Summaries of Data Structures

Step 1

Array List (Dynamic Array)

Summary Description

- Array Lists (or Dynamic Arrays) are simply arrays wrapped in a container that handles automatically resizing the array when it becomes full
- As a result, if we know which index of the Array List we wish to access, we can do so in O(1) time because of the array's random access property
- Array Lists are typically implemented such that the elements are contiguous in the array (i.e., there are no empty slots in the array)

Time/Space Complexities of an Unsorted Array List

Worst-Case Time Complexity (Unsorted Array List)

- **Find: O(*n*)** — We need to iterate over all *n* elements to find the query
- **Insert: O(*n*)** — If we insert at the front of the Array List, we need to move over each of the *n* elements
- **Remove: O(*n*)** — If we remove from the front of the Array List, we need to move over each of the *n* elements

Average-Case Time Complexity (Unsorted Array List)

- **Find: O(*n*)** — The average number of checks is $\frac{1+2+...+n}{n} = \frac{\sum_{i=1}^{n} i}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2}$
- **Insert: O(*n*)** — The average number of "element moves" is $\frac{n+(n-1)+...+1+0}{n} = \frac{\sum_{i=0}^{n} i}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2}$
- **Remove: O(*n*)** — The average number of "element moves" is $\frac{(n-1)+(n-2)+...+1+0}{n} = \frac{\sum_{i=0}^{n-1} i}{n} = \frac{n(n-1)}{2n} = \frac{n}{2}$

Best-Case Time Complexity (Unsorted Array List)

- **Find: O(1)** — The query is the first element in the Array List
- **Insert: O(1)** — If we insert at the end of the Array List, we don't need to move any elements
- **Remove: O(1)** — If we remove from the end of the Array List, we don't need to move any elements

Space Complexity (Unsorted Array List)

- **O(*n*)** — The two extremes are *just before* resizing (completely full, so the array is of size *n*) or *just after* resizing (array is half full, so the array is of size 2*n*)

Time/Space Complexities of a Sorted Array List

Worst-Case Time Complexity (Sorted Array List)

- **Find: O(log *n*)** — We can perform Binary Search to find an element
- **Insert: O(*n*)** — If we insert at the front of the Array List, we need to move over each of the *n* elements

- **Remove: O(*n*)** — If we remove from the front of the Array List, we need to move over each of the *n* elements

Average-Case Time Complexity (Sorted Array List)

- **Find: O(log *n*)** — The derivation is too complex for a summary slide, but it's the average case of Binary Search
- **Insert: O(*n*)** — The average number of "element moves" is $\frac{n+(n-1)+...+1+0}{n} = \frac{\sum_{i=0}^{n} i}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2}$
- **Remove: O(*n*)** — The average number of "element moves" is $\frac{(n-1)+(n-2)+...+1+0}{n} = \frac{\sum_{i=0}^{n-1} i}{n} = \frac{n(n-1)}{2n} = \frac{n}{2}$

Best-Case Time Complexity (Sorted Array List)

- **Find: O(1)** — The query is the first element in the Array List we check via Binary Search
- **Insert: O(1)** — If we insert at the end of the Array List, we don't need to move any elements
- **Remove: O(1)** — If we remove from the end of the Array List, we don't need to move any elements

Space Complexity (Sorted Array List)

- **O(*n*)** — The two extremes are *just before* resizing (completely full, so the array is of size *n*) or *just after* resizing (array is half full, so the array is of size 2*n*)

Step 2

Linked List

Summary Description

- Linked Lists are composed of nodes connected to one another via pointers
- We typically only keep global access to two pointers: a *head* pointer (which points to the first node) and a *tail* pointer (which points to the last node)
- In a Singly-Linked List, each node maintains one pointer: a *forward* pointer that points to the next node in the list
- In a Doubly-Linked List, each node maintains two pointers: a *forward* pointer that points to the next node in the list, and a *previous* pointer that points to the previous node in the list
- Unlike an Array List, we do *not* have direct access to any nodes other than *head* and *tail*, meaning we need to iterate node-by-node one-by-one to access inner nodes
- Once we know where in the Linked List we want to insert or remove, the actual insertion/removal is O(1) because we just change pointers around
- As a result, in each of the cases (worst, average, and best), the time complexity of insert and remove are the same as the time complexity of find, because you call find and then perform a O(1) operation to perform the insertion/removal

Time/Space Complexities of a Linked List

Worst-Case Time Complexity

- **Find: O(*n*)** — In both Singly- and Doubly-Linked Lists, if our query is the middle element, we need to iterate over $\frac{n}{2}$ nodes
- **Insert: O(*n*)** — Perform find, which is O(*n*) in the worst case, and then perform O(1) pointer changes
- **Remove: O(*n*)** — Perform find, which is O(*n*) in the worst case, and then perform O(1) pointer changes

Average-Case Time Complexity

- **Find: O(*n*)** — The average number of checks in a Singly-Linked List is $\frac{1+2+...+n}{n} = \frac{\sum_{i=1}^{n} i}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2}$, and in a Doubly-Linked List, if we know the index we want to access, it is $2\frac{\frac{n}{2}+1}{2} = \frac{n}{2} + 1$
- **Insert: O(*n*)** — Perform find, which is O(*n*) in the average case, and then perform O(1) pointer changes
- **Remove: O(*n*)** — Perform find, which is O(*n*) in the average case, and then perform O(1) pointer changes

Best-Case Time Complexity

- **Find: O(1)** — If our query is the first element we check (or if we specify *head* or *tail* as our query)
- **Insert: O(1)** — Perform find, which is O(1) in the best case, and then perform O(1) pointer changes
- **Remove: O(1)** — Perform find, which is O(1) in the best case, and then perform O(1) pointer changes

Space Complexity

- **O(*n*)** — Each node contains either 1 or 2 pointers (for Singly- or Doubly-Linked, respectively) and the data, so O(1) space for each node, and we have exactly *n* nodes

Step 3

Skip List

Summary Description

- Skip Lists are like Linked Lists, except every node has multiple *layers*, where each layer is a forward pointer
- We denote the number of layers a given node has as the node's *height*
- We typically choose a maximum height, *h*, that any node in our Skip List can have, where *h* << *n* (the total number of nodes)
- We are able to "skip" over multiple nodes in our searches (unlike in Linked Lists, where we had to traverse through the nodes one-by-one), which allows us to mimic Binary Search when searching for an element in the list
- To determine the height of a new node, we repeatedly flip a weighted coin that has probability *p* to land on heads, and we keep adding layers to the new node until we encounter our first tails
- Just like with a Linked List, to insert or remove an element, we first run the regular find algorithm and then perform a single O(*h*) operation to fix pointers

Time/Space Complexities of a Skip List

Worst-Case Time Complexity

- **Find: O(*n*)** — If all of our nodes have the same height (low probability, but possible), we just have a regular Linked List
- **Insert: O(*n*)** — Perform find, which is O(*n*) in the worst case, and then perform a single O(*h*) pointer fix (*h* << *n*)
- **Remove: O(*n*)** — Perform find, which is O(*n*) in the worst case, and then perform a single O(*h*) pointer fix (*h* << *n*)

Average-Case Time Complexity

- **Find: O(log *n*)** — The formal proof is too complex for a summary slide
- **Insert: O(log *n*)** — Perform find, which is O(log *n*) in the average case, and then perform a single O(*h*) pointer fix (*h* << *n*)
- **Remove: O(log *n*)** — Perform find, which is O(log *n*) in the average case, and then perform a single O(*h*) pointer fix (*h* << *n*)

Best-Case Time Complexity

- **Find: O(1)** — If our query is the first element we check
- **Insert: O(*h*)** — Perform find, which is O(1) in the best case, and then perform a single O(*h*) pointer fix (*h* << *n*)
- **Remove: O(*h*)** — Perform find, which is O(1) in the best case, and then perform a single O(*h*) pointer fix (*h* << *n*)

Space Complexity

- **Worst-Case: O(*n* log *n*)** — The formal proof is too complex for a summary slide
- **Average-Case: O(log *n*)** — The formal proof is too complex for a summary slide

Step 4

Heap

Summary Description

- A Heap is a complete binary tree that satisfies the *Heap Property*
- *Heap Property*: For all nodes *A* and *B*, if node *A* is the parent of node *B*, then node *A* has *higher priority* (or equal priority) than node *B*
- A *min*-heap is a heap in which every node is *smaller* than (or equal to) all of its children (or has no children)
- A *max*-heap is a heap where every node is *larger* than (or equal to) all of its children (or has no children)
- It is common to implement a Heap as an Array List because all of the data will be localized in memory (faster in practice)
- If a Heap is implemented as an Array List, the root is stored in index 0, the next node (in a level-order traversal) is at index 1, then at index 2, etc.
- If a Heap is implemented as an Array List, for a node *u* stored at index *i*, *u*'s parent is stored at index $\lfloor \frac{i-1}{2} \rfloor$, *u*'s left child is stored at index $2i + 1$, and *u*'s right child is stored at index $2i + 2$

Time/Space Complexities of a Linked List

Worst-Case Time Complexity

- **Peek: O(1)** — We just return the root, which is a O(1) operation
- **Insert: O(log *n*)** — If our new element has to bubble up the entire tree, which has a height of O(log *n*)
- **Pop: O(log *n*)** — If our new root has to trickle down the entire tree, which has a height of O(log *n*)

Average-Case Time Complexity

- **Peek: O(1)** — We just return the root, which is a O(1) operation
- **Insert: O(log *n*)** — The formal proof is too complex for a summary slide
- **Pop: O(log *n*)** — The formal proof is too complex for a summary slide

Best-Case Time Complexity

- **Peek: O(1)** — We just return the root, which is a O(1) operation
- **Insert: O(1)** — If our new element doesn't have to bubble up at all
- **Pop: O(1)** — If our new root doesn't have to trickle down at all

Space Complexity

- **O(*n*)** — We typically implement a Heap as an Array List

Binary Search Trees

Summary Description

- A Binary Search Tree is a binary tree in which any given node is larger than all nodes in its left subtree and smaller than all nodes in its right subtree
- A Randomized Search Tree is a Binary Search Tree + Heap (a "Treap") in which each node has a *key* and a *priority*, and the tree maintains the *Binary Search Tree Property* with respect to *keys* and the *Heap Property* with respect to *priorities*
- An AVL Tree is a self-balancing Binary Search Tree in which, for all nodes in the tree, the *heights* of the two child subtrees of the node differ by at most one
- A Red-Black Tree is a self-balancing Binary Search Tree in which all nodes must be "colored" either red or black, the root of the tree must be black, red nodes can only have black children, and every path from any node *u* to a null reference must contain the same number of black nodes
- AVL Trees are stricter than Red-Black Trees in terms of balance, so AVL Trees are typically faster for find operations
- Red-Black Trees only do one pass down the tree for inserting elements, whereas AVL trees need one pass down the tree and one pass back up, so Red-Black Trees are typically faster for insert operations

Time/Space Complexities of a Regular Binary Search Tree

Worst-Case Time Complexity (Regular Binary Search Tree)

- **Find: O(*n*)** — If we insert the elements in ascending/descending order, we get a Linked List
- **Insert: O(*n*)** — If we insert the elements in ascending/descending order, we get a Linked List
- **Remove: O(*n*)** — If we insert the elements in ascending/descending order, we get a Linked List

Average-Case Time Complexity (Regular Binary Search Tree)

- **Find: O(log *n*)** — The formal proof is too complex for a summary slide
- **Insert: O(log *n*)** — Effectively the same algorithm as find, with the actual insertion being a O(1) pointer rearrangement
- **Remove: O(log *n*)** — The formal proof is too complex for a summary slide

Best-Case Time Complexity (Regular Binary Search Tree)

- **Find: O(1)** — If the query is the root
- **Insert: O(1)** — If the root only has one child and the node we are inserting becomes the root's other child
- **Remove: O(1)** — If we are removing the root and the root only has one child

Space Complexity (Regular Binary Search Tree)

- **O(*n*)** — Each node contains either 3 pointers (parent, left child, and right child) and the data, so O(1) space for each node, and we have exactly *n* nodes

Time/Space Complexities of a Randomized Search Tree

Worst-Case Time Complexity (Randomized Search Tree)

- **Find: O(*n*)** — If the elements are in ascending/descending order in terms of *both* keys *and* priorities, we get a Linked List

- **Insert: O(*n*)** — If the elements are in ascending/descending order in terms of *both* keys *and* priorities, we get a Linked List
- **Remove: O(*n*)** — If the elements are in ascending/descending order in terms of *both* keys *and* priorities, we get a Linked List

Average-Case Time Complexity (Randomized Search Tree)

- **Find: O(log *n*)** — The formal proof is too complex for a summary slide
- **Insert: O(log *n*)** — The formal proof is too complex for a summary slide
- **Remove: O(log *n*)** — The formal proof is too complex for a summary slide

Best-Case Time Complexity (Randomized Search Tree)

- **Find: O(1)** — If the query is the root
- **Insert: O(1)** — If the root only has one child and the node we are inserting becomes the root's other child
- **Remove: O(1)** — If we are removing the root and the root only has one child

Space Complexity (Randomized Search Tree)

- **O(*n*)** — Each node contains either 3 pointers (parent, left child, and right child), the key, and the priority, so O(1) space for each node, and we have exactly *n* nodes

Time/Space Complexities of an AVL Tree

Worst-Case Time Complexity (AVL Tree)

- **Find: O(log *n*)** — AVL Trees must be balanced by definition
- **Insert: O(log *n*)** — AVL Trees must be balanced by definition, and the rebalancing is O(log *n*)
- **Remove: O(log *n*)** — AVL Trees must be balanced by definition, and the rebalancing is O(log *n*)

Average-Case Time Complexity (AVL Tree)

- **Find: O(log *n*)** — The formal proof is too complex for a summary slide
- **Insert: O(log *n*)** — The formal proof is too complex for a summary slide
- **Remove: O(log *n*)** — The formal proof is too complex for a summary slide

Best-Case Time Complexity (AVL Tree)

- **Find: O(1)** — If the query is the root
- **Insert: O(log *n*)** — AVL Trees must be balanced, so we have to go down the entire O(log *n*) height of the tree
- **Remove: O(log *n*)** — The formal proof is too complex for a summary slide

Space Complexity (AVL Tree)

- **O(*n*)** — Each node contains either 3 pointers (parent, left child, and right child) and the data, so O(1) space for each node, and we have exactly *n* nodes

Time/Space Complexities of a Red-Black Tree

Worst-Case Time Complexity (Red-Black Tree)

- **Find: O(log *n*)** — Red-Black Trees must be balanced (formal proof is too complex for a summary slide)
- **Insert: O(log *n*)** — Red-Black Trees must be balanced (formal proof is too complex for a summary slide), so we have to go down the entire O(log *n*) height of the tree, and the rebalancing occurs with O(1) cost during the insertion
- **Remove: O(log *n*)** — Red-Black Trees must be balanced (formal proof is too complex for a summary slide), and the rebalancing occurs with O(1) cost during the removal

Average-Case Time Complexity (Red-Black Tree)

- **Find: O(log *n*)** — The formal proof is too complex for a summary slide
- **Insert: O(log *n*)** — The formal proof is too complex for a summary slide
- **Remove: O(log *n*)** — The formal proof is too complex for a summary slide

Best-Case Time Complexity (Red-Black Tree)

- **Find: O(1)** — If the query is the root
- **Insert: O(log *n*)** — Red-Black Trees must be balanced (formal proof is too complex for a summary slide), so we have to go down the entire O(log *n*) height of the tree, and the rebalancing occurs with O(1) cost during the insertion
- **Remove: O(log *n*)** — The formal proof is too complex for a summary slide

Space Complexity (Red-Black Tree)

- **O(*n*)** — Each node contains either 3 pointers (parent, left child, and right child) and the data, so O(1) space for each node, and we have exactly *n* nodes

Step 6

B-Tree and B+ Tree

Summary Description

- A B-Tree is a self-balancing tree in which internal nodes must have at least *b* and at most 2*b* children (for some predefined *b*), all leaves must be on the same level of the tree, the elements within a given node must be in ascending order, and child pointers appear *between* elements and on the edges of the node
- For two adjacent elements *i* and *j* in a B-Tree (where *i* < *j*, so *i* is to the left of *j*), all elements down the subtree to the left of *i* must be smaller than *i*, all elements down the subtree between *i* and *j* must be greater than *i* and less than *j*, and all elements down the subtree to the right of *j* must be greater than *j*
- A B+ Tree can be viewed as a variant of the B-tree in which each internal node contains only keys and the leaves contain the actual data records
- In a B+ Tree, *M* denotes the maximum number of children any nodes can have, and *L* denotes the maximum number of data records
- In a B+ Tree, every node has at most *M* children, every internal node except the root has at least ⌈M/2⌉ children, the root has at least 2 children (if it's not a leaf), an internal node with *k* children must contain *k*–1 elements, all leaves must be on the same level of the tree, internal nodes only contain "search keys" (no data records), and the smallest data record between search keys *i* and *j* (where *i* < *j*) must equal *i*

Time/Space Complexities of a B-Tree

Worst-Case Time Complexity (B-Tree)

- **Find: O(log *n*)** — B-Trees must be balanced by definition
- **Insert: O(b*log *n*)** — B-Trees must be balanced by definition, and the re-balancing is O(b*log *n*) because worst case scenario, we need to shuffle around O(*b)* keys in a node at each level to keep the sorted order property
- **Remove: O(b*log *n*)** — B-Trees must be balanced by definition, and the re-balancing is O(b*log *n*) because worst case scenario, we need to shuffle around O(*b)* keys in a node at each level to keep the sorted order property

Average-Case Time Complexity (B-Tree)

- **Find: O(log *n*)** — The formal proof is too complex for a summary slide
- **Insert: O(log *n*)** — The formal proof is too complex for a summary slide
- **Remove: O(log *n*)** — The formal proof is too complex for a summary slide

Best-Case Time Complexity (B-Tree)

- **Find: O(1)** —If the query is the middle element of the root (so Binary Search finds it first)
- **Insert: O(log n)** — All insertions happen at the leaves. In the best case, no re-sorting or re-balancing is needed
- **Remove: O(log n)** — Removing from any location will require the re-adjustment of child pointers *or* traversing the entire height of the tree. In the best case, no re-sorting or re-balancing is needed

Space Complexity (B-Tree)

- **O(*n*)** — Every node must be at least $\frac{1}{3} \leq \frac{b-1}{2b-1} < \frac{1}{2}$ full, so in the worst case, every node is $\frac{1}{3}$ full, meaning we allocated O(3*n*) space, which is O(*n*)

Time/Space Complexities of a B+ Tree

Worst-Case Time Complexity (B+ Tree)

- **Find: O(log *n*)** — B+ Trees must be balanced by definition
- **Insert: O(*M**log *n* + *L*)** — B+ Trees must be balanced by definition and the re-balancing is O(M*log *n* + *L*) because worst case scenario, we need to shuffle around *O(M)* keys in an internal node, O(*L)* data records in a leaf node, for O(log *n*) levels of the tree in order to keep the sorted order property
- **Remove: O(*M**log *n* + *L*)** — B+ Trees must be balanced by definition and the re-balancing is O(M*log *n* + *L*) because worst case scenario, we need to shuffle around *O(M)* keys in an internal node, O(*L)* data records in a leaf node, for O(log *n*) levels of the tree in order to keep the sorted order property

Average-Case Time Complexity (B+ Tree)

- **Find: O(log *n*)** —The formal proof is too complex for a summary slide
- **Insert: O(log *n*)** — The formal proof is too complex for a summary slide
- **Remove: O(log *n*)** —The formal proof is too complex for a summary slide

Best-Case Time Complexity (B+ Tree)

- **Find: O(log n)** — All data records are stored at the leaves
- **Insert: O(log n)** — All insertions happen at the leaves. In the best case, no re-sorting or re-balancing is needed
- **Remove: O(log n)** — Removing from any location will require the re-adjustment of child pointers *or* traversing the entire height of the tree. In the best case, no re-sorting or re-balancing is needed

Space Complexity (B+ Tree)

- **O(*n*)** — Every node must be at least $\frac{\frac{M}{2}L}{ML} = \frac{1}{2}(\frac{ML}{ML}) = \frac{1}{2}$ full, so in the worst case, every node is $\frac{1}{2}$ full, meaning we allocated O(2*n*) space, which is O(*n*)

Step 7

Hash Table and Hash Map

Summary Description

- A Hash Table is an array that, given a key *key*, computes a hash value from *key* (using a hash function) and then uses the hash value to compute an index at which to store *key*
- A Hash Map is the exact same thing as a Hash Table, except instead of storing just *keys*, we store (*key*, *value*) pairs
- The capacity of a Hash Table should be prime (to help reduce collisions)
- When discussing the time complexities of Hash Tables/Maps, we typically ignore the time complexity of the hash function
- The load factor of a Hash Table, $\alpha = \frac{N}{M}$ (*N* = size of Hash Table and *M* = capacity of Hash Table) should remain below ~0.75 to keep the Hash Table fast (a smaller load factor means better performance, but it also means more wasted space)
- For a hash function *h* to be valid, given two equal keys *k* and *l*, *h*(*k*) must equal *h*(*l*)
- For a hash function *h* to be good, given two unequal keys *k* and *l*, *h*(*k*) should ideally (but not necessarily) equal *h*(*l*)
- A good hash function for a collection that stores *k* items (e.g. a string storing *k* characters, or a list storing *k* objects, etc.) should perform some non-commutative arithmetic that utilizes each of the *k* elements
- In Linear Probing (a form of Open Addressing), collisions are resolved by simply shifting over to the next available slot
- In Double Hashing (a form of Open Addressing), collisions are resolved in a way similar to Linear Probing, except instead of only shifting over one slot at a time, the Hash Table has a second hash function that it uses to determine the "skip" for the probe
- In Random Hashing (a form of Open Addressing), for a given key *key*, a pseudorandom number generator is created seeded with *key*, and the possible indices are given by the sequence of numbers returned by the pseudorandom number generator
- In Separate Chaining (a form of Closed Addressing), each slot of the Hash Table is actually a data structure itself (typically a Linked List), and when a key hashes to a given index in the Hash Table, simply insert it into the data structure at that index
- In Cuckoo Hashing (a form of Open Addressing), the Hash Table has to hash functions, and in the case of a collision, the new key pushes the old key out of its slot, and the old key uses the other hash function to find a new slot

Time/Space Complexities of a Hash Table/Map with Linear Probing

Worst-Case Time Complexity (Linear Probing)

- **Find: O(*n*)** — If all the keys mapped to the same index, we would need to probe over all *n* elements
- **Insert: O(*n*)** — If all the keys mapped to the same index, we would need to probe over all *n* elements
- **Remove: O(*n*)** — If all the keys mapped to the same index, we would need to probe over all *n* elements

Average-Case Time Complexity (Linear Probing)

- **Find: O(1)** — The formal proof is too complex for a summary slide
- **Insert: O(1)** — The formal proof is too complex for a summary slide
- **Remove: O(1)** — The formal proof is too complex for a summary slide

Best-Case Time Complexity (Linear Probing)

- **Find: O(1)** — No collisions
- **Insert: O(1)** — No collisions
- **Remove: O(1)** — No collisions

Space Complexity (Linear Probing)

- **O(*n*)** — Hash Tables typically have a capacity that is at most some constant multiplied by *n* (the constant is predetermined)

Time/Space Complexities of a Hash Table/Map with Double Hashing

Worst-Case Time Complexity (Double Hashing)

- **Find: O(*n*)** — If we are extremely unlucky, we may have to probe over all *n* elements
- **Insert: O(*n*)** — If we are extremely unlucky, we may have to probe over all *n* elements
- **Remove: O(*n*)** — If we are extremely unlucky, we may have to probe over all *n* elements

Average-Case Time Complexity (Double Hashing)

- **Find: O(1)** — The formal proof is too complex for a summary slide
- **Insert: O(1)** — The formal proof is too complex for a summary slide
- **Remove: O(1)** — The formal proof is too complex for a summary slide

Best-Case Time Complexity (Double Hashing)

- **Find: O(1)** — No collisions
- **Insert: O(1)** — No collisions
- **Remove: O(1)** — No collisions

Space Complexity (Double Hashing)

- **O(*n*)** — Hash Tables typically have a capacity that is at most some constant multiplied by *n* (the constant is predetermined)

Time/Space Complexities of a Hash Table/Map with Random Hashing

Worst-Case Time Complexity (Random Hashing)

- **Find: O(*n*)** — If each number generated by our generator mapped to an occupied slot, we would need to generate *n* numbers
- **Insert: O(*n*)** — If each number generated by our generator mapped to an occupied slot, we would need to generate *n* numbers
- **Remove: O(*n*)** — If each number generated by our generator mapped to an occupied slot, we would need to generate *n* numbers

Average-Case Time Complexity (Random Hashing)

- **Find: O(1)** — The formal proof is too complex for a summary slide (ignoring the time complexity of the pseudorandom number generator)
- **Insert: O(1)** — The formal proof is too complex for a summary slide (ignoring the time complexity of the pseudorandom number generator)
- **Remove: O(1)** — The formal proof is too complex for a summary slide (ignoring the time complexity of the pseudorandom number generator)

Best-Case Time Complexity (Random Hashing)

- **Find: O(1)** — No collisions (ignoring the time complexity of the pseudorandom number generator)
- **Insert: O(1)** — No collisions (ignoring the time complexity of the pseudorandom number generator)
- **Remove: O(1)** — No collisions (ignoring the time complexity of the pseudorandom number generator)

Space Complexity (Random Hashing)

- **O(*n*)** — Hash Tables typically have a capacity that is at most some constant multiplied by *n* (the constant is predetermined)

Time/Space Complexities of a Hash Table/Map with Separate Chaining

Worst-Case Time Complexity (Separate Chaining)

- **Find: O(*n*)** — If all the keys mapped to the same index (assuming Linked List)
- **Insert: O(*n*)** — If all the keys mapped to the same index (assuming Linked List)
- **Remove: O(*n*)** — If all the keys mapped to the same index (assuming Linked List)

Average-Case Time Complexity (Separate Chaining)

- **Find: O(1)** — The formal proof is too complex for a summary slide
- **Insert: O(1)** — The formal proof is too complex for a summary slide
- **Remove: O(1)** — The formal proof is too complex for a summary slide

Best-Case Time Complexity (Separate Chaining)

- **Find: O(1)** — No collisions
- **Insert: O(1)** — No collisions
- **Remove: O(1)** — No collisions

Space Complexity (Separate Chaining)

- **O(*n*)** — Hash Tables typically have a capacity that is at most some constant multiplied by *n* (the constant is predetermined), and each of our *n* nodes occupies O(1) space

Time/Space Complexities of a Hash Table/Map with Cuckoo Hashing

Worst-Case Time Complexity (Cuckoo Hashing)

- **Find: O(1)** — Keys can only map to two slots

- **Insert: O(*n*)** — If we run into a cycle and bound it by *n* (otherwise we could face an infinite loop), we rebuild the table in-place
- **Remove: O(1)** — Keys can only map to two slots

Average-Case Time Complexity (Cuckoo Hashing)

- **Find: O(1)** — Keys can only map to two slots
- **Insert: O(1)** — The formal proof is too complex for a summary slide
- **Remove: O(1)** — Keys can only map to two slots

Best-Case Time Complexity (Cuckoo Hashing)

- **Find: O(1)** — Keys can only map to two slots
- **Insert: O(1)** — No collisions
- **Remove: O(1)** — Keys can only map to two slots

Space Complexity (Cuckoo Hashing)

- **O(*n*)** — Hash Tables typically have a capacity that is at most some constant multiplied by *n* (the constant is predetermined)

Step 8

Multiway Trie

Summary Description

- A Multiway Trie is a tree structure in which, for some alphabet Σ, edges are labeled by a single character in Σ and nodes can have at most |Σ| children
- For a given "word node" in a Multiway Trie, the key associated with the "word node" is defined by the concatenation of edge labels on the path from the root of the tree to the "word node"
- We use *k* to denote the length of the longest key in the Multiway Trie and *n* to denote the number of elements it contains

Time/Space Complexities of a Skip List

Worst-Case Time Complexity

- **Find: O(*k*)** — Perform a O(1) traversal for each of the key's *k* letters
- **Insert: O(*k*)** — Perform a O(1) traversal for each of the key's *k* letters along with O(1) node/edge creations
- **Remove: O(*k*)** — Perform the find algorithm, and then remove the "word node" label from the resulting node

Average-Case Time Complexity

- **Find: O(*k*)** — If all keys are length *k*, then we do $\frac{k+k+...+k}{n} = \frac{nk}{k} = k$ on average, translating to O(*k*). If only one key is length *k*, even if all other keys are length 1, then we do $\frac{1+1+...+1+k}{n} = \frac{n-1+k}{n} = 1$ for *n* >> *k*, translating to O(1). If the lengths of keys are distributed uniformly between 1 and *k*, then the expected length of a key is $\frac{k}{2}$, translating to O(*k*)
- **Insert: O(*k*)** — Same proof as average-case find
- **Remove: O(*k*)** — Same proof as average-case find

Best-Case Time Complexity

- **Find: O(*k*)** — If all keys are length *k* (otherwise, it will simply be the length of the query string)
- **Insert: O(*k*)** — If all keys are length *k* (otherwise, it will simply be the length of the query string)
- **Remove: O(*k*)** — If all keys are length *k* (otherwise, it will simply be the length of the query string)

Space Complexity

- **Worst-Case: O** $\left(|\Sigma|^{k+1}\right)$ — If we were to have every possible string of length *k*, the first level of our tree would have 1 node, the next level would have |Σ| nodes, then |Σ|², etc., meaning our total space usage would be $|\Sigma|\left(1+|\Sigma|+|\Sigma|^2+\ldots+|\Sigma|^k\right)=|\Sigma|\left(\sum_{i=0}^{k}|\Sigma|^i\right)=|\Sigma|\left(\frac{|\Sigma|^{k+1}-1}{|\Sigma|-1}\right)\approx|\Sigma|^{k+1}$

Step 9

Ternary Search Tree

Summary Description

- A Ternary Search Tree is a trie in which each node has at most 3 children: a *middle*, *left*, and *right* child
- For every node *u*, the *left child* of *u* must have a value *less than u*, and the *right child* of *u* must have a value greater than *u*
- The *middle child* of *u* represents the next character in the current word
- For a given "word node," define the path from the root to the "word node" as *path*, and define *S* as the set of all nodes in *path* that have a middle child also in *path*. The word represented by the "word node" is defined as the concatenation of the labels of each node in *S*, along with the label of the "word node" itself

Time/Space Complexities of a Ternary Search Tree

Worst-Case Time Complexity

- **Find: O(*n*)** — If we insert the elements in ascending/descending order, we effectively get a Linked List
- **Insert: O(*n*)** — If we insert the elements in ascending/descending order, we effectively get a Linked List
- **Remove: O(*n*)** — If we insert the elements in ascending/descending order, we effectively get a Linked List

Average-Case Time Complexity

- **Find: O(log *n*)** — The formal proof is too complex for a summary slide
- **Insert: O(log *n*)** — The formal proof is too complex for a summary slide
- **Remove: O(log *n*)** — The formal proof is too complex for a summary slide

Best-Case Time Complexity

- **Find: O(*k*)** — If our query, whose length is *k*, was the first word that was inserted into the tree
- **Insert: O(*k*)** — If our new word, whose length is k, is a prefix of the first word that was inserted into the tree
- **Remove: O(*k*)** — If our query, whose length is k, was the first word that was inserted into the tree

Space Complexity

- **O(*n*)** — The formal proof is too complex for a summary slide

Step 10

Disjoint Set

Summary Description

- The Disjoint Set ADT is defined by the "union" and "find" operations: "union" merges two sets, and "find" returns which set a given element is in
- We can implement the Disjoint Set ADT very efficiently using Up-Trees
- Under Union-by-Size (also known as "Union-by-Rank"), when you are choosing which sentinel node to make the parent of the other sentinel node, you choose the sentinel node whose set contains the most elements to be the parent
- Under Union-by-Height, when you are choosing which sentinel node to make the parent of the other sentinel node, you choose the sentinel node whose height is smaller to be the parent
- Under Path Compression, any time you are performing the "find" operation to find a given element's sentinel node, you keep track of every node you pass along the way, and once you find the sentinel node, directly connect all nodes you traversed directly to the sentinel node
- Even though Union-by-Height is slightly better than Union-by-Size, Path Compression gives us the biggest bang for your buck as far as speed-up goes, and because tree heights change frequently under Path Compression (thus making Union-by-Height difficult to perform), we typically choose to perform Union-by-Size
- In short, most Disjoint Sets are implemented as Up-Trees that perform Path Compression and Union-by-Size

Time/Space Complexities of an Up-Tree

Worst-Case Time Complexity

- **Find: O(log *n*)** — Under Union-by-Size (the formal proof is too complex for a summary slide)
- **Union: O(log *n*)** — Under Union-by-Size (the formal proof is too complex for a summary slide)

Amortized Time Complexity

- **Find: O(1)** — Technically the inverse Ackermann function, which is a small constant for all practical values of *n* (the formal proof is too complex for a summary slide)
- **Union: O(1)** — Technically the inverse Ackermann function, which is a small constant for all practical values of *n* (the formal proof is too complex for a summary slide)

Best-Case Time Complexity

- **Find: O(1)** — If our query is a sentinel node
- **Union: O(1)** — If our queries are both sentinel nodes

Space Complexity

- **O(*n*)** — Each element occupies exactly one node, and each node occupies O(1) space

Step 11

Graphs and Graph Representation

Summary Description

- A Graph is simply a set of nodes (or "vertices") *V* and a set of edges *E* that connect them

- Edges can be either "directed" (i.e., an edge from *u* to *v* does not imply an edge from *v* to *u*) or "undirected" (i.e., an edge from *u* to *v* can also be traversed from *v* to *u*)
- Edges can be either "weighted" (i.e., there is some "cost" associated with the edge) or "unweighted"
- We call a graph "dense" if it has a relatively large number of edges, or "sparse" if it has a relatively small number of edges
- Graphs are typically represented as Adjacency Matrices or Adjacency Lists
- For our purposes in this text, we disallow "multigraphs" (i.e., we are disallowing "parallel edges": multiple edges with the same start and end node), meaning our graphs have at most $|V|^2$ edges

Time/Space Complexities of an Adjacency Matrix

Time Complexity (Adjacency Matrix)

- We can **check if an edge exists** between two vertices *u* and *v* (and check its cost, if the graph is "weighted") in **O(1)** time by simply looking at cell (*u*, *v*) of our Adjacency Matrix
- If we want to **iterate over all outgoing edges** of a given node *u*, we can do no better than **O(|*V*|)** time in the worst case because we would have to iterate over the entire *u*-th row of the Adjacency Matrix (which has |*V*| columns)

Space Complexity (Adjacency Matrix)

- **O(|*V*|²)** — If each of our |*V*| vertices has an edge to each of the other |*V*| vertices (the most dense our graph can be)

Time/Space Complexities of an Adjacency List

Time Complexity (Adjacency List)

- We can **check if an edge exists** between two vertices *u* and *v* (and check its cost, if the graph is "weighted") by searching for node *v* in the list of edges in node *u*'s slot in the Adjacency List, which would take **O(|*E*|)** time in the worst case (if all |*E*| of our edges came out of node *u*)
- If we want to **iterate over all outgoing edges** of a given node *u*, because an adjacency list has direct access to this list, we can do so in the least amount of time possible, which would be **O(|*E*|)** only in the event that all |*E*| of our edges come out of node *u*

Space Complexity (Adjacency List)

- **O(|*V*|+|*E*|)** — We must allocate one slot for each of our |*V*| vertices, and we place each of our |*E*| edges in their corresponding slot

Step 12

Graph Traversal Algorithms

Summary Description

- In all graph traversal algorithms we discussed, we choose a specific vertex at which to begin our traversal
- In Breadth First Search, we explore the starting vertex, then its neighbors, then their neighbors, etc. In other words, we explore the graph in layers spreading out from the starting vertex
- Breadth First Search can be easily implemented using a Queue to keep track of vertices to explore
- In Depth First Search, we explore the current path as far as possible before going back to explore other paths

- Depth First Search can be easily implemented using a Stack to keep track of vertices to explore
- In Dijkstra's Algorithm, we explore the shortest possible path at any given moment
- Dijkstra's Algorithm can be easily implemented using a Priority Queue, ordered by shortest distance from starting vertex, to keep track of vertices to explore
- For our purposes in this text, we disallow "multigraphs" (i.e., we are disallowing "parallel edges": multiple edges with the same start and end node), meaning our graphs have at most $|V|^2$ edges

Time/Space Complexities of Breadth First Search

Time Complexity (Breadth First Search)

- Exploring the entire graph using Breadth First Search would take time $\mathbf{O(|V|+|E|)}$ because we have to potentially visit all $|V|$ vertices and traverse all $|E|$ edges, where each visit/traversal is O(1)

Space Complexity (Breadth First Search)

- $\mathbf{O(|V|+|E|)}$ — We might theoretically have to keep track of every possible vertex and edge in the graph during our exploration
- If we wanted to keep track of the entire current path of every vertex in our Queue, the space complexity would blow up

Time/Space Complexities of Depth First Search

Time Complexity (Depth First Search)

- Exploring the entire graph using Depth First Search would take time $\mathbf{O(|V|+|E|)}$ because we have to potentially visit all $|V|$ vertices and traverse all $|E|$ edges, where each visit/traversal is O(1)

Space Complexity (Depth First Search)

- $\mathbf{O(|V|+|E|)}$ — We might theoretically have to keep track of every possible vertex and edge in the graph during our exploration
- Because we are only exploring a single path at a time, even if we wanted to keep track of the entire current path, the space required to do so would only be $O(|E|)$ because a single path can have at most $|E|$ edges

Time/Space Complexities of Dijkstra's Algorithm

Time Complexity (Dijkstra's Algorithm)

- We must initialize each of our $|V|$ vertices, and in the worst case, we will insert (and remove) one element into a Priority Queue for each of our $|E|$ edges, resulting in an overall worst-case time complexity of $\mathbf{O(|V| + |E| \log |E|)}$ overall if our Priority Queue is implemented intelligently (e.g. using a Heap)

Space Complexity (Dijkstra's Algorithm)

- $\mathbf{O(|V|+|E|)}$ — We might theoretically have to keep track of every possible vertex and edge in the graph during our exploration

Minimum Spanning Trees: Prim's and Kruskal's Algorithms

Summary Description

- Given a graph *G*, a Spanning Tree of *G* is a tree that hits every node in *G*
- Given a graph *G*, a Minimum Spanning Tree of *G* is a Spanning Tree of *G* that has the minimum overall cost (i.e., that minimizes the sum of all edge weights)
- We discussed two algorithms that can find a Minimum Spanning Tree in an arbitrary graph *G* equally efficiently
- Prim's Algorithm starts with a one-node tree and repeatedly finds a minimum-weight edge that connects a node in the tree to a node that is not in the tree, and adds that connecting edge to the tree
- Kruskal's Algorithm starts with a forest of one-node trees and repeatedly finds a minimum-weight edge that connects two previously unconnected trees in the forest and merges the two trees using the edge

Time Complexity of Prim's Algorithm

Worst-Case Time Complexity (Prim's Algorithm)

- **O(|*V*| + |*E*| log |*E*|)** — The formal proof is too complex for a summary slide

Time Complexity of Kruskal's Algorithm

Time Complexity (Adjacency List)

- **O(|*V*| + |*E*| log |*E*|)** — The formal proof is too complex for a summary slide