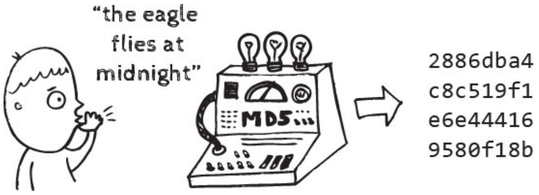
Adam Blank Lecture 10 Winter 2016

CSE 332

Data Abstractions

CSE 332: Data Abstractions

Hashing: Part I

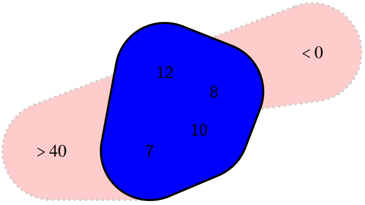


A Special Purpose ADT: "BoundedSet"

BoundedSet ADT

Data	**Set** of **numerical** keys where $0 \le k \le B$ for some $B \in \mathbb{N}$
insert(**key**)	Adds **key** to set
find(**key**)	Returns true if **key** is in the set and false otherwise
delete(**key**)	Deletes **key** from the set

The only difference between Set and BoundedSet is that BoundedSet comes with an upper bound of B.



Some Ideas for BoundedSet Implementations

- Use any of the dictionaries we've already learned! This gets us $\mathcal{O}(\lg n)$ behavior for each of the operations.
- **Direct Address Table:**

false	false	false	false	false	false	false	false	false
has[0]	has[1]	has[2]	has[3]	has[4]	has[5]	has[6]	has[7]	has[8]

```
void add(int value)          { this.data[value] = true; }
boolean contains(int value) { return this.data[value]; }
void remove(int value)       { this.data[value] = false; }
```

- **BitSet:** Stores one or more ints and uses the ith bit to represent the number i.

$$(1234)_{10} = (00000000000000000000010011010010)_2 = \{1,4,6,7,10\}$$

```
void add(int value)          { this.set |= 1 << value; }
boolean contains(int value) { return (this.set >> value) & 1; }
void remove(int value)       { this.set &= ~(1 << value); }
```

Neat Fact: BitSets are often good enough in practice!

Issues with BoundedSets?

Looking ahead, we know HashTables are coming. The clear question to ask is we now have several dictionary choices; so, which do we use when? Why bother using HashTables if BoundedSets are good enough?

BoundedSets Use A Lot of Space!

- Given an input file with four billion integers, provide an algorithm to generate an integer which is not contained in the file. (If you have 1GB of memory? If you have 10MB of memory?)
  Perhaps interestingly, a BitSet is the **best** solution here. The trick is that ints are bounded! If we used a tree, we'd want a B-Tree (why?). A hash table would end up using too much memory.
- Store a set of prime numbers less than $1,000,000$
  There are relatively few prime numbers $(78498/100000 < 8\%)$. The right choice here will be a hash table. The problem with a BoundedSet will end up being space: the BitSet will be way too sparse.
- Determine if a String has all unique characters.
  This is going to be another BitSet problem! Characters are just ints in disguise. The range of ACII characters can easily fit in a BitSet.
- Store a set of students in a course by their Student ID Number.
  We're going to want a HashTable for this one. The number of students is at most around 1000; the number of Student IDs is 1,000,000. The BitSet would be wasting a ton of space!

Issues with BoundedSets?

BoundedSets Only Allow Integer Keys!

If we ever want our keys to be something complicated like Strings or arbitrary Objects, our implementations of BoundedSet aren't going to work. Notice that chars are fine though!

BoundedSets Are Very Bad For Certain Operations!

- Given an input file with four billion integers, determine the the number of unique integers in the file.
  A B-Tree will work better here. In a BitSet, to get the size, we need to loop over the entire key space. In a tree, it's stored upon insertion.
- Store a set of prime numbers with easy access to "previous" and "next".
  The right choice here will be a tree (probably an AVL tree). None of our other data structures give us a useful way of getting "previous" or "next".
- Give a sorted list of Student IDs in the course.
  We already figured out we can't use a BitSet for this one. A HashTable will end up being really bad too. A big downside to HashTables is that they provide no guarantee about ordering!

Putting it all together: Although BoundedSet (and HashTable) are basically the same ADT, they sacrifice operations related to **ordering** (printSorted, findMin, findMax, pred, succ) for better runtime on the core operations.

BoundedSet → HashTable

Putting all these observations together, we see the following:

- Use a Tree if we care about the ordering of the data.
- Use a BitSet if we have **int** keys and the data is not sparse.
- Use a HashTable if the key space is **much** larger than the number of expected items **or** we need non-integer keys

Hash Tables

- Provides $\mathcal{O}(1)$ core Dictionary operations (**on average**)
- We call the key space the "universe": U and the Hash Table T
- We should use this data structure **only** when we expect $|U| >> |T|$
- (Or, the key space is non-integer values.)

These Requirements Are Really Common!

- Compilers: all possible variables vs. defined ones
- Databases: student names vs. actual students
- . . .

BoundedSet → HashTable

Game Plan

To get from BoundedSets to HashTables, we need to make several generalizations/fixes:

- Avoid sparseness of the table
  Solution: Map multiple keys to the same table location
- Allow non-integer keys
  Solution: Provide a mapping from Type → $\mathbb{N}$.
- Deal with "collisions"
  What do we do when two keys are in the same location?

We will handle these one at a time.

Fixing Sparseness

Course Roster

Store a set of students in a course by their Student ID Number.

If we use a BoundedSet, we will need 1,000,000 bytes which is severe overkill for a 20 person class. The solution is to choose a mapping from $U \to T$. The traditional choice is to mod by the table size:

$$\texttt{keyToIndex}(k) = k \bmod |T|$$

Let's look at a few examples:

$U = \{0, 1, \ldots, 1000\}$, $|T| = 10$

Insert: $7, 18, 41, 34, 10$

10	41		34				7	18	
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]

$U = \{0, 1, \ldots, 1000\}$, $|T| = 10$

Insert: $20, 40, 60, 80, 100$

T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]

These all go into the 0 bucket!

Fixing Sparseness: PRIMES!

Our last example showed us that we can get **really bad behavior** with this technique. What happened? Why was that so bad?

The more factors the table size has, the worse the distribution

In general, if x and y are co-prime:

$$ax \equiv bx \pmod{y} \text{ iff } a \equiv b \pmod{y}$$

Technique: Choose $|T|$ to always be prime

- Real-life data has patterns
- The pattern is unlikely to follow a prime sequence
- Some collision strategies only work well with prime table sizes

Investigating Table Size

Consider $|T| = 60$. Note that $60 = 2^2 \times 3 \times 5$. Consider the following insertion sequences:
5, 10, 15, 20, . . . 10, 20, 30, . . . 2, 4, 6, 8, . . .
All of these waste significant amounts of the table!
What if we have $|T| = 61$ instead? These "more likely patterns" won't waste the table.

Non-Integer Keys

Course Roster

Store a set of students in a course by their UWNetID.

We need to find a way to map from $U \to$ int. This idea is called a **hash function**.

Hash Function

A **hash function** is a mapping from the key set (U) to int. Ideally, whatever function we use would have the following properties:

- **Uniform Distribution of Outputs**: There are 2^{32} 32-bit ints; so, the probability that the hash function maps to any individual output should be $\frac{1}{2^{32}}$.
- **Low Computational Cost**: We will be computing the hash function a lot; so, we need it to be very easy to compute.

So, what do hash functions look like in practice?

Hashing Non-ints

Here's some ideas for hash functions for Strings:

- $h(s_0 s_1 \cdots s_{m-1}) = 1$
  This hash function is very fast, but it maps everything to the same index.
- $h(s_0 s_1 \cdots s_{m-1}) = \sum_{i=0}^{m-1} s_i$
  This hash function ignores crucial information about the string: the positions of the characters.
- $h(s_0 s_1 \cdots s_{m-1}) = 2^{s_0} 3^{s_1} 5^{s_2} 7^{s_3} 11^{s_4} \ldots$
  This hash function maps every string to a unique number, but it's difficult to compute.
- $h(s_0 s_1 \cdots s_{m-1}) = \sum_{i=0}^{m-1} 31^i s_i$
  This hash function is a nice compromise. It does have collisions, but all information about the String is used.

Combining Hash Functions

A Few Tricks

- Use all 32 bits (careful, that includes negative numbers)
- Use different overlapping bits for different parts of the hash (This is why a factor of 31^i works better than 256^i)
- When smashing two hashes into one hash, use bitwise-xor
- Rely on expertise of others; consult books and other resources
- If keys are known ahead of time, choose a perfect hash

Hashing a `Person` Object

```
class Person {
   String first; String middle; String last;
   Date birthdate;
}
```

- An inherent trade-off: hashing-time vs. collision-avoidance
- Use all the fields?

Putting The Pieces Together

$$\underbrace{\texttt{E} \xrightarrow{\text{hash function}} \texttt{int}}_{\text{Hash Table Client}} \underbrace{\xrightarrow{\text{mod }|T|} \text{Table Index} \xrightarrow{\text{collision?}} \text{Fixed Table Index}}_{\text{Hash Table Library}}$$

Client Responsibilities

- The client is responsible for choosing a "good" hash function (fast & spreads out outputs)
- The client should avoid "wasting" any part of `E` or the bits of the `int`

Library Responsibilities

- The library is responsible for mapping the integer to a table index
- The library is responsible for choosing the table size
- The library is responsible for keeping track of collisions

Collisions

Definition (Collision)

A **collision** is when two distinct keys map to the same location in the hash table.

A good hash function attempts to avoid as many collisions as possible, but they are inevitable.

How do we deal with collisions?

There are multiple strategies:

- Separate Chaining
- Open Addressing
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

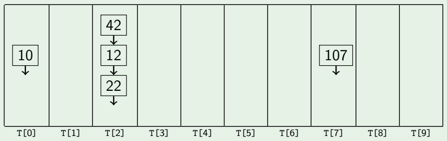
Today, we'll discuss **Separate Chaining**; next time, we'll discuss open addressing.

Separate Chaining

Idea

If we hash multiple items to the same location, store a `LinkedList` of them.

Example (Insert: $10, 22, 107, 12, 42$)



What is the worst case time for find?

Well, if the hash function were $h(k) = c$, then we'd get a linked list of size n in one bucket. So, it's $\mathcal{O}(n)$.

Load Factor

Definition (Load Factor (λ))

The **load factor** of a hash table is a measure of "how full" it is. We define it as follows:

$$\lambda = \frac{N}{|T|}$$

If we're using separate chaining, the average number of elements per bucket is λ.

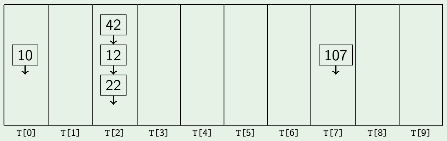
If we do inserts followed by random finds. . .

- Each unsuccessful `find` compares against λ items
- Each successful `find` compares against λ items

For separate chaining, we should keep $\lambda \approx 1$

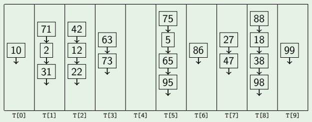
Load Factor Examples

Example (What is the Load Factor?)



What is λ for this hash table? $\lambda = \frac{N}{|T|} = \frac{5}{10} = 0.5$

Example (What is the Load Factor?)

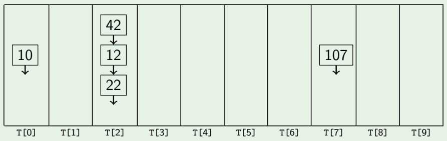


What is λ for this hash table? $\lambda = \frac{N}{|T|} = \frac{21}{10} = 2.1$

Separate Chaining Delete

The algorithm for `delete` is just the reverse of `insert`. We remove it from the linked list:

Example (Delete: 12)



Just like `insert`, the worst case runtime is $\mathcal{O}(n)$, but average is $\mathcal{O}(1)$.