# Table of Contents

# Big-O Runtimes!

## Lists

| Operation | Singly Linked List | Array |
|---|---|---|
| Look up an arbitrary location | $O(n)$ | $O(1)$ |
| Insert after a given element | $O(1)$ | $O(n)$ |
| Remove after a given element | $O(1)$ | $O(n)$ |
| Insert at an arbitrary location | $O(n)$ | $O(n)$ |
| Remove at an arbitrary location | $O(n)$ | $O(n)$ |
| Search for an input value | $O(n)$ | $O(n)$ |
| Special cases | Insert front $\rightarrow O(1)$ | Insert back (not full) $\rightarrow O(1)$* |

## Stack & Queue

| Operation | Stack | Queue |
|---|---|---|
| Push to top | $O(1)$ | - |
| Lookup top | $O(1)$ | - |
| Remove top | $O(1)$ | - |
| Push to back | - | $O(1)$ |
| Lookup front | - | $O(1)$ |
| Remove front | - | $O(1)$ |

## Trees

| Operation | Binary Tree | BST | AVL |
|---|---|---|---|
| find | $O(n)$ | $O(h), O(n)$ | $O(log\ n)$ |
| insert | $O(1)$ | $O(h), O(n)$ | $O(log\ n)$ |
| delete | $O(n)$ | $O(h), O(n)$ | $O(log\ n)$ |
| traverse | $O(n)$ | $O(n)$ | $O(n)$ |

## Heaps

| Operation | Heap |
|---|---|
| construction | $O(n)$ |
| insert | $O(log\ n)$ |
| removeMin | $O(log\ n)$ |

## Disjoint Sets

| Technique/Operation | find | union |
|---|---|---|
| Canonical keys (indices) | $O(1)$ | $O(n)$ |
| Canonical keys (stored as -1) | $O(n)$ | $O(1)$ |
| Smart union | $O(log\ n)$ | $O(log\ n)$ |
| Smart union + path compression | $O(1)$* | $O(1)$* |

## Graphs

$|V| = n,\ |E| = m$

| Operation | Edge List | Adjacency Matrix | Adjacency List |
|---|---|---|---|
| space | $O(n + m)$ | $O(n^2)$ | $O(n + m)$ |
| insertVertex | $O(1)$* | $O(n)$* | $O(1)$* |
| removeVertex | $O(n + m)$ | $O(n)$ | $O(deg(v))$ |
| insertEdge | $O(1)$* | $O(1)$ | $O(1)$* |
| removeEdge | $O(m)$ | $O(1)$ | $O(min(deg(u), deg(v)))$ |
| incidentEdges | $O(m)$ | $O(n)$ | $O(deg(v))$ |
| areAdjacent | $O(m)$ | $O(1)$ | $O(min(deg(u), deg(v)))$ |

## Graph Algorithms

$|V| = n,\ |E| = m$

| Algorithm | Sparse | Dense |
|---|---|---|
| BFS | $O(n + m)$ | $O(n + m), O(n^2)$ |
| DFS | $O(n + m)$ | $O(n + m), O(n^2)$ |
| Kruskal's | $O(n + n\ log\ n)$ | $O(n + n^2\ log\ n)$ |
| Prim's | $O(n\ log\ n + n\ log\ n)$ | $O(n^2\ log\ n)$ |
| Dijkstra's | $O(n\ log\ n + n\ log\ n)$ | $O(n^2\ log\ n)$ |
| Floyd-Warshall's | $O(n^3)$ | $O(n^3)$ |

## Hash Table

Overall $O(1)$*** **IN EXPECTATION**
On its own → Overall $O(n)$
$\alpha = \frac{\#\ items}{\#\ positions} = \frac{n}{m}$

| Operation | Open Hashing | Closed Hashing |
|---|---|---|
| insert | $O(1)$ | $O(\frac{1}{1-\alpha})$ |
| find/remove | $O(1 + \alpha)$ | $O(\frac{1}{1-\alpha})$ |

## Bloom Filter

Overall $O(1)$ → ignores collisions

| Operation | Bloom Filter |
|---|---|
| insert | $O(1)$ |
| find | $O(1)$ |
| remove | n/a |

# C++ Review (Week 1)

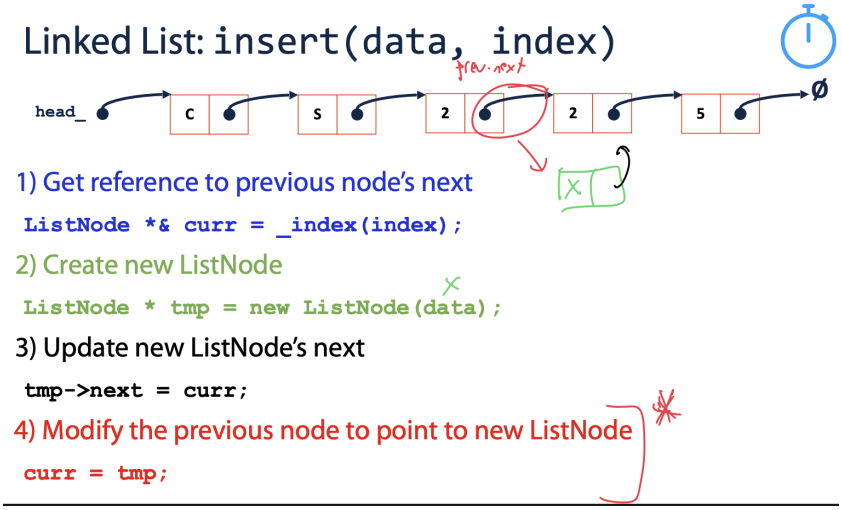
- Pointers are variables that store memory addresses (like a sign that points you to a house)
    - To get the value the pointer is pointing to, use the dereference operator *
    - Can be used both on the stack and the heap
        - Book* b = &book
        - Book* b = new Book();
- Stack - local variable storage
    - Local memory is managed by the computer
- Heap (free store) - dynamically allocated storage
    - Heap memory is managed by the **new** and **delete** keywords
- The way a parameter is used in a function can change what it does
    - Pass by value - creates a copy (will NOT update the variable in main if changes are made here)
    - Pass by pointer - an address on the heap (changes will persist because the pointer can be dereferenced)
    - Pass by reference - an alias to an existing variable (changes will persist since it is redefined in a local scope)
- If we are storing pointers, it is NOT our job to allocate memory
    - If we did not allocate, it is not our problem to deallocate
- Rule of Three: If any of these functions is necessary to define, then all three must be defined
    - Copy constructor - make a deep copy of an already existing object
    - Copy assignment operator - assign value from existing object to **this** object
    - Destructor - deletes an object (frees allocated memory)

# Lists ADT

- Templated functions are a recipe for code using generic types
- Abstract data types (ADT) describe a data type as a combination of data being stored by the data type and operations that can be performed on the data type
- A list in an ordered (not sorted) collection of items that can be similar or different, and can be a fixed size or can be resized

# Linked Lists (Week 2)

- Linked lists are made up of nodes that store data and store a next pointer to the next data
    - Storing data as references allows us to modify it directly if necessary
    - The List class will have a head_ pointer that accesses the beginning of a linked list
- To insert a node at the head of a linked list, create the node first, then make its next pointer the current head, and update head_ to be the node we just made
    - All of these actions are constant, so the runtime is $O(1)$
- To insert a node at an index, we need to create the node, traverse the list up until index-1, assign the node's next pointer to the new node's next pointer, then update the node (position we are at) next pointer to point to the node we just made
    - Creating the node and updating the pointers takes $O(1)$ time, but traversing the list takes $O(n)$, so the runtime is $O(n)$



- Random access is $O(n)$ in a singly linked list (worst case, index is all the way at the end)
- Similar to insert, removing a node takes $O(n)$ time (but if we're given a reference to the node right before it, it is $O(1)$ time)

## Array (Week 3)

- An array is allocated as continuous memory
  - Values needed for efficiency are data, size, and capacity (which are all memory addresses)
  - data_ is a pointer to the start
  - size_ is a pointer to the next available space
  - capacity_ is a pointer past the end of the array
    - Capacity is reached when size_ == capacity_
- Random access is $O(1)$ since we can get the data through the index
- Inserting at the front takes $O(n)$ time since we have to shift all the other values to the right by one space to make room for the new data
- Inserting at index (assuming we are not at capacity) requires finding the position of interest and shifting any data to the right of it one space to the right to make room for the new data
  - This results in an $O(n)$ runtime (worst case, index = 0)
- Inserting at the back (assuming we are not at capacity) takes constant work since no shifting is necessary
  - This results in $O(1)$ time
- Amortization gives us the precise total work over n calls
  - If the array resizes by a constant k, the runtime is $O(n)$
  - If the array is resized by a factor (double for example) the runtime is $O(1)$*

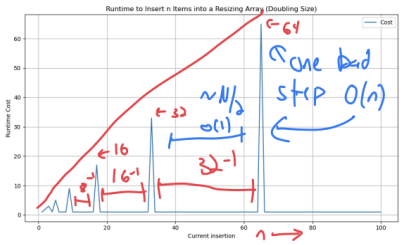
**Amortized:** $O(1)^*$

Precise total work over N calls

$\frac{2N-1}{N} \cong 2$ work per insert

**Big O:** $O(n)$

Upperbound on worst case



## Runtime Table

| | Singly Linked List | Array |
|---|---|---|
| Look up an arbitrary location | $O(n)$ | $O(1)$ |
| Insert after a given element | $O(1)$ | $O(n)$ |
| Remove after a given element | $O(1)$ | $O(n)$ |
| Insert at an arbitrary location | $O(n)$ | $O(n)$ |
| Remove at an arbitrary location | $O(n)$ | $O(n)$ |
| Search for an input value | $O(n)$ | $O(n)$ |
| Special cases | Insert front $\rightarrow O(1)$ | Insert back (not full) $\rightarrow O(1)$* |

# Stacks and Queues (Week 3)

- A stack is like a list, but can only do three operations:
    - push - puts an item on the top of the stack
    - pop - removes the top item on the stack
    - top - returns the top item of the stack
    - All are $O(1)$ operations
- Follows a **last-in, first-out (LIFO)** pattern
- The implementation is similar to that of a linked list and array
- Tradeoffs:
    - All access rules are $O(1)$ in exchange for no random access

- A queue stores an ordered collection of objects like a list, but can only do three operations:
    - enqueue - put an item at the **back** of the queue
    - dequeue - remove the **front** item of the queue
    - front - return the front item of the queue
    - All are $O(1)$ operations
- Follows a **first-in, first-out (FIFO)** pattern
- It is easier to implement a queue as an array using unsigned ints
- Resizing the queue is $O(n)$ for copying the values over, but is still $O(1)$*

# Iterators (Week 3)

- Iterators provide a way to access items in a container without exposing the underlying structure of the container
- For a class to implement iterators, it needs two functions called begin() and end()
    - begin - returns an Iterator object pointing at the first item
    - end - returns an Iterator object pointing to one entry past the end of the dataset
- Iterator is defined in a class inside of a class
    - Must implement…
        - Iterator& operator ++() - move to next position
        - const T& operator *() - dereference operator
        - bool operator !=(const Iterator&) - compare two Iterator objects
    - Allows us to use **auto** keyword to iterate

# Trees ADT (Week 4)

- Trees are made up of nodes and edges
    - Unlike linked lists, the nodes can have multiple edges
- There are three types of nodes:
    - Root - no parent
    - Leaf - no children
    - Internal - have parent and children nodes
- Relationships
    - Siblings - nodes that have the same parent
    - Neighbor - nodes that have an edge with each other
    - Descendent - node that has mutual nodes with an upper node
    - Ancestor - node that has mutual nodes with a lower node
- The height of a null tree is -1 (if there were a root node, the height would be 0)
    - height(T) = max (height(TL), height(TR)) + 1
- A tree is full only if:
    - The tree is null
    - The tree has a root, left subtree, and right subtree that are empty
    - The tree has a root, left subtree, and right subtree that are not empty
- A tree is perfect if:
    - The tree is null
    - The tree has a root, left subtree, and right subtree, where the left and right subtrees are also perfect
        - # nodes = $2^{h+1} - 1$
        - # leaf nodes = $2^{h}$
- A tree is complete if:
    - The tree is perfect except for the last level (leaves can be at different levels)

- The tree data structure has insert, remove, access, make, and empty functions
- One tradeoff of trees is having lots of wasted pointers
    - If there are n data items, there are n+1 null pointers
- Types of traversals:
    - In-order: visit left subtree, root, right subtree
    - Pre-order: visit root, left subtree, right subtree
    - Post-order: visit left subtree, right subtree, root
    - The runtime for these traversals is $O(n)$ because we are visiting every node once
    - Additionally, these traversals involve recursion
- By looking at the preorder and postorder outputs, we can determine the possibility of what the original tree looks like
    - To be confident, we need the in-order and either the pre- or post-order
- To do level-order traversal, we can use a queue to keep track of the nodes (this is how BFS works!)
    - Enqueue the root, then enqueue its children while the queue is not empty
- Any traversals listed above are good for tree search in a tree with no structure, since we have to look through all the nodes anyway

- Tree search:

| Search type | Traversal type | Time complexity | Space complexity |
|---|---|---|---|
| Depth-First Search (DFS) | In, Pre, Post | $O(n)$ | $O(h)$<br><br>*height of the tree since each level lower adds a recursive call to the stack frame |
| Breadth-First Search (BFS) | Level | $O(n)$ | $O(2^h)$<br><br>*width of the tree since the queue is storing nodes |

# BSTs (Week 5)

- In an infinite tree, BFS is recommended because using DFS will trigger an infinite recursive loop (there is no maximum depth)
- If we restrict the depth, we can get a lower space complexity for DFS while keeping the same runtime

- Dictionary ADT stores key-value pairs
    - Used in many data structures as an underlying base
    - Contains insert, remove, and find functions
    - Keys must be unique, but values do not have to be

- In BSTs, for any x in the left subtree, the value must be less than the root
- Similarly, for any y in the right subtree, the value must be greater than the root
- These definitions must be applied recursively!
- The worst-case structure for a BST is when the height is the same as the number of nodes (essentially a linked list)
    - This causes the worst-case runtimes of find, insert, and remove to all be $O(n)$
- BST find()
    - Check base case for if root is null or root->key is key
    - Call find on left subtree if root->key > key
    - Call find on right subtree if root->key < key
- BST remove()
    - Case 1: if it is a leaf, delete it like a linked list
    - Case 2: if it has one child, then have a temp, set it to the node we want to delete, assign the parent node to the target's child, then remove temp (which is the target)
    - Case 3: if it has two children, find the target node and its IOP/IOS (in-order predecessor and successor), swap the target with the IOP/IOS, and recursively call on the target's new location
        - Basically, it recurses until it hits one of the other two cases

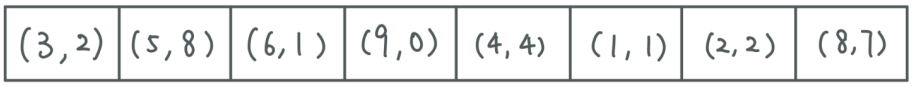
Runtimes (Up to Week 5)

| Operation | BST | Binary Tree | Sorted Array | Unsorted Array | Sorted Linked-List | Unsorted Linked-List |
|---|---|---|---|---|---|---|
| find | $O(h)$, $O(n)$ | $O(n)$ | $O(log\ n)$ | $O(n)$ | $O(n)$ | $O(n)$ |
| insert | $O(h)$, $O(n)$ | $O(1)$ | $O(n)$ | $O(n)$, $O(1)$* | $O(n)$ | $O(1)$ |
| delete | $O(h)$, $O(n)$ | $O(n)$ | $O(n)$ | $O(n)$ | $O(n)$ | $O(n)$ |
| traverse | $O(n)$ | $O(n)$ | $O(n)$ | $O(n)$ | $O(n)$ | $O(n)$ |

n = # of elements

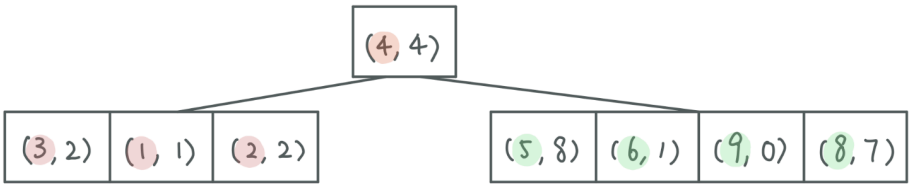
# KD Trees (Week 5 & 6)

- A KD tree is a k-dimensional tree (nodes have $k$ number of properties)
    - Can use the quick select algorithm
- BSTs are useful for 1D range-based searches and nearest-neighbor searches
    - However, the runtime is $O(n)$
- To build a KD tree…
    - Find the median point along a dimension and partition nodes
    - Go to the next dimension
    - Recursively build left subtree
    - Recursively build right subtree

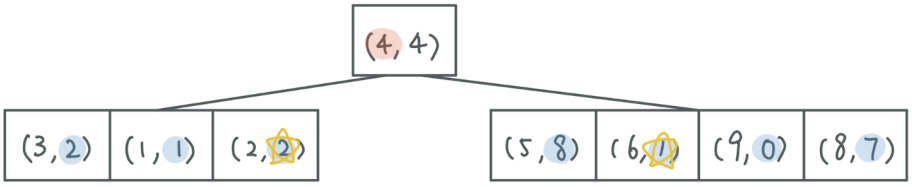



Suppose we are given this array to construct a kd-Tree.

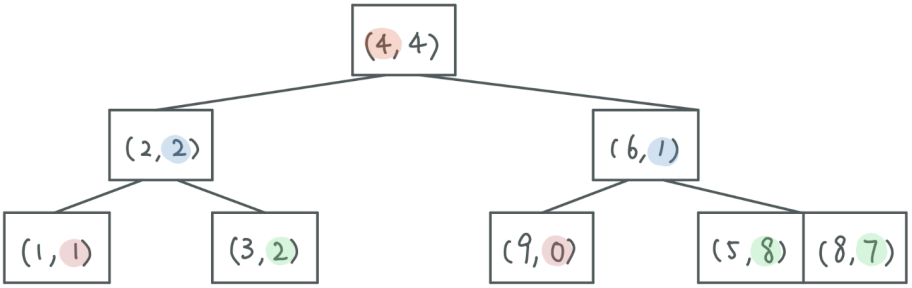

(4, 4) is the median in terms of *x* coordinate, make it our subroot.




Partition the array so that every point with *x* coordinate smaller than 4 is on the left side of (4, 4), and every point with *x* coordinate larger than 4 is on the right side of (4, 4).




Find the median in terms of *y* coordinate on each of the subarrays.



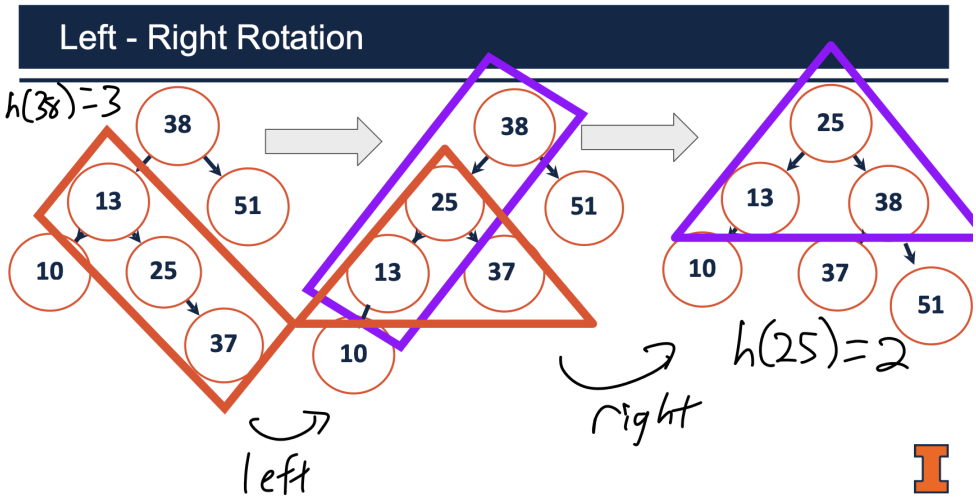

Partition each subarray by its median, and make the median the subroot. Repeat this process until the array only consists of one node.

- For the quick select algorithm…
    - Partition elements about the median
    - Smaller values on the left of the median
    - Larger values on the right of the median
    - Must be FASTER than $O(n \log n)$
- The four steps of the algorithm are:
    - Select a random pivot
    - Swap the pivot to the end
    - Partition elements
    - Swap pivot value to its proper place
- In the worst case, the pivot is either the max or min value
    - Work decreases by 1
    - Runtime is $O\left(n^2\right)$
    - This is highly unlikely due to picking the pivot randomly, so let's look at the average case
- In the average case, the runtime of quick-select is $O(n)$
- The runtime of building a KD tree is $O(n \log n)$

- A lambda function is an in-line function without a name
    - Good for short functions that are called infrequently
    - Larger code size, but executes faster
- [capture](input){body}
    - Capture - takes variables from the current program state
    - Input - function parameters
    - Body - function body

# AVL Trees (Week 6 & 7)

- "Mountain" trees are balanced
- "Stick" trees are unbalanced
- An AVL tree is a self-balancing BST
- Balanced trees will guarantee better runtimes
- The balance (b) of a node is the height of its right subtree minus the height of its left subtree
    - Positive - right subtree is taller
    - Negative - left subtree is taller
- A tree is balanced if the $|b| \leq 1$
- A tree can be balanced by rotating it to the left or right
    - Converts sticks
    - Balances have the same sign
    - The rotations modify 3 pointers
- In situations where a singular rotation does not resolve the imbalance, we have to perform a left-right or a right-left rotation
    - This occurs when a node has a balance of 1 and its parent has a balance of (opposite sign) 2




- All rotations take $O(1)$ time, and rotations preserve the BST property!
    - $O(h) \rightarrow O(log\ n)$

## AVL Trees

Three issues for consideration:

1. Detecting Imbalance - How do I detect an imbalance? ✓ $|b| > 1$
2. Rotations - How do they work? Modifying pointers to make "sticks" → "mountains"
3. Selecting a rotation - Which one do I choose?

balance r −2 2

$T_L$ −1 1 — Right Rotation; Left-Right Rotation

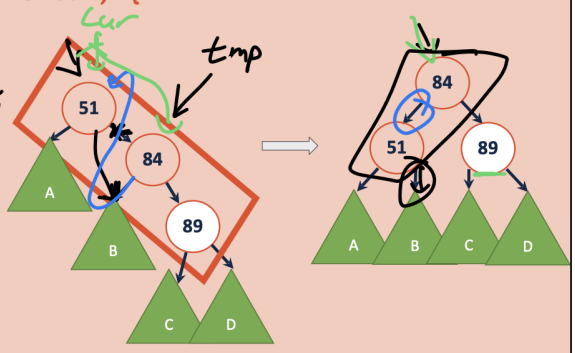
$T_R$ 1 −1 — Left Rotation; Right-Left Rotation

- To insert into an AVL tree…
  - Find (using the BST property) and insert the node
  - Check if nodes are balanced (recursively)
  - Rotate to fix the imbalance
  - Update the height

```
1  template <typename K, typename V>
2  void AVL<K, V>::_ensureBalance(TreeNode *& cur) {
3    // Calculate the balance factor:
4    int balance = height(cur->right) - height(cur->left);
5
6    // Check if the node is currently not in balance:
7    if ( balance == -2 ) {   <- heavy to left
8      int l_balance =   // check left subtree balance
9            height(cur->left->right) - height(cur->left->left);
10     if ( l_balance == -1 ) { ____Right Rotation____; }
11     else                   { ____Left Right Rotation____; }
12   } else if ( balance == 2 ) {   <- heavy to right
13     int r_balance =  // check right subtree balance
14           height(cur->right->right) - height(cur->right->left);
15     if( r_balance == 1 ) { ____Left Rotation____; }
16     else                 { ____Right-Left Rotation____; }
17   }
18
19   _updateHeight(cur);
20 };
21
```

Rotations

```
1  template <typename K, typename V>
2  void AVL<K, V>::rotateLeft(TreeNode *& cur) {
3    // Modify Pointers
4       TreeNode* tmp = cur->right
5
6       cur->right = cur->right->left;
7
8       tmp->left = cur
9
10
11      cur = temp
12
13
14
15
16
17   // Update Heights
18        cur -> height
19        cur -> left -> height
20
21
};
```




- When checking the balance, we will **never** have a case where the balance is more than 2

- The height of an AVL tree is log n
    - find is faster in an AVL tree bc it is shorter
    - insert/remove is faster in a KD tree

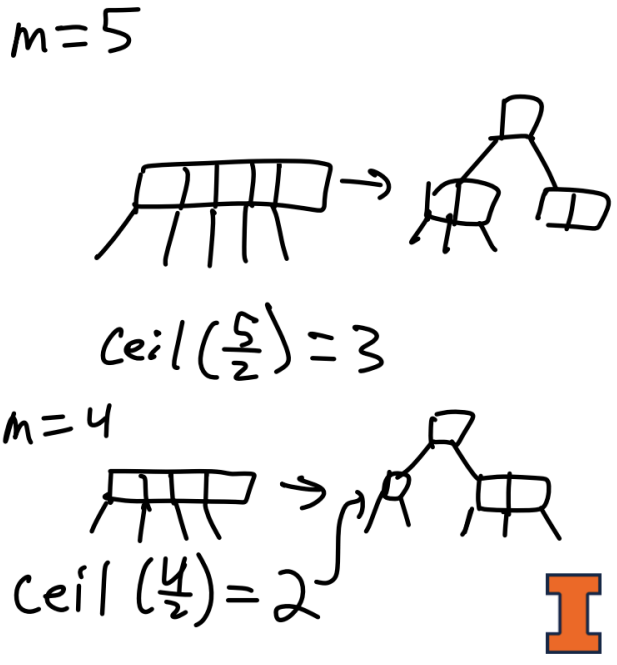
Runtimes (Up to Week 7)

| Operation | Sorted Array | Unsorted Array | Sorted Linked-List | Unsorted Linked-List | Binary Tree | BST | AVL |
|---|---|---|---|---|---|---|---|
| find | $O(log\ n)$ | $O(n)$ | $O(n)$ | $O(n)$ | $O(n)$ | $O(h)$, $O(n)$ | $O(log\ n)$ |
| insert | $O(n)$ | $O(n)$, $O(1)$* | $O(n)$ | $O(1)$ | $O(1)$ | $O(h)$, $O(n)$ | $O(log\ n)$ |
| delete | $O(n)$ | $O(n)$ | $O(n)$ | $O(n)$ | $O(n)$ | $O(h)$, $O(n)$ | $O(log\ n)$ |
| traverse | $O(n)$ | $O(n)$ | $O(n)$ | $O(n)$ | $O(n)$ | $O(n)$ | $O(n)$ |

$n$ = # of elements

# BTrees (Week 7)

- The goal of a BTree is to minimize the number of seeks (shorter tree) and have more data at each node
- Like an AVL tree, it is self-balancing and still has the search property
- However, a BTree has more nodes per level and more children for each node
- A BTree of order m is an m-way tree
    - All keys in a node are ordered
    - All nodes contain no more than m-1 keys
    - All internal nodes have one more child than keys
    - All leaves are on the same level
- Insertions always start at the leaf
    - Split (throw up the median) if m keys are reached in a node
    - Split can make its way up the tree recursively (new root keeps getting pushed up)
- The root node can be a leaf or have $[2,\ m]$ children
    - This is the case because it means it split at some point
- All non-root internal nodes have $[\ \lceil \frac{m}{2} \rceil,\ m]$ children



## BTree Search

```
1  template <typename K, typename V>
2  V BTree<K, V>::_find(BTreeNode * node, const K & key) const {
3    unsigned i;
4    for (i = 0; i < node.keys_ct_ && key > node.keys_[i]; i++) { }
5
6    if (i < node.keys_ct_ && key == node.keys_[i]) {
7      return node.values_[i];
8    }
9
10   if (node.isLeaf()) {
11     return V();
12   }
13
14   BTreeNode nextChild = node._fetchChild(i);
15   return _find(nextChild, key);
16 }
```

stopping at end

node equal or larger

found key

return not found

recursively call on child

seek

- A BTree is a tree of arrays!
- BTree Find
  - Base:
    - If root is empty, return
    - If leaf, do array find an return
  - Recursive:
    - Array find for match or the first greatest value
    - Recurse on the appropriate child
  - The index of the first greater value is the index of the child we want to visit
- BTree Insert
  - Split when m number of nodes is reached

# Heaps (Week 8)

- A complete binary tree T is a min-heap if
  - T is null
  - T has a root and a subtree where the root is less than its children and the subtrees are also min-heaps
- A complete tree can be described without pointers
  - This can happen through arrays
- It is better to keep an empty element at the front of the array
  - This allows the insert time to be $O(1)$ instead of $O(1)$*
- The children of a "node" can be accessed by finding $2i$ or $2i + 1$
  - Conversely, the parent of a "node" can be accessed by doing $\lfloor \frac{i}{2} \rfloor$
- For insertions:
  - Insert at the back of the array
  - Correct the min-heap if the new item breaks the properties
  - Swap with the parent if necessary
    - Steps 2 and 3 are recursive!
  - HeapifyUp has an $O(log\ n)$ runtime
- For removeMin:
  - Swap root with the last item in the array, then remove the min value
  - Restore the heap property using HeapifyDown on the root
    - If child is smaller than node, swap node with the smallest child (recursive process)
  - HeapifyDown has an $O(log\ n)$ runtime
- For buildHeap (3 ways):
  - A sorted array IS a min-heap
    - Runtime will depend on the sorting algorithm used, which should optimally be $O(n\ log\ n)$
  - Use heapifyUp from the first item (can't use last item because it is only true for already sorted arrays)
    - Start at either 1 or 2 and end at index size_−1 (1 is a base case)
    - Will be $O(n\ log\ n)$ for building
  - Use heapifyDown from the bottom (FAST)
    - Will be $O(n)$ for building

- Overall, the min-heap has:
    - $O(n)$ for construction
    - $O(log\ n)$ for insert
    - $O(log\ n)$ for remove
- The array has an efficient insert/remove back, array swaps, and improved construction time
- Heap Sort has an $O(n\ log\ n)$ runtime
    - Build heap - $O(n)$
    - Call removeMin n times - $O(n\ log\ n)$
    - Reverse the array as needed - $O(n)$

# Disjoint Sets (Week 9)

- Structure matters more in disjoint sets
- Functions:
    - find() returns a set representation of the element
        - Checks relationships between elements
    - union() merges two sets of elements
        - Called when elements are not already in the same set
    - makeSet() creates a set of elements
- Key ideas:
    - Every item exists in exactly one set
    - Every item in each set has the same representation
    - Every set has a different representation
- Canonical items are like iterators and represent a set
- One way to implement is to store the canonical keys as indices in an array
    - find() has random access $\rightarrow O(1)$
    - union() needs to walk across the array to modify the canonical items $\rightarrow O(n)$
- Another way to implement is storing canonical keys as -1
    - find() walks up the chain to find the canonical item $\rightarrow O(n)$
    - union() changes one value to be the canonical item $\rightarrow O(1)$
        - O(n) if value is not given
- For this, we need an UpTree
    - An UpTree is an array implemented as a tree
    - All set elements point to the index of their respective canonical item
    - The best case UpTree is $O(h)$, and the worst case is $O(n)$
- UpTree functions:
    - find() checks if it is canonical; otherwise, it calls itself to find the canonical item $\rightarrow O(h)$
    - union() [without find] sets one canonical item equal to another $\rightarrow O(1)$
    - Smart union stores the negative height instead of -1 inside canonical items
        - Store $-1 * (height + 1)$
        - With a smart union, a union by size or height will both return a height of $O(log\ n)$
        - This causes find() and union() to both be $O(log\ n)$

- Applying path compressions allows both runtimes to reach $O(1)$*
- Union by rank:
    - If the rank of two UpTrees is the same, the merged UpTree has rank+1
    - If one of the UpTrees has a bigger rank than the other, the rank of the merged UpTree is the rank of the bigger one
- By using iterative logs, we reach the Inverse Ackermann constant (according to CS), making union and find $O(1)$*

# Graphs (Week 9 & 10)

- A tree is a type of graph that is acyclic and has a clear root
- A graph has no root and may contain cycles
- Graphs are one of the most useful data structures
- Graph vocab:
    - Vertex: nodes of a graph
    - Edges: connections between nodes
    - Degree: # of edges touching a vertex
        - In a directed graph, you can have an in-degree and an out-degree
    - Adjacency: two vertices are adjacent if they are connected by an edge
    - Path: sequence of vertices/edges between two nodes
    - Cycle: a path from a node to itself
    - Simple graph: no self-loops or multi-edges
    - Directed: edges are one-way connections
    - Undirected: traversable in either direction
    - Reachability: a vertex $v_2$ is reachable from $v_1$ if there is a path from $v_1$ to $v_2$
    - Weights: a value associated with an edge
    - Spanning Tree: a connected graph with no cycles
        - Contains $n - 1$ edges to connect n vertices
- Graphs are a set of vertices and a set of edges
    - A subgraph is a subset of vertices and edges from the original graph
    - Complete subgraph: every pair of vertices is adjacent
    - Connected subgraph: a path exists between every pair of vertices
    - Connected components: a connected subgraph that is not part of a larger subgraph (every node in the subgraph is reachable)
- Minimum edges:
    - Unconnected graph → 0
    - Connected (simple) graph → $n - 1$
- Maximum edges:
    - Connected simple graph → $(n - 1) + (n - 2) + \ldots = \frac{n(n-1)}{2}$
- So, the <u>number</u> of edges is $n - 1 \leq m \leq \frac{n(n-1)}{2}$

- Functions:
    - insertVertex()
    - insertEdge()
    - removeVertex()
    - removeEdge()
    - getEdges()
    - areAdjacent()
    - origin()
    - destination()
- For the following section, $n = |V|$ and $m = |E|$
    - incidentEdges = get all edges
- Edge List
    - Overview:
        - insertVertex and insertEdge are $O(1)$*
        - removeVertex, removeEdge, incidentEdges, areAdjacent are all $O(m)$
    - Pros:
        - Very fast vertex/edge insert
        - Easy to implement and use
        - Storage costs are minimal
    - Cons:
        - Most graph access functions are slow $\rightarrow O(m)$
- Adjacency Matrix
    - Overview:
        - insertEdge, removeEdge, and areAdjacent are $O(1)$
        - incidentEdges is $O(n)$
        - insertVertex and removeVertex are $O(n)$-$O\left(n^2\right)$
            - Tombstoning (“procratinating”) can be used for removing rows/columns after a vertex is removed and say it is “deleted”
        - Space is always $O\left(n^2\right)$
    - Pros:
        - Very fast edge lookup/modification
    - Cons:
        - Takes lots of storage (worst memory cost for sparse graphs)
        - Adding vertices (or removing) is bad

- Adjacency List
  - Overview:
    - Vertices are stored in a map with linked list values
    - Each node is a pointer to an edge in an edge list
    - incidentEdges is $O(deg(v))$
    - areAdjacent is $O(min[deg(v_1), deg(v_2)])$
    - insertEdge is $O(1)$*
    - removeEdge is $O(min[deg(v_1), deg(v_2)])$

**|V|= n, |E|= m**

| Expressed as O(f) | Edge List | Adjacency Matrix | Adjacency List |
|---|---|---|---|
| Space | n+m | $n^2$ | n+m |
| insertVertex(v) | 1* | n* | 1* |
| removeVertex(v) | n+m | n | deg(v) |
| insertEdge(u, v) | 1 | 1 | 1* |
| removeEdge(u, v) | m | 1 | min( deg(u), deg(v) ) |
| incidentEdges(v) | m | n | deg(v) |
| areAdjacent(u, v) | m | 1 | min( deg(u), deg(v) ) |

# Graph Algorithms

## Search Algorithms (Week 10 & 11)

- To complete a traversal, we need…
    - A starting vertex
    - A way to keep track of nodes visited
    - A way to keep track of the current node

- BFS Traversal:
    - Steps:
        - Initialize a queue (to track nodes to visit)
        - Create labels for distance from start and a predecessor
        - While !queue.empty()
        - Remove from queue and process all children (label dist and prev, then add unvisited children)
    - The runtime is $O(V + E)$ because it looks through every node AND edge
    - The traversal depends on the start position, along with the order of edges at each node
    - BFS can count connected components and detect cycles
        - A single cross-edge says we have a cycle (edge labels)
    - BFS distance is always the shortest distance from the source to any vertex (vertex labels)
    - The endpoints of a cross edge never differ in distance by more than 1

- DFS Traversal:
    - Steps:
        - Initialize a stack (to track nodes to visit)
        - Create labels for distance from start and a predecessor
        - While !stack.empty()
        - Access top from stack and process one child (label dist and prev, then add unvisited children)
        - pop() if no unvisited children
    - DFS can make a spanning tree
    - Distance does not have meaning here
    - Edge labels still have meaning and still tell us cycles
    - The runtime is $O(E)$ because it visits every vertex once and looks at all children

- BFS vs DFS:
    - BFS solves unweighted MST, shortest path, cycle detection
        - Memory is bounded by tree width
    - DFS solves unweighted MST and cycle detection
        - Memory is bounded by longest path

# MST Algorithms (Week 11)

- The input graph must be connected, undirected, and have edge weights
    - If there are no edge weights, MST would be solved by BFS or DFS
- The output is a graph that is a spanning graph of G, is connected and acyclic, and has a minimum total weight among all spanning trees
- In a connected graph, the number of edges is bounded by $n - 1 \leq m \leq n^2$

- Kruskal's Algorithm:
    - This algorithm is globally greedy
    - For this algorithm, we need the global minimum edge weight, and if two vertices are already connected
    - This means we need a priority queue of edges (sorted by weight) and a disjoint set of vertices
        - The priority queue can be a minheap or a sorted array
            - Minheap shrinks over time and applies heapifyDown
            - Sorted array is not destroyed during the algorithm (useful if we need a sorted array for applications later)
        - The disjoint set starts with the vertices as their own sets
    - This stops when n-1 edges are recorded (minimum edges to connect a graph of n vertices)
    - The runtime of this algorithm is $O(n + m\,log(m))$

## Kruskal's Algorithm

```
1  KruskalMST(G):
2    DisjointSets forest
3    foreach (Vertex v : G.vertices()):
4      forest.makeSet(v)
5
6    PriorityQueue Q    // min edge weight
7    Q.buildFromGraph(G.edges())
8
9    Graph T = (V, {})
10
11   while |T.edges()| < n-1:
12     Vertex (u, v) = Q.removeMin()
13     if forest.find(u) != forest.find(v):
14        T.addEdge(u, v)
15        forest.union( forest.find(u),
16                      forest.find(v) )
17
18   return T
19
```

**|V|= n, |E|= m**

**What is the Big O?**

2 — 4: O(n)

6 — 7: Heap: O(m)
Sorted List: O(m log m)

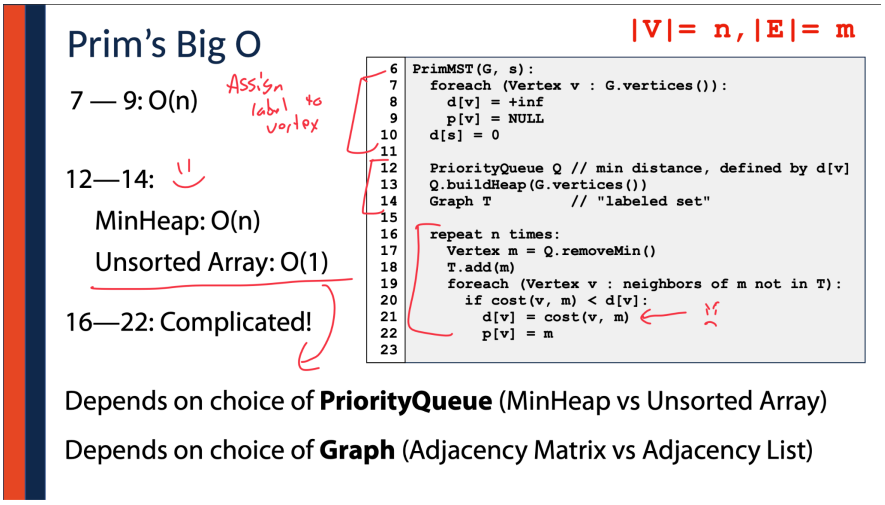
11: m x <12-17>

12—17: Heap: O(log m)
Sorted List: O(1)

$O(1 + \cancel{n} + m \log m)$

Disjoint set we treat as O(1) b/c path compression w/ smart union

- Prim's Algorithm:
  - This algorithm is <u>locally</u> greedy
  - For this algorithm, we need a source node. We check its neighbors, then its neighbors, and so on
  - This means we need a priority queue of nodes and a map of distances with respect to their neighbors
    - The priority queue can be a minheap or an <u>unsorted</u> array
  - This stops after n loops (all nodes are connected)
  - The runtime of this algorithm is complicated




  - With a heap, the runtime is $O(n^2 + m \log n)$ in an adjacency matrix and $O(n \log n + m \log n)$ in an adjacency list
    - This is a horrible runtime because we need to rebuild the heap m times
  - With an unsorted array, the runtime is $O(n^2)$
    - This is because we go through each vertex's neighbors n times
  - Use an adjacency list heap for sparse graphs and an unsorted array for dense graphs

- Kruskal vs. Prim

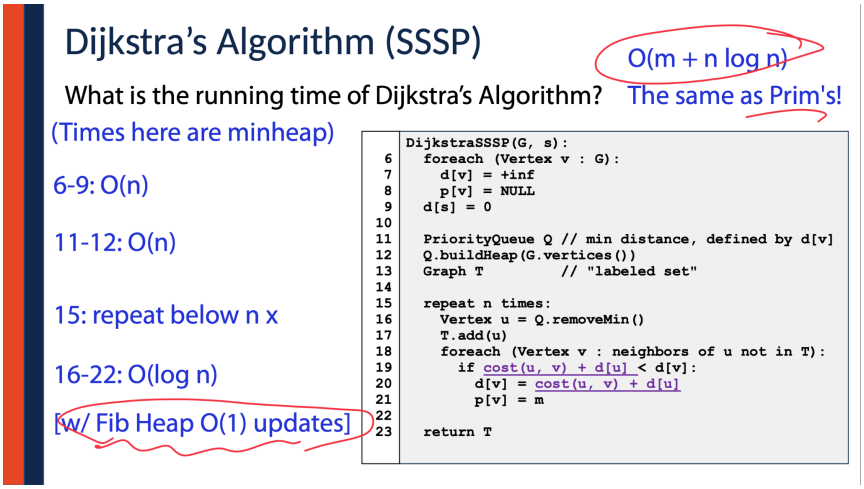
  In a sparse graph, Kruskal's is $O(n + n \log n)$ and Prim's is $O(n \log n + n \log n)$

  In a dense graph, Kruskal's is $O(n + n^2 \log n)$ and Prim's is $O(n^2 \log n)$

# SSSP Algorithms (Week 11 & 12)

- The input graph must be connected, undirected, and have edge weights
    - If there are no edge weights, SSSP would be solved by BFS
- The output is a graph with the shortest weighted path from the source node to the end position

- Dijkstra's Algorithm
    - Solves the single-source shortest path problem
    - This algorithm is very similar to Prim's!
        - The key difference is comparing the edge weights and cost to determine if a detour is better than the original path direction
        - We have distance to v, so is it better to detour to u to arrive at v?
    - This guarantees we will not visit any vertex except through an optimal path
    - The runtime is the exact same as Prim's! $\rightarrow O(m + n \log n)$




    - We will only visit nodes through their smallest path
        - If a length is smaller, we will always visit that first
    - This algorithm does NOT handle negative edge weights well (becomes a loop until negative infinity)
        - Without a cycle, we can handle it, but it is very inefficient
        - This happens because we assume that the next item out of the priority queue is the next smallest item; negative weights break this assumption

- Floyd-Warshall Algorithm
    - This algorithm can handle negative-weight edges (but not negative-weight cycles)
    - Checks every permutation systematically to determine the best path
    - The runtime of this algorithm is $O\left(n^3\right)$

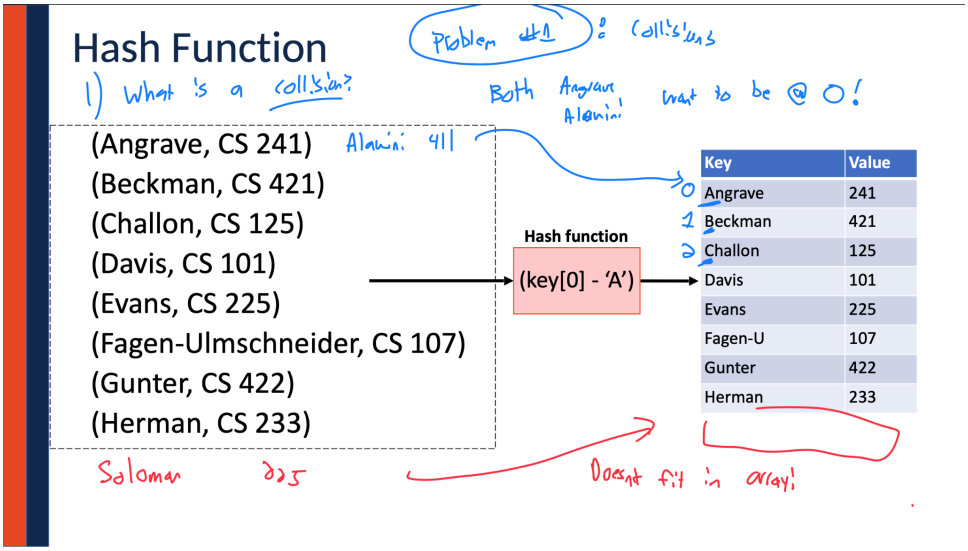
# Probability in CS (Week 12)

- A randomized algorithm is an algorithm that uses a source of randomness somewhere in its implementation
- This is useful if we don't want to brute force the algorithm without sacrificing lots of storage and memory
- One example is Quick Primes with Fermat's Primality Test
    - Prime numbers are correctly identified as prime, and non-prime numbers have a very small, non-zero chance of being identified as prime in $O(1)$ time
- Cases of randomization in algorithms
    - Assume input data is random to estimate average-case performance → NEVER DO THIS
    - Use randomness inside an algorithm to estimate the expected running time
        - Will work 100% of the time, but may be slow
    - Use randomness inside an algorithm to approximate a solution in a fixed time
        - Runs fast, but may not be correct
        - An example of this is selecting a Quicksort pivot

# Hash Tables (Week 12 & 13)

- Sometimes a data structure is too ordered/structured
  - An AVL tree can find in $O(log\ n)$ time, but as a trade-off, it costs $O(log\ n)$ to insert
- Randomized data structures rely on <u>expected</u> performance
  - This cheats trade-offs!

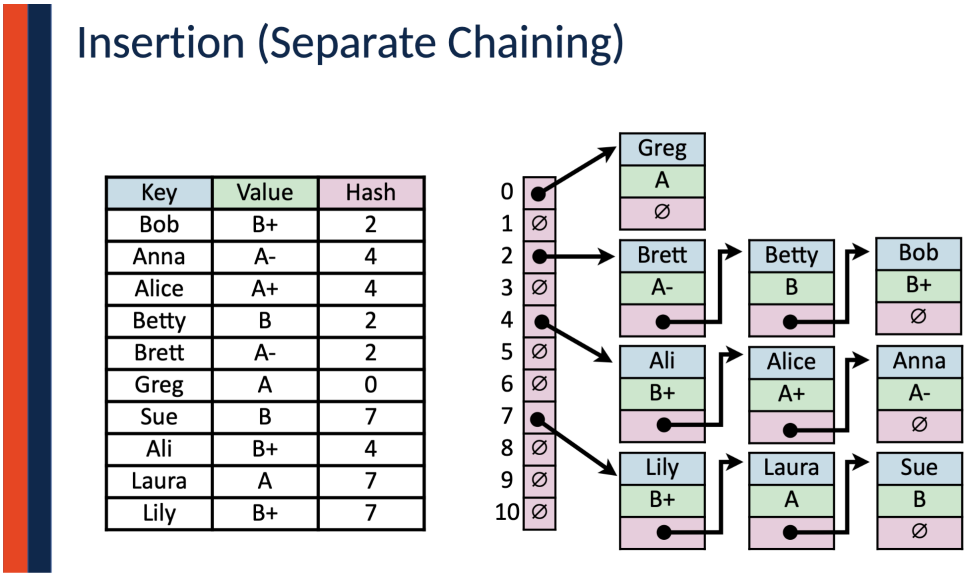
- A Hash Table consists of three things:
  - A hash function $h(k)$: $k \rightarrow int$
  - A data storage structure, which is an array
  - Some way to handle hash collisions
- A hash function maps a key space to integers (universe of keys)
- A hash function must be:
  - Deterministic: $(\forall k_1, k_2)\ ((k_1 == k_2) \Rightarrow h(k_1) = h(k_2))$
  - Efficient: $O(1)$
  - Defined for a certain size table:




- A general hash function should have a hash and a compression

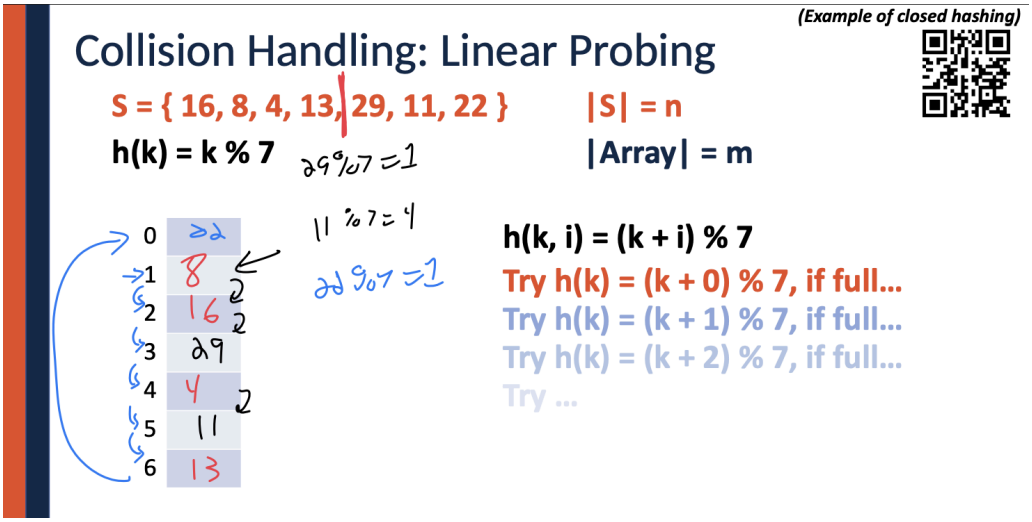
- A hash collision occurs when multiple keys hash to the same value
- Simple Uniform Hashing Assumption (SUHA) is uniform and independent
    - All positions in the hash table are equally likely for any input $(\frac{1}{m})$
    - All key's hash values are independent of other keys
- Load factor (collision chance): $\alpha = \frac{\#\ items}{\#\ positions} = \frac{n}{m}$
    - Open hashing: $0 \leq \alpha \leq \infty$
    - Closed hashing: $0 \leq \alpha < 1$

- There are two ways to handle hash collisions: open hashing and closed hashing
- Open hashing: stores KV pairs externally to the hash table
    - Resolved by adding to a linked list (separate chaining)
    - Can store infinite items




    - Results in $O(1)$ insert, but $O(n)$ find and remove (by key) → BAD
    - Adding randomness can fix this
        - After SUHA fixes the hash, find() and remove() are expected $1 + \alpha$, where $1$ is the $O(1)$ lookup and $\alpha$ is the linked list lookup
        - $\alpha$ is the expected length of each linked list

- Closed hashing: stores KV pairs internally in the hash table
  - Resolved by adding to an array (open addressing)
  - Can store no more than $m$ items
  - Linear probing places an element at an index (or at the next available one if it is occupied)




    - find() - Hash the input key, look at the position
      - If present, return the value; otherwise, look at the next available space
      - Stop when the object is found, we searched every position, or we find a blank space
    - remove() - Hash the input key, find the actual location (if it exists), and remove the object at the hash value
      - Use tombstoning to avoid resizing the array
  - Quadratic probing changes insertions at the next available index
    - This can be more problematic because we might take more than $m$ steps to find an available space
    - Individual collisions still cause large chains
  - Double hashing adds a second hash function to handle collisions
    - Less consistent, but still more deterministic
    - This decreases the probability of collisions by a lot
    - However, this can lead to an infinite loop if the hash functions are not selected carefully
      - The table size has to be a prime number
      - $h_2(k)$ CANNOT be 0
      - $h_1$ and $h_2$ must be independent of each other

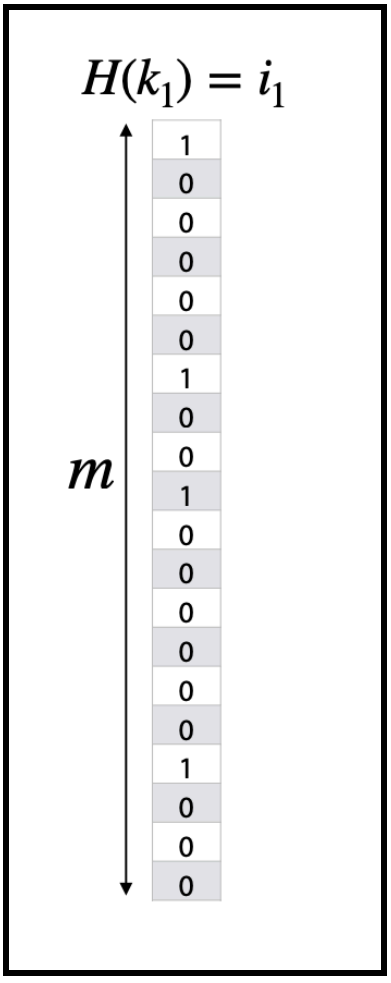
- Runtimes for Open Hashing
    - insert() $\rightarrow 1$
    - find()/remove() $\rightarrow 1 + \alpha$
- Runtimes for Closed Hashing
    - insert() $\rightarrow \frac{1}{1-\alpha}$
    - find()/remove() $\rightarrow \frac{1}{1-\alpha}$
- As $\alpha$ increases, the runtime approaches $\infty$
- If $\alpha$ is constant, the running time is constant
- To resize a hash table, double the size of the array to the nearest prime, change compression, and rehash every item

- The hash table is overall $O(1)$***
    - Star 1: Picking a random hash (expectation)
    - Star 2: SUHA
    - Star 3: pseudo-amortized (resizing hash table)
- Open hashing is better for big records, while closed hashing is better for structure speed

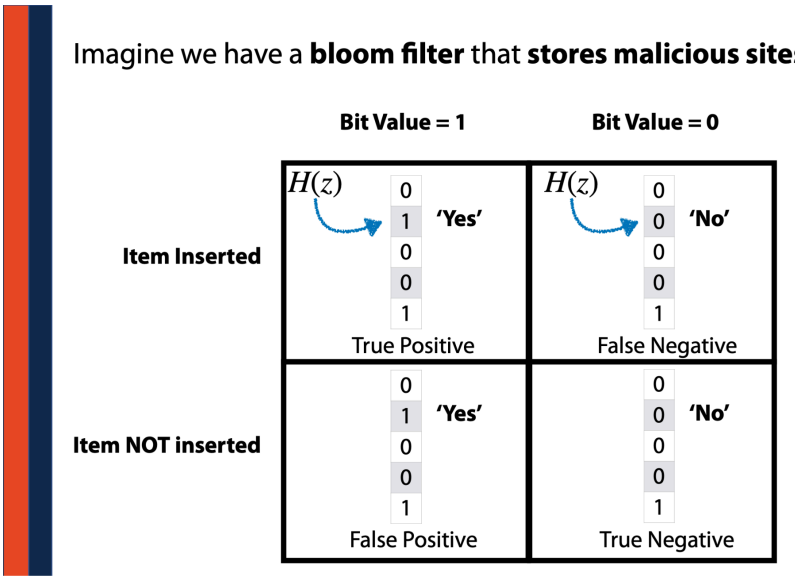
# Bloom Filters (Week 14)

- AVL Trees, unsorted arrays, and hash tables are the big takeaways from Data Structures
    - All are somewhat different, but are all bounded by $O(n)$ for memory
- How can we build a search index on a collection of objects in a memory-bound environment?
    - Throw out information that isn't needed → lossy data compression
    - Compress the dataset → lossless compression
    - This can reduce storage costs!

- A bloom filter is a hash table WITHOUT its key-value pairs and is also a probabilistic data structure
    - Constructor
        - Give one or more hash functions and the input array size
    - Insert:
        - Hash the item
        - Insert by setting the bit from 0 to 1
        - If there's a collision, ignore it because the bit is already 1 (Any bit set to 1 stays 1)
        - Results in $O(1)$ runtime!
    - Find:
        - Hash the item
        - Lookup the value of the bit
        - If the value is 0, there is a 100% chance the item is not in the dataset. If the value is 1, the item might be present, but it could be a collision.
        - This is an $O(1)$ process
    - The tradeoff with the bloom filter is that we don't track hash collisions. Because of that, we cannot delete items.



- Probabilistic Accuracy:
  - True positive - The outcome is true, and it is actually true
  - True negative - The outcome is false, and it is actually false
  - False positive - The outcome is true, but it is actually false
  - False negative - The outcome is false, but it is actually true




  - A false negative is the worst flag because there is something wrong, but the model does not detect it!
    - For bloom filters, this is when an item is inserted, but when we call find, there is nothing there, even though there should be something there
  - A one-sided error is when <u>either</u> a false positive or a false negative is possible
    - If any of the $k$ different bloom filters returns 0, the item definitely does not exist (100% chance it does not exist)
    - If all the $k$ different bloom filters return 1, the item may or may not exist (never a 100% chance that the item exists because it may be the result of collisions)
  - Expected FPR: $\left(1-\left(1-\frac{1}{m}\right)^{nk}\right)^{k} \approx \left(1-e^{\frac{-nk}{m}}\right)^{k}$
    - When determining the false positive rate, it is best to have a $k$ (number of hash functions) that is not too big or too small
- Bloom filters can be merged using the bitwise union (| operation) or the bitwise intersection (& operation)
- Bloom filters cannot be resized because they are a lossy data structure

# Cardinality and MinHash (Week 14 & 15)

- Cardinality is a measure of how many unique items are in a set
  - In small datasets, just add the elements to std::set and take the size
  - However, it is not realistic to count the number of objects in a large dataset, so we must estimate
- Imagine a hat with a random subset of numbered cards from 0 to 999
  - If the minimum number is 95, this tells us that there are about 10 cards in the hat… but how?
  - Let min = 95 and assume there is a uniform distribution. We want to estimate the cardinality of the set.
    - This is given by $1000 = 95(N + 1)$, where $N$ is the number of items and $N + 1$ is the number of partitions

- Imagine we have a SUHA hash $h(k)$ over the range $m$
  - Inserting a new key is equivalent to adding a card to the hat
  - Tracking only the minimum value is a sketch that estimates the cardinality
  - By the definition of SUHA, $E[X_{N+1}] = \frac{1}{N+1}$ (chance of being the smallest item)
  - To get a better estimate, run more trials
    - This gives $E[M_k] = \frac{k}{N+1}$
- Given any dataset and a SUHA hash function, we can estimate the number of unique items by tracking the $k$-th minimum hash value
  - Higher $k \rightarrow$ better estimate, but higher storage cost (and slower runtime)
- To measure the similarity of sets $A$ & $B$, we need to measure how similar the sets are in relation to the total size of both sets
  - Jaccard coefficient: $J = \frac{A \cap B}{A \cup B}$, where $0 \leq J \leq 1$
  - A higher Jaccard coefficient means the sets are more similar
- The MinHash sketch is built by tracking $k$ minima, but only uses the $k$-th minima to get cardinality
- To construct a MinHash sketch, we need…
  - A dataset
  - A hash function
  - Some size $k$

- Methods of sketching
    - BF sketch: hash every item once and store it in a bloom filter
    - Cardinality sketch: hash every item once and store the $k$-th minimum value
    - MinHash sketching: hash every item once and store the bottom $k$ MinHash values
- MinHash sketching allows us to estimate cardinality and set similarity at the expense of losing the original dataset