***Disclaimer: Content in this cheat sheet is biased towards the Spring 2019 offering of CS 61B with Josh Hug, and may not include everything in scope for future iterations of the course. If there’s a new topic or any errors, shoot me a message!***

#### **Data Structures Runtime Table:** Ordered as (Average Runtime / Worst Runtime) - inspired by <https://www.bigocheatsheet.com/>

| **Data Structure** | **Access** | **Search/Contains** | **Insertion** | **Deletion** |
| --- | --- | --- | --- | --- |
| **Array (Resizing)** | Θ(1) | Θ(n) | Θ(n) | Θ(n) |
| **Stack/Queue** | Θ(n) | Θ(n) | Θ(1) | Θ(1) |
| **Linked-List**  *Best version SLL and DLL* | Θ(n) | Θ(n) | Θ(1)  *Insert at front for SLL* | Θ(1) |
| **Hash Table**  *Constant amortized time* | No indexing | Θ(1) / Θ(n)  *Good/Bad Distribution* | Θ(1) / Θ(n) | Θ(1) / Θ(n) |
| **BST** | Θ(log(n)) / Θ(n)  *Bushy/Spindly* | Θ(log(n)) / Θ(n) | Θ(log(n)) / Θ(n) | Θ(log(n)) / Θ(n) |
| **B-Tree/Red-Black** | Θ(log(n)) | Θ(log(n))  *Builds from the ground up!* | Θ(log(n)) | Θ(log(n)) |
| **KD-Tree** | Θ(log(n)) / Θ(n) | Θ(log(n)) / Θ(n)  *May need to check bad side* | Θ(log(n)) / Θ(n) | Θ(log(n)) / Θ(n) |
| **Trie** | No indexing | O(L)  *L is length of longest string* | O(L) | O(L) |
| **Heap** | Θ(1)  *getMin() is constant, removeMin() is NOT* | N/A  *changePriority() is logN, otherwise only min/max node matters* | Θ(log(n))  *Insert to bottom, bubble up* | Θ(log(n))  *Move last elem to top, bubble down* |

***Θ(1) < Θ(log(n)) < Θ(n) < Θ(nlog(n)) < Θ(n2) < Θ(2n) < Θ(n!)***

**NOTES:** *N* insertions/deletions are different than a single one! Would need to **multiply by *N***

Resizing then adding is additive, not multiplicative (Ex. Array resize = Θ(n + 1))

#### **Sorting Algorithms Runtime Table:** inspired by <https://www.bigocheatsheet.com/>

| **Sort** | **Best Case** | **Worst Case** | **Additional Space** | **Notes/Stability** |
| --- | --- | --- | --- | --- |
| **Selection Sort** | Θ(n2) | Θ(n2) | Θ(1) | NOT Stable |
| **Heap Sort**  *In-place, bottom-up heapify* | Θ(n)  *No bubble down,*  *all duplicates* | Θ(nlog(n))  *Requires no recursion!*  *:O* | Θ(1) | NOT Stable  *Uses a MAX-heap* |
| **Merge Sort** | Θ(nlog(n)) | Θ(nlog(n)) | Θ(n) | Stable  *Better than Quicksort for Linked-Lists* |
| **Insertion Sort** | Θ(n)  *No Inversions* | Θ(n2) | Θ(1) | Stable |
| **Quicksort**  *With* [*tail call optimization*](https://www.geeksforgeeks.org/quicksort-tail-call-optimization-reducing-worst-case-space-log-n/)*, can have Θ(log(n)) space* | Θ(nlog(n))  *May be Θ(n) on duplicates, depends* | Θ(n2)  *Always picking extreme value as pivot* | Θ(n)  *If 2 recursive calls, both sides* | Depends on Strategy  *3-way-partitioning = stable*  *Hoare = NOT stable* |
| **Counting Sort**  *N = # keys*  *R = Size of Alphabet* | Θ(N +R) | Θ(N + R) | Θ(N + R) | Stable  *Single Alphabet Keys Only* |
| **LSD Radix Sort**  *W = Width of Longest Key* | Θ(W(N +R)) | Θ(W(N +R)) | Θ(N + R) | Stable  *Strings of Alphabet Keys* |
| **MSD Radix Sort** | Θ(N +R) | Θ(W(N +R)) | Θ(N+WR) | Stable |

***= Comparison Sorts = Counting Sorts \*Outlined Boxes are average runtimes***

**NOTES:** ***Quicksort*** *considered fastest comparison sort if using Hoare Partitioning,* ***Merge Sort*** *is the fastest STABLE comparison sort,* ***Insertion Sort*** *is the fastest for ALMOST SORTED lists*

**Comparison vs. Counting**: For sufficiently large collections from any alphabets, **Counting Sort** *fastest*, BUT for, say, 100 elements with huge alphabet size, **Quicksort** may be faster

[Comparison Sorts Animations](https://www.toptal.com/developers/sorting-algorithms)

#### **Binary Search Trees (BSTs):**

* Elements in a BST must implement *comparable*, .compareTo() method
* Everything to the left of a node is smaller, everything to the right of a node is larger
* Usually no duplicate items, but strategies to include them include using a Map<T: Integer> to store multiplicity or update Node class to include a middle child

***Hibbard Deletion (Θ(log(n)) / Θ(n)):***

* When removing any node with **2 children**, replace it with the largest node in its left branch or the smallest node in its right branch (if **1 child**, simply replace the node with child, if **0 children**, simply delete node)
* Worst case when has both left and right child, and branches are spindly towards the middle

#### **B-Trees (2-3/2-3-4) and Red-Black Trees (LLRB/2-3-4):**

* Insert into node until node overflows, then push middle (or left middle) element to the top
* B-Tree nodes **always** have the max number of children (# items in node + 1) or 0 children, all leaves are the same distance to the source
* Red-Black Trees are just a BST representation of a B-Tree, with red nodes to represent nodes that are on the same level as each other





*\* All diagrams modified from Josh Hug*



***Left-Leaning Red-Black Tree Rotations:***

* ALWAYS insert into LLRB with a red node, below are some common cases
  + If inserting on **right** with no siblings, **rotate left** on its parent
  + If inserting on **right** with a red node already to left, “**flip**” the two red nodes so their parent becomes the red node
  + If inserting on **left** with the parent as a red node, we have double left-leaning nodes. **Rotate right** on its parent then **color flip**
  + Inserting on **left** when the parent is not a red node is perfectly okay!
* Check [these slides](https://docs.google.com/presentation/d/1FVENq6nVfWEHohE8j3oQC6uutxWOghacBfJixFV3KQU/edit#slide=id.g1ee126c17d_0_0) for visuals (*Note: Professor Hug uses red links instead of nodes)*

#### **Tree Traversals:**

* **Pre-Order** - traverse current node, then left child, then right child
* **In-Order** - traverse left child, then current node, then the right child (*sortedQueueToTree)*
* **Post-Order** - traverse left child, then right child, then the current node
* **Quick-n-Dirty Trick:** start from the root, draw a path heading *counterclockwise*...
  + **Pre-Order** - nodes traversed in order of when you reach **left** of the node
  + **In-Order** - nodes traversed in order of when you reach the **bottom** of the node
  + **Post-Order** - nodes traversed in order of when you reach the **right** of the node
* **Depth First Search (DFS) -** Use Stack (LIFO) as fringe, add root. While fringe is not empty, remove next from fringe, add its children then process removed node
* **Breadth First Search (BFS) -** Use Queue (FIFO) as fringe, same process as DFS

\* *Check the* [*Graphs*](#_wg8yjn7uavb8) *tab for more about DFS and BFS*

#### **Hashing:**

* Usually **External Chaining**, buckets are represented as an array (indexed in Θ(1)!), and collisions are handled by adding to the end of a Linked List in that bucket

All hashing follows **3 steps:**

1. Calculate object’s hashcode
2. Take hashcode mod number of buckets to figure out which bucket to insert into
3. Insert object to the end of the bucket’s Linked List

***Good Hashcode Properties:*** *Must return an integer*

* **Deterministic:** repeated calls to hashcode() return the same thing (no Random)
* **Uniform:** keys spread evenly across buckets (don’t use the first letter of a word as a hashcode!)
* **Quick:** Hashcode is relatively quick to compute, as close to constant as possible

**Other Properties:**

* If exceed a load factor (# objects / # buckets), resize by a **multiplicative** factor, and REHASH all the items; many objects may be in a different bucket
* When **.contains**([*item*]) is called, first hashcode is called on *item*, then bucket is calculated with hashcode % # buckets. Then for every item in the linked list, **.equals** is called until it reaches the end or finds the item.

***WARNINGS:***

* Mutating an object does **NOT** rehash it (can lead to mistakes in **.contains**)
* On exams, **DO NOT** assume a good hash function or amortized constant runtime!

#### **Heaps and Priority Queues:**

***Heap Properties:***

* **Complete:** Every level except last must be completely filled, nodes as far left as possible
* **Min-Heap:** Every node is less than or equal to child (Max heap - every node greater than or equal to child)
* 0th index nulled for easy child and parent indexing
* **Priority Queue** is an ABSTRACT DATA TYPE implemented with a heap (DATA STRUCTURE)!

***Index References of Node in Heap with 0th index null:***

* **Parent** **Index** = index / 2
* **Left Child Index** = index \* 2
* **Right Child Index** = index \* 2 + 1

***Methods:***

* **Insert** - insert to as bottom and as left as possible, *bubble up*
* **deleteMin** - remove the top node and replace with bottom right, *bubble down*
* **changePriority (Θ(log(n))**  - *bubble up* or *down*depending on if priority became higher or lower

***Exam Tips:***

* The only thing you know about a min-heap is that its children must be greater than or equal to it, and the **root node is the** **min value**. Thus, for a min-heap with unique values, any node that has **no children** **can be the largest value** and any node that doesn’t have **more than half of the total number of nodes as children or descendants** can be the median value

For a really cool use of Min-Heap, check out problem 9 of [this midterm ;)](https://tbp.berkeley.edu/exams/6137/download/)

#### **KD-Trees (Multidimensional Data):**



* Partition space using a BST that compares the same dimension every *k* levels (in the example above, *k* = 2, compares x-dimension at even depths and y-dim at odd depths)

***Nearest method (finds the nearest point/node to a given point):***

* ALWAYS check “good” side first (side closer to the given point, depends on dimension), check bad side only if it could contain a closer point than the current best distance from given point
* “Prune” by ignoring partitions that cannot contain node closer than current nearest point
* In the example to the right (using the kd-tree above), once we finish checking the left side of point **C**, it is not worth checking the right side of **C**, as no point can possibly be closer than the purple point, which is farther than 2.2 units away from the given point (visualized by the orange dotted circle around the point **(0, 7)**)
* [Hug’s nearest method animation](https://docs.google.com/presentation/d/1DNunK22t-4OU_9c-OBgKkMAdly9aZQkWuv_tBkDg1G4/edit#slide=id.g54b6045cf5_150_2806)

#### **Tries:**

* Absolutely beautiful creatures - highlighted nodes represent the end of a word
* Used in Autocomplete (Strings broken into chars)
* Good to review **add** and **keysWithPrefix** methods
* Random runtimes (Also check [runtime table](#_amsylmbkupet)):
  + *Inserting N* strings length *M* -> **Θ(NM)**
  + *Finding all keys (length L) with a prefix of another key* -> **Θ(NL)**
  + *Finding the longest key that is a prefix of another key* -> **Θ(NL)**

#### **Disjoint Sets:**

* Great for determining CONNECTIVITY (friends on facebook, generating maze, etc.)
* Useful in other algorithms such as Kruskal’s Algorithm in [MSTs](#_vbvmjb16tp1s)

***Methods:***

* **connect(x, y)** - connects two objects (can be vertices, integers, whatever)
* **isConnected(x, y)** - checks to see if x, y is connected, returns boolean

***Weighted Quick Union:***

* Objects in WQU are connected as a tree, always connect the ROOT of one set to the ROOT of another set.
* Information stored as an array, where the value of an index is the index’s parent (not root), and the ROOT is stored as a negative number corresponding to the size of each set
* To keep tree balanced, connect tree with less weight to the tree with more weight
  + Improves upon a naive QuickUnion structure that can result in spindly trees (connect 1-2, 2-3, 3-4, 4-5 … )

\*\*For other implementations of Disjoint Sets from QuickFind to WeightedQuickUnion with path compression, check these [slides](https://docs.google.com/presentation/d/1I9Jzt95GvxRqwGMzN7DSEIREKhMZ0qEUzPyWLl6CU5g/edit#slide=id.g4fd335df8a_0_8)

#### **Graphs:**

* All trees are graphs! But graphs can have **cycles**, can be **directed** (edges travel one-way) or **undirected** (edges can travel both ways),

***Traversals Revisited (BFS and DFS don’t consider Edge Weights!!):***

***DFS and BFS Runtime:*** *O(V+E)****, Space Complexity:*** *O(V) [for storing “visited” array]*

***Two Representation of Graphs:***

#### **Graph Algorithms (Shortest Paths with Edge Weights):**

***Dijkstra’s Algorithm* (**[***Demo***](https://docs.google.com/presentation/d/1_bw2z1ggUkquPdhl7gwdVBoTaoJmaZdpkV6MoAgxlJc/pub?start=false&loop=false&delayms=3000&slide=id.g771336078_0_180)**)*:***

* Dijkstra is just BFS if there are no edge weights or identical edge weights
* ***May Fail*** for negative edge weights! (won’t if only neg weights are from start node)
* **Exam Tip:** If multiple paths give same distance and you want to find the one with the fewest edges, add a tiny constant number to each edge of the graph to ensure path with the least amount of edges is returned
* **General Steps:**

1. Insert all vertices into priority queue initialized with priority infinity
2. Remove the vertex at top of queue, if a shorter distance is found from source to vertex **change the priority** of the vertex in the queue to the smaller number

*Assuming E > V, Total runtime is O(E log V)*

***A\* Algorithm - Single target in mind rather than the shortest path to all vertices (***[***Demo***](https://docs.google.com/presentation/d/177bRUTdCa60fjExdr9eO04NHm0MRfPtCzvEup1iMccM/edit#slide=id.g771336078_0_180)***):***

* Same as Dijkstra’s, store priority as distance from source + heuristic (estimated distance to goal)
* Unlike Dijkstra's, may not need to visit all vertices! Stop once the goal is visited
* Runtime depends heavily on the heuristic function (take 188 for some examples :) )

***Topological Sort - What if we have negative edge weights? (***[***Demo***](https://docs.google.com/presentation/d/1srzyUpmXrwSLHaAZDYVCpmPFFsq5q-sTwPwdCJkIQVI/edit#slide=id.g99668982c_1_693)***):***

* Perform a DFS postorder on every vertex that doesn’t have a directed edge going into it (indegree 0), *reverse* the list returned by DFS postorder to get topological order
* Finds shortest path tree on a **Directed Acyclic Graph** even with negative edges! (visits vertices in topological order, relaxing edges as we go (*O(V + E)*)
  + To find the longest path tree, simply negate all edge weights, run the topological shortest paths algorithm on DAG)

\*Unlike a graph we may run Dijkstra's on, multiple vertices may have an indegree 0 (multiple “source” nodes)! Thus, these source nodes will not have an into arrow when diagrammed.

#### **Minimum Spanning Trees:**

* Tree (no cycles, connected) that includes all vertices in a graph with minimum weight, \*only works on ***undirected graphs***
* **Cut Property:** assign graph’s nodes to 2 different sets (**cut**); given any cut, minimum weight **crossing edge** (edge from one set to the other) is in the MST
* **Cycle Property:** The largest edge in a cycle will not be in the MST.
* If edges are *NOT UNIQUE*, there is a chance the MST is *NOT UNIQUE*
* Can’t use Dijkstra's method (not exactly, anyways), no notion of a “source node”
* Results with *V - 1 edges* (given that trees have no cycles and must be connected, why can’t the number of edges be anything else?)

**Exam Tip** - To see if adding an edge gives better MST, consider the largest edge of the MST coming out of that vertex

***Prim’s Algorithm (***[***Demo***](https://docs.google.com/presentation/d/1GPizbySYMsUhnXSXKvbqV4UhPCvrt750MiqPPgU-eCY/edit#slide=id.g9a60b2f52_0_0)***):***

* Very similar to Dijkstra's with one caveat - consider distance from ***TREE***, not SOURCE
* Runtime same as Dijkstra’s, see table on the previous page
* **General Steps:**

1. Insert all vertices into priority queue initialized with priority infinity
2. Remove the vertex at top of the queue (thereby “setting” the edge corresponding to the priority of the vertex as an edge in the MST under construction), if a shorter distance is found from tree to vertex **change the priority (weight of edge)** of the vertex in the queue to the smaller number

***Kruskal’s Algorithm (***[***Demo***](https://docs.google.com/presentation/d/1KpNiR7aLIEG9sm7HgX29nvf3yLD8_vdQEPa0ktQfuYc/edit#slide=id.g9a60b2f52_0_0)***):***

* Consider edges from smallest weight to largest, add the smallest edge to the MST unless it creates a cycle, and stop at *V - 1 edges*
* **General Steps (Implementing in Code):**

1. Insert the edges in a PQ, with priority equal to edge weight
2. Remove smallest weight edge that doesn’t create a cycle, add to MST

* How do we check for a cycle? [**Disjoint Sets!**](#_26qxc3m9yt0g)
  + For every edge we add to MST, we connect the vertices with WQU.
  + Before we add the edge to MST, first check if two vertices are already connected. If so, adding the edge will create a cycle



* The asterisk \* implies we are using **WeightedQuickUnion with path compression**, which in addition to always connecting the smaller set to the larger set, whenever we call **isConnected** or **connected** (the method will check the ROOT of the set), once the function finds the root, every vertex in the tree we visited to find the root will be connected directly to the root rather than its parent - this shortens the height of tree

#### **Rapid Fire Comparison Sorts:**

* Can’t do better than Θ(NlogN) worst case - QuickSort, MergeSort, HeapSort
* ***Selection Sort (***[***Demo***](https://docs.google.com/presentation/d/1p6g3r9BpwTARjUylA0V0yspP2temzHNJEJjCG41I4r0/edit#slide=id.g463de7561_042)***)*:** Find smallest item, move to front. Selection sort rest
* ***Heapsort (***[***Demo***](https://docs.google.com/presentation/d/1SzcQC48OB9agStD0dFRgccU-tyjD6m3esrSC-GLxmNc/edit)***)*:**

1. Starting from the bottom of the heap (end of array), bubble down on each element to create a valid MAX heap (why won’t min heap work? Try it!).
2. Swap the largest item with the last item of the heap, the back of the array should be increasingly sorted (don’t forget to bubble down the swapped item!)

* ***Merge Sort (***[***Demo***](https://docs.google.com/presentation/d/1h-gS13kKWSKd_5gt2FPXLYigFY4jf5rBkNFl3qZzRRw/edit)***)*:** Split list into two halves, merge sort each half. Then combine sorted halves together - compare the smallest item of each half not yet put in the merged array in order from smallest to largest. (if one half is done, can just put rest of other half)
* ***Insertion Sort (***[***Demo***](https://docs.google.com/presentation/d/10b9aRqpGJu8pUk8OpfqUIEEm8ou-zmmC7b_BE5wgNg0/edit#slide=id.g463de7561_042)***)*:** For each item from the beginning, move it toward the front of the list until it reaches an element it is larger than (“inserting” it into its correct sorted position of every element before it). Front of list should be increasingly sorted
  + Amazing for lists that are almost sorted, few inversions
* ***3-Way Partitioning Quicksort (***[***Demo***](https://docs.google.com/presentation/d/1QjAs-zx1i0_XWlLqsKtexb-iueao9jNLkN-gW9QxAD0/edit#slide=id.g463de7561_042)***)*:** Generate a random pivot, traverse the list and add every item smaller than it to its left and every item larger to its right (break items equal to it arbitrarily, either add ALL of them to its left or ALL to right). Quicksort left side and quicksort right side, then combine as follows: sorted left side, SINGLE pivot (should be only one item), sorted right side
* ***Hoare Partitioning Quicksort (***[***Demo***](https://docs.google.com/presentation/d/1DOnWS59PJOa-LaBfttPRseIpwLGefZkn450TMSSUiQY/pub?start=false&loop=false&delayms=3000)***):*** Generate left and right pointers and set leftmost element as pivot, where the left pointer stops at larger or equal items vs. pivot and right pointer stops at smaller or equal items vs. pivot. First move the left pointer to right until stop (could be the very first element!), then move the right pointer to the left until stopped. When both pointers have stopped, swap the elements at each pointer and increment the left pointer right once and right pointer left once. Once the left pointer has been incremented to be to the right of the “right” pointer, swap the original “right” pointer with the pivot (leftmost element) to complete the sorting.

#### **Rapid Fire Counting Sorts:**

* Cheeky - sacrifices efficient space complexity for runtime
* Alphabet can be any measure from suite of cards to the population of a country
  + For “strings” of alphabet keys (usually multiple digit numbers), use Radix Sort
* ***Counting Sort (***[***Demo***](https://docs.google.com/presentation/d/1vmVKHRSwb5WN1rHvktplbPGecHChxOwWa7ovRuiLzbA/edit#slide=id.g582f6c5a07_0_0)***)*:**

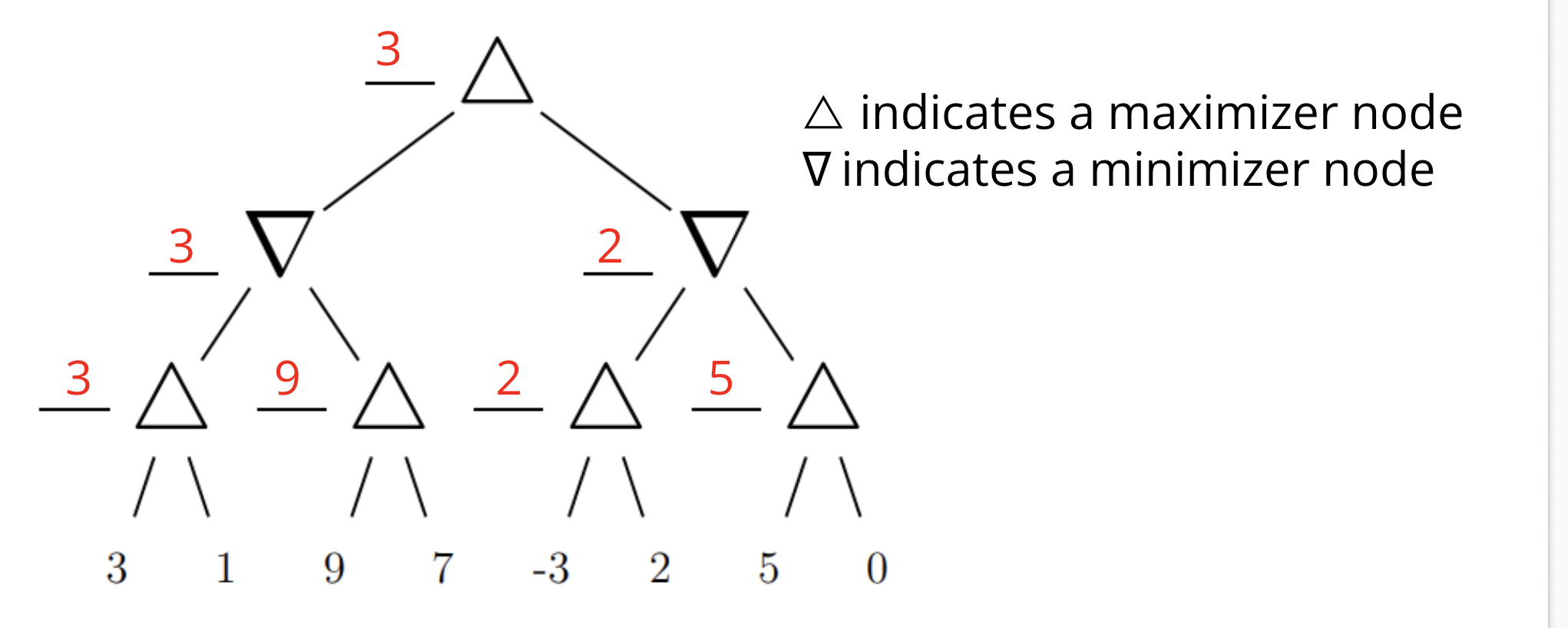
1. For each item in your “alphabet” keys, count the number of items that correspond with each key
2. Calculate starting points by adding number of items in the key - this will be the next key’s starting point (think of it as allocating # spaces for a key with # items)
3. Create a new array with the same size as the original, then traverse each item of original and add to its allocated space (based on starting point) in the new array

* ***Least Significant Digit Radix Sort (***[***Demo***](https://docs.google.com/presentation/d/1YBYV2ymAFiHHbSNcC1DRRi2C56blMxEsTZ7G_QE3q_8/edit#slide=id.g465fbc6ea_173)***)*:** Sort each digit independently from rightmost digit (ones place) to the leftmost digit.
  + For varying length keys (499 vs. 38), treat the empty spaces as a 0 for integers or “empty” space less than any other character for other alphabets (499 vs. 038)
* ***Most Significant Digit Radix Sort (***[***Demo***](https://docs.google.com/presentation/d/1YBYV2ymAFiHHbSNcC1DRRi2C56blMxEsTZ7G_QE3q_8/edit#slide=id.g465fbc6ea_129)***)*:** Sort each digit independently from leftmost digit (most significant) to the rightmost digit.
  + To ensure array doesn’t become unsorted when going into less significant digits, group all the items with same MSD together, sort them separately, insert everything in a newly created array when each subproblem is size 1

#### **Student Notes:**

***Choosing the best sorting algorithm - Li Zixi (Summer 2019):***

* If the elements have a fixed radix (i.e. length/digits), radix sort is better.
* For arrays with ≤ O(N) inversions, Insertion Sort can complete in linear time. (It must take at least Θ(N) time to inspect the whole array. For example, Insertion sort will be Θ(N) for an array with O(N0.5) inversions. For arrays with ≥ O(N) inversions, O(NlogN) algorithms would be better. (since O(NlogN) would be better than even O(N1.01))
* Insertion sort is best when the list is short.
* Mergesort performs better for linked lists. Quicksort performs best for arrays.

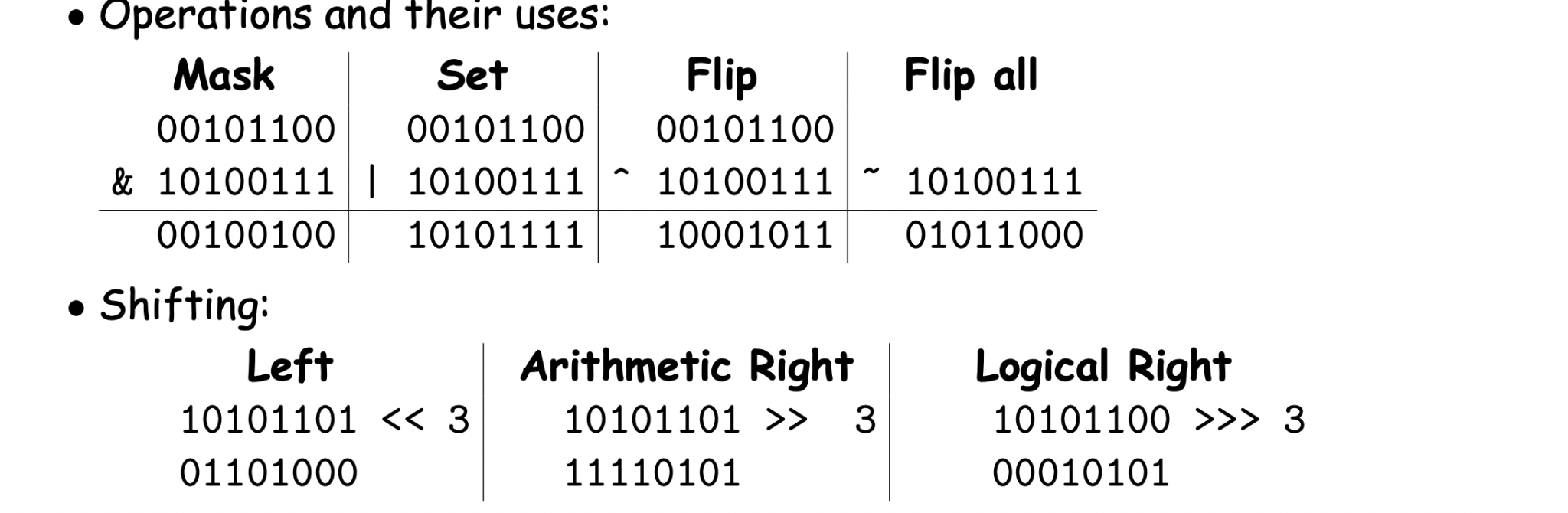
***Game Trees (minimax and alpha-beta pruning) - Dhaval Patel (Fall 2021):***

* Calculate node values in a depth-first search traversal
* Value of game tree is the root value
* “min” nodes take the minimum of children
* “max” nodes take the maximum of children

**Alpha-Beta Pruning**

* Maintain two new values while running the depth-first minimax traversal
* The root node gets α=-∞ and β=∞.
* When you explore a child node, the child inherits α and β values from its parent.
* Maximizer nodes try to maximize value of children, and also the value of α
  + When a maximizer node sees a child with value v greater than β, it can stop exploring its remaining children
  + Otherwise, when a maximizer node sees a child with value v greater than α, update α to v
* Minimizer nodes try to minimize value of children, and also the value of β
  + When a minimizer node sees child with value v less than α, it can stop exploring its remaining children
  + When a minimizer node sees child with value v less than β, update β to v
* [Minimax GeeksForGeeks article](https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/)

***Bitwise Operations - Dhaval Patel (Fall 2021):***

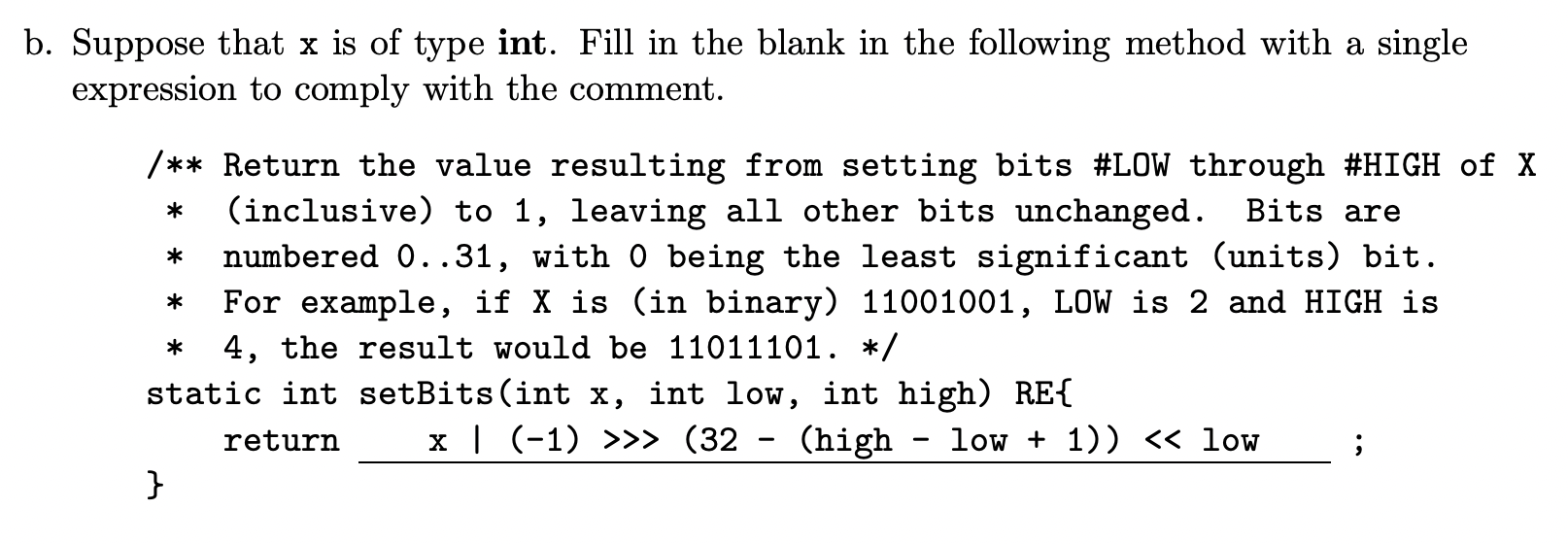
* Shift left adds a number of 0s to the right side shifting everything to the left!
* Shift logical right (>>>) adds a number of 0s to the left side shifting everything to the right!
* Shift arithmetic right (>>) adds a number of 1s to the left side shifting everything to the right! 

^ = flip

~ = flip all

& = mask

| = set



* Note how the return statement uses shifts, variables, and numbers to alter the int