CSCI 136: Data Structures and Advanced Programming
Lecture 29
Hash collisions
Instructor: Dan Barowy
Williams

Topics
Hash collisions
Graphs

Your to-dos

1. Read before Wed: Bailey, Ch. 16.4.
2. Lab 9 (solo lab), due Tuesday 5/3 by 10pm.

Note about lab 9:
You may use the structure5 Hashtable implementation.
Practice Quiz

Hash tables

Hash codes

Hashing so important that every object in Java has a built-in hash function.

<table>
<thead>
<tr>
<th>hashCode</th>
</tr>
</thead>
<tbody>
<tr>
<td>public int hashCode()</td>
</tr>
<tr>
<td>Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by java.util.</td>
</tr>
<tr>
<td>The general contract of hashCode is:</td>
</tr>
<tr>
<td>• Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.</td>
</tr>
<tr>
<td>• If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.</td>
</tr>
<tr>
<td>• It is not required that if two objects are unequal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.</td>
</tr>
<tr>
<td>As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java programming language.)</td>
</tr>
<tr>
<td>Returns:</td>
</tr>
<tr>
<td>a hash code value for this object.</td>
</tr>
<tr>
<td>See Also:</td>
</tr>
<tr>
<td>equals(java.lang.Object), System.identityHashCode(java.lang.Object)</td>
</tr>
</tbody>
</table>

Hash collisions

A hash collision is when two or more distinct keys have the same hash value.

index(“Dan”) → 6

index(“Benedict Cumberbatch”) → 6
Perfect hash function

A perfect hash function is a hash function that ensures that distinct keys map to distinct indices. I.e., there are no collisions.

Problem: It's pretty darn hard to come up with a perfect hash function.

1. You need to know all possible keys in advance.
2. If the number of possible keys is large, it is expensive to compute \(O(n^2)\) time and expensive to store \(O(n)\) space.

With a good hash table implementation, “imperfect” hash functions are usually good enough.

Dealing with collisions

There are two approaches to dealing with collisions:

1. Change your hash function.
2. Change your hash table design.

The easier of the two approaches turns out to be #2.

Open addressing

Open addressing is a method for resolving collisions in a hash table. Collisions are resolved by probing, which is a predetermined method for searching the hash table (aka a probe sequence). On insertion, probing finds the first available bucket. On lookup, probing searches until either the key is found or an empty space is found.
Linear probing

Suppose our keys are Strings and our hash function is

```java
((int) key.charAt(0)) % A.length
```

(i.e., a low-quality hash function).

0 1 2 3 4 5 6 7

key: “Dan”, value: -11
index(“Dan”) → 4

key: “Dirk”, value: 20
index(“Dirk”) → 4

collision!

Linear probing works by scanning for \( h(key) + c \times i \), where \( c \) is a constant (usually 1) and \( i \) is the \( i \)th attempt.

0 1 2 3 4 5 6 7

key: “Dan”, value: -11
index(“Dan”) → 4

key: “Dirk”, value: 20
index(“Dirk”) → 4

retry

collision!

Linear probing

Linear probing works by scanning for \( h(key) + c \times i \), where \( c \) is a constant (usually 1) and \( i \) is the \( i \)th attempt.

0 1 2 3 4 5 6 7

key: “Don”, value: -11
index(“Don”) → 4

key: “Dirk”, value: 20
index(“Dirk”) → 4

Collision!

Likelihood of collisions grows as cluster grows.

Our table is still half empty! This is bad!
Linear probing

\[ h(key) + c \times i \]

Changing \( c \) helps some.

E.g., \( c = 2 \).

key: “Dan”, value: -11
index(“Dan”) \( \rightarrow \) 4

key: “Dirk”, value: 20
index(“Dirk”) \( \rightarrow \) 4

Changing \( c \) helps some.
But it can also **make the problem worse**.

key: “Dan”, value: -11
index(“Dan”) \( \rightarrow \) 4

key: “Dirk”, value: 20
index(“Dirk”) \( \rightarrow \) 4

Deletions are also problematic.

Addressed by leaving a sentinel value at deleted location.

Deletions are also problematic.

Doesn’t reclaim space until all colliding entries deleted.
External chaining

External chaining is a method for resolving collisions in a hash table. Collisions are resolved by storing more than one value in a bucket, e.g., using a list.

Same bad hash function:

```
(((int) key.charAt(0)) % A.length)
```

collision!

key: “Dan”, value: -11
index(“Dan”) → 4

key: “Dirk”, value: 20
index(“Dirk”) → 4

External chaining: deletion

Deletion is trivial.

Complexity

<table>
<thead>
<tr>
<th>Method</th>
<th>Successful</th>
<th>Unsuccessful</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear probes</td>
<td>(\frac{1}{2} \left( 1 + \frac{1}{1-\alpha} \right))</td>
<td>(\frac{1}{2} \left( 1 - \frac{1}{(1-\alpha)^2} \right))</td>
</tr>
<tr>
<td>Double hashing</td>
<td>(\frac{1}{\alpha} \ln \frac{1}{1-\alpha})</td>
<td>(\frac{1}{1-\alpha})</td>
</tr>
<tr>
<td>External chaining</td>
<td>(1 + \frac{1}{2\alpha})</td>
<td>(\alpha + e^{-\alpha})</td>
</tr>
</tbody>
</table>

Figure 15.11 Expected theoretical performance of hashing methods, as a function of \(\alpha\), the current load factor. Formulas are for the number of association compares needed to locate the correct value or to demonstrate that the value cannot be found.
Hash Table Expansion

When a hash table fills up, we should expand, just as with a Vector. But there are some problems...

Hash Table Expansion

Virtually every hash function relies on the size of the underlying array to do the hashing. Recall:

```java
int index(K key) {
    return abs(h(key) % A.length);
}
```

When a hash table expands, we usually address this by rehashing all elements during a copy. Why is this OK?

Hash Table Expansion

Another issue: hash table performance degrades severely as it fills up.

Recall that we can have an effectively full hash table even when there is actually space.

\[ h(key) + c \times i \]

where \( c = 2 \)

Hash Table Expansion

Therefore, we resize before the table is likely to be full.

Let \( n \) be the number of elements stored in a hash table.

Let \( m \) be the number of buckets.

Load factor = \( n \ / \ m \)

When the load factor is reached, the hash table is resized.
Hash Table Expansion

There are two ways to find a good load factor.

1. Careful analysis of the probability of attempting to insert more than one element into the same bucket, combined with a preference for acceptable average slowdown.

2. Empirical measurement, combined with a preference for acceptable average slowdown.

A load factor of 0.7-0.8 is generally accepted to be a good threshold.

Recap & Next Class

Today:
- Hash collisions
- Graphs

Next class:
- Graph algorithms