CSE 250
Lecture 29
Hash Tables
Maps

• Sets, Bags: Collections of Elements of type A
  - add(a: A)
  - remove(a: A)
  - apply(a: A)
    • set.apply(a:A): Boolean // is the element part of set
    • bag.apply(a:A): Int // # of copies of the element in bag
• Maps: Like Sets, but where A is a 2-tuple (key, value)
  - The identity of the element is determined by key
Maps

- Map[K, V]
  - add(k: K, v: V) // also called put(k, v)
    - Insert (k, v) into the map
    - If an element associated with key k already exists, replace it.
  - remove(k): V
    - Remove the element associated with key k, return the corresponding value
  - apply(k: K): V // also called get(k)
    - Return the value corresponding to key k.
Maps

- Map[K,V] as a Sorted Sequence
  - \(O(\log(n))\) apply (but very cache-friendly)
  - \(O(n)\) add
  - \(O(n)\) remove
- Map[K,V] as a balanced Binary Search Tree
  - \(O(\log(n))\) apply
  - \(O(\log(n))\) add
  - \(O(\log(n))\) remove
- Map[K,V] as a LSM tree
Finding Items

The most expensive part of finding records is **finding** them.
(i.e., where is the record located?)

So... skip the search
Alternative Idea: Assign Bins

- Create an array of size $N$
- Pick an $O(1)$ function to assign each record a number in $[0,N)$
  - First letter of name $\rightarrow [0, 26)$
Alternative Idea: Assign Bins

Athos B C D'Artagnan ... Porthos ... Y Z
Alternative Idea: Assign Bins

- Pros
  - O(1) Insert
  - O(1) Find
  - O(1) Remove
- Cons
  - Wasted Space (Only 3/26 slots used)
  - Duplication (What about Aramis?)
Bucket-Based Organization

- Wasted Space
  - Not ideal, but not wrong
  - $O(1)$ access time might be worth it!
  - Also depends on choice of function (more on this later)
- Duplication
  - We need to deal with duplicates!
Duplication

• **Idea**: Make buckets bigger (e.g., B elements)
  - **Pro**: Up to B duplicates in a bucket, Still O(1) (O(B)) find
  - **Con**: No more than B duplicates in a bucket

• **Idea**: Make buckets arbitrarily large (e.g., Linked List)
  - **Pro**: No more overflow
  - **Con**: O(n) worst-case find.
## Buckets + Linked Lists

<table>
<thead>
<tr>
<th>Athos</th>
<th>B</th>
<th>C</th>
<th>D'Artagnan</th>
<th>...</th>
<th>Porthos</th>
<th>...</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>...</td>
<td>∅</td>
<td>...</td>
<td>∅</td>
<td>∅</td>
</tr>
</tbody>
</table>

- **Aramis**
  - ∅
Buckets + Linked Lists

class BucketMap[K, V](_numBuckets: Int, _lookupFunction: K => Int) {

  val _buckets = new Array[List[(K, V)]](_numBuckets)
  for(i <- 1 until _numBuckets) { _buckets(i) = Nil }

  def apply(key: K): V = {
    val bucketPosition = _lookupFunction(k)
    var element = _buckets(bucketPosition)
    while(element != Nil){
      if(key == element.head._1) {
        return element.head._2; }
      element = element.tail
    }
    throw new NoSuchElementException(key.toString)
  }
}
Picking a Lookup Function

- Desirable Features for $h(x)$
  - Fast
    - needs to be $O(1)$
  - “Unique”
    - As few duplicate bins as possible
Picking a Lookup Function

Ideal!
... but unachievable

apply(k) is O(1)
Picking a Lookup Function

Worst Case!

apply(k) is O(n)
Picking a Lookup Function

Almost Ideal!
... and achievable

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

- First Letter of UBIT
First letter of UBIT

36 ‘j’s
No ‘u’s
Other Functions

- First Letter of UBIT name
  - Unevenly Distributed: $O(n)$ apply
- Identity Function on UBIT #
  - Need a 50m+ element array
Other Functions

- **Identity Function:** $(x: \text{Int}) \Rightarrow x$
  - **Problem:** Can return values over $N$
  - **Solution:** Cap return value by Modulus with $N$
    - $(x: \text{Int}) \Rightarrow x \% N$
# Modulus

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
</tbody>
</table>
Modulus
Modulus

UBIT # % 26

substr(UBITName, 0, 1)

But still relies on UBIT # being random!
Picking a Lookup Function

- **Wacky Idea**: Have $h(x)$ return a random value in $[0, N)$
  - `Random.nextInt % N`

  (Yes, it makes apply impossible, but bear with me)
Picking a Lookup Function

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

\[ \mathbb{E} [b_{i,j}] = \frac{1}{N} \]
Picking a Lookup 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 \( i \neq i' \)

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

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

(h(i) can’t be related to h(i’))
Picking a Lookup 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(): \( O \left( \frac{n}{N} \right) \)

**Worst-Case** Runtime of insert, apply, remove(): \( O(n) \)
Hash Functions

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

• hash(x) is pseudorandom
  1) hash(x) ~ uniform random value in [0, INT_MAX)
  2) hash(x) always returns the same value
  3) hash(x) uncorrelated with hash(y) for x ≠ y
Using Hash Functions

- hash(x: Int): Int
  
  What about strings?

```scala
def hashString(str: String): Int = {
  var accumulator: Int = SEED
  for (character <- str) {
    accumulator = hash(accumulator * character.toInt)
  }
  return accumulator
}
```

(simplified, don’t actually do exactly this)

Arbitrary starting constant (hash("") )
call hash() str.length times
Hash Functions

- hash(x: Object): Int
  - In Java/Scala, call x.hashCode
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 $\alpha_{max}$ and start requiring that $\alpha \leq \alpha_{max}$

What happens when this constraint is violated?