Today’s Agenda

- Analyzing ADT Implementations
  - Arrays
  - Binary Search Trees
- Hash tables
  - Hash functions
  - What makes a “good” hash function?
- Other uses of hashing
Analyzing ADT Implementations
Analyzing ADT Implementations

For all of our ADTs (Vector, Set, etc) our goal is to achieve fast

- Contains
- Add
- Remove
Implementing Set

- Let’s use an array!
- We need dynamic memory (on the heap!)
- Let’s focus on 2 versions: unsorted array and sorted array
Unsorted Array

Need to check if the element is contained in the array to ensure no duplicates

Contains
Add
Remove
Unsorted Array

Need to check if the element is contained in the array to ensure no duplicates

Contains \(0(n)\)
Add
Remove
Unsorted Array

Need to check if the element is contained in the array to ensure no duplicates

Contains $O(n)$

Add $O(n)$

Remove
Unsorted Array

Need to check if the element is contained in the array to ensure no duplicates

Contains $0(n)$
Add $0(n)$
Remove $0(n)$
Sorted Array

Binary search speeds up lookups!

Contains
Add
Remove
Sorted Array

Binary search speeds up lookups!

Contains $\mathcal{O}(\log(n))$

Add

Remove
Sorted Array

Still need to shift elements over 😞

Contains: $O(\log(n))$
Add: $O(n)$
Remove
Sorted Array

Still need to shift elements over 😞

<table>
<thead>
<tr>
<th>Operation</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contains</td>
<td>$O(\log(n))$</td>
</tr>
<tr>
<td>Add</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Remove</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
Next step for lookup-based structures...

Binary Search Trees
Stanford library Map and Set classes are backed by binary search trees
Binary Search Trees

Assuming a balanced binary search tree

Contains
Add
Remove
Binary Search Trees

Assuming a balanced binary search tree

Contains $O(\log(n))$

Add

Remove
Binary Search Trees

Assuming a balanced binary search tree

- Contains: $O(\log(n))$
- Add: $O(\log(n))$
- Remove: $O(\log(n))$
Binary Search Trees

Assuming a balanced binary search tree

Contains $O(\log(n))$

Add $O(\log(n))$

Remove $O(\log(n))$
Can we do better than $O(\log(n))$? 😐
UG2 Package Center

- The package center gets a lot of packages throughout the quarter

- They store packages by keeping a small number of buckets for groups of packages

- They have a rule that assigns packages to buckets

- When a student comes in to pick up their package, they know exactly which bucket to go to
To: Neel Kishnani

Unique ID: NEELK
Bin Number: G-B1A1

11/15/2021 4:19 PM

JJD014600009239261945
Let’s introduce a special function called a hash function.
We’ll use this **hash function** to assign elements to buckets
Hash Functions

Important property:
The same input should produce the same output

- Functions with this property are deterministic
- More on deterministic functions in CS103!

For the purposes of CS106B, assume our hash function returns an int
- The input can be of any type though! (string, double, int, etc.)
Input: 12
The output of a hash function is called a hash code!

Input: 12

Hash Code: 106107
Input: 137
Input: 1016
Hash Code: 309731
Input: 12
Input: 12
Hash Code: 106107
A new data structure

- Let’s go back to our array and treat each slot as a bucket for elements, just like the package center!

- We’ll assign each element we need to insert into a bucket and store it there
Use a hash function to assign elements to buckets 🛍
This data structure is called a **Hash Table**
HashTable::HashTable() {
   // Initialize array of buckets
   _elements = new int[NUM_BUCKETS];
}
An idea for a hash function

Return the element itself!

```c
int hash1(int elem) {
    return elem;
}
```
void HashTable::insert(int elem) {
    int bucket = hash1(elem);
    _elements[bucket] = elem;
}

Break
Logistics

- Assignment 7 is out now and due 12/7
  - Huffman Coding!
  - Last assignment of the quarter – congrats!

- Final Exam:
  - 8:30-11:30AM on Monday December 12th
  - Same format as midterm
  - Practice materials up on course website
  - Review sessions (plural) happening next week
Resume
int hash1(int elem) {
    return elem;
}

Hash Function:
int hash1(int elem) {
    return elem;
}

Hash Function:

Input: 3
int hash1(int elem) {
    return elem;
}

Hash Function: 

Input: 3
Hash Code: 3
```c
int hash1(int elem) {
    return elem;
}
```

Input: 3

Hash Code: 3

Hash Function: The hash code is the bucket we put the element in

[0] [ ] [ ] [ ] [ ]
[1] [ ] [ ] [ ] [ ]
[2] [ ] [ ] [ ] [ ]
[3] [3] [ ] [ ] [ ]
[4] [ ] [ ] [ ] [ ]
```c
int hash1(int elem) {
    return elem;
}
```

Hash Function: 3

Input: 0

[0] [1] [2] [3] [4]

[0] [1] [2] [3] [4]
```c
int hash1(int elem) {
    return elem;
}
```

Input: 0

Hash Code: 0

Hash Function: [0] [1] [2] [3] [4]

[3] 3
```c
int hash1(int elem) {
    return elem;
}
```

**Input:**

0

**Hash Code:**

0

**Hash Function:**

0

0

3

0

0

4
int hash1(int elem) {
    return elem;
}

Hash Function:

Input: 17000
Hash Function:

```c
int hash1(int elem) {
    return elem;
}
```

Input: 17000

Hash Code: 17000
Hash Function:
```
int hash1(int elem) {
    return elem;
}
```

Input: 17000

Hash Code: 17000

We need to enlarge our array - lots of wasted space!!
Issue #1

This hash function could lead to a sparse hash table
Hash Function:

```c
int hash1(int elem) {
    return elem;
}
```

Input: -3

[0] 0
[1]
[2]
[3] 3
[4] ...
[17000] 17000
Issue #2

This hash function doesn’t handle negative inputs
We want to **limit the range of possible buckets**
A better(?) hash function

Let’s use the % operator!

```c
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```
int hash2(int elem) {
    return abs(elem) % numBuckets;
}

Input: 3
int hash2(int elem) {
    return abs(elem) % numBuckets;
}

Input: 3
Hash Code: 3
```c
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Input: 3
Hash Code: 3
int hash2(int elem) {
    return abs(elem) % numBuckets;
}

Input: 17000
```c
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```

Input: 17000
Hash Code: 0
Handles this large value!

```c
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```

Input: 17000
Hash Code: 0
```c
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```

Input: -6

<table>
<thead>
<tr>
<th>Element</th>
<th>Hash Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>17000</td>
</tr>
<tr>
<td>[1]</td>
<td></td>
</tr>
<tr>
<td>[2]</td>
<td></td>
</tr>
<tr>
<td>[3]</td>
<td>3</td>
</tr>
<tr>
<td>[4]</td>
<td></td>
</tr>
</tbody>
</table>
```c
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```

Input: -6
Hash Code: 1
Input: -6
Hash Code: 1

Handles this negative value!
```c
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```

Input: 8
```
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```

**Input:** 8

**Hash Code:** 3

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>17000</td>
</tr>
<tr>
<td>[1]</td>
<td>-6</td>
</tr>
<tr>
<td>[2]</td>
<td></td>
</tr>
<tr>
<td>[3]</td>
<td>3</td>
</tr>
<tr>
<td>[4]</td>
<td></td>
</tr>
</tbody>
</table>
```c
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```

Input: 8

Hash Code: 3
Hash Collisions

- Our hash function assigned two different elements to the same bucket
  - We call this a collision

- We have to decide what to do when collisions happen

- Idea: instead of having our array store int, let’s have it a linked list
  - Each bucket will now be a ListNode*  
  - When we have a collision, we can add the new element to the front of the list in $O(1)$
In the header file...

private:
    ListNode** _elements;

In the cpp file...

HashTable::HashTable() {
    // Initialize array of buckets
    _elements = new ListNode*[NUM_BUCKETS]();
}

A double pointer! This means that each array element is a pointer. More in CS107!

Initialize each bucket to the nullptr
This is called a 

Chaining Hash Table
int hash2(int elem) {
    return abs(elem) % numBuckets;
}

Input: 2
```c
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```

Input: 2
Hash Code: 2
int hash2(int elem) {
    return abs(elem) % numBuckets;
}

Input: 2
Hash Code: 2
```cpp
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```

Input: 10
int hash2(int elem) {
    return abs(elem) % numBuckets;
}

Input: 10
Hash Code: 0
int hash2(int elem) {
    return abs(elem) % numBuckets;
}

Input: 7
int hash2(int elem) {
    return abs(elem) % numBuckets;
}

Input: 7
Hash Code: 2
Inserting into this chaining hash table is \( O(1) \)
void HashTable::insert(int elem) {
    if (contains(elem)) return;
    int bucket = hash2(elem);
    ListNode* front = _elements[bucket];

    // Create new front of list, tack previous onto end
    ListNode* cur = new ListNode{elem, front};
    _elements[bucket] = cur;
}
Say you got the following elements as inputs next:

17, 22, 92, 77
With several collisions, our `contains` and `remove` will be $O(n)$

Where $n$ is the number of elements in the relevant bucket
Our goal is to get a “good” hash function that:

- Distributes elements evenly (“spread”)

- Maintains a reasonable load factor
Load Factor

- The average number of elements in each bucket
  - If the load factor is low: wasted space
  - If the load factor is high: slow operations

- The load factor of a hash table with $n$ elements and $b$ buckets is:

\[
\frac{n}{b}
\]
Good Hash Functions

- There’s tons of research in designing hash functions
- Beyond the scope of this class
  - CS161, CS166, CS265
HashSet

Assuming we have a good hash function

Contains
Add
Remove
HashSet

Assuming we have a good hash function

Contains: $\mathcal{O}(n/b)$
Add
Remove
HashSet

Assuming we have a good hash function

Contains $0(n/b)$

Add $0(n/b)$

Remove
HashSet

Assuming we have a good hash function

Contains \(0(n/b)\)
Add \(0(n/b)\)
Remove \(0(n/b)\)
With $b$ chosen to be close to $n$, we can approximate $O(1)$ contains, add, and remove
That’s just about as good as we can do! ✓
The Stanford library HashSet and HashMap are implemented with hash tables!
<table>
<thead>
<tr>
<th>Method</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HashMap</strong></td>
<td></td>
</tr>
<tr>
<td><code>clear()</code></td>
<td>O(N)</td>
</tr>
<tr>
<td><code>containsKey(key)</code></td>
<td>O(1)</td>
</tr>
<tr>
<td><code>equals(map)</code></td>
<td>O(N)</td>
</tr>
<tr>
<td><code>firstKey()</code></td>
<td>O(1)</td>
</tr>
<tr>
<td><code>get(key)</code></td>
<td>O(1)</td>
</tr>
<tr>
<td><code>isEmpty()</code></td>
<td>O(1)</td>
</tr>
<tr>
<td><code>keys()</code></td>
<td>O(N)</td>
</tr>
<tr>
<td><code>lastKey()</code></td>
<td>O(1)</td>
</tr>
<tr>
<td><code>mapAll(fn)</code></td>
<td>O(N)</td>
</tr>
<tr>
<td><code>put(key, value)</code></td>
<td>O(1)</td>
</tr>
<tr>
<td><code>remove(key)</code></td>
<td>O(1)</td>
</tr>
<tr>
<td><strong>HashSet</strong></td>
<td></td>
</tr>
<tr>
<td><code>add(value)</code></td>
<td>O(1)</td>
</tr>
<tr>
<td><code>clear()</code></td>
<td>O(N)</td>
</tr>
<tr>
<td><code>contains(value)</code></td>
<td>O(1)</td>
</tr>
<tr>
<td><code>difference(otherSet)</code></td>
<td>O(N)</td>
</tr>
<tr>
<td><code>equals(set)</code></td>
<td>O(N)</td>
</tr>
<tr>
<td><code>first()</code></td>
<td>O(1)</td>
</tr>
<tr>
<td><code>isEmpty()</code></td>
<td>O(1)</td>
</tr>
<tr>
<td><code>intersect(otherSet)</code></td>
<td>O(N)</td>
</tr>
<tr>
<td><code>isSubsetOf(otherSet)</code></td>
<td>O(N)</td>
</tr>
<tr>
<td><code>isSupersetOf(otherSet)</code></td>
<td>O(N)</td>
</tr>
<tr>
<td><code>last()</code></td>
<td>O(1)</td>
</tr>
<tr>
<td><code>mapAll(fn)</code></td>
<td>O(N)</td>
</tr>
<tr>
<td><code>remove(value)</code></td>
<td>O(1)</td>
</tr>
</tbody>
</table>
Other uses of hash functions
Hash Functions

- Broadly, hash functions map a value to a unique integer value
- Presents in several CS domains
- The magic of hash functions:
  - They can take in any value and boil it down to a unique number
  - Images, ADTs, files, etc.
- Thought question: how would you hash a string?
  - Length?
  - ASCII representation?
  - What about an image?
Hash Functions

Goal: different values should produce very different hash codes
### User table (bcrypt)

<table>
<thead>
<tr>
<th>Username</th>
<th>Password</th>
</tr>
</thead>
<tbody>
<tr>
<td>alice</td>
<td>$2b$10$aQNe4MK0HDhrkus8GZGQLNj11nsx12VTMTDBkykiLjRbb.fIuGC</td>
</tr>
<tr>
<td>bob</td>
<td>$2b$10$TSbaMNCCqG.xNkDVszwwh09Fpb.eeW6aUSIFzGkPoQrs5RahskOU0</td>
</tr>
<tr>
<td>charlie</td>
<td>$2b$10$5KcQQNEfnkPBYxeiq52ZeePXL75j30HG7zngfesGuc0js37X41e</td>
</tr>
<tr>
<td>dakotah</td>
<td>$2b$10$i8n7Lsq13ygE0m3cQ8oEUBjPnGcGBUA4zvJhnsKgyDEZdEd2EFXa</td>
</tr>
</tbody>
</table>
CS145: Data Management and Data Systems

Big Scale

Roadmap

**Hashing-Sorting** solves "all" known data scale problems :=)

+ Boost with a few patterns – Cache, Parallelize, Pre-fetch

**THE BIG IDEA**

Note: Works for Relational, noSQL (e.g., mysql, postgres, BigQuery, BigTable, MapReduce, Spark)
Cryptographic Hash Functions

- Hash functions used in a security context
- One-way function: can’t reverse
- Collision resistant
- Most popular: SHA-256
- More in CS155, CS 253, CS255
END