Announcements

- WA3 released, due 4/23/23 @ 11:59PM
  - Make sure you have the correct version
Clarification on Red-Black Trees

Is the following a valid Red-Black Tree?

What depth are the EmptyTree nodes?
Clarification on Red-Black Trees

Is the following a valid Red-Black Tree?

What depth are the EmptyTree nodes?
Is the following a valid Red-Black Tree?

What depth are the EmptyTree nodes?

By just considering depth from the root this tree seems to fit the necessary constraints...

But just like AVL trees, this property must hold for ALL nodes in the tree...
Clarification on Red-Black Trees

Consider the subtree rooted at 3

What are the depth of the EmptyTree nodes with respect to 3?
Clarification on Red-Black Trees

Consider the subtree rooted at 3

What are the depth of the EmptyTree nodes with respect to 3?
Clarification on Red-Black Trees

Consider the subtree rooted at 3

What are the depth of the EmptyTree nodes with respect to 3?

This does not meet Red-Black constraints on depth

...and we can see it is impossible to color these nodes by the rules
Clarification on Red-Black Trees

Since we are unable to color the subtree rooted at 3, we are unable to color the entire tree.
Back to 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 $O(\log(n))$
For most of these operations, the expensive part is finding the record...
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 an 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

A  B  …  F  G  H  …  Z
Assigning Bins

add("Halloween")
Assigning Bins

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

![Diagram showing bin assignment with "Halloween" in the G bin]
Assigning Bins

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

This computation is $O(1)$
Assigning Bins

```python
add("Friday the 13th") → "Friday the 13th"[0] == "F" == 5
```

![Bins Diagram]

A  B  ...  Friday the 13th  G  Halloween  ...  Z
Assigning Bins

\[
\text{add("Get Out")} \rightarrow \text{"Get Out"[0] == "G" == 6}
\]
Assigning Bins

add("Babadook") → "Babadook"[0] == "B" == 1

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

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

Find in constant time!
Assigning Bins

find("Scream") \rightarrow "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)
Bin-Based Organization

Wasted Space
- Not ideal...but not wrong
- $O(1)$ access time might be worth it
- Also depends on the choice of function

Duplication
- We need to be able to handle duplicates
Bin-Based Organization

Wasted Space
- Not ideal...but not wrong
- $O(1)$ access time might be worth it
- Also depends on the choice of function

Duplication
- We need to be able to handle duplicates

What about "buckets" instead of "bins" (store multiple items per location)
Handling "Duplicates"

How can we store multiple items at each location?
Bigger Buckets

Fixed Size Buckets ($B$ elements)

Pros
● Can deal with up to $B$ dupes
● Still $O(1)$ find

Cons
● What if more than $B$ dupes?

Arbitrarily Large Buckets (List)

Pros
● No limit to number of dupes

Cons
● $O(n)$ worst-case find
Assigning Bins

add("Frankenstein")?
Assigning Bins

add("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?*
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
Picking a Hash Function

Elements/Bucket

Buckets
Picking a Hash Function

apply(k) is $O(1)$
Picking a Hash Function

- Elements/Bucket
- Buckets

apply(k) is $O(1)$

Ideal!

...but unachievable
Picking a Hash Function
Picking a Hash Function

apply(k) is $O(n)$
Picking a Hash Function

Worst Case!

apply(k) is $O(n)$
Picking a Hash Function

Elements/Bucket vs Buckets
Picking a Hash Function

apply(k) is something like $O(1)$?
Picking a Hash Function

Almost Ideal!

...and achievable

apply(k) is something like $O(1)$?
Other Functions

First Letter of UBIT Name
• Unevenly distributed, $O(n)$ worst case apply
First Letter of UBIT Name

- 36 'j's
- No 'u's
Other Functions

First Letter of UBIT Name
- Unevenly distributed, $O(n)$ worst case apply

Identity Function on UBIT #
- Need a 50m+ element array
Other Functions

First Letter of UBIT Name
- Unevenly distributed, $O(n)$ worst case apply

Identity Function on UBIT #
- Need a 50m+ element array
- **Problem:** For reasonable $N$, identity function returns something $> N$
Other Functions

First Letter of UBIT Name
- Unevenly distributed, $O(n)$ worst case apply

Identity Function on UBIT #
- Need a 50m+ element array
- **Problem:** For reasonable $N$, identity function returns something $> N$
- **Solution:** Cap return value of function to $N$ with modulus
  - $(x: \text{Int}) \Rightarrow x \% N$
Identity of UBIT # mod 26
Comparison

UBIT # % 26

substr(UBITName, 0, 1)
Comparison

UBIT # % 26

This still relies on UBIT # being "randomly distributed"

`substr(UBITName, 0, 1)`
What else could we use that would evenly distribute values to locations?
Picking a Hash Function

What else could we use that would evenly distribute values to locations?

**Wacky Idea:** Have $h(x)$ return a random value in $[0,N)$

(This makes apply impossible...but bear with me)
Random Hash Function

\( n = \) number of elements in any bucket

\( N = \) number of buckets

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

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

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

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

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

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

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

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

\[ \mathbb{E} \left[ \sum_{i=0}^{n} b_{i,j} \right] = \frac{n}{N} \]
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 \( b_{ij'} \) 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'))
Random Hash Function

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

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

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

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

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

((h(i) can’t be related to h(i’)))

...given this information, what do the runtimes of our operations look like?
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: \( O(n) \)
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
- $\text{hash}(x) \sim$ uniform random value in $[0, \text{INT}_\text{MAX})$
- $\text{hash}(x)$ always returns the same value for the same $x$
- $\text{hash}(x)$ is uncorrelated with $\text{hash}(y)$ for all $x \neq y$
Hash Functions + Buckets

Everything is: $O \left( \frac{n}{N} \right)$

Let’s call $\alpha = \frac{n}{N}$ the load factor.
Hash Functions + Buckets

Everything is: $O\left(\frac{n}{N}\right)$

Let's call $\alpha = \frac{n}{N}$ the load factor.

**Idea:** Make $\alpha$ a constant
Hash Functions + Buckets

Everything is: \( O\left(\frac{n}{N}\right) \)

Let’s call \( \alpha = \frac{n}{N} \) the load factor.

**Idea:** Make \( \alpha \) a constant

Fix an \( \alpha_{\text{max}} \) and start requiring that \( \alpha \leq \alpha_{\text{max}} \).
Hash Functions + Buckets

Everything is: \( O \left( \frac{n}{N} \right) \)

Let’s call \( \alpha = \frac{n}{N} \) the load factor.

**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?*
Hash Functions + Buckets

Everything is: \( O \left( \frac{n}{N} \right) \)

Let's call \( \alpha = \frac{n}{N} \) the load factor.

**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!*