SEARCHING

HASH TABLES

Ruben Acuña

Fall 2018

Minor Revision 11/16/20

2 INTRODUCTION

BASIC IDEA

- In a previous module, we saw the application of binary search trees for implementing symbol tables.
- It was reasonably fast (O(logn)) for both searches and inserts – unlike the list and array implementation.

- A new idea: go back to arrays and plan to map each key to a specific array index.
  - Vital: must implement a **hash function** that takes a Key and produces an integer **hash** that encodes its identity.
  - If we do this, one can imagine using the hash as an array index… constant time look ups!

HASH FUNCTIONS

- Let us say that we want to store into an array with M indices.
  - There will be N elements in this array.

- *The hash should encode the identity of a key.*

- Ultimately, we need to produce hashes in the **range [0, M-1]** and that are **uniformly distributed** over that interval.

- There is a time space trade off here:
  - If M is larger, then we use lots of space but get O(1) (its just an array).
  - If N is larger, then we use lots of time to find elements, O(n), like a list.

PRODUCING A HASH VALUE

- Positive Integers:
  - If k is some positive integer, one may hash it as itself, provided it's not too large. Otherwise we need to consider a modulus.
- Floating-point numbers:
  - If k is some floating-point number, one may hash it as round(kM)….?
- Strings:
  - Convert each digit to number.

```
int hash = 0;
for (int i = 0; i < s.length(); i++)
   hash = (31 * hash + s.charAt(i)) % M;
```

- Compound Keys:
  - Mix up data.

```
int hash = (((area * 31 + exch) % M) * 31 + ext) % M;
```

SIZING A HASH BY M

- If the hash value is large, then we may need to shrink it to fit a smaller domain by doing k%M.
- What should M be?
  - We want to be sure that the modulus operation will space things out.
- Choose something prime and far away from any numbers that look like a power of 2 or 10.
- We are trying to produce a hash that isn't biased to specific part of the input number.

key	hash (M = 100)	hash (M = 97)
212	12	18
618	18	36
302	2	11
940	40	67
702	2	23
704	4	25
612	12	30
606	6	24
772	72	93
510	10	25
423	23	35
650	50	68
317	17	26
907	7	34
507	7	22
304	4	13
714	14	35
857	57	81
801	1	25
900	0	27
413	13	25
701	1	22
418	18	30
601	1	19

Modular hashing

JAVA IMPLEMENTATION

- In Java, the method *public int hashCode()* is the standard way to access a class's hash.
- The default implementation of this is the address of the object.

- The rules:
  - For any collection of keys (of objects of the same class), running hashCode on them should uniformly distribute those values over [-MAXINT, MAXINT].
  - x.equals(y) if and only if x.hashCode() == y.hashCode().

- Note this method returns an integer. However, the maximum size of an integer is larger than M!
- The easy way to fix this is to compute: (x.hashCode() & 0x7fffffff) % M

JAVA SAMPLE

- Simple:
  - Follow the idea of mixing data with a prime factor: 31x+y.
  - If something already has a hashCode function implemented, use it.

(this is roughly the approach used in the Java libraries.)

```
public class Transaction {
    ...
    private final String who;
    private final Date when;
    private final double amount;
    public int hashCode() {
        int hash = 17;
        hash = 31 * hash + who.hashCode();
        hash = 31 * hash + when.hashCode();
        hash = 31 * hash
          + ((Double) amount).hashCode();
        return hash;
    }
    ...
}
```

REQUIREMENTS FOR A GOOD HASH FUNCTION

- Hash functions should produce values that are:
  - Consistent (deterministic)
  - Efficient to compute
  - Uniformly distributed

The last requirement, uniform distribution, is something we won't really touch on. It is a TCS and number theory topic that is beyond our class.

In fact, we will simply assume all our hash functions have this property – i.e., that it is a **perfect hash**.

- This gives a goal to aim for when creating a hash function.
- As well as something to build off when implementing the hashtable.

STORING KEYS & VALUES

- Now we have some idea of how to map a key to a hash, and what a hash should look like.
- The second part of implementing hashtables is to store keys/values into an array of size M.
- The simplest way would be using the hash as an array index.
- However, remember that a hash can be any integer – even bigger than M unless M is huge.
  - Making M huge is kind of pointless though… why?
- Well, take the modulus of a hash to get an index that fits. Easy.
- No matter what though, we will end up hashing different keys to the same value – we need **collision-resolution**.

Let's try hashing with:
hash(x)=x%3.

And inserting:
insert(0)→hash is 0
insert(1)→hash is 1
insert(2)→hash is 2
insert(3)→hash is 0
insert(4)→hash is 1
insert(5)→hash is 2
and so on…

This is a problem!

COLLSION-RESOLUTION

- There two basic ideas we can try:
  1. Make each place in the array store a collection of all the elements that have that hash – **chaining**.
  2. When hashing to an occupied index, looking for the next available space – **linear probing**.
     - Of course, there can be a complicated rule to find out what the next available space is.

12 SEPARATE CHAINING

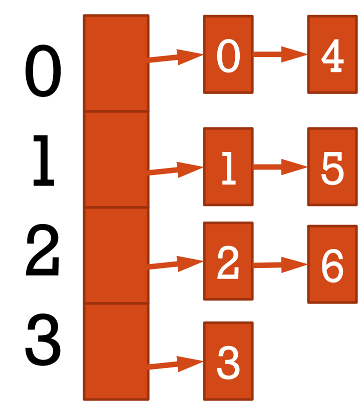
SEPARATE CHAINING

- Here's the idea:
  - Create an array of size M.
    - Each index will represent one hash value after mod'ing.
  - At each position of the array, put a linked list.
  - When inserting or searching an element, use the hash function to get the index of the array – it will return a list where that element must be.
  - Treat inserting or searching on that list as in any normal list situation.

Let M=4.

Try hashing with:
hash(x)=x%4.

And inserting:
insert(0)→hash is 0
insert(1)→hash is 1
insert(2)→hash is 2
insert(3)→hash is 3
insert(4)→hash is 0
insert(5)→hash is 1
insert(6)→hash is 2
and so on…



ALGORITHM TRACE

- Assume you have a separate chaining hashtable with M=5. Give the final hashtable after adding these keys: 3, 11, 7, 0, 14, 1. Use the hash function hash(k) = k mod 5. Your drawing should include the main size M array, and lists located at each index. Do not worry about including the values that would be paired with the keys in a real hashtable.

CHAINING IMPLEMENTATION

```
public class SeparateChainingHashST<Key, Value> {
    private int N; // number of key-value pairs
    private int M; // hash table size
    private SequentialSearchST<Key, Value>[] st;

    public SeparateChainingHashST() {
        this(997);
    }

    public SeparateChainingHashST(int M) {
        this.M = M;
        st = (SequentialSearchST<Key, Value>[]) new SequentialSearchST[M];
        for (int i = 0; i < M; i++)
            st[i] = new SequentialSearchST();
    }
```

CHAINING IMPLEMENTATION

- Hash:
  - Mask out the sign bit.
  - Apply a mod.

Get:
  - Add element to inner ST in array.

Put:
  - Same as get.

```
private int hash(Key key) {
    return (key.hashCode() & 0x7fffffff) % M;
}

public Value get(Key key) {
    return (Value) st[hash(key)].get(key);
}

public void put(Key key, Value val) {
    st[hash(key)].put(key, val);
}
```

ADDITIONS

- How hard would it be to implement delete(Key key)?
- How hard would it be to implement resize(int newM)?

PERFORMANCE

- Since there are a total of N elements spread across M indices, there are on average N/M elements in each index.
  - Search:
    - ~N/2M
  - Insert:
    - ~N/M

The textbook has an elaborate explanation for why the number of elements in each list will be close to a constant factor of N/M – perhaps to be covered on another day.

19

LINEAR PROBING

LINEAR PROBING

- Here's the idea:
  - Create an array of size M, where M is at least as big as N. (Why?)
  - At each position of the array, we put a key/value.
  - When inserting or searching an element, use the hash function to get the index of the array.
  - If that index is empty, use it. Otherwise, move forward one element, check if it is empty, and so on.

Let M=4.

Try hashing with:
hash(x)=x%4.

And inserting:
insert(0)→hash is 0
insert(1)→hash is 1
insert(4)→hash is 0
insert(5)→hash is 1
and so on…

0	1	2	3
0			

0	1	2	3
0	1		

0	1	2	3
0	1	4	

0	1	2	3
0	1	4	5

ALGORITHM TRACE

- Assume you have a linear probe hashtable with M=11. Simulate the hashtable to add these keys: 3, 11, 7, 0, 14, 1. Use the hash function hash(k, i) = (k mod 11 + i) mod 11, where *i* is number of times the algorithm has tried to insert the key. For your answer, create a table that shows the main size M array. Do not worry about including the values that would be paired with the keys in a real hashtable.

LINEAR, PT1

- Parallel arrays – not the best.
- Does the default constructor look right?

```
public class LinearProbingHashST<Key, Value>
{
    private int N; // number of key-value pairs in the tab
    private int M; // size of linear-probing table
    private Key[] keys; // the keys
    private Value[] vals; // the values

    public LinearProbingHashST() {
        this(16);
    }

    public LinearProbingHashST(int size) {
        M = size;
        keys = (Key[]) new Object[M];
        vals = (Value[]) new Object[M];
    }

    private int hash(Key key) {
        return (key.hashCode() & 0x7fffffff) % M;
    }
}
```

LINEAR, PT2

- Remember that we want to have a 'shifting' action occur if an index is already full.

```
    public void put(Key key, Value val) {
        if (N >= M/2)
            resize(2*M);
        int i;
        for (i = hash(key); keys[i] != null; i = (i + 1) % M)
            if (keys[i].equals(key)) {
                vals[i] = val;
                return; }
        keys[i] = key;
        vals[i] = val;
        N++;
    }

    public Value get(Key key) {
        for (int i = hash(key); keys[i] != null; i = (i + 1) % M)
            if (keys[i].equals(key))
                return vals[i];
        return null;
    }
}
```

LINEAR, PT3

```
private void resize(int cap) {
    LinearProbingHashST<Key, Value> t;
    t = new LinearProbingHashST<>(cap);
    for (int i = 0; i < M; i++)
        if (keys[i] != null)
            t.put(keys[i], vals[i]);
    keys = t.keys;
    vals = t.vals;
    M = t.M;
}
```

ADDITIONS

- How hard would it be to implement delete(Key key)?

PERFORMANCE

A little complicated – Knuth ended up proving it.

As elements are added into an array, short consecutive regions of occupied indices will form from collisions.

- Note: larger regions will grow faster than smaller ones.

Proof defines a value α, which is N/M, meaning the portion of the array filled by elements. Clearly, the more elements are in the array, the more likely collisions are to occur. Based on α, Knuth defined:

$$search\ hit: \frac{1}{2}(1 + \frac{1}{1-\alpha}) \qquad search\ miss/insert: \frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$$

In general, aim for $\alpha = N / M \approx ½$.

PERFORMANCE SUMMARY

Algorithm (data structure)	avg: search hit	avg: insert	Efficiently support ordered operations?
sequential search (unordered linked list)	N/2	N	No
binary search (ordered array)	lg N	N	Yes
binary search trees	1.39lg N	1.39lg N	Yes
chaining	N/(2M)	N/M	No
linear probing	<1.5	<2.5*	No

*Still a chance of resizing!