COMPGC05 Algorithmics
Section 5: Hash Tables

Jens Krinke
Centre for Research on Evolution, Search & Testing
Software Systems Engineering Group
Department of Computer Science
University College London

This lecture is being recorded.

- This lecture is being recorded in video format by the University College London (UCL).
- The recording may be used by UCL for the purposes of offering it to students and staff. This may include conversion to digital format and storing and publication on UCL's servers.
- Any person who does not wish to appear in the recording should contact the lecturer, Dr Jens Krinke.

Section 5. Hash Tables

Hash Table
Hash Functions
Collisions
Probing
Chaining

Textbook Chapter 9

Hash Tables

- Hashing is also known as
  - Key-to-address translation
  - Scatter storage
- Hashing potentially enables us to do table look up with O(1) efficiency!!!
  (how else do you think Google manages to be fast?).

» But W(n) is still O(n).

Simple Example

- Normally lookup-table retrieval is at best log(n), but suppose you have a LUT where the keys are integers between 0 and M-1.
- Then you could 'cheat' and use the keys to index an array
- So retrieval would be O(1).

Hash Functions

What about payroll numbers?

- These go from 11000000 to 11000675.
- We could index an array by a function of the payroll number:
  index = PN - 11000000;

Hash Tables

- The general idea is to use an array of size M to store entries in a LUT, and
- Use a **Hash Function**, h, to convert the key value to an array index.
- h: {all possible keys}→{0, 1, …, M-1}
- Given {key, value}, set objectArray[h(key)] = value.
- This is to good to be true, there must be a problem with it. What is it?

Collisions

- Typically there are many more possible keys than array indices (e.g. suppose keys are surnames).
- So the Hash function will be many-to-one.
- So some non-equal keys will have the same hash value. (e.g. 'griffin' and 'smith' might have the same value).
- So when we store a data item, we may find another item already where we want to put it.
- We need to handle these collisions.

Hash Table Implementation

- Hash table implementation calls for two choices:
- Choice of hash function that
  - minimises collisions
  - is simple and efficient to compute
- Collision resolution strategy.

Hashing Strings. 1.

- Suppose that the keys of the LUT are strings (pretty general-purpose).
- And suppose we want to store entries in a table of fixed size M.
- So, we require that the hash function h(key) gives values in the range 0,…,(M-1).

Hashing Strings. 1.

- One way to implement h(key) is to convert String's to integers via UNICODEs.
- Each character is represented by a number.
- In Java, we can use a cast to get the integer UNICODE value of a character:

```
char c = 'S';
int a = (int)c; //a equals 83.
```


Hashing Strings. 2.

So, we can add up all UNICODEs and use the sum modulo the size of the table.

```
public static int hash(String key, int M) {
  int n = 0;

  for (int i=0; i<key.length(); i++) {
    n += (int)key.charAt(i);
  }

  return n % M;
}
```

N.B. There are better ways of doing this.

What size should the Hash Table be?

- Rule of Thumb 1.
  (memory wastage vs. collision avoidance)
  - table size should be slightly bigger than the number of entries.
- Rule of Thumb 2.
  (collision avoidance):
  - Choose M to be a prime number
  - M not too close to a power of 2.

Collision Resolution. 1.

- Whatever hash function we use there will **always** be collisions.
- The **Overflow Chaining** solution is to allow array slots to hold multiple entries.
- Each array element is a linked list of table entries.
- W(n) is O(n)
- But move-to-front or migrate-to-front can be used for each individual list.

Collision Resolution. 2.

- Another collision resolution strategy is **Open Addressing**.
- If the slot specified by the hash function is empty, use it; otherwise use a slot nearby.
- The process of looking for an empty slot is called **probing**.
- **Linear probing** looks at slots h(k), then h(k)+1, h(k)+2, etc

Class outline for open addressing

```
public class HashTableLProbe {
  protected class Entry {
    protected String key;
    protected Object value;
    ...
  }

  protected Entry[] entryArray;

  public HashTableLProbe() {
    entryArray = new Entry[50];
  }
  public HashTableLProbe(int size) {
    entryArray = new Entry[size];
  }

  //public interface
}
```

Insertion, deletion & retrieval with open addressing

- Retrieval starts by looking at the entry indexed by the hash value, then looks as the next entry, etc.
- Search can be recognised as unsuccessful when an empty entry is found.
- **But entry removal complicates this considerably!**
- Need to use **tombstones** to mark slots that are now empty but weren't always.


```
public void insert(String key, Object value)
throws TableOverflowException {
  //Compute the hash value
  int index = hash(key, entryArray.length);

  //Probe linearly to find empty slot.
  int count = 0;

  while (entryArray[index] != null &&
         (!entryArray[index].key.equals("Tombstone")) &&
         count != entryArray.length) {
    index = (index + 1) % entryArray.length;
    count += 1;
  }

  if (count == entryArray.length) {
    throw new TableOverflowException();
  } else {
    entryArray[index] = new Entry(key, value);
  }
}
```



```
public Object retrieve(String key)
throws KeyNotFoundInTableException {
  //Compute the hash value
  int index = hash(key, entryArray.length);

  //Probe linearly looking for match.
  int count = 0;

  while (entryArray[index] != null &&
         (!entryArray[index].key.equals(key)) &&
         count != entryArray.length) {
    index = (index + 1) % entryArray.length;
    count += 1;
  }

  if (entryArray[index]==null
      || count==entryArray.length) {
    throw new KeyNotFoundInTableException();
  }

  return entryArray[index].value;
}
```



```
public void delete(String key)
throws KeyNotFoundInTableException {
  //Compute the hash value
  int index = hash(key, entryArray.length);

  //Probe linearly looking for match.
  int count = 0;

  while (entryArray[index] != null &&
         (!entryArray[index].key.equals(key)) &&
         count != entryArray.length) {
    index = (index + 1) % entryArray.length;
    count += 1;
  }

  if (entryArray[index]==null
      || count==entryArray.length) {
    throw new KeyNotFoundInTableException();
  }

  entryArray[index].key = "Tombstone";
}
```

Example

```
HashTableLProbe myLUT = new HashTableLProbe(10);

myLUT.insert("Priscilla", new Integer(41));
myLUT.insert("Travis", new Integer(34));
myLUT.insert("Samuel", new Integer(28));
myLUT.insert("Helena", new Integer(39));
myLUT.insert("Andrew", new Integer(14));
myLUT.insert("Kay", new Integer(24));
myLUT.insert("John", new Integer(67));

myLUT.delete("Travis");
myLUT.delete("John");
myLUT.delete("Kay");
myLUT.insert("Dani", new Integer(15));
myLUT.insert("John", new Integer(67));
myLUT.insert("Travis", new Integer(34));
```


Example

Priscilla	931	1
Travis	633	3
Samuel	615	5
Helena	589	9
Andrew	609	9
Kay	293	3
John	399	9
Dani	380	0

0		
1		Priscilla
2		
3		Travis
4		
5		Samuel
6		
7		
8		
9		

Example

Travis	633	3
John	399	9
Kay	293	3
Dani	380	0

0	Andrew		Dani
1	Priscilla		
2	John		
3	Travis		Travis
4	Kay		
5	Samuel		
6			
7			
8			
9	Helena		John

```
myLUT.delete("Travis");
myLUT.delete("John");
myLUT.delete("Kay");
myLut.insert("Dani", new Integer(15));
myLUT.insert("John", new Integer(67));
myLUT.insert("Travis", new Integer(34));
```



```
public void delete(String key)
throws KeyNotFoundInTableException {
  ...

  if (entryArray[index]==null
      || count==entryArray.length) {
    throw new KeyNotFoundInTableException();
  }

  // Better: check for end of chain
  if (entryArray[(index + 1) % entryArray.length]
      != null) {
    entryArray[index].key = "Tombstone";
  } else {
    entryArray[index] = null;
  }
}
```

```
public void delete(String key)
throws KeyNotFoundInTableException {
  ...

  // Better: check for end of chain
  if (entryArray[(index + 1) % entryArray.length]
      != null) {
    entryArray[index].key = "Tombstone";
  } else {
    entryArray[index] = null;

    // Cleanup obsolete tombstones
    index = (index - 1) % entryArray.length;
    while (entryArray[index] != null
           && entryArray[index].key.equals("Tombstone")) {
      entryArray[index] = null;
      index = (index - 1) % entryArray.length;
    }
  }
}
```


Double Hashing

- Linear probing causes bunching in the table.
- We can use **quadratic probing** instead:
- Looks in slots, h(k), h(k)+1, h(k)+4, h(k)+9, etc.
- Helps stop runs merging.
- **Double hashing** generalises this idea.

Double Hashing

- Probe offset is a function of the original index:

```
index = hash(key, entryArray.length);

while (entryArray[index] != null) {
  index = (index + hash2(index)) % M;
}
```

- `hash2(index)` must never be 0
- `M` should be prime, `hash2` should be quite random and relatively prime to `M`.

Hash Table Performance. 1.

- Performance of open addressing degenerates rapidly as the table fills up.
- The table should be expanded when it gets 85% to 90% full.
- On average, 5 probes are required for a hash table that is 66% full.
- **But** this number is independent of n, so retrieval is still O(1)!

Hash Table Performance. 2.

- Overflow chaining hash tables have much more **graceful degradation.**
- No catastrophic loss of performance as the number of entries nears the array size.
- Con: much larger storage requirements.
- Overall Conclusion: Careful open addressing is generally preferred.

Do Exercise 4

Summary

- Hash Table
- Hash Functions
- Collisions
- Probing
- Chaining

END