

for (int j=1; j<M; j++)

table[i][j] = (c[i] == d[j]) ? table[i-1][j-1] + 1 : Math.max(table[i-1][j], table[i][j-1]);

return table;