Announcements

- Current progress on evaluations: 51%
- WA4 should be released tonight
<table>
<thead>
<tr>
<th>Analysis Tools/Techniques</th>
<th>ADTs</th>
<th>Data Structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asymptotic Analysis, (Unqualified) Runtime Bounds</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amortized Runtime</td>
<td>Seq, Buffer</td>
<td>ArrayBuffer</td>
</tr>
<tr>
<td>Recursive analysis, divide and conquer, Average/Expected Runtime</td>
<td>Seq</td>
<td>Linked Lists</td>
</tr>
<tr>
<td></td>
<td>Stack, Queue, PriorityQueue</td>
<td></td>
</tr>
<tr>
<td>Analysis Tools/Techniques</td>
<td>ADTs</td>
<td>Data Structures</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------</td>
<td>-----------------</td>
</tr>
<tr>
<td></td>
<td>Stack, Queue, PriorityQueue</td>
<td></td>
</tr>
<tr>
<td>Graphs</td>
<td>EdgeList, Adjacency List, Adjacency Matrix</td>
<td></td>
</tr>
<tr>
<td>Heaps, Trees</td>
<td>BST, AVL Tree, Red-Black Tree</td>
<td></td>
</tr>
<tr>
<td>HashTables</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Miscellaneous</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Sequences (what can you do with them?)

- Enumerate every element in sequence
  - ie: print out every element, sum every element
- Get the "nth" element
  - ie: what is the first element? what is the 42nd element?
- Modify the "nth" element
  - ie: set the first element to x, set the third element to y
Abstract Data Types (ADTs)

- The specification of what a data structure can do

What's in the box? ...we don't know, and in some sense...we don't care

Usage is governed by **what** we can do, not **how** it is done
The Seq ADT

apply(idx: Int): [A]
   Get the element (of type A) at position idx

iterator: Iterator[A]
   Get access to view all elements in the sequence, in order, once

length: Int
   Get the number of elements in the seq
The `mutable.Seq` ADT

- **apply(idx: Int): [A]**
  - Get the element (of type `A`) at position `idx`

- **iterator: Iterator[A]**
  - Get access to view all elements in the sequence, in order, once

- **length: Int**
  - Count the number of elements in the seq

- **insert(idx: Int, elem: A): Unit**
  - Insert an element at position `idx` with value `elem`

- **remove(idx: Int): A**
  - Remove the element at position `idx`, and return the removed value
Array [T] : Seq[T]

What does an Array of n items of type T actually look like?

- 4 bytes for n (optional)
- 4 bytes for sizeof(T) (optional)
- $n \times \text{sizeof(T)}$ bytes for the data

```
| n  | sizeof(T) | a(0) | a(1) | a(2) | a(3) | a(4) | ... |
```
What does an `ArrayBuffer` of `n` items of type `T` actually look like?

- 4 bytes for `n` (optional)
- 4 bytes for `sizeof(T)` (optional)
- 4 bytes for the number of used fields
- `n * sizeof(T)` bytes for the data

| n | `sizeof(T)` | u | a(1) or None | a(2) or None | a(3) or None | a(4) or None | ... |
Linked Lists

HEAD

None
Linked Lists

HEAD

A

None
Linked Lists

HEAD

A → B → None
Linked Lists

A ➔ B ➔ C ➔ None

HEAD ➔ A ➔ B ➔ C ➔ None
Linked Lists

HEAD

A -> B 

C 

F 

E 

D 

H 

G 

I 

J 

K 

L 

None
Implementing apply

For Array and ArrayBuffer:
- Let $a$ be the memory address of the first element of the array
- Let $s$ be the size of each element in the array
- Then we know element $i$ is located at address $a + s \times i$

For LinkedList:
- We know where the first element is located (and maybe the last)
- We have no idea where the $i^{th}$ element is
- All we can do is follow the references until we get there
Comparing Random Access for Array vs List

Array

List
Comparing Random Access for Array vs List

Array

List

Let’s ignore the specific numbers and clean things up a bit...
Comparing Random Access for Array vs List
Comparing Random Access for Array vs List

What differentiates these two algorithms is how they scale with input size (the shape of the function)
Idea: Capture this behavior by treating the number of steps as a function of the input size.
Growth Functions

Not a function in code...but a mathematical function:

\[ f(n) \]

n: The “size” of the input

ie: number of users, rows, pixels, etc

f(n): The number of “steps” taken for input of size n

ie: 20 steps per user, where n = |Users|, is 20 x n
Some Basic Assumptions:

Problem sizes are non-negative integers

\[ n \in \mathbb{Z}^+ \cup \{0\} = \{0, 1, 2, 3, \ldots\} \]

We can’t reverse time... (obviously)

\[ f(n) > 0 \]

Smaller problems aren’t harder than bigger problems

\[ \forall n_1 < n_2, f(n_1) \leq f(n_2) \]
Runtime as a Function
Which is better? $3x|\text{Users}| + 5$ or $|\text{Users}|^2$
Attempt #3: Asymptotic Analysis

Case 1: $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$  
(f grows faster; g is better)

Case 2: $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$  
(g grows faster; f is better)

Case 3: $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \text{some constant}$  
(f and g “behave” the same)
Goal of “Asymptotic Analysis”

We want to organize growth functions into different Complexity Classes

Within the same complexity class, functions “behave the same”

To do this, focus on the dominating term...
## Why Focus on Dominating Terms?

<table>
<thead>
<tr>
<th>( f(n) )</th>
<th>10</th>
<th>20</th>
<th>50</th>
<th>100</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \log(\log(n)) )</td>
<td>0.43 ns</td>
<td>0.52 ns</td>
<td>0.62 ns</td>
<td>0.68 ns</td>
<td>0.82 ns</td>
</tr>
<tr>
<td>( \log(n) )</td>
<td>0.83 ns</td>
<td>1.01 ns</td>
<td>1.41 ns</td>
<td>1.66 ns</td>
<td>2.49 ns</td>
</tr>
<tr>
<td>( n )</td>
<td>2.5 ns</td>
<td>5 ns</td>
<td>12.5 ns</td>
<td>25 ns</td>
<td>0.25 µs</td>
</tr>
<tr>
<td>( n\log(n) )</td>
<td>8.3 ns</td>
<td>22 ns</td>
<td>71 ns</td>
<td>0.17 µs</td>
<td>2.49 µs</td>
</tr>
<tr>
<td>( n^2 )</td>
<td>25 ns</td>
<td>0.1 µs</td>
<td>0.63 µs</td>
<td>2.5 µs</td>
<td>0.25 ms</td>
</tr>
<tr>
<td>( n^5 )</td>
<td>25 µs</td>
<td>0.8 ms</td>
<td>78 ms</td>
<td>2.5 s</td>
<td>2.9 days</td>
</tr>
<tr>
<td>( 2^n )</td>
<td>0.25 µs</td>
<td>0.26 ms</td>
<td>3.26 days</td>
<td>( 10^{13} ) years</td>
<td>( 10^{284} ) years</td>
</tr>
<tr>
<td>( n! )</td>
<td>0.91 ms</td>
<td>19 years</td>
<td>( 10^{47} ) years</td>
<td>( 10^{141} ) years</td>
<td>🤯</td>
</tr>
</tbody>
</table>
Asymptotic Analysis
Asymptotic Analysis

- Functions that grow faster
- Functions that grow "at the same rate"
- Functions that grow slower
Asymptotic Analysis

\( \Theta(f) \) is the set of functions that grow at the same rate.

If \( g(n) \in \Theta(f) \) then \( g \) and \( f \) behave the same.
Asymptotic Analysis

$O(f)$ is the set of functions that grow slower (or the same as) $f$

If $g(n) \in O(f)$ then $g \leq f$
\( \Omega(f) \) is the set of functions that grow faster (or the same as) \( f \)

If \( g(n) \in \Omega(f) \) then \( g \geq f \)
Recap of Runtime Complexity

**Big-O — Upper Bound**
- Growth functions in the **same or smaller** complexity class
- \( f(n) \in O(g(n)) \) iff \( f(n) \leq c \cdot g(n) \) for some constant \( c \), and \( n > n_0 \)

**Big-\( \Omega \) — Lower Bound**
- Growth functions in the **same or bigger** complexity class
- \( f(n) \in \Omega(g(n)) \) iff \( f(n) \geq c \cdot g(n) \) for some constant \( c \), and \( n > n_0 \)

**Big-\( \Theta \) — Tight Bound**
- Growth functions are in the **same** complexity class
- \( f(n) \in \Theta(g(n)) \) iff \( f(n) \in O(g(n)) \) and \( f(n) \in \Omega(g(n)) \)
apply:

- Array/ArrayBuffer: $\Theta(1)$
- LinkedList: $O(n)$

Iteration:

- Array/ArrayBuffer: $\Theta(n)$
- LinkedList: $O(n)$
insert
- ArrayBuffer: $O(n)$ ← move elements and maybe resize
- LinkedList: $O(n)$ ← find the insertion point, then insert

prepend/append
- ArrayBuffer: $O(n)$ ← move elements and maybe resize
- LinkedList: $O(1)$ ← assuming we have head/tail, no need to search
Back to Seq

insert
- **ArrayBuffer:** $O(n)$ ← move elements and maybe resize
- **LinkedList: $O(n)$** ← find the insertion point, then insert

prepend/append
- **ArrayBuffer: $O(n)$** ← move elements and maybe resize
- **LinkedList: $O(1)$** ← assuming we have head/tail, no need to search

Most of these appends are $O(1)$...can we include that context in our analysis?
Cost of $n$ appends in a row...

$\Theta(X) + (X-1) \cdot \Theta(1)$ for $X$ appends

$X \cdot \Theta(1)$ for $X$ appends

$\Theta(X \cdot 2) + (X \cdot 2 - 1) \cdot \Theta(1)$ for $X \cdot 2$ appends
newLength = data.size * 2

So...how many red boxes for \( n \) inserts? \( \Theta(\log(n)) \)

How much work for box \( j \)?

\[
\Theta(IS \cdot 2^j) + \sum_{1}^{IS \cdot 2^j} \Theta(1) = \Theta(2^j)
\]

How much work for \( n \) inserts?

\[
\sum_{j=0}^{\Theta(\log(n))} \Theta(2^j)
\]

Total for \( n \) insertions: \( \Theta(n) \)
Amortized Runtime

If \( n \) calls to a function take \( O(T(n)) \)...

We say the **Amortized Runtime** is \( O(T(n) / n) \)

The **amortized runtime** of `append` on an `ArrayBuffer` is: \( O(n/n) = O(1) \)

The **unqualified runtime** of `append` on an `ArrayBuffer` is: \( O(n) \)
Aside on Summations

When analyzing code, we often have multiple steps one after another.

To determine the total runtime, we add the number of steps:

```
for i ← 0 to n, by -1:
    for j ← 0 to i:
        // do something
```

The above inner loop first does 1 iteration, then 2 iterations, then 3, then 4...

Total number of iterations: $1 + 2 + 3 + 4 + \ldots + n = \sum_{j=1}^{n} j$.

Understanding summation and summation rules are important!
What about algorithms with a random component, ie QuickSort?
QuickSort: Worst-Case Runtime

What is the worst-case runtime?

$$T_{\text{quick sort}}(n) \in O(n^2)$$

Remember: This is called the unqualified runtime...we don't take any extra context into account
Is the worst case runtime representative?

**No!** (the actual runtime will almost always be faster)

But what **can** we say about runtime?
QuickSort Runtime

Now we can write our runtime function in terms of random variables:

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n \leq 1 \\
T(0) + T(n - 1) + \Theta(n) & \text{if } n > 1 \land X = 1 \\
T(1) + T(n - 2) + \Theta(n) & \text{if } n > 1 \land X = 2 \\
T(2) + T(n - 3) + \Theta(n) & \text{if } n > 1 \land X = 3 \\
\vdots \\
T(n - 2) + T(1) + \Theta(n) & \text{if } n > 1 \land X = n - 1 \\
T(n - 1) + T(0) + \Theta(n) & \text{if } n > 1 \land X = n 
\end{cases}
\]
QuickSort Runtime

...and convert it to the expected runtime over the variable $X$

$$E[T(n)] = \begin{cases} 
\Theta(1) & \text{if } n \leq 1 \\
E[T(X - 1)] + E[T(n - X)] + \Theta(n) & \text{otherwise}
\end{cases}$$

This looks like the runtime of MergeSort, so now our hypothesis is that our **Expected Runtime** is $n \log(n)$
What guarantees do you get?

If \( f(n) \) is a **Tight Bound**
   The algorithm always runs in \( cf(n) \) steps

If \( f(n) \) is a **Worst-Case Bound**
   The algorithm always runs in at most \( cf(n) \)

If \( f(n) \) is an **Amortized Worst-Case Bound**
   \( n \) invocations of the algorithm **always** run in \( cnf(n) \) steps

If \( f(n) \) is an **Average Bound**
   ...we don't have any guarantees

← Unqualified runtime
A Bit More on Tight Bounds

If $f(n)$ is a Tight Upper Bound
Then $f(n)$ is an upper bound AND there is no smaller upper bound

If $f(n)$ is a Tight Lower Bound
Then $f(n)$ is a lower bound AND there is no larger lower bound

If $f(n)$ is a Tight Upper Bound AND a Tight Lower Bound
Then $f(n)$ is a tight bound, or a $\Theta$ bound
## Seq Summary So Far

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>apply(i)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(i), O(n)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>update(i, val)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(i), O(n)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>insert(i, val)</td>
<td>Θ(n)</td>
<td>O(n)</td>
<td>Θ(i), O(n)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>remove(i, val)</td>
<td>Θ(n)</td>
<td>Θ(n-i), O(n)</td>
<td>Θ(i), O(n)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>append(i)</td>
<td>Θ(n)</td>
<td>O(n), Amortized Θ(1)</td>
<td>Θ(i), O(n)</td>
<td>Θ(1)</td>
</tr>
</tbody>
</table>
Ways to Access Elements of a Sequence

By Index: Get the element at a particular *position*

By Reference: Get the element with a particular *reference*

(Search) By Value: Find the element with a particular *value* (or *key*)
- For an unsorted array or list, this takes $O(n)$ time (have to check all)
- For a sorted array, only need $O(\log n)$ steps
Variants on Sequences (more ADTs)

Stack
- LIFO: last in first out
- push elements to the top of the stack
- pop elements from the top of the stack

Queue
- FIFO: first in first out
- enqueue elements to the end of the queue
- dequeue elements from the front of the queue

PriorityQueue
- Elements ordered by priority
- dequeue removes the highest priority element
Recap

Stacks: Last In First Out (LIFO)
- Push (put item on top of the stack) \( \Theta(1) \) (or amortized \( O(1) \))
- Pop (take item off top of stack) \( \Theta(1) \)
- Top (peek at top of stack) \( \Theta(1) \)

Queues: First in First Out (FIFO)
- Enqueue (put item on the end of the queue) \( \Theta(1) \) (or amortized \( O(1) \))
- Dequeue (take item off the front of the queue) \( \Theta(1) \)
- Head (peek at the item in the front of the queue) \( \Theta(1) \)

Stacks and Queues can be easily implemented with Arrays and Linked Lists. Priority Queues can be...but not very efficiently
A New ADT... PriorityQueue

PriorityQueue[A <: Ordering]

enqueue(v: A): Unit
  Insert value v into the priority queue

decqueue: A
  Remove the greatest element in the priority queue

head: A
  Peek at the greatest element in the priority queue
Priority Queues

Two mentalities...

**Lazy:** Keep everything a mess ("Selection Sort")

**Proactive:** Keep everything organized ("Insertion Sort")
Lazy Priority Queue

**Base Data Structure:** Linked List

- `enqueue(v: A): Unit`
  - Append `v` to the end of the linked list. \( O(1) \)

- `dequeue/head : A`
  - Traverse the list to find the largest value. \( O(n) \)
Proactive Priority Queue

**Base Data Structure:** Linked List

- `enqueue(v: A): Unit`
  - Insert `v` in reverse sorted order. $O(n)$

- `dequeue/head : A`
  - Refer to the first item in the list. $O(1)$
Priority Queues

<table>
<thead>
<tr>
<th>Operation</th>
<th>Lazy</th>
<th>Proactive</th>
</tr>
</thead>
<tbody>
<tr>
<td>enqueue</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>dequeue</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>head</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

Can we do better?
**Idea:** Keep the priority queue "kinda" sorted.

Hopefully "kinda" sorted is cheaper to maintain than a full sort, but still gives us some of the benefits.
Binary Heaps

Organize our priority queue as a directed tree

**Directed:** A directed edge from $a$ to $b$ means that $a \geq b$

**Binary:** Max out-degree of 2 (easy to reason about)

**Complete:** Every "level" except the last is full (from left to right)

**Balanced:** TBD (basically, all leaves are roughly at the same level)

>This makes it easy to encode into an array (later today)
Valid Max Heaps
The Heap ADT

enqueue(elem: A): Unit  
Place an item into the heap  \[\text{[AKA } \text{pushHeap]}\]

dequeue: A  
Remove and return the maximal element from the heap  \[\text{[AKA } \text{popHeap]}\]

head: A  
Peek at the maximal element in the heap

length: Int  
The number of elements in the heap
Heap.enqueue

**Idea:** Insert the element at the next available spot, then fix the heap.

1. Call the insertion point `current`
2. While `current != root` and `current > parent`
   a. Swap `current` with `parent`
   b. Repeat with `current ← parent`
Heap.enqueue

What if we enqueue 6?
What if we enqueue 6?
Place in the next available spot
What if we enqueue 6?
Swap with parent if it is bigger than the parent
Heap.enqueue

What if we enqueue 6?
Continue swapping upwards...
Heap.enqueue

What if we enqueue 6?

Stop swapping when we are no longer bigger than our parent
Heap.dequeue

Idea: Replace root with the last element then fix the heap

1. Start with current ← root
2. While current has a child > current
   a. Swap current with its largest child
   b. Repeat with current ← child
Heap.dequeue

What if we call dequeue?
Heap.dequeue

What if we call dequeue?
Remove and return the root
What if we call dequeue?

Make the last item the new root
Heap.dequeue

What if we call dequeue?

Check for our largest child
What if we call dequeue?

If the largest child is bigger than us, swap
Heap::dequeue

What if we call dequeue?
Continue swapping down the tree as necessary...
What if we call dequeue?
Continue swapping down the tree as necessary...
What if we call dequeue?
Stop swapping when our children are no longer bigger
## Priority Queues

<table>
<thead>
<tr>
<th>Operation</th>
<th>Lazy</th>
<th>Proactive</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>enqueue</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(\log(n))$</td>
</tr>
<tr>
<td>dequeue</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(\log(n))$</td>
</tr>
<tr>
<td>head</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
Storing Heaps

How can we store this heap in an array buffer?

Enqueue always inserts at the arrays end (then we fix up)
Runtime Analysis

enqueue
- **Append to ArrayBuffer**: amortized $O(1)$ (unqualified $O(n)$)
- **fixUp**: $O(\log(n))$ fixes, each one costs $O(1) = O(\log(n))$
- **Total**: amortized $O(\log(n))$ (unqualified $O(n)$)

dequeue
- **Remove end of ArrayBuffer**: $O(1)$
- **fixDown**: $O(\log(n))$ fixes, each one costs $O(1) = O(\log(n))$
- **Total**: worst-case $O(\log(n))$
Heapify

Given an arbitrary array (show as a tree here) turn it into a heap
Heapify

Start at the lowest level, and call `fixDown` on each node (0 swaps per node)
Heapify

Do the same at the next lowest level (at most one swap per node)
Heapify

Do the same at the next lowest level (at most one swap per node)
Heapify

Continue upwards (now at most 2 swaps per node)
Heapify

Continue upwards (now at most 2 swaps per node)
Heapify

Continue upwards (now at most 2 swaps per node)
Heapify

Continue upwards (now at most 2 swaps per node)
Heapify

Continue upwards (now at most 2 swaps per node)

This whole process only takes $O(n)$ time!
Graphs
Let's Talk About Graphs

A graph is a pair \((V,E)\) where:

- \(V\) is a set of vertices
- \(E\) is a set of vertex pairs called edges
- Edges and vertices may also store data (labels)
Edge Types

Directed Edge (asymmetric relationship)
- Ordered pair of vertices \((u, v)\)
- Origin \(u\) → destination \(v\)

Undirected Edge (symmetric relationship)
- Unordered pair of vertices \((u, v)\)

Directed Graph: All edges are directed

Undirected Graph: All edges are undirected
Terminology

Endpoints of an edge
$U, V$ are endpoints of an edge

Adjacent Vertices
$U, V$ are adjacent

Degree of a vertex
$X$ has degree $5$
Terminology

Edges indecent on a vertex
\(a, b, d\) are incident on \(V\)

Parallel Edges
\(h, i\) are parallel

Self-Loop
\(j\) is a self-loop

Simple Graph
A graph without parallel edges or self-loops
Terminology

Path
A sequence of alternating vertices and edges
- begins with a vertex
- ends with a vertex
- each edge preceded/followed by its endpoints

Simple Path
A path such that all of its vertices and edges are distinct

$U, c, W, e, X, g, Y, f, W, d, V$ is not simple

$V, b, X, h, Z$ is simple
Terminology

Cycle
A path the begins and ends with the same vertex. Must contain at least one edge.

Simple Cycle
A cycle such that all of its vertices and edges are distinct.

\[ U, c, W, e, X, g, Y, f, W, d, V, a, U \] is a cycle that is not simple.

\[ V, b, X, g, Y, f, W, c, U, a, V \] is a simple cycle.
Graph Properties

Proof: Each edge is counted twice

\[ \sum_{v} \text{deg}(v) = 2m \]
A (Directed) Graph ADT

Two type parameters (Graph\([V,E]\))

- \(V\): The vertex label type
- \(E\): The edge label type

Vertices

- ...are elements (like Linked List Nodes)
- ...store a value of type \(V\)

Edges

- ...are also elements
- ...store a value of type \(E\)
Attempt 1: Edge List

Data Model:

A List of Edges
(ArrayBuffer)

A List of Vertices
(ArrayBuffer)
Attempt 1: Linked Edge List

Data Model:

A List of Edges
(DoublyLinkedList)

A List of Vertices
(DoubleLinkedList)
Edge List Summary

- addEdge, addVertex: $O(1)$
- removeEdge: $O(1)$
- removeVertex: $O(m)$
- vertex.incidentEdges: $O(m)$
- vertex.edgeTo: $O(m)$
- Space Used: $O(n) + O(m)$
Edge List Summary

Linked List [Vertex]

Vertex

Linked List [Edge]
How can we improve?

**Idea:** Store the in/out edges for each vertex!
Storing the list of incident edges in the vertex saves us the time of checking every edge in the graph.

The edge now stores additional nodes to ensure removal is still \( \Theta(1) \)
Adjacency List Summary

- addEdge, addVertex: $O(1)$
- removeEdge: $O(1)$
- removeVertex: $O(\text{deg}(\text{vertex}))$
- vertex.incidentEdges: $O(\text{deg}(\text{vertex}))$
- vertex.edgeTo: $O(\text{deg}(\text{vertex}))$
- Space Used: $O(n) + O(m)$
### Adjacency Matrix

<table>
<thead>
<tr>
<th></th>
<th>U</th>
<th>V</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>-</td>
<td>a</td>
<td>-</td>
</tr>
<tr>
<td>V</td>
<td>-</td>
<td>-</td>
<td>b</td>
</tr>
<tr>
<td>W</td>
<td>c</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

![Graph Diagram](image)

- U to V with weight **a**
- V to U with weight **b**
- W to V with weight **c**

The diagram shows the connections between nodes U, V, and W with their respective weights.
Adjacency Matrix Summary

- `addEdge`, `removeEdge`: $O(1)$
- `addVertex`, `removeVertex`: $O(n^2)$
- `vertex.incidentEdges`: $O(n)$
- `vertex.edgeTo`: $O(1)$
- Space Used: $O(n^2)$

How does this relate to space of edge/adjacency lists? *If the matrix is "dense" it's about the same*
A **subgraph**, $S$, of a graph $G$ is a graph where:
- $S$'s vertices are a subset of $G$'s vertices
- $S$'s edges are a subset of $G$'s edges

A **spanning subgraph** of $G$...
- Is a subgraph of $G$
- Contains all of $G$'s vertices
A graph is **connected**... If there is a path between every pair of vertices

A **connected component** of $G$...

- Is a maximal connected subgraph of $G$
  - "maximal" means you can't add a new vertex without breaking the property
  - Any subset of $G$'s edges that connect the subgraph are fine
A **spanning tree** of a connected graph...

...Is a spanning subgraph that is a tree

...It is not unique unless the graph is a tree
DFS vs BFS

DFS (LIFO order...Stacks)

BFS (FIFO order...Queues)

BACK Edge($v,w$): $w$ is an ancestor of $v$ in the discovery tree

CROSS Edge($v,w$): $w$ is at the same or next level as $v$
## DFS Traversal vs BFS Traversal

<table>
<thead>
<tr>
<th>Application</th>
<th>DFS</th>
<th>BFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spanning Trees</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Connected Components</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Paths/Connectivity</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Cycles</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Shortest Paths</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Articulation Points</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
Depth-First Search Complexity

In summary...

1. Mark the vertices UNVISITED \(O(|V|)\)
2. Mark the edges UNVISITED \(O(|E|)\)
3. DFS vertex loop \(O(|V|)\)
4. All calls to DFSOne \(O(|E|)\)

\(O(|V| + |E|)\)
Breadth-First Search Complexity

In summary...

1. Mark the vertices UNVISITED $O(|V|)$
2. Mark the edges UNVISITED $O(|E|)$
3. Add each vertex to the work queue $O(|V|)$
4. Process each vertex $O(|E|)$

$O(|V| + |E|)$
Djikstra's Algorithm

- DFS uses a Stack to manage the search (LIFO order)
- BDS uses a Queue to manage the search (FIFO order)
- Djiktra's uses a PriorityQueue to manage the search (priority order)
  - Finds the shortest path in a weighted graph
  - Runs in $\sim O(|V| \log |V|)$

114
Trees, Sets, Bags
(Even More) Tree Terminology

**Rooted, Directed Tree** - Has a single root node (node with no parents)

**Parent of node X** - A node with an out-edge to X (max 1 parent per node)

**Child of node X** - A node with an in-edge from X

**Leaf** - A node with no children

**Depth of node X** - The number of edges in the path from the root to X

**Height of node X** - The number of edges in the path from X to the deepest leaf
(Even More) Tree Terminology

**Level of a node** - Depth of the node + 1

**Size of a tree** \((n)\) - The number of nodes in the tree

**Height/Depth of a tree** \((d)\) - Height of the root/depth of the deepest leaf
(Even More) Tree Terminology

**Binary Tree** - Every vertex has at most 2 children

**Complete Binary Tree** - All leaves are in the deepest two levels

**Full Binary Tree** - All leaves are at the deepest level, therefore every vertex has exactly 0 or 2 children, and $d = \log(n)$
A **Binary Search Tree** is a **Binary Tree** in which each node stores a unique key, and the keys are ordered.

**Constraints**
- No duplicate keys
- For every node $X_L$ in the left subtree of node $X$: $X_L.key < X.key$
- For every node $X_R$ in the right subtree of node $X$: $X_R.key > X.key$

$X$ **partitions** its children
## BST Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>find</td>
<td>$O(d)$</td>
</tr>
<tr>
<td>insert</td>
<td>$O(d)$</td>
</tr>
<tr>
<td>remove</td>
<td>$O(d)$</td>
</tr>
</tbody>
</table>

What is the runtime in terms of $n$? $O(n)$

Does it need to be that bad?
# BST Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>find</td>
<td>$O(d)$</td>
</tr>
<tr>
<td>insert</td>
<td>$O(d)$</td>
</tr>
<tr>
<td>remove</td>
<td>$O(d)$</td>
</tr>
</tbody>
</table>

What is the runtime in terms of $n$? $O(n)$

$log(n) \leq d \leq n$
Tree Depth vs Size

If height(left) ≈ height(right)

If height(left) ≪ height(right)

\[ d = O(\log(n)) \]

We want this, not this

\[ d = O(n) \]
Keeping Depth Small - Two Approaches

Option 1

Keep tree **balanced**: subtrees +/-1 of each other in height

(add a field to track amount of "imbalance")

Option 2

Keep leaves at some minimum depth (*d/2*)

(Add a color to each node marking it as "red" or "black")
Rebalancing Trees (rotations)
Rebalancing Trees (rotations)

Rotate(A, B)
Rebalancing Trees (rotations)

```plaintext
B
 /  \
A   \
 //  \
X Y Z

Rotate(A, B)
```
Rebalancing Trees (rotations)

A became B's left child

B's left child became A's right child

Rotate(A, B)
Rebalancing Trees (rotations)

A became B's left child
B's left child became A's right child

Is ordering maintained?

Rotate(A, B)
Rebalancing Trees (rotations)

A became B's left child

B's left child became A's right child

Is ordering maintained? Yes!

Rotate(A, B)
Rebalancing Trees (rotations)

A became B's left child
B's left child became A's right child

*Is ordering maintained? Yes!*

*Complexity?*

Rotate(A, B)
Rebalancing Trees (rotations)

A became B's left child

B's left child became A's right child

Is ordering maintained? Yes!

Complexity? $O(1)$
Rebalancing Trees (rotations)

A became B's left child

B's left child became A's right child

Is ordering maintained? Yes!

Complexity? $O(1)$

This is called a left rotation

(right rotation is the opposite)

Rotate(A, B)
An **AVL tree** (Adelson-Velsky and Landis) is a **BST** where every subtree is depth-balanced

**Remember:** Tree depth = height(root)

**Balanced:** $|\text{height(root.right)} - \text{height(root.left)}| \leq 1$
AVL Trees

An **AVL tree** (Adelson-Velsky and Landis) is a **BST** where every subtree is depth-balanced.

**Remember:** Tree depth = height(root)

**Balanced:**
\[|\text{height(root.right)} - \text{height(root.left)}| \leq 1\]
Question: Does the AVL property result in any guarantees about depth?

YES! Depth balance forces a maximum possible depth of $\log(n)$

Proof Idea: An AVL tree with depth $d$ has "enough" nodes
Inserting Records

To insert a record into an AVL Tree:

1. Find the insertion point (remember it is a BST) \( O(d) = O(\log n) \)
2. Insert the new leaf and set balance factor to 0 \( O(1) \)
3. Trace path back up to root and update balance factors \( O(d) = O(\log n) \)
   a. If a balance factor becomes +/-2 then rotate to fix \( O(1) \)
Maintaining Balance - Another Approach

Enforcing height-balance is too strict (May do “unnecessary” rotations)

Weaker (and more direct) restriction:

- Balance the depth of EmptyTree nodes
- If $a, b$ are EmptyTree nodes, then enforce that for all $a, b$:
  - $\text{depth}(a) \geq (\text{depth}(b) \div 2)$
  - or
  - $\text{depth}(b) \geq (\text{depth}(a) \div 2)$
If no EmptyTree has depth less than $d/2$, then this part of the tree must be full. $n \geq 2^{d/2}$ nodes

$log(n) \geq d/2$

$2 \log(n) \geq d \rightarrow d \in O(\log(n))$

Therefore enforcing these constraints means that tree depths is $O(\log(n))$...

So how do we enforce them (efficiently)?
Red-Black Trees

To Enforce the Depth Constraint on EmptyTree nodes:

1. Color each node red or black
   a. The # of black nodes from each EmptyTree node to root must be same
   b. The parent of a red node must always be black

2. On insertion (or deletion)
   a. Inserted nodes are red (won't break 1a)
   b. Repair violations of 1b by rotating and/or recoloring
      i. Make sure repairs don't break 1a
Label each EmptyTree with the number of black nodes along the path back to the root. All 3 in this case ✓
Red-Black Tree

**Note:** Each insertion creates at most one red-red parent-child conflict
- $O(1)$ time to recolor/rotate to repair the parent-child conflict
- May create a red-red conflict in grandparent
  - Up to $d/2 = O(\log(n))$ repairs required, but each repair is $O(1)$
- **Insertion therefore remains** $O(\log(n))$

**Note:** Each deletion removes at most one black node (red doesn't matter)
- $O(1)$ time to recolor/rotate to preserve black-depth
- May require recoloring (grand-)parent from black to red
  - Up to $d = O(\log(n))$ repairs required
- **Deletion therefore remains** $O(\log(n))$
HashTables
Map Implementations

Map\[K,V\] as a Sorted Sequence
- apply \(O(\log(n))\) for Array, \(O(n)\) for Linked List
- add \(O(n)\)
- remove \(O(n)\)

Map\[K,V\] as a balanced Binary Search Tree
- apply \(O(\log(n))\)
- add \(O(\log(n))\)
- remove \(O(\log(n))\)
Finding Items

For most of these operations, the expensive part is finding the record...

So...let's skip the search
Assigning Bins

**Idea:** What if we could assign each record to a location in an Array

- Create and array of size $N$
- Pick an $O(1)$ function to assign each record a number in $[0,N)$
  - ie: If our records are names, first letter of name $\rightarrow [0,26)$
Assigning Bins

add("Halloween") → "Halloween"[0] == "H" == 7

This computation is $O(1)$
Assigning Bins

\[
\text{add("Friday the 13th") } \rightarrow \text{"Friday the 13th"[0] == "F" == 5}
\]
Assigning Bins

add("Get Out") → "Get Out"[0] == "G" == 6
Assigning Bins

\[
\text{add( "Babadook" ) } \rightarrow \text{"Babadook"[0] == "B" == 1}
\]
Assigning Bins

find("Get Out") → "Get Out"[0] == "G" == 6

Find in constant time!

A  Babadook  ...  Friday the 13th  Get Out  Halloween  ...  Z
Assigning Bins

find("Scream") → "Scream"[0] == "S" == 18

Determine that "Scream" is not in the Set in constant time!
Assigning Bins

Pros
● $O(1)$ insert
● $O(1)$ find
● $O(1)$ remove

Cons
● Wasted space (4/26 slots used in the example, will we ever use "Z"?)
● Duplication (What about inserting Frankenstein)
Assigning Bins

```
add("Frankenstein")?
```

<table>
<thead>
<tr>
<th>A</th>
<th>Babadook</th>
<th>...</th>
<th>Friday the 13th</th>
<th>Get Out</th>
<th>Halloween</th>
<th>...</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ø</td>
<td>Ø</td>
<td></td>
<td>∅</td>
<td>Ø</td>
<td>Ø</td>
<td></td>
<td>Ø</td>
</tr>
</tbody>
</table>

Making each bucket a linked list solves the collision problem →

Frankenstein
Now we can handle as many duplicates as we need. But are we losing our constant time operations?

*How many elements are we expecting to end up in each bucket?*

*Depends partially on our choice of Hash Function*
Picking a Hash Function

Desirable features for $h(x)$:

- Fast — needs to be $O(1)$
- "Unique" — As few duplicate bins as possible
Hash Functions In the Real-World

Examples
- SHA256 ← Used by GIT
- MD5, BCRYPT ← Used by unix login, apt
- MurmurHash3 ← Used by Scala

*$\text{hash}(x)$* is pseudo-random
- $\text{hash}(x)$ ~ uniform random value in $[0, \text{INT\_MAX})$
- $\text{hash}(x)$ always returns the same value for the same $x$
- $\text{hash}(x)$ is uncorrelated with $\text{hash}(y)$ for all $x \neq y$
Refresher on Modulus

The modulus function takes any integers \( n \) and \( d \), and returns a number \( r \) in the range \([0, d)\), such that \( n = q \times d + r \). (It returns the remainder of \( n / d \))

If my hash table has 7 buckets, and I insert an element with hash code 73, what bucket would it go in? \( 73 \% 7 = 3 \)
Pseudo-Random Hash Function

\[ n = \text{number of elements in any bucket} \]
\[ N = \text{number of buckets} \]
\[ b_{i,j} = \begin{cases} 
1 & \text{if element } i \text{ is assigned to bucket } j \\
0 & \text{otherwise} 
\end{cases} \]

\[ \mathbb{E}[b_{i,j}] = \frac{1}{N} \]
Pseudo-Random Hash Function

\[ n = \text{number of elements in any bucket} \]
\[ N = \text{number of buckets} \]
\[ b_{i,j} = \begin{cases} 
1 & \text{if element } i \text{ is assigned to bucket } j \\
0 & \text{otherwise} 
\end{cases} \]

\[ \mathbb{E} \left[ \sum_{i=0}^{n} b_{i,j} \right] = \frac{n}{N} \]
Pseudo-Random Hash Function

\[ n = \text{number of elements in any bucket} \]
\[ N = \text{number of buckets} \]

\[ b_{i,j} = \begin{cases} 
1 & \text{if element } i \text{ is assigned to bucket } j \\
0 & \text{otherwise} 
\end{cases} \]

Only true if \( b_{i,j} \) and \( b_{i',j} \) are uncorrelated for any \( i \neq i' \)

\[ \mathbb{E} \left[ \sum_{i=0}^{n} b_{i,j} \right] = \frac{n}{N} \]

The expected number of elements in any bucket \( j \)

(h(i) can’t be related to h(i’))
Pseudo-Random Hash Function

\[ n = \text{number of elements in any bucket} \]

\[ N = \text{number of buckets} \]

\[ b_{i,j} = \begin{cases} 
1 & \text{if element } i \text{ is assigned to bucket } j \\
0 & \text{otherwise} 
\end{cases} \]

Only true if \( b_{ij} \) and \( b_{i'j} \) are uncorrelated for any \( i \neq i' \)

\( (h(i) \text{ can't be related to } h(i')) \)

The expected number of elements in any bucket \( j \)

\[ \mathbb{E} \left[ \sum_{i=0}^{n} b_{i,j} \right] = \frac{n}{N} \]

...given this information, what do the runtimes of our operations look like?
Pseudo-Random Hash Function

\[ n = \text{number of elements in any bucket} \]
\[ N = \text{number of buckets} \]

\[ b_{i,j} = \begin{cases} 
1 & \text{if element } i \text{ is assigned to bucket } j \\
0 & \text{otherwise} 
\end{cases} \]

**Expected** runtime of insert, apply, remove: \( O(n/N) \)

**Worst-Case** runtime of insert, apply, remove: \( O(n) \)
Idea: Make $\alpha$ a constant

Fix an $\alpha_{\text{max}}$ and start requiring that $\alpha \leq \alpha_{\text{max}}$

What do we do when this constraint is violated? **Resize!**
Hash Function Recap

- We now have pseudo-random hash functions that run in $O(1)$
  - They act as if they are uniformly random
    - Will evenly distribute elements to buckets
    - hash($x$) is uncorrelated with hash($y$)
  - They are deterministic (hash($x$) will always return the same value)
- We can use these hash functions to determine which bucket an arbitrary element belongs in in $O(1)$ time
- There are expected to be $n/N$ elements in that bucket
  - So runtime for all operations is expected $O(1) + O(n/N)$

Next goal: Make this a constant
Rehashing

When we insert an element that would exceed the load factor we:

1. Resize the underlying array from $N_{\text{old}}$ to $N_{\text{new}}$
2. Rehash all of the elements from their old bucket to their new bucket
   a. Element $x$ moves from $\text{hash}(x) \mod N_{\text{old}}$ to $\text{hash}(x) \mod N_{\text{new}}$

How long does this take?

1. Allocate the new array: $O(1)$
2. Rehash every element from the old array to the new: $O(N_{\text{old}} + n)$
3. Free the old array: $O(1)$

Total: $O(N_{\text{old}} + n)$
Recap of HashTables (so far...)

**Current Design:** HashTable with Chaining
- Array of buckets
- Each bucket is the head of a linked list (a "chain" of elements)
Runtime for apply(x)

**Expected Runtime:**
1. Find the bucket (call our hash function): $O(c_{hash}) = O(1)$
2. Find the record in the bucket: $O(\alpha \cdot c_{equality}) = O(1)$
3. Total: $O(c_{hash} + \alpha \cdot c_{equality}) = O(1)$

**Unqualified Worst-Case:**
1. Find the bucket (call our hash function): $O(c_{hash}) = O(1)$
2. Find the record in the bucket: $O(n \cdot c_{equality}) = O(n)$
3. Total: $O(c_{hash} + n \cdot c_{equality}) = O(n)$
Runtime for \texttt{remove}(x)

**Expected Runtime:**
1. Find the bucket (call our hash function): $O(c_{\text{hash}}) = O(1)$
2. Find the record in the bucket: $O(\alpha \cdot c_{\text{equality}}) = O(1)$
3. Remove (by reference): $O(1)$
4. Total: $O(c_{\text{hash}} + \alpha \cdot c_{\text{equality}} + 1) = O(1)$

Only one extra constant-time step to remove

**Unqualified Worst-Case:**
1. Find the record in the bucket: $O(n \cdot c_{\text{equality}}) = O(n)$
2. Total: $O(c_{\text{hash}} + n \cdot c_{\text{equality}} + 1) = O(n)$
Runtime for `insert(x)`

**Expected Runtime:**
1. Find the bucket (call our hash function): $O(c_{hash}) = O(1)$
2. Remove $x$ from bucket if present: $O(\alpha \cdot c_{equality} + 1)$
3. Prepend to bucket: $O(1)$
4. Rehash if needed: $O(n \cdot c_{hash} + N)$ (amortized $O(1)$)
5. **Total:** $O(c_{hash} + \alpha \cdot c_{equality} + 3) = O(1)$

**Unqualified Worst-Case:**
1. Remove $x$ from bucket if present: $O(n \cdot c_{equality} + 1) = O(n)$
2. **Total:** $O(c_{hash} + n \cdot c_{equality} + 3) = O(n)$

One additional constant-time step to prepend, and then potentially the need to rehash, but that is amortized $O(1)$.
Hash Tables with Chaining

- hash(A) = 4
- hash(B) = 5
- hash(C) = 5
- hash(D) = 2
- hash(E) = 6
- hash(F) = 2

Collisions are resolved by adding the element to the buckets linked list.
Hash Tables with Open Addressing

hash(A) = 4 ← no collision
hash(B) = 5
hash(C) = 5
hash(D) = 2
hash(E) = 6
hash(F) = 4

With Open Addressing collisions are resolved by "cascading" to the next available bucket
HashTables with Open Addressing

hash(A) = 4
hash(B) = 5 ← no collision
hash(C) = 5
hash(D) = 2
hash(E) = 6
hash(F) = 4

With Open Addressing collisions are resolved by "cascading" to the next available bucket
Hash Tables with Open Addressing

hash(A) = 4
hash(B) = 5
hash(C) = 5 ← collision! Search for next free bucket
hash(D) = 2
hash(E) = 6
hash(F) = 4

With Open Addressing collisions are resolved by "cascading" to the next available bucket
HashTables with Open Addressing

hash(A) = 4
hash(B) = 5
hash(C) = 5 ← collision! Search for next free bucket
hash(D) = 2
hash(E) = 6
hash(F) = 4

With Open Addressing collisions are resolved by "cascading" to the next available bucket
# Hash Tables with Open Addressing

<table>
<thead>
<tr>
<th>Hash Value</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>B</td>
</tr>
<tr>
<td>5</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>D</td>
</tr>
<tr>
<td>6</td>
<td>E</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
</tr>
</tbody>
</table>

With Open Addressing collisions are resolved by "cascading" to the next available bucket.
With Open Addressing collisions are resolved by "cascading" to the next available bucket.

hash(A) = 4
hash(B) = 5
hash(C) = 5
hash(D) = 2
hash(E) = 6  ← collision! cascade to 0
hash(F) = 4
HashTables with Open Addressing

hash(A) = 4
hash(B) = 5
hash(C) = 5
hash(D) = 2
hash(E) = 6
hash(F) = 4 ← collision! Cascade all the way to 1

With Open Addressing collisions are resolved by "cascading" to the next available bucket.
Cuckoo Hashing

Idea: Use two hash functions, \( \text{hash}_1 \) and \( \text{hash}_2 \)

To insert a record \( X \):

1. If \( \text{hash}_1(X) \) and \( \text{hash}_2(X) \) are both available, pick one at random
2. If only one of those buckets is available, pick the available bucket
3. If neither is available, pick one at random and evict the record there
   a. Insert \( X \) in this bucket
   b. Insert the evicted record following the same procedure
Hash Tables with Cuckoo Hashing

\[
\begin{align*}
\text{hash}_1(A) &= 1 & \text{hash}_2(A) &= 3 \\
\text{hash}_1(B) &= 2 & \text{hash}_2(B) &= 4 \\
\text{hash}_1(C) &= 2 & \text{hash}_2(C) &= 1 \\
\text{hash}_1(D) &= 4 & \text{hash}_2(D) &= 6 \\
\text{hash}_1(E) &= 3 & \text{hash}_2(E) &= 4
\end{align*}
\]
HashTables with Cuckoo Hashing

\[
\begin{align*}
\text{hash}_1(A) &= 1 & \text{hash}_2(A) &= 3 \\
\text{hash}_1(B) &= 2 & \text{hash}_2(B) &= 4 \\
\text{hash}_1(C) &= 2 & \text{hash}_2(C) &= 1 \\
\text{hash}_1(D) &= 4 & \text{hash}_2(D) &= 6 \\
\text{hash}_1(E) &= 3 & \text{hash}_2(E) &= 4
\end{align*}
\]
C can't go in either bucket, so evict one at random (let's say B) and reinsert the evicted element.
HashTables with Cuckoo Hashing

hash_1(A) = 1  hash_2(A) = 3
hash_1(B) = 2  hash_2(B) = 4
hash_1(C) = 2  hash_2(C) = 1
hash_1(D) = 4  hash_2(D) = 6
hash_1(E) = 3  hash_2(E) = 4

B can only go in 4 now, but 4 is free
HashTables with Cuckoo Hashing

$\text{hash}_1(A) = 1 \quad \text{hash}_2(A) = 3$
$\text{hash}_1(B) = 2 \quad \text{hash}_2(B) = 4$
$\text{hash}_1(C) = 2 \quad \text{hash}_2(C) = 1$
$\text{hash}_1(D) = 4 \quad \text{hash}_2(D) = 6$
$\text{hash}_1(E) = 3 \quad \text{hash}_2(E) = 4$

$B$ can only go in 4 now, but 4 is free
### HashTables with Cuckoo Hashing

<table>
<thead>
<tr>
<th>Index</th>
<th>A</th>
<th>C</th>
<th>B</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash$_1$</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>hash$_2$</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

- $hash_1(A) = 1$  
- $hash_2(A) = 3$
- $hash_1(B) = 2$  
- $hash_2(B) = 4$
- $hash_1(C) = 2$  
- $hash_2(C) = 1$
- $hash_1(D) = 4$  
- $hash_2(D) = 6$
- $hash_1(E) = 3$  
- $hash_2(E) = 4$
HashTables with Cuckoo Hashing

hash_1(A) = 1  hash_2(A) = 3
hash_1(B) = 2  hash_2(B) = 4
hash_1(C) = 2  hash_2(C) = 1
hash_1(D) = 4  hash_2(D) = 6
hash_1(E) = 3  hash_2(E) = 4
Hash Tables with Cuckoo Hashing

Hash functions:

- \( \text{hash}_1(A) = 1 \)
- \( \text{hash}_2(A) = 3 \)
- \( \text{hash}_1(B) = 2 \)
- \( \text{hash}_2(B) = 4 \)
- \( \text{hash}_1(C) = 2 \)
- \( \text{hash}_2(C) = 1 \)
- \( \text{hash}_1(D) = 4 \)
- \( \text{hash}_2(D) = 6 \)
- \( \text{hash}_1(E) = 3 \)
- \( \text{hash}_2(E) = 4 \)

What if we try to insert \( F \) which hashes to either 1 or 3?
HashTables with Cuckoo Hashing

\[
\begin{align*}
\text{hash}_1(A) &= 1 & \text{hash}_2(A) &= 3 \\
\text{hash}_1(B) &= 2 & \text{hash}_2(B) &= 4 \\
\text{hash}_1(C) &= 2 & \text{hash}_2(C) &= 1 \\
\text{hash}_1(D) &= 4 & \text{hash}_2(D) &= 6 \\
\text{hash}_1(E) &= 3 & \text{hash}_2(E) &= 4
\end{align*}
\]

What if we try to insert \( F \) which hashes to either 1 or 3? **We will loop infinitely trying to evict...so limit the number of eviction attempts then do a full rehash**
Cuckoo Hashing

So with Cuckoo Hashing, we may have to rehash early, and may follow long chains of evictions inserting, but...

What is the runtime of apply/remove?
Cuckoo Hashing

So with Cuckoo Hashing, we may have to rehash early, and may follow long chains of evictions inserting, but...

What is the runtime of apply/remove?

1. Check 2 different buckets: $O(1)$
2. That's it...no chaining, cascading etc...

Apply and remove are **GUARANTEED** $O(1)$ with Cuckoo Hashing
...So the expected runtime of all operations is $O(1)$

*Why would you ever use any other data structure?*

- HashTables do not preserve ordering
- HashTables may waste a lot of memory
- Rehashing can be expensive
- Only **guarantee** on lookup time is that it is $O(n)$
Misc Topics
Algorithmic Complexity

Remember: $O(f(n))$ placed bounds on growth functions in general. Not necessarily only for runtime growth functions...

Runtime Bounds (or Runtime Complexity)
- The algorithm takes $O(...)$ time

Memory Bounds (or Memory Complexity)
- The algorithm needs $O(...)$ storage

I/O Bounds (or I/O Complexity)
- The algorithm performs $O(...)$ accesses to slower memory
The Memory Hierarchy (simplified)

Cache

Memory (RAM)

Solid State Drives (SSDs)

Hard Disk Drives (HDDs, "Spinning Rust")

Faster

Bigger
Improving on Fence Pointers ISAM Index

1. Binary Search @ Level 0 to find Level 1 page

2. Load and search Level 1 page to find Level 2 page

3. Load and search Level 2 page to find data page

4. Load and search data page to find the record
What if the data changes?
Keep free space in your pages…but not too much free space

If a new element gets added to a full page, split it into two pages

Ensure all pages are at least half full
Lossy Sets

LossySet[A]

add(a: A): Insert a into the set

apply(a: A):
  • If a is in the set ALWAYS return true
  • If a is not in the set USUALLY return false (returning true is OK)
Lossy Set

What does this gain for us?

Idea: If apply doesn't always need to be right, we don't need to store everything
class LossyHashSet[A](_size: Int) extends LossySet[A] {
  val bits = new Array[Boolean](_size)
  def add(a: A): Unit = {
    val bucket = a.hashCode % _size
    bits(bucket) = true
  }
  def apply(a: A): Boolean = {
    val bucket = a.hashCode % _size
    return bits(bucket)
  }
}
Lossy Set Example

add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
apply("Scream")? TRUE
apply("Saw")? TRUE
apply("The Candyman")? FALSE
apply("Dracula")? FALSE
apply("Friday the 13th")? TRUE
k-D Trees

- Can generalize to $k > 2$ dimensions
  - Depth 0: Partition on Dimension 0
  - Depth 1: Partition on Dimension 1
  - ...
  - Depth $k-1$: Partition on Dimension $k-1$
  - Depth $k$: Partition on Dimension 0
  - Depth $k+1$: Partition on Dimension 1
  - Depth $i$: Partition on Dimension $(i \mod k)$

- In practice, `range()` and `knn()` become $\sim O(n)$ for $k > 3$
  - If a subtree’s range overlaps with the target in even one dimension, we need to search it. (*Curse of Dimensionality*)
k-D Tree

Partitions on x

Partitions on y

10,4

5,4

19, 10

9,1

1,6

13,6

16,15
Quad/Oct Trees Revisited

Idea: Let's organize the data (spatially) in a tree structure

- 2D space → use a quad tree
- 3D space → use an oct tree (each node has at most 8 children)

Unlike last time, let's partition the space we are simulating, rather than the points in the space.
Create a quad-tree by recursively partitioning the space

- Divide the space evenly until there is only one element per partition
- Internal tree nodes represent the partitions, leaves are the actual elements
Space Partitioning - 2D Example
Other Problems: Ray/Path Tracing

Which object does this ray of light hit?
Do we have to check every single object?
How can we organize these objects?
Other Problems: Ray/Path Tracing

**Idea:** Build a hierarchy of bounding boxes (BVH - Bounding volume hierarchy)
Other Problems: Ray/Path Tracing

These bounding boxes form a tree...
We can check if the ray intersects a bounding box.
If it does, explore its children.
If not, ignore it.
High-Level Summary

- We've seen both trees and hash tables as effective ways to organize our data if we know we are going to be searching it often.

- **HashTables** can be great for exact lookups.
  - Think PA4: you may want to lookup a person with an exact (birthday, zipcode) pair, and HashTable lets you do that very fast.

- **Trees** and tree like structures work very well for "fuzzier" searches.
  - What is "close" to this point? What object might this projectile hit? etc.
  - The input to your search is not necessarily an exact element in your tree, but the tree organizes the data in a way that directs your search.
Thanks for a great semester!