CS 2230
CS II: Data structures

Meeting 29: Hashing
Brandon Myers
University of Iowa

https://en.wikipedia.org/wiki/Hash_function
Today’s big ideas

• A new way to store data: *every value gets its own location* (e.g., Integer 3 lives at index 3).

• Problems with storing things by value
  a) How do we use up less memory?
  b) How do we generalize beyond positive integers?

• Fix for problem (a): multiple values share the same location and we deal with the consequences

• Fix for problem (b): introducing *hashing*
Roadmap

Propose to **represent a set of integers as an array of booleans** so that we can search in \( O(1) \) time

Wow that's fast! But it has the problem that it requires too much memory!

Reduce the size of the array, but now elements **collide** on the same index

Deal with collisions with a variety of methods ("chaining, probing")

Represent sets of any object by using a **hash function** to turn the object into an integer
What are ways we can represent a Set of integers?
<table>
<thead>
<tr>
<th>Data structure</th>
<th>how to search for a specific value</th>
<th>if you know where it is stored (e.g., index or reference)</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsorted array of integers</td>
<td>search from start until we find it</td>
<td>go to the index</td>
</tr>
<tr>
<td><img src="image1" alt="Array" /></td>
<td><img src="image2" alt="Search process" /></td>
<td><img src="image3" alt="Index" /></td>
</tr>
<tr>
<td><code>10 14 4 15 7 21</code></td>
<td><code>10 14 4 15 7 21</code></td>
<td><code>10 14 4 15 7 21</code></td>
</tr>
<tr>
<td></td>
<td><code>find(4)</code></td>
<td><code>get(2)</code></td>
</tr>
<tr>
<td></td>
<td><img src="image3" alt="Index" /></td>
<td><img src="image3" alt="Index" /></td>
</tr>
<tr>
<td>Data structure</td>
<td>how to search for a specific value</td>
<td>if you know where it is stored (e.g., index or reference)</td>
</tr>
<tr>
<td>-------------------------</td>
<td>------------------------------------</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td>unsorted array of integers</td>
<td>search from start until we find it find(4)</td>
<td>go to the index get(2)</td>
</tr>
<tr>
<td><strong>10 14 4 15 7 21</strong></td>
<td><strong>10 14 4 15 7 21</strong></td>
<td><strong>0 1 2 3 4 5</strong></td>
</tr>
<tr>
<td>sorted array of integers</td>
<td>binary search find(7)</td>
<td>go to the index get(1)</td>
</tr>
<tr>
<td><strong>4 7 10 14 15 21</strong></td>
<td><strong>4 7 10 14 15 21</strong></td>
<td><strong>0 1 2 3 4 5</strong></td>
</tr>
<tr>
<td>Data structure</td>
<td>how to search for a specific value</td>
<td>if you know where it is stored (e.g., index or reference)</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>--------------------------------------------------------------------------</td>
<td>---------------------------------------------------------</td>
</tr>
<tr>
<td>unsorted array of integers</td>
<td>search from start until we find it</td>
<td>go to the index</td>
</tr>
<tr>
<td><img src="image" alt="Unsorted Array" /></td>
<td>find(4)</td>
<td>get(2)</td>
</tr>
<tr>
<td>sorted array of integers</td>
<td>binary search</td>
<td>go to the index</td>
</tr>
<tr>
<td><img src="image" alt="Sorted Array" /></td>
<td>find(7)</td>
<td>get(1)</td>
</tr>
<tr>
<td>binary search tree of integers</td>
<td>search from root</td>
<td>go to the node</td>
</tr>
<tr>
<td><img src="image" alt="Binary Search Tree" /></td>
<td>find(4)</td>
<td></td>
</tr>
<tr>
<td>Data structure</td>
<td>how to search for a specific value</td>
<td>if you know where it is stored (e.g., index or reference)</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td>unsorted array of integers</td>
<td>search from start until we find it</td>
<td>go to the index</td>
</tr>
<tr>
<td></td>
<td><img src="image1" alt="Unsorted Array Diagram" /></td>
<td><img src="image2" alt="Unsorted Array Diagram" /></td>
</tr>
<tr>
<td>sorted array of integers</td>
<td>binary search</td>
<td>go to the index</td>
</tr>
<tr>
<td></td>
<td><img src="image3" alt="Sorted Array Diagram" /></td>
<td><img src="image4" alt="Sorted Array Diagram" /></td>
</tr>
<tr>
<td>binary search tree of integers</td>
<td>search from root</td>
<td>go to the node</td>
</tr>
<tr>
<td></td>
<td><img src="image5" alt="Binary Search Tree Diagram" /></td>
<td><img src="image6" alt="Binary Search Tree Diagram" /></td>
</tr>
<tr>
<td>huge array of booleans (true means</td>
<td>use the value as an index</td>
<td>same as search</td>
</tr>
<tr>
<td>the value is in the set)</td>
<td><img src="image7" alt="Huge Array of Booleans Diagram" /></td>
<td><img src="image8" alt="Huge Array of Booleans Diagram" /></td>
</tr>
</tbody>
</table>

### Examples

1. **Unsorted Array**
   - **Array:** 10 14 4 15 7 21
   - **Search for 4:** Start from the beginning, search until we find it: 10 14 4 15 7 21
   - **Result:** Go to the index

2. **Sorted Array**
   - **Array:** 4 7 10 14 15 21
   - **Search for 7:** Binary search
     - **Result:** Go to the index

3. **Binary Search Tree**
   - **Root:** 6
   - **Search for 4:** Start from root
     - **Result:** Go to the node

4. **Huge Array of Booleans**
   - **Array:** [F T T F ... F]
   - **Search for 3:** Use the value as an index
     - **Result:** Same as search
This data structure is great! Find any value in O(1) time!

```java
boolean[] set = new boolean[Integer.MAX_INT+1];
set[1] = true; // add 1
set[2] = true; // add 2
```

Problems?
Let $M(n)$ be the amount of memory this Set uses, where $n$ = number of elements in the Set. Which is true and the best bound?

a) $M(n) \in O(1)$  
b) $M(n) \in O(\log n)$  
c) $M(n) \in O(n)$  
d) $M(n) \in O(\text{Integer.MAX_VALUE})$  
e) $M(n) \in O(n * \text{Integer.MAX_INT})$
For example...

Integer.MAX_VALUE = 2^{31} - 1

boolean data type is 1 to 2 bytes

2^{31} - 1 * 2 bytes = ~4GB even if your set is nearly empty!

If you are clever and represent the boolean as 1 bit each (0=false, 1=true) then you can get down to 268MB

Even if 268MB fits in your computer’s RAM, reality bites you: if your elements are uniformly randomly distributed across those 268MB then the elements of your set won’t all be in your computer’s fast cache memory, which has a capacity in the 100s of KB (take CS:2630 to learn more!)
Fixing the memory problem

Limit the array to a smaller capacity, say 6

how to add(\(i\)): mark true at index \(i \mod \text{capacity}\)

(bonus: we can also store negative integers now)

a new problem! It looks like 1 is in the set (and 13,19,25, ...) even though we only added 7
add(2)

add(7)

Since many values (1, 7, 13, 19, 25, ...) map to index 1, we need to keep track of which key is stored there.

We’ll have the array store Integers, where null means the bucket is empty and a non-null value is the key stored there.

```java
Integer[] set = new Integer[6]; // capacity=6
set[2 % 6] = 2;
set[7 % 6] = 7;
```
Suppose our set is initially empty as above. What will it look like after the following elements are added? -1, 19, 17, 21, and 8

- a) -1 19 17 21 8 null null
- b) -1 8 17 19 21 null null
- c) null 19 8 21 null 17 -1
- d) 21 8 null 17 null null 19 -1
Administrivia

Last call to sign up for HW8 partners

See ICON announcement (up or coming soon) on how to prepare for the crash course on Git version control in discussion section next week
Collisions!

null | 7 | 2 | null | null | null

0 | 1 | 2 | 3 | 4 | 5

```
add(13)  // 13 % 6 = 1
```

uhoh...
You know that feeling... when someone takes your parking spot
Dealing with collisions

add(13)  // 13 % 6 = 1

Linear probing
Go to the next spot until you find an opening

Chaining
Each bucket is a linked list of elements stored there

...and other techniques!
Suppose our set is initially empty as above. What will it look like after the following elements are added, assuming we use linear probing? 9, 18, 23, 17

a) null null \textcolor{red}{9} 23 \textcolor{red}{18} 17 null

c) null null \textcolor{red}{9} 18 23 \textcolor{red}{17} null

b) null null 23 \textcolor{red}{17} 4 null null

d) null null \textcolor{red}{9} null 18 null null
How should we implement remove() if we are using linear probing? (e.g., remove(7))

a) set the bucket to null
b) Remove the element and move all elements after it left by one space
c) Move all elements after it (up to the next null) left by one space
d) leave a special marker in the bucket that means it is deleted
e) there is no good way to allow remove()
The hash set in your smartphone’s processor that you didn’t know about

all your data (e.g., running programs, the OS, and their data)

cache stores a subset of your data

it is small but that makes it fast!
The hash set in your smartphone’s processor that you didn’t know about

The cache is basically a hash set
here’s one where each bucket can only hold 1 key

Direct Mapped Cache

hash function is take some of the bits of the memory address

compare the key with the key in the bucket
The hash set in your smartphone’s processor that you didn’t know about

The cache is basically a hash set
here’s one where each bucket can hold up to 4 keys

4-Way Set Associative Cache

the key
Address

hash function is take
some of the bits of the
memory address

compare the key with the keys in the bucket
Putting non-integers into a set

```java
String[] set = new String[capacity];
set[???] = "Cat";
```

Where should we put the string “Cat”? 
Putting non-integers into a set

```java
String[] set = new String[capacity];
set[???] = "Cat";
```

Where should we put the string “Cat”?

use a *hash function*

a hash function is just any function that turns an object into an integer
Suppose the hash function for a string is the length

What is the contents after inserting “Cat”, “Dog”, “Froggy”? Assume we use Linear Probing.

<table>
<thead>
<tr>
<th>a)</th>
<th>null</th>
<th>null</th>
<th>null</th>
<th>“Cat”</th>
<th>“Dog”</th>
<th>”Froggy”</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td>b)</td>
<td>“Froggy”</td>
<td>null</td>
<td>null</td>
<td>“Cat”</td>
<td>“Dog”</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>c)</td>
<td>“Cat”</td>
<td>”Dog”</td>
<td>”Froggy”</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>d)</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>“Cat”</td>
<td>“Dog”</td>
<td>null</td>
<td>“Froggy”</td>
</tr>
</tbody>
</table>
a hash function is just any function that turns an object into an integer

for example, Oracle Java distribution’s hash function for Strings

```java
public class String {
    // a string is stored as an array of "chars" (characters)
    private final char value[];

    // hash function for String
    public int hashCode() {
        int h = hash;
        if (h == 0 && value.length > 0) {
            char val[] = value;

            for (int i = 0; i < value.length; i++) {
                h = 31 * h + val[i];
            }
            hash = h;
        }
        return h;
    }
}
```
A paraphrase of Object.hashCode specification in the Java API

• The general contract of hashCode is:
  • during the same run of your program, hashCode on a specific object must always return the same result
  
  • o1.equals(o2) ⇒ o1.hashCode() == o2.hashCode()
  
  • Important to know that

  o1.hashCode() == o2.hashCode DOES NOT IMPLY o1.equals(o2)

  i.e., it is okay for two different objects to have the same hashCode (and pretty much impossible to avoid)

https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#hashCode()
If I have the following code

```java
Map<Cat, Dog> x = new HashMap<Cat, Dog>();
```

which of the following statements is true

a) If you override Cat.equals you must override Cat.hashCode
b) You must override Dog.equals and Dog.hashCode
c) You must override Cat.equals, Cat.hashCode, Dog.equals, and Dog.hashCode
Running time of successful find?

• Linear probing

• expected length of a sequence of non-nulls: \( O \left( 1 + \frac{1}{1-\alpha} \right) \)
  where \( \alpha \) is the load factor

• where \( \alpha = \frac{\# \text{occupied}}{\text{capacity}} \) (\( \alpha \) is called the load factor)

• worst case: \( O(n) \) if the table is allowed to get nearly full (i.e. \( \alpha \) is very close to 1)
  
  Since the running time depends on \( \alpha \), we should decrease it by growing the array when \( \alpha \) becomes too large

  Rule of thumb: if \( \alpha \) increases beyond 0.5 or 0.75, grow the capacity
Running time of successful find?

• Chaining
  • What is the expected length of the longest chain? What is the average length of a chain?

• Of course, we want our hash function to distribute keys well (if everything hashes to a constant number of buckets, lookup time would be $O(n)$)

• If you are lucky enough for the items to be uniformly distributed across buckets then the average length of chains would be $1/\alpha$

• However, the birthday paradox from (see, Discrete Math) tells us that the probability of some collisions is high even if keys are drawn from uniform distribution

• Therefore, $\alpha$ should still be kept sufficiently smaller than 1
HW8: Sliding block puzzle solver

Slide blocks until the 2x2 block is in the middle of the bottom row.

Figure 1

Figure 2

<table>
<thead>
<tr>
<th>Goal configuration</th>
<th>Goal file</th>
</tr>
</thead>
<tbody>
<tr>
<td>row 0</td>
<td>1 1 3 1</td>
</tr>
<tr>
<td>1</td>
<td>1 1 4 2</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>column 0</td>
<td>1 1 3 2</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
HW8: Sliding block puzzle solver

Easy/medium/hard puzzles

First solution should be simple: use a straightforward representation of a “tray” of blocks and brute force search

Open-ended: pick a combination of data structures and search algorithm that solves the most puzzles within a given time
Today’s big ideas

• A new way to store data: *every value gets its own location* (e.g., Integer 3 lives at index 3).

• Problems with storing things by value
  a) How do we use up less memory?
  b) How do we generalize beyond positive integers?

• Fix for problem (a): multiple values share the same location and we deal with the consequences

• Fix for problem (b): introducing *hashing*
Resources

Visualizations of probing and chaining hash tables!

http://www.cs.usfca.edu/~galles/visualization/OpenHash.html

http://www.cs.usfca.edu/~galles/visualization/ClosedHash.html

http://www.cs.usfca.edu/~galles/visualization/ClosedHashBucket.html
Acknowledgements

Cache diagrams – Ferry24Milan

Created by Jaime Carrion
from Noun Project