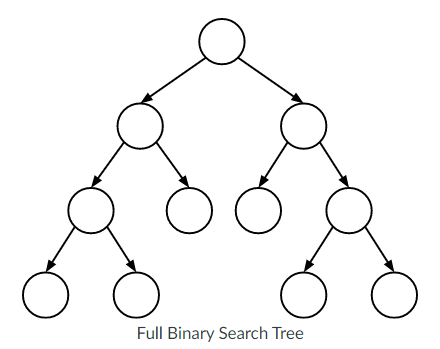
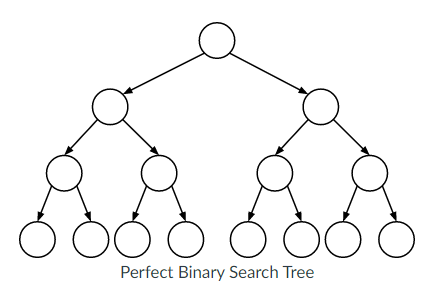
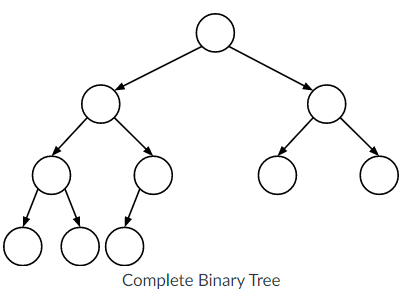
MODULE 6: Trees, Especially Binary Search Trees

TREES

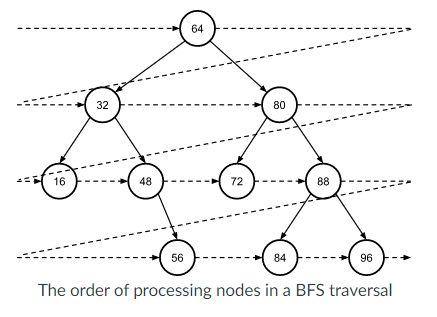
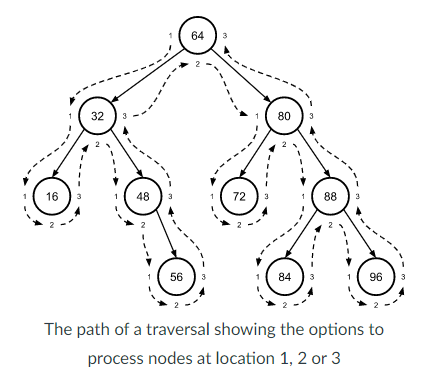
* Trees represent a collection of data as a hierarchical structure, encoding the hierarchical relationships between different data elements.
* Definitions
  + Parent: A node is a parent if it has an edge that points directly downward to another node
  + Child: A node is a child if it has an edge that it is being pointed by, aka the parent
  + Sibling: A node is a sibling is it shares the same parent node
  + Descendant: A node is a descendant if it comes after another node
  + Ancestor: Opposite of descendant
  + Root: beginning node
  + Interior node: if it has at least one child
  + Leaf: No children
  + Subtree: a portion of a tree
  + Path: collection of edges
  + Path length: length of a path
  + Depth: length of the path from root to a specific node
  + Height: Maximum depth
* Each node in the structure may have only one parent.
* The edges of the structure may not form any cycles.

BINARY TREES

* Special type of tree where each node has at most two children.
* Types of BST:
  + Full: every interior node has exactly two children
  + Perfect: All leaves are at the same depth
    - If we know height *h*, then we know:
      * Leaves = 2h
      * Nodes = 2h+1-1
    - If we know that a perfect BST has n nodes, then we know its height is approximately log(n)
  + Complete: perfect except for its deepest level whose nodes are all as far left as possible

Traversals

* Depth-first: explores a tree subtree by subtree, visiting all node’s descendants before visiting any of its siblings.
  + Moves as far downward in the tree as it can go before moving across in the tree.
    - N – visit/process the current node itself.
    - L – traverse the left subtree of the current node.
    - R – traverse the right subtree of the current node.
  + Pre-order: NLR – process the current node before traversing either of its subtrees.
  + In-order traversal: LNR – traverse the current node’s left subtree before processing the node itself, and then traverse the node’s right subtree.
  + Post-order traversal: LRN – traverse both current node’s subtrees (left, then right) before processing the node itself.
* Elegant recursive implementation:
* inOrder(N):
* if N is not NULL:
* inOrder(N.left)
* process N
* inOrder(N.right)
* Breadth-first: explores a tree level by level, visiting every node at a given depth in the tree before moving downward and traversing the nodes at the next-deepest level.
  + Moves as far across the tree as it can go before moving down in the tree.
* levelOrder(bst):
* q ← new, empty queue
* enqueue(q, bst.root)
* while q is not empty:
* N ← dequeue(q)
* if N is not NULL:
* process N
* enqueue(q, N.left)
* enqueue(q, N.right)



OPERATIONS

* Finding an element
  + Take a value, examine one node at a time.
    - One pointer at current node beginning at root.
    - If current node is None, the value DNE.
    - If found, return.
    - Otherwise, do a BST. If less than, cur.left. Otherwise, cur.right.
* find(bst, kq):
* N ← bst.root
* while N is not None:
* if N.key equals kq:
* return success
* else if kq < N.key:
* N ← N.left
* else:
* N ← N.right
* return failure
* Inserting a new element
  + Always inserted as leaves.
  + Must find the right location for the new leaf node—insert at a location that maintains the BST property at all nodes in the tree.
  + Proceed until reaching a None node to insert to.
* insert(bst, k, v):
* P ← None
* N ← bst.root
* while N is not None:
* P ← N
* if k < N.key:
* N ← N.left
* else:
* N ← N.right
* create a new node as the child of P containing k,v
  + Track the location of the new node’s parent, but otherwise search proceeds.
  + If P is None at the end of the search, this indicates that the BST is empty, and the new node should be inserted as the root of the tree. If P isn’t None, then the new node will be inserted as either the left or right child of P depending on whether k is less than or greater than/equal to P’s key.
* Removing an element
  + Requires A LOT of special cases.
  + Leaf node = just set to None.
  + Parent of a single child = set the child as the parent.
  + Otherwise, replace with the in-order successor aka parent’s right’s leftmost child.
* remove(bst, k):
* N, PN ← find the node to be removed and its parent
* based on key k, as in the find() function
* if N has no children:
* update PN to point to None instead of N
* else if N has only a child:
* update PN to point to N’s child instead of N
* else:
* S, PS ← find N’s in-order successor and its
* parent, as described above
* S.left ← N.left
* if S is not N.right:
* PS.left ← S.right
* S.right ← N.right
* update PN to point to S instead of N
* free N

RUNTIME COMPLEXITIES

* **Main factor of all three main BST operations arises from need to search within the tree**.
  + Find: search for the key.
  + Insert: search for the correct location.
  + Remove: search for both key and in-order successor.
* All operations begin at the root and moves down the tree one level until it reaches the bottom of the tree or finds the node of interest.
* Each operation performs a number of search iterations equal at most to the height of the tree.
* Each iteration of these searches performs constant-time work (comparisons)
* Thus, total amount of work done searching in all three of these operations is O(h).
* For a fixed number of nodes n, the height h of the BST can vary greatly. It can be as little as log(n) or as great as n.

Module Learning Outcomes

After successful completion of this module, you should be able to do the following (in addition to answering the questions listed below):

1. Describe the properties of a tree and a binary search tree
   1. In a tree, can a node have more than 1 parent? -- No
   2. What additional properties does a binary search tree have over a standard tree? – Binary trees have at most two children and has the children separated in a way such that keys that are less than the root node are on the left while keys greater than the root node are on the right. This repeats for each parent and child relationship throughout the tree.
2. Implement a binary search tree
   1. How can a binary tree be implemented using an array data structure?

– using a stack, you can use operations such as push, pop, top, and is empty or a queue with enqueue, dequeue and is empty. By initializing a node object with attributes left and right to represent that node’s children, the left and right pointers will point to other node objects. The node objects can then be saved in either a stack or queue array data structure, and operations can be done:

* 1. What functions or properties do nodes need to have in a binary tree?

1. Add: To add a new node into the array, you need to make sure if this is the first addition. If so, make it the root node. Otherwise, navigated through the BST beginning at the root until you reach None. The traversal should be comparing the node’s value against each node it passes to check if it is less or greater than it to move either left or right.
   1. Continually redefine the node as the current comparing node as its left or right child until it reaches a current that is None—that means you add it there!
2. Contains: Same thing as add except once the comparative node contains the same value of interest, return True. If the comparative node is None, you’ve reached the end of the subtree and the value DNE.
3. Remove\_first: First, check if the tree is empty—if so do nothing. Otherwise,
   1. Cases to check
      1. No right subtree 🡪 self.root = self.root.left (set root as the left node)
      2. No left sub tree 🡪 self.root = self.root.right (set root as the right node)
      3. Only one subtree of right subtree on the right 🡪 self.root.right.left = self.root.left, self.root = self.root.right (set the right subtree’s left node as the root’s left, and then the root as the right)
      4. Both subtrees exist, find and replace the root with the leftmost node of the right subtree. 🡪 keep the entire right subtree, and move left until you reach None. Track the previous node and the left node, continually replacing them as you move. After you reach the replacement/successor, replace the root node by setting the replacement node’s left as the root’s left, and the replacement nodes right as the root’s right. Finally, set the root as the replacement.
4. Remove: First, check if the tree is empty—if so, do nothing. Then check if the value to remove is the root. If so, just remove\_first(). Otherwise,
   1. Navigate through the tree to search for the value using BST
      1. If found, keep track of it using a variable 🡪 cur
      2. If not found, return False
      3. While you’re looking through, have a flag to check if the value exists on the left or right 🡪 found\_left = True/False
   2. Get the replacement node that should replace cur.
      1. Cases to check
         1. No right subtree:
            1. Replacement = cur.left
         2. No left subtree:
            1. Replacement = cur.right
         3. Right subtree only has one subtree
            1. Replacement = cur.right
            2. Replacement.left = cur.left
         4. Otherwise, find in-order successor:
            1. Replacement = cur.right
            2. Move to the left of replacement until None 🡪 replacement found
            3. Update pointers with removing node

Replacement\_previous.left = replacement.right

Replacement.left = cur.left

Replacement.right = cur.right

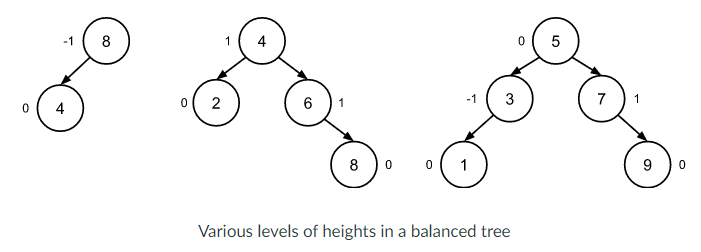
* 1. Dereference the removed node, replace with successor node. This completes the pointers.
     1. If found\_left 🡪 prev\_cur = replacement. Otherwise prev\_cur.right = replacement

1. Pre-order traversal:
   1. First check if the tree is empty 🡪 return an empty queue.
   2. Otherwise, recurse starting at the root node. If the node is not None, enqueue the node’s value and recurse the left and right (traversing the left and right subtrees). Then return the queue.
2. In-order traversal:
   1. First check if the tree is empty 🡪 return an empty queue.
   2. Otherwise, recurse starting at the root node. If the node is not None, enqueue as the bottom of a node is passed. So, recurse left, enqueue, then recurse right. Then return the queue.
3. Post-order traversal:
   1. First check if the tree is empty 🡪 return empty queue.
   2. Otherwise, recurse left and right, then enqueue. Then return the queue.
4. By-level traversal:
   1. First check if the tree is empty 🡪 return empty queue.
   2. Otherwise, track an additional queue that will have enqueue and dequeue nodes to navigate through level by level. Enqueue into the original queue the values of those nodes.
      1. Enqueue the self.root. Then while the helper is not empty, cur = helper.dequeue. If that cur node isn’t None, enqueue its value into the queue and then enqueue the left and right of the cur node into helper.
5. Is full: if the node is none, true. If no children, true. If children, recurse left and right. Else return false.
6. Is complete: if node is none: true. Else, have a tracker for the indices of the left and right using the array: left is index \* 2 + 1 to find children, right is index\*2 + 2.
7. Is perfect:
8. Size:
9. Height:
10. Number of leaves:
11. Finding unique values:
    1. How does having a binary tree simplify the implementation over a non-binary tree? – Maximum of 2 children, subtrees hold an ordered property, can be empty, limitation on the degree of a node, only left or right subtree,

MODULE 7: AVL Trees

* When we talk about balance with BST, we are referring to trees in which all nodes have depth approximately logn or less.
  + Balance is important because the primary operations on BSTs all have O(h) runtime complexity.
  + O(h) operations will be fast in a balanced BST because h will be approximately log(n). An unbalanced BST will result in h being closer to n.
* There is nothing in the mechanics of plain BSTs to ensure that a tree is balanced.
  + Important to realize that for a given set of keys the shape of BST depends on the order in which those keys are inserted into the tree.

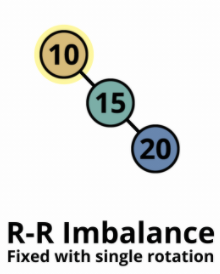
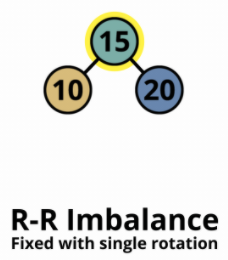
What is the definition of balancing?

* Height balance is a measurable form of BST balance.
  + A BST is height balanced if at every node in the tree the subtree heights of the node’s left and right subtrees differ by at most 1.
  + Important b/c it guarantees an overall height that is within a constant factor of log(n).
    - Because BST operations have O(h) runtime complexity, operations in a height balanced BST are guaranteed to have O(logn) runtime complexity.
* Node’s balance factor is a metric used to figure out whether the subtree rooted at that node is height balanced.
  + **Balance factor of the node N is the height of N’s right subtree minus the height of N’s left subtree:**
    - balanceFactor(N) = height(N.right) - height(N.left)
  + *NOTE: height of a null node is -1.*
  + **An entire BST is height balanced when every node in the tree has a balance factor of -1, 0, or 1.**
  + If a node has a **negative balance factor, it is considered LEFT-HEAVY**. This implies the node’s left subtree has greater height than its right subtree.
  + If a node has a **positive balance factor, it is considered RIGHT-HEAVY**.

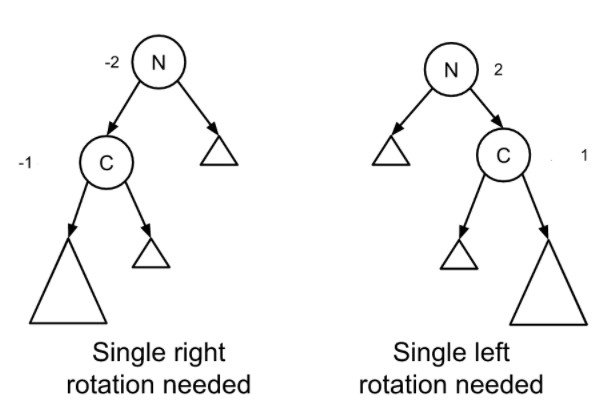
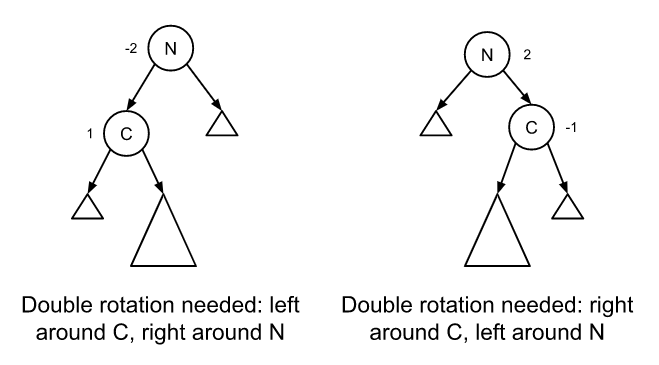
AVL TREE ROTATIONS

* AVL tree is one of several existing types of self-balancing BST.
* AVL is an acronym for the tree’s inventors.
* Includes mechanisms to ensure that the tree always exhibits height balance.
  + Check the height balance of the tree after each insertion and removal of an element and perform rebalancing operations known as rotations whenever height balance is lost.

ROTATIONS

* A rotation is a simple operation that restructures an isolated region of the tree by performing a limited number of pointer updates that result in one node moving “upwards” in the tree and another moving “downwards.”
  + Done in such a way as to preserve the BST property among all nodes in the tree.
* Each rotation has a center and a direction.
  + Center is the node at which the rotation is performed.
    - Either a left or right rotation around the center node can be done.
  + Left rotation: moves nodes in counterclockwise direction. Center moving downwards and nodes to its right moving upwards.
  + Right rotation moves modes in clockwise direction, with center moving downwards and nodes to its left moving upwards.
* 🡪 
* In this left rotation, node with key 10 is the center of the rotation. It moves downward in the tree which its right child, the node with key 15, moves upward.

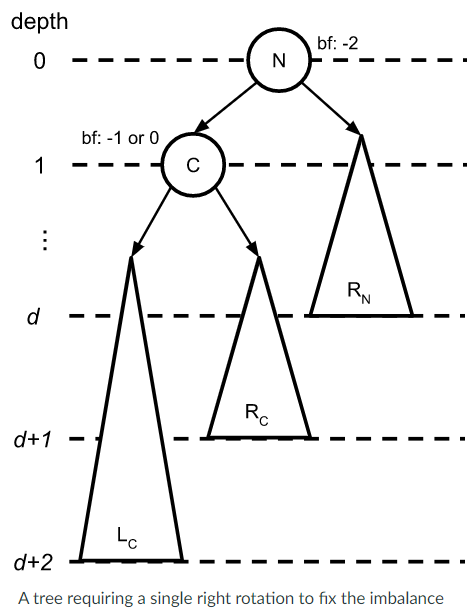
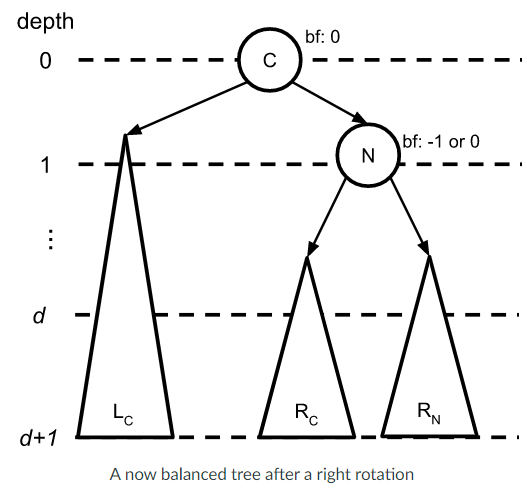
DETERMING THE ROTATIONS NEEDED

* A rotation of some kind will be needed any time an insertion into or removal from an AVL tree leaves the tree with a node whose balance factor is either -2 or 2.
* In other words: a rotation is needed when height balance is lost at a specific node in the tree.
* If N has a balance factor of -2, this means N is left-heavy. Regardless of the direction of heaviness, the heavier of N’s children will be regarded as C.
* C will also be either left or right heavy.
* If N and C are heavy in the same direction (same sign on balance factor) then a single rotation is needed around N in the opposite direction as N’s heaviness.
* 
* If they are heavy on opposite directions, a double rotation is needed.
  + If N is left-heavy and C is right heavy, rotate left around C then right around N.
  + If N is right-heavy and C is left-heavy, rotate right around C then left around N.
* 

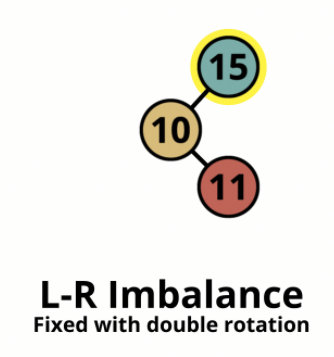
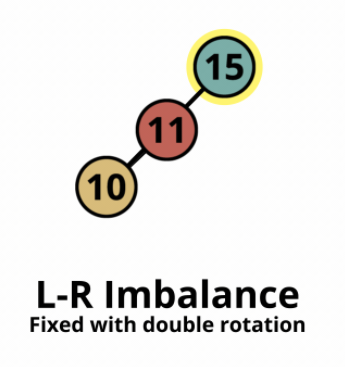
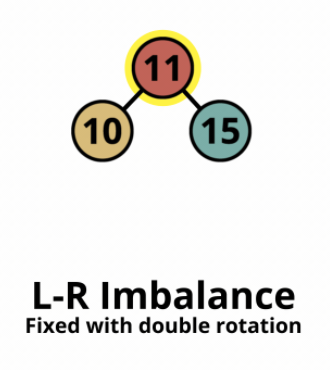
IMPLEMENTATION

Single Rotations

* Left-left imbalance – N is left-heavy and N’s left child C is also left-heavy
  + To fix a left-left imbalance at N, we apply a single right rotation around N.
* Right-right imbalance – N is right-heavy and N’s right child C is also right-heavy
  + To fix a right-right imbalance at N, we apply a single left rotation around N.
* Rotation results in its center N moving downward in the tree and N’s original child C moving up ward to become the new root of the subtree. If we were doing a left rotation, the operation would be mirrored.

 🡪 

Double Rotations

* Left-right imbalance – N is left-heavy and N’s left child C is right-heavy
  + To fix a left-right imbalance, we apply a left rotation around C followed by a right rotation around N.
* Right-left imbalance – N is right-heavy and N’s right child C is left-heavy
  + To fix a right-left imbalance, we apply a right rotation around C followed by a left rotation around N.
* 🡪  🡪
  + **The first rotation is simply intended to align imbalances on the same side**
    - **Create a left-left imbalance or a right-right imbalance**
    - **Then do a single rotation**

Now that we’ve established how the tree will be represented, we’ll more concretely specify how a rotation works. Here’s pseudocode for a left rotation centered around a node N:

rotateLeft(N):

C ← N.right

N.right ← C.left

if N.right is not NULL:

N.right.parent ← N

C.left ← N

N.parent ← C

updateHeight(N)

updateHeight(C)

return C

And here’s the code for a right rotation around N:

rotateRight(N):

C ← N.left

N.left ← C.right

if N.left is not NULL:

N.left.parent ← N

C.right ← N

N.parent ← C

updateHeight(N)

updateHeight(C)

return C

In both rotation functions, note that we return C, which has become the new root of the subtree at which the rotation was performed. We’ll use this return value later to update N’s old parent to point to C (and vice versa).

In the rotation functions, we also use a function updateHeight(), which updates the height of a node whose subtrees may have been restructured:

updateHeight(N):

N.height ← MAX(height(N.left), height(N.right)) + 1

Here, N’s new height is one more than the larger of the heights of its left and right subtrees (we add 1 to account for N itself). The height() function here simply returns a node’s height or –1 if that node is NULL.

Just to review the way these pieces work:

* Rotating left or right around a given node works as described above and simply involves trading a few pointers.
* If we perform a rotation, we must re-compute the subtree heights for both the node that moved downwards during the rotation (i.e., N) and the node that moved upwards during the rotation (i.e., C).

With those pieces concretely specified, we can now incorporate restructuring functionality into the AVL tree’s insert and remove operations. Here’s pseudocode for the insert operation:

avlInsert(tree, key, value):

insert key, value into tree like normal BST insertion

N ← newly inserted node

P ← N.parent

while P is not NULL:

rebalance(P)

P ← P.parent

Similarly, here’s pseudocode for the AVL tree’s remove operation:

avlRemove(tree, key):

remove key from tree like normal BST removal

P ← lowest modified node (e.g. parent of removed node)

while P is not NULL:

rebalance(P)

P ← P.parent

The key piece of both operations is the rebalance() function, which actually performs rebalancing at each node. Here’s pseudocode for that function:

rebalance(N):

if balanceFactor(N) < -1:

if balanceFactor(N.left) > 0:

N.left ← rotateLeft(N.left)

N.left.parent ← N

newSubtreeRoot ← rotateRight(N)

newSubtreeRoot.parent ← N.parent

N.parent.left or N.parent.right ← newSubtreeRoot

else if balanceFactor(N) > 1:

if balanceFactor(N.right) < 0:

N.right ← rotateRight(N.right)

N.right.parent ← N

newSubtreeRoot ← rotateLeft(N)

newSubtreeRoot.parent ← N.parent

N.parent.left or N.parent.right ← newSubtreeRoot

else:

updateHeight(N)

here’s what happens during a rotation:

* A limited (constant) number of pointers is updated (runs in constant time).
* The height of two nodes is updated (runs in constant time, since each update looks only at the heights of the node’s two children).
* If a tree is height balanced (like an AVL tree is), then its height is guaranteed to be within a constant factor of log(n). Thus, since rebalance() itself has O(1) complexity, the AVL tree’s insert and remove operations each have an overall complexity of O(log n).

KEY QUESTIONS

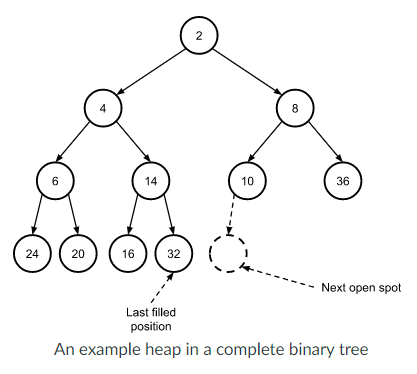
* Describe the issues with having an unbalanced tree.
  + What is the worst case search time in a tree that does not implement any sort of balancing? 🡪 o(n) where h is n
  + In a balanced tree what is the worst case search time? O(log(n))
* Describe the steps involved in a rotation.
  + Are we able to rotate a node that is not connected to leaf nodes? yes
  + If we rotate a node that is 1 level away from the root in a tree that is 5 levels deep, how many levels of the tree are affected? Only the root level and the level below it would be affected
* Implement a self-balancing AVL tree.
  + What steps are involved in a rotation if you are using an array to store the tree?
  + What steps are in involved in a rotation if you are using links between parents and children to store the tree?

MODULE 8: PRIORITY QUEUES AND HEAPS

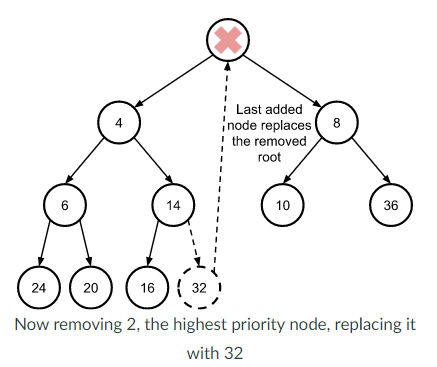
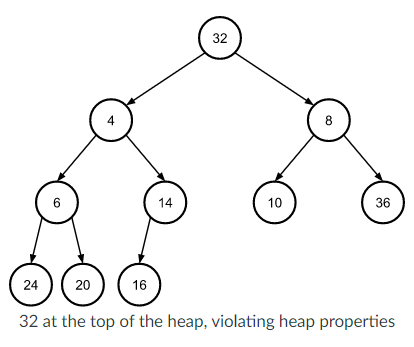
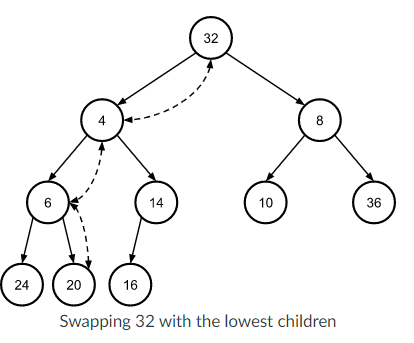
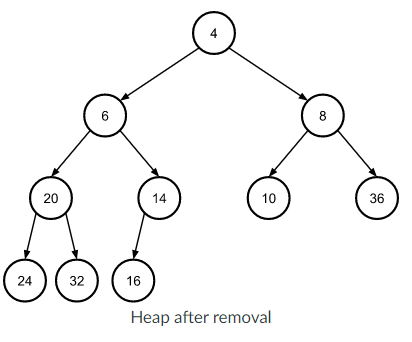
Priority queues and heaps:

* The priority queue is an ADT that associates a priority value with each element.
  + The element with the highest priority is the first one dequeued.
    - Typically corresponds to the element with the lowest priority value.
* A heap data structure is typically a complete binary tree in which every node’s value is less than or equal to the value of its children.
  + Specifically called a minimizing binary heap/min heap.
  + Max heaps are where each node’s value is greater than or equal to the value of its children.

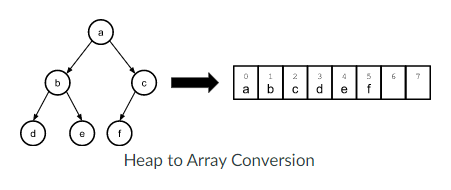
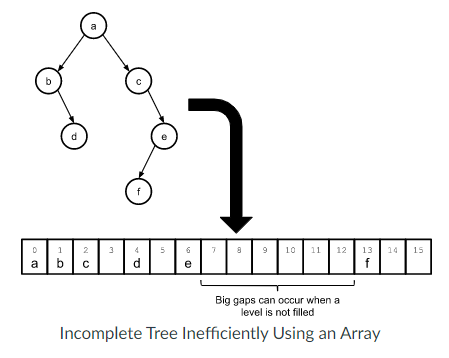
Heaps and priority queue basics

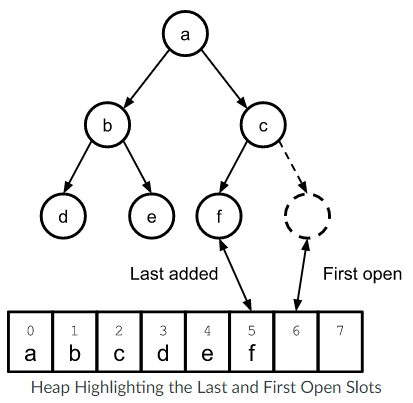
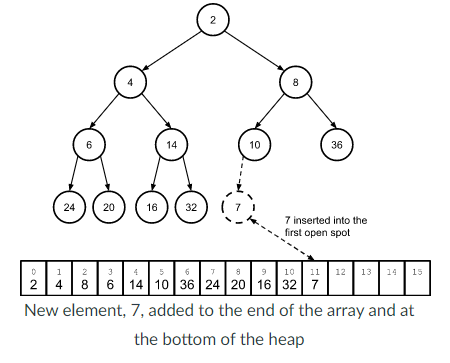
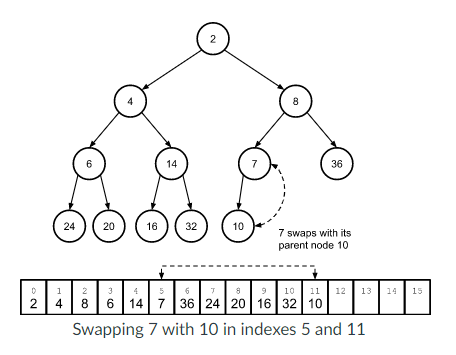
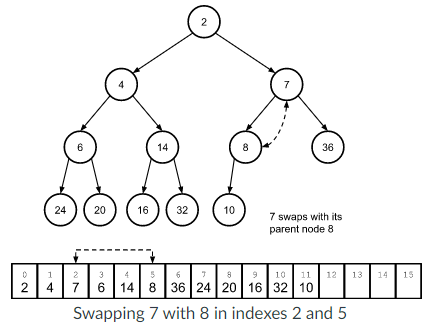
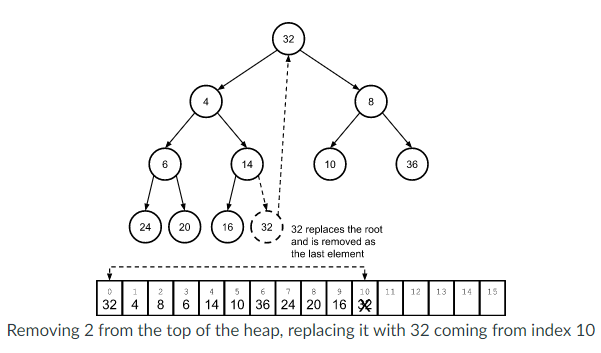
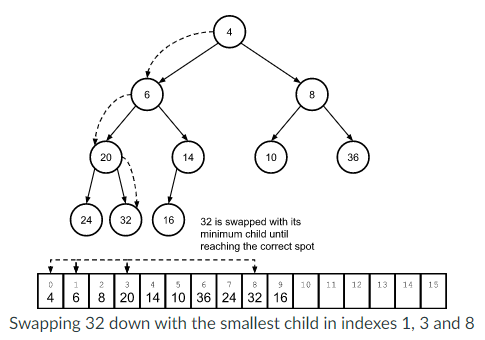
* The priority queue ADT has an interface that looks like:
  + Insert() – inserts an element with a specified priority value.
  + First() – returns the element with the lowest priority value (the “first” element in the priority queue).
  + Remove\_first() – removes and returns the element with the lowest priority value.
* 
* **A priority queue is typically implemented using a data structure called a heap.**
* Example of a min heap with only priority values:
  + NOTE: these priority values would also have some sort of attached data that we would be storing. We keep track of the last filled position and the next open spot.
  + 

Heap operations

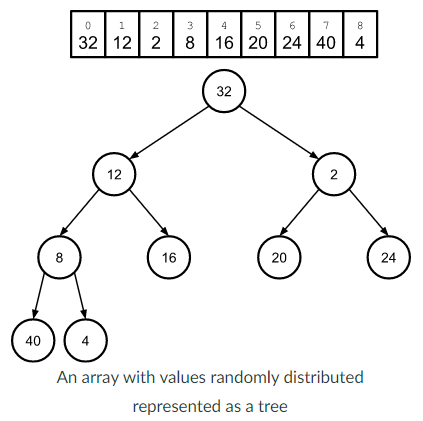
* Adding
  + A min/max heap is maintained through the addition and removal of nodes via percolations, which move nodes up and down the tree according to their priority values.
  + Adding is placing a value into the next open spot, then percolate it up the heap until its priority value is less than (greater for max) both its children.
  + Percolation happens by:
* while new priority value < parent’s priority value:
* swap new node with parent
* Removing
  + The root node’s priority value is ALWAYS the lowest in a min heap. Therefore, first() and remove\_first() operations always access and remove the root node.
  + If we remove the root node, how do we replace it?
    - Must maintain completeness of the BST.
    - To replace the root node after it is removed, we replace it with the element last in the heap and then fix the heap by percolating that node down.
    - 
  + The last node is removed after its value replaces the root:
    - 
  + Then percolate the replacement node down the tree:
* while priority > smallest child priority:
* swap with smallest child
  + Thus:
  + 🡪 

Heap Implementation

* Overview
  + Root node stored at index 0.
    - i is the index at which the node of interest is stored.
  + The left and right children of a node at the index are stored at indices 2\*i+1 and 2\*i+2 respectively.
  + Parent node is (i-1)/2 <<- using the floor that results from integer division.
  + Keeping track of the last element and the first open spot in the array is simple:
    - It is simply the last element in the array and the empty spot that follows the last element:
    - 
* Inserting into an array-based heap:
  + Put new element at the end of the array.
  + Compute the inserted element’s parent index aka (i-1)/2.
  + Compare the inserted element with the value of its parent.
  + If the value of the parent is greater than the value of the inserted element, swap the elements in the array and repeat step 2.
    - Do not repeat if the element has reached the beginning of the array.
  + 
  + 
  + 
* Removing from an array-based heap
  + Remember the value of the first element which must be returned later.
  + Replace the value of the first element in the array with the value of the LAST element and remove the last element.
  + If the array is not empty 🡪 compute indices of the children of the replacement element (2\*i+1 and 2\*i +2)
    - If both elements fall beyond the bounds of the array, stop here.
  + Compare the value of the replacement element with the minimum value of its two children (or possibly one child).
  + If the replacement element’s value is greater than its minimum child’s value, swap those two elements in the array and repeat from step 3.
  + 
  + 

Building a heap from an unsorted array



* Follow similar procedure as downward percolation in the removal method to build a heap from unsorted arrays.
* Remember: each individual leaf subtree is a proper heap of one element.
* Beginning at the leaf nodes, which we know are valid heaps, we go up to the parent of a leaf node and check if it is greater than the child. If it is, switch the parent node with the smaller of its children, making the subtree valid and then move to the next node. Repeat until all subtree heaps are valid, making the whole tree valid.

Heapsort

* An efficient O(nlogn), in-place sorting algorithm.
* First thing it does is build a heap out of the array using the procedure described above.
* Then, to complete the sort, we use a procedure like heap removal method with a few small tweaks.
  + Keep a running counter k that is initialized to one less than the size of the array (i.e., the last element).
  + Instead of replacing the first element in the array (the min) with the last element (the kth element), we swap those two elements in the array.
    - The array itself remains the same size and we decrement k.
  + When percolating the replacement value down to its correct place in the array, we stop at the kth element.
    - Thus, the heap is effectively shrinking by 1 with each iteration.
* We repeat this procedure until k reaches the beginning of the array.
* The sorting procedure maintains two properties while it runs:
  + The elements of the array beyond k are sorted with the min element at the end of the array.
  + The array through element k always forms a heap, with the minimum remaining value at the beginning of the array.
* Taking the heap array we built above, heapsort will run as follows, with k initialized to the last element in the array:
* Describe the operations associated with a priority queue.
  + Is it possible to access any item in a priority queue?
    - No, we can only access the item with the highest priority.
  + When storing data in a priority queue, what must also be provided?
    - Value must be inserted with its associated priority.
* Describe the properties of a heap.
  + How is a heap different from a binary search tree?
    - The value of any given node is larger than the value of its children.
    - Lets us insert elements into a tree more easily bc it has fewer limits than a bst but still lets us quickly find the largest element of a collection.
  + What is heap sort?
    - An efficient O(nlogn), in-place sorting algorithm.

MODULE 9: Hash maps

* Hash maps are tools used to help figure out where elements are in an array.
  + Basically, run the values through a formula that converts whatever we are storing into a number.
  + Take the modulo of that number based on the size of the array to find which space in the array contains that value.

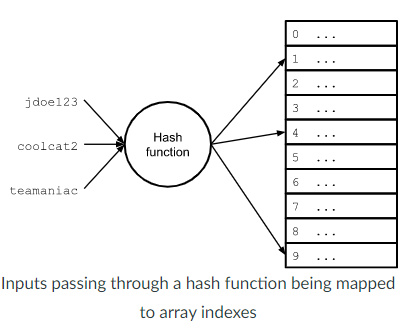
Intro to Maps and Hash Tables

* When insertion, lookup, and removal are the only operations we need, we can use the map data type.
* A map is also known as a dictionary or an associative array. It is built into Python in syntax: {key 1: value1,…}

Maps

* Allows you to use a **simple array**, storing key/value structs.
  + This would give us O(n) insertions and lookups, or O(logn) lookups, if we ordered the array by key.
  + We could use an **AVL tree**, also storing key/value structures.
    - This would give us O(logn) insertions and lookups.
* We can improve this by using a hash table.

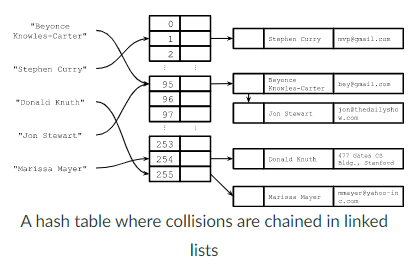
Hash table

* Like an array, except elements can be indexed by values other than integers and more than one element may share an index.
* Key to implementing a hash table is a hash function.
  + **A hash function** takes values of some type and maps them to an integer index value.
  + We use this value both to store and retrieve data out of an actual array.
  + 
* **Often the hash function computes an index in two steps:**
* hash = hash\_function(key)
* index = hash % array\_size
* The first step computes a value based on the key.
  + We may not always be working with arrays of a size that exactly matches the values the has function can produce.
  + This means we need to use % to make sure the value gets assigned to an index that actually exists in the array.
* When choosing/designing a hash function, you need:
  + **Determinism**: a given input should always map to the same hash value.
  + **Uniformity**: the inputs should be mapped as evenly as possible over the output range.
    - A non-uniform function can result in many collisions, where multiple elements are hashed to the same array index.
  + **Speed**: low computational burden.
* Example of well-known, widely used hash function:
* def hash\_djb2(s):
* hash = 5381
* for x in s:
* hash = (( hash << 5) + hash) + ord(x) #hash \* 33 + c
* return hash & 0xFFFFFFFF
  + Simple and fast, produces good distribution.

Perfect and minimally perfect hash functions

* **Perfect hash functions** are one that results in no collisions—every input get a unique output.
* **Minimally perfect hash functions** are ones that results in no collisions for a table size that EQUALS exactly the number of elements.
* def string\_hash(my\_string):
* return (ord(my\_string[0]) - ord('a')) % 6

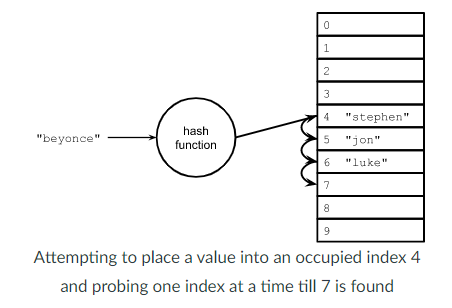
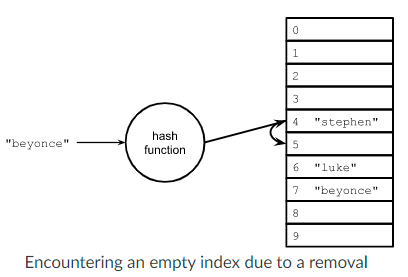
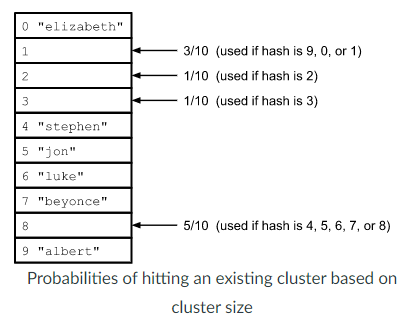
**Collision resolution with chaining**

* We can eliminate collisions entirely if we allow multiple keys to share the same table entry.
* For multiple keys, linked lists can be used to store the individual keys that map to the same entry.
  + Commonly referred to as buckets or chains.
* When collision occurs, the new element is simply added to the collection at its corresponding hash index.
* 
* **In a chained hash table, accessing the value for a particular key would follow:**
  + Compute the element’s bucket using the hash function.
  + Search the data structure at that bucket for the element using the key.
* **Adding or removing an element** would be similar except we would add or remove the element to or from the appropriate bucket’s data structure.
* **Load factor** of a hash table is the average number of elements in each bucket:
* 𝝺=n/m 🡪 load factor is equal to the total number of elements stored divided by the number of buckets
  + **n** is the total number of elements stored in the table.
  + **m** is the number of buckets.
  + **Lambda** is the load factor.
* In a chained hash table, the load factor can be greater than 1.
* As the load factor increases, operations on the table slow.
  + For linked list-based chained tables, the average number of links traversed for **successful searche**s is lambda/2.
  + For **unsuccessful searches**, the average number of links traverse is equal to lambda.
* Thus, to maintain strong performance with a hash table, we often **double the number of buckets when the load factor reaches a certain limit**.
  + The hash table array could be implemented with a dynamic array whose resizing behavior is based on the load factor.
* To perform the resize, recompute the hash function for each element with the new number of buckets.

**Average case complexity of a linked list-based chained hash table:**

* Assume the hash function has a good distribution.
* The average case for all operations is O(lambda).
* If the number of buckets is adjusted according to the load factor, then the number of elements is a constant factor of the number of buckets
  + 𝝺 = n/m = O(m)/m = O(1)).
* **Thus, the average case performance of all operations can be kept to constant time. The worst case complexity is O(n) since all the elements might end up in the same bucket.**

**Collision resolution with open addressing**

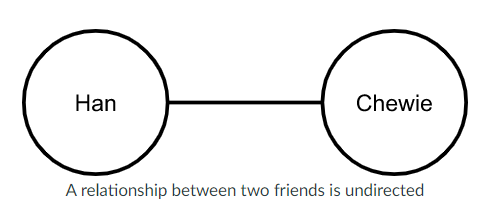
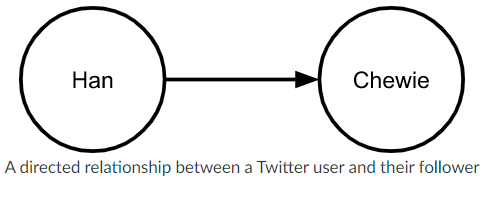
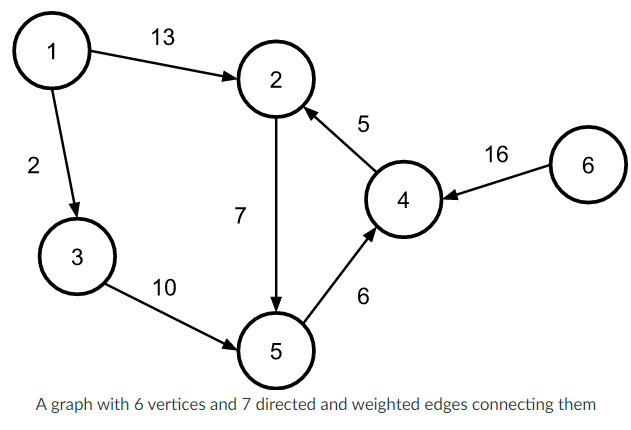
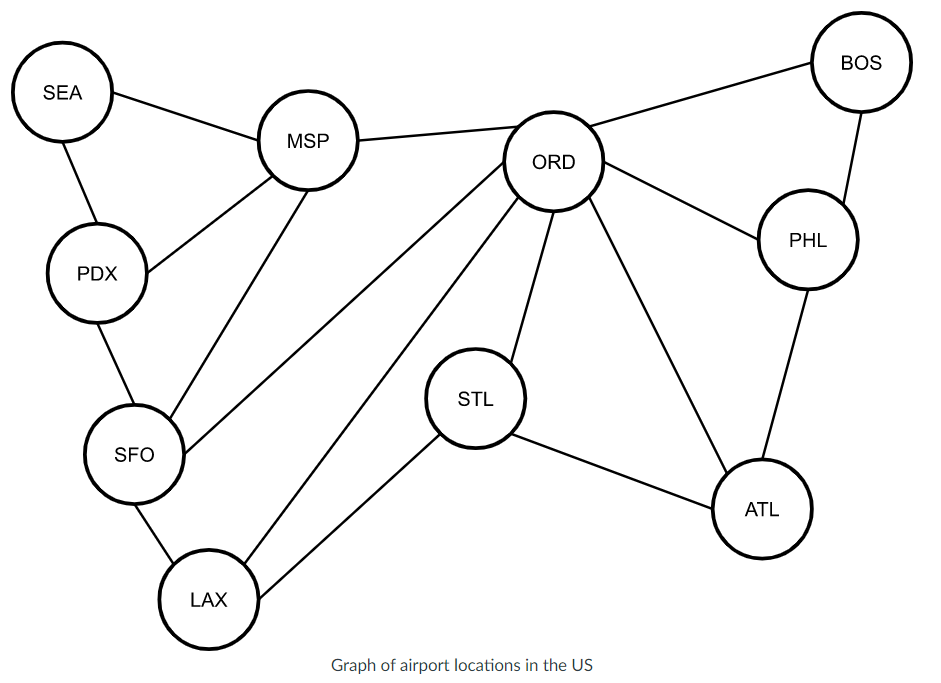
* Open addressing involves probing for an empty spot in the hash table array if a collision occurs.
* All hashed elements are *stored directly in the hash table array*.
* Procedure for adding:
  + Use the hash function to compute an initial index i\_initial for the element.
  + If the hash table array at index i\_initial is empty, insert the element there and stop.
  + Otherwise, compute the next index I in the probing sequence and repeat.
* Procedure for search for an empty position is called probing. It is the same as for inserting an element except we probe until we find either the element we’re looking for or an empty spot, the latter of which means the element doesn’t exist. It follows 3 main different schemes:
  + Linear probing: i = initial + j
    - 
    - Reached the end of the array while probing? Simply wrap around the beginning and rewrite the scheme—m is the total number of spots in the hash table array:
      * i = (initial + j) % m (m is the number of buckets)
  + Quadratic probing: i = initial + j^2
    - Reached the end of the array while probing? Simply wrap around the beginning and rewrite the scheme:
      * i = (i = initial + j^2) % m
  + Double hashing: i = initial + j \* h\_2(key)
    - Here, h\_2 is a second, independent hash function.
    - Reached the end of the array while probing? Simply wrap around the beginning and rewrite the scheme:
      * i = (initial + j \* h\_2(key)) % m
* **Remove an element**:
  + This may disrupt probing for elements after it. For example, if we removed “jon” in the example above and then searched for “Beyonce”:
    - 
  + Without taking an extra measure, “beyonce” would not be found because the probe would end at the now-empty index 5.
  + To get around this problem: use a special value known as a *tombstone*.
    - This value can be replaced when adding a new entry, since if there’s any value there, it won’t halt the search for an element.
  + With a tombstone value of \_TS\_ inserted for the removed “jon”, the search can proceed normally.
* **Clustering** is an issue where elements are placed into the table into clusters of adjacent indices. For example, using linear probing, the probability of a new entry being added to an existing cluster increases as the size of the cluster increases, since the larger cluster yields more chances for a collision.
  + 
  + Using quadratic probing and double hashing can help reduce clustering in an open addressing scheme.
  + Open addressing requires the load factor to be less than 1.
* Just as dynamic array is doubled when necessary, a common solution of a **full hash table** is to move all values into a new and larger table when the load factor becomes larger than some threshold.
  + To do so, a new table is created and every entry in the old table is rehashed, this time dividing by the new table size to find the index to use for the new table.
* When the load factor is very low, we have a lot of unused space allocated.
  + There is a tradeoff between speed and space with open addressing.

Complexity analysis of open address hashing

* Assuming truly uniform hashing:
  + To insert a given item into the table (that’s not already there), the probability for success is:
    - **p = (m-n)/m-0**
      * There are m total slots and n filled slots, so m-n open slots.
      * probability p
  + If the first probe fails, the probability that the second probe succeeds is:
    - p = (m-n) / (m-1)
      * There are still (m - n) remaining open slots, but now we only have a total of (m – 1) slots to look at since we examined one already.
  + If first two probes fail, the probability that third probe succeeds is:
    - P = (m-n)/(m-2)
    - Denominator is (m-2) because two slots have been examined and so forth.
  + **In other words, for each probe, the probability of success is at least p because:**
    - **(m-n)/(m-c) >= (m-n)/m = p**
  + Here, we are dealing with a geometric distribution, so the expected number of probes until success is:
    - 1/p = 1/ ( (m-n) / m ) = 1 / (1 – n/m) = 1 / (1 – lambda)
      * In other words, **the expected number of probes for any given operation is:**
        + O(1/(1-𝝺).
    - This suggests that, **if we limit the load factor to a constant and reasonably small number**, our operations will be O(1) on average.
      * For example, if we have lambda = 0.75, we expect 4 probes on average.
      * For lambda = 0.9, we would expect 10 probes.
* Describe how a hash function is used in a hash map. 🡪 maps values to keys to make pairs.
* Do we need to make sure our hash function only returns values smaller than the size of the array? Yes! Or you can increase the number of buckets but that can waste space.
* What is an example of a functional but very poorly performing hash function? 🡪 Hash based on number of letters.
* Explain how we can handle collisions in a hash map. 🡪 chaining or probing
* What data structure is often used in conjunction with an array to make a hash map? 🡪 Linked lists for chaining (?)
* The big O complexity of which data structure ends up prevailing if we pick a bad hash function? 🡪 linked lists O(n) in the event that all values pile into the same key and you must traverse the linked list until the end of the chain.

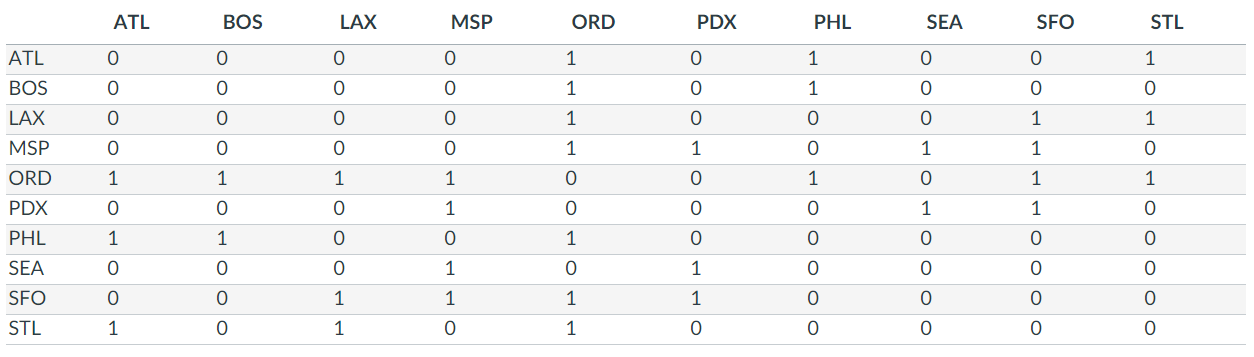
MODULE 10: GRAPHS

Graphs

* A graph is a structure representing a collection of objects or states, where some pairs of those objects are related or connected in some way.
* Graphs are used in:
  + Social networks
  + Computer graphics
  + Machine learning
  + Computer vision
  + Logistics and optimization
  + Computer networking
* Graphs are made up of vertices (things in the graph) and edges (connections between those things.
* Graph components
  + **Vertices** represent objects, states, locations, etc. These form a set where each vertex is unique (no two vertices represent the same object/state).
    - V = {v1, v2, v3, … vn}
  + **Edges** represent relationships or connects between vertices.
    - These are represented as vertex pairs.
      * E = {(vi, vj), …}
    - Edges can be directed or undirected.
    - If there is an edge between vi and vj, then vi and vj are said to be adjacent (or they are neighbors).
    - Edges can be weighted or unweighted. Weighted edges represent something about that edge. For example, it might represent the distance between vertices.
  + An undirected edge is like a friend relationship in FB.
    - If Han and Chewie are friends, there would be an undirected edge between them in the FB graph:
    - 
  + A directed edge is like a follow relationship in Twitter.
    - If Han follows Chewie, there would be a directed edge between them in the Twitter graph:
      * 
      * Here, we say the edge is directed from Han to Chewie.
      * We can also say that Han is the head of this edge, and that Chewie is its tail.
      * Or Chewie is a direct successor of Han, and that Han is a direct predecessor of chewie.
      * Or that Chewie is reachable from Han.
  + An example of a small graph with 6 vertices and 7 directed, weighted edges:
    - 
* Graphs represent general relationships between objects.
  + A node may have connections to any number of nodes.
    - An undirected graph is connected if all vertices are reachable from all other vertices.
    - A directed graph is strongly connected if all vertices are reachable from all other vertices.
  + There can be multiple paths (or no path) from one node to another.
    - A path in an undirected graph is a sequence of vertices, where each adjacent pair of vertices are adjacent in the graph.
      * Informally, we can also think of a path as a sequence of edges.
    - A simple path visits each edge at most once.
      * Simple paths are often sed for a trail with no repeated vertices, conflicting with the definition we have just mentioned.
      * Because of this variation in terminology, you need to make sure which set of definitions are used in a particular problem when you consider traversing edges of a graph.
      * A directed path is a sequence of directed edges in the graph.
  + A component of an undirected graph is an induced subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the rest of the graph.
    - A vertex with no incident edges is itself a component.
    - A graph that is itself connected has exactly one component, consisting of the whole graph.
    - Components are also sometimes called connected components.
  + There can be cycles in the graph.
    - A cycle is a closed walk that enters and leaves each vertex at most once.
    - An undirected graph is acyclic if no subgraph is a cycle.
      * Acyclic graphs are also called forests.
    - A directed graph is acyclic if it does not contain a directed cycle.
      * Directed acyclic graphs are often called dags.
* Two main ways to represent a graph:
* 

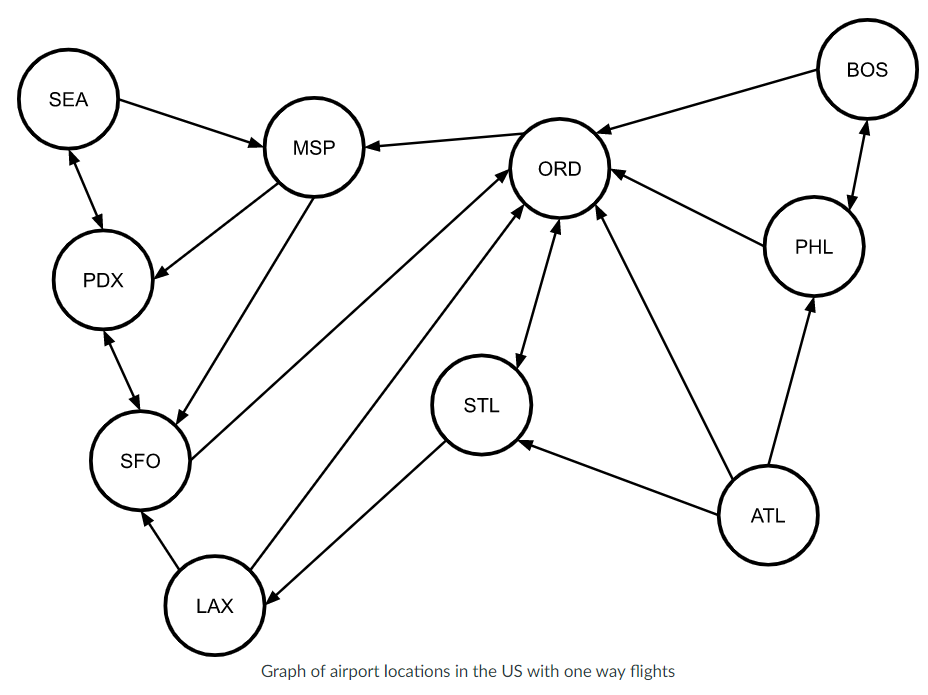
**UNDIRECTED GRAPHS**

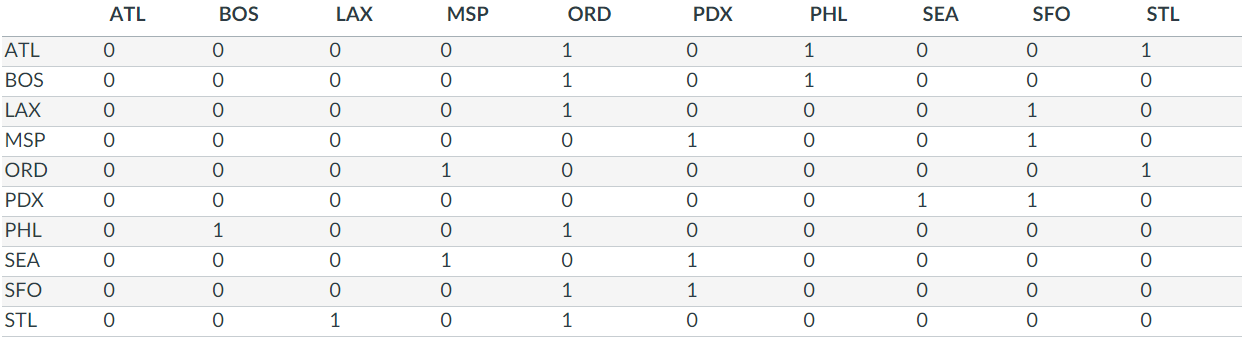
* + **An adjacency list**, in which each vertex stores a list of its adjacent vertices.
* ATL: [ORD, PHL, STL],
* BOS: [ORD, PHL],
* LAX: [ORD, SFO, STL],
* MSP: [ORD, PDX, SEA, SFO],
* ORD: [ATL, BOS, LAX, MSP, PHL, SFO, STL],
* PDX: [MSP, SEA, SFO],
* PHL: [ATL, BOS, ORD],
* SEA: [MSP, PDX],
* SFO: [LAX, MSP, ORD, PDX],
* STL: [ATL, LAX, ORD]
  + - Space complexity: O(|V|+|E|)
  + **An adjacency matrix**, which is a two-dimensional matrix whose rows and columns represent vertices.
    - If there is an edge between vi and vj, the value at location (I, j) in the matrix will be a non-zero. Note that the matrix is symmetric.



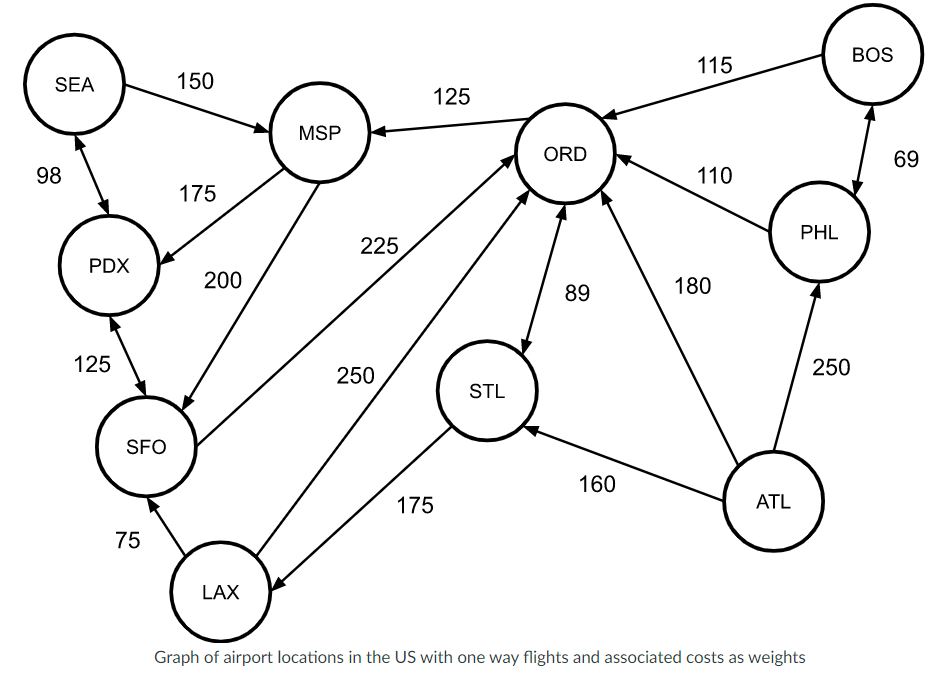
* + - Space complexity: O(|V|^2)
  + The adjacency list is more space efficient when the graph is sparse (relatively few edges).

**DIRECTED GRAPHS**

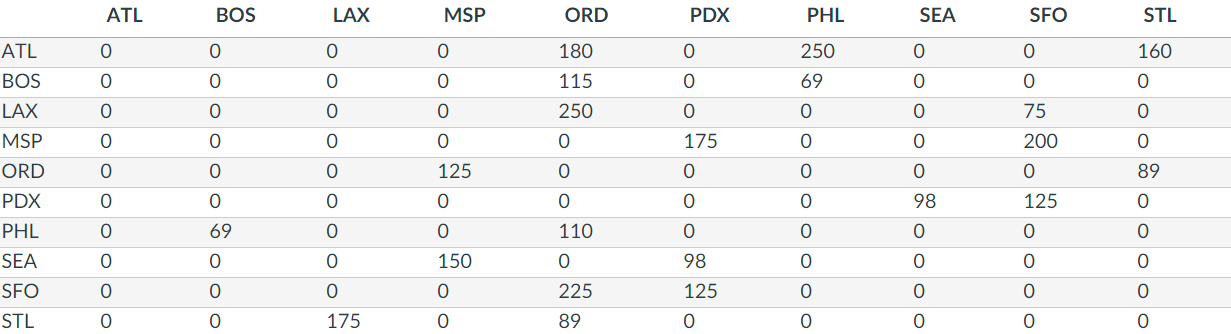
* For directed graphs:
  + 
  + Adjacency list
* ATL: [ORD, PHL, STL],
* BOS: [ORD, PHL],
* LAX: [ORD, SFO],
* MSP: [PDX, SFO],
* ORD: [MSP, STL],
* PDX: [SEA, SFO],
* PHL: [BOS, ORD],
* SEA: [MSP, PDX],
* SFO: [ORD, PDX],
* STL: [LAX, ORD]
  + Adjacency matrix (NOTE: no longer symmetric)



* With weight in a directed graph



* + Adjacency list would store the weights/costs along with the edges:
* ATL: [{ORD: 180}, {PHL: 250}, {STL: 160}],
* BOS: [{ORD: 115}, {PHL: 69}],
* LAX: [{ORD: 250}, {SFO: 75}],
* MSP: [{PDX: 175}, {SFO: 200}],
* ORD: [{MSP: 125}, {STL: 89}],
* PDX: [{SEA: 98}, {SFO: 125}],
* PHL: [{BOS: 69}, {ORD: 110}],
* SEA: [{MSP: 150}, {PDX: 98}],
* SFO: [{ORD: 225}, {PDX: 125}],
* STL: [{LAX: 175}, {ORD: 89}]
  + Adjacency matrix holds weights/costs instead of binary values



* + - Could use a special value to indicate there is no edge if we wanted to allow a special case.

Working with Graphs

* In almost every case, the thing we are **most interested in with graphs is if we can get from node A to node B**. We might also be interested in knowing the distance between nodes and which path is the shortest.
* Single source reachability -- DFS
  + Question: Which nodes are reachable from some specific node?
  + Algorithm:
    - 1. Initialize an empty set of reachable vertices.
    - 2. Initialize an empty stack. Add Vi to the stack.
    - 3. If the stack is not empty, pop a vertex v from the stack.
    - 4. If v is not in the set of reachable vertices:
      * Add it to the set of reachable vertices.
      * Add each vertex that is a direct success of v to the stack.
    - 5. Repeat from 3.
* Depth-first search and breadth-first search
  + **DFS** is an algorithm for exploring a tree where we travel a particular path as far as we can take it before trying another path.
    - In other words, in DFS, the neighbors of a node’s neighbor are explored before exploring the node’s other neighbors.
    - DFS can be implements using a stack like in the reachability algorithm.
      * If you replace the stack with a queue, this results in a BFS.
  + **BFS** explores a tree by traveling all paths to a given depth, then travelling all those paths one step deeper, then travelling them one step deeper, etc.
    - In other words, in BFS, all of a node’s neighbors are explored before exploring its neighbors’ neighbors.
    - BFS travels all paths of length 1, then travels all paths of length 2, then travels all paths of length 3, etc.
  + General algorithm for DFS (stack) and BFS (queue).
    - 1. Initialize an empty set of visited vertices.
    - 2. Initialize an empty stack (DFS) or queue (BFS). Add Vi to the stack/queue.
    - 3. If the stack/queue is not empty, pop/dequeue a vertex v.
    - 4. Perform any desired processing on v.
      * E.g., check if v meets a desired condition.
    - 5. DFS only: if v is not in the set of visited vertices:
      * Add v to the set of visited vertices.
      * Push each vertex that is direct successor of v to the stack.
    - 6. BFS only:
      * Add v to the set of visited vertices.
      * For each direct successor v’ of v:
        + If v’ is not in the set of visited vertices, enqueue it into the queue.
    - 7. Repeat from 3.
  + BFS or DFS usually used when we are looking for a node with a particular characteristic.
    - Ex. Both algorithms can be used to find a path from start to finish in a maze.
  + Comparisons between DFS and BFS:
    - DFS is a backtracking search.
      * If we’re looking for a node with a specific characteristic and DFS takes a path that doesn’t contain such a node, it will backtrack to try a different path.
      * In an infinite graph, DFS can become lost down an infinite path without ever finding a solution.
    - BFS is complete and optimal: if a solution exists in the graph, BFS is guaranteed to find it, and it will find the shortest path to that solution.
      * However, BFS may take a long time to find a solution if the solution is deep in the graph.
    - DFS may find a deep solution more quickly.
    - Both algorithms have O(V) space complexity in the worst case.
      * However, BFS may take up more space in practice.
        + If the graph has a high branching factor, i.e., if each node has many neighbors, BFS can take a lot of memory to maintain all of the paths it’s exploring on the queue.
* **Dijkstra’s algorithm: single source lowest-cost paths**
  + **Dijkstra’s algorithm finds the shortest/lowest-cost path from a specified vertex in a graph to all other reachable vertices in the graph.**
  + Structured very much like DFS and BFS, except for, in this algorithm, we will use a priority queue to order our search.
    - The priority values used in the queue correspond to the cumulative distance to each vertex added to the priority queue.
    - Thus, we are always exploring the remaining node with the minimum cumulative cost.
  + Algorithm:
    - Initialize an empty map/hash table representing visited vertices.
      * Key is the vertex v.
      * Value is the min distance d to vertex v.
    - Initialize an empty priority queue, and insert Vs into it with distance (priority) 0.
    - While the priority queue is not empty:
      * Remove the first element (a vertex) from the priority queue and assign it to v. Let d be v’s distance (priority).
      * If vi s not in the map of visited vertices:
        + Add v to the visited map with distance/cost d.
        + For each direct successor Vi of v:

Let di equal the cost/distance associated with edge (v, vi).

Insert vi to the priority queue with distance (priority) d+di

* + This version of the algorithm only keeps track of the minimum distance to each vertex, but it can be easily modified to keep track of the min-distance path, too. You can do this by augmenting the visited vertex map and the priority queue to keep track of the vertex previous to each one added.
  + The complexity of this version of the algorithm is O(|E|log|E|).
    - The innermost loop is executed at most |E| times, and the cost of the instructions inside the loop is O(log|E|). The inner cost comes from inserting into the priority queue.