Habanero Extreme Scale Software Research Project
Comp215: Performance

Zoran Budimlić (Rice University)
“To suffer the penalty of too much haste, which is too little speed.”

- Plato
Never sacrifice correctness for performance

Correct but slow always beats buggy but fast!
Always think carefully how your changes are affecting the code
  Regression tests are your friend
  Some systems won’t even let you commit until you pass the regression tests
There are exceptions to this rule, though
  Depends on the definition of “correctness”
  Approximate computation
Do not sacrifice elegance or design for performance

Remember Dan’s Friday lecture
Maybe your change didn’t introduce a bug, but it complicated the code so much that the next change is bound to do so
Better algorithm beats better coding

An O(n) algorithm almost always beats an O(n^2) algorithm

Why “almost”?

- Know your input domain
- If all you ever do is sorting lists that are at most 10 elements, then O(n^2) might be good enough
  - In fact, it might be faster than O(n log(n))
- A “slower on average” algorithm that always meets the deadline may be much better than a faster algorithm that sometimes misses it
- Make sure you document these kinds of decisions

Charles Leiserson: O(2D) vs O(2^D)
Engineering does matter

Use efficient data structures
Avoid unnecessary indirection
Optimize the common path
Follow the 80/20 rule:
  80% of work is done by 20% of code
Might even be a 90/10 rule!
Know your tools

What the compiler can/cannot do
- Common scalar optimizations
- Optimizing the memory layout
- Tail recursion removal

What the runtime can/cannot do
- How efficient is the garbage collector

How efficient are the libraries
- java.lang.String

Profiling your code
- Filter the actual results from the noise
- Java profiling has limited usability
Arrays

We love our lists!
  Recursive
  Immutable
  Generic

But sometimes, arrays are MUCH faster
  If your data can be indexed
  $O(1)$ deletion/lookup (vs $O(n)$ for lists)
  $O(1)$ insertion (same as lists)
But, but, but mutation!!!!!
Yes, mutation!

Not uniformly evil
Sometimes unavoidable
  Cyclic data structures
  When past events matter
  Bank transactions

Functional: return effects (return a new Queue containing the result of the operation)

Mutating: modify data

When used carefully, mutating code can have the appearance of functional code
  Clients of your code can have a stateless view of the code
  Internals of your code can have mutation
“The only place success comes before work is in the dictionary.”

- Vince Lombardi
Dictionaries

Dynamic-set data structure for storing items indexed using keys

Supports operations: Insert, Search, and Delete

Applications:
- Symbol table of a compiler
- Memory-management tables in operating systems
- Large-scale distributed systems

We already have this! Treaps! (and no mutation!)
- $O(\log(n))$ insert, search and delete

Hash Tables:
- Effective way of implementing dictionaries
- Generalization of ordinary arrays
- $O(1)$ expected for insert, search and delete
Direct-address tables vs. Hash tables

U - Universe of possible keys
K - set of keys actually stored in the dictionary
Use ordinary arrays to implement a dictionary
Direct addressing: array[index]
Element whose key is k is obtained by indexing into the k\textsuperscript{th} position of the array.
Applicable when we can afford to allocate an array with one position for every possible key
   i.e. when the universe of keys U is small: |U| \sim |K|
Dictionary operations can be implemented to take O(1) time
Impractical when |K| \ll |U|
   Use hash tables for this, with size proportional to |K|
Need to define a \textit{function} that maps keys to slots in the hash table
Hashing

Hash function $h$: Mapping from $U$ to the slots of a hash table $T[0..m-1]$

$$h : U \rightarrow \{0, 1, \ldots, m-1\}$$

With arrays, key $k$ maps to slot $A[k]$

With hash tables, key $k$ maps or “hashes” to slot $T[h[k]]$

$h[k]$ is the hash value of key $k$

If keys are the same (according to Java’s equals() method), the result of the hash function has to be the same

Different keys do not guarantee different hash values
Hashing

\[ h(k_1) = h(k_3) \]

Hash Table (size m)

\( U \) (universe of keys)

\( K \) (actual keys)

Collision

\[ h(k_2) = h(k_5) \]
Issues with hashing

Multiple keys can hash to the same slot — collisions are possible
  Design hash functions such that collisions are minimized
  But avoiding collisions is impossible
  Design collision-resolution techniques

Search will cost $O(n)$ time in the worst case.
  However, all operations can be made to have an expected complexity of $O(1)$. 
Collision Resolution

Chaining:

Store all elements that hash to the same slot in a linked list

Store a pointer to the head of the linked list in the hash table slot

Open Addressing:

All elements stored in hash table itself

When collisions occur, use a systematic (consistent) procedure to store elements in free slots of the table
Collision resolution by chaining

\[ h(k_1) = h(k_4) \]

\[ h(k_2) = h(k_5) = h(k_6) \]

\[ h(k_3) = h(k_7) \]

\[ h(k_8) \]

\[ m-1 \]
Collision resolution by chaining
Complexity of operations with chaining

**Insert**
- Insert x at the head of the list
  - $O(1)$

**Search**
- Search for x in the list at the hash(key(x))
  - Worst-case complexity $O(n)$
  - Good hash function is the key
  - Expected complexity $O(1)$

**Delete**
- Search for x in the list at the hash(key(x))
- Remove x from the list
  - Same complexity as Search
Load factor

Load factor: \( L = \frac{n}{m} \) (average number of keys per slot)
- \( m \) - number of slots in the table
- \( n \) - total number of keys in the table

Rehashing

Common practice: rehash the table when \( L \) reaches 50%
Create a new (larger) table with new hash function
Scan the old table, insert every element in new table
Expensive
  - But, amortized cost of inserts is still \( O(1) \)
Hashing: The Division Method

Division method for a hash function:

\[ H(x) = x \% M \]

where the key \( x \) is converted an integer somehow and \( M \) is a constant.

Generally \( M \) is the size of the hash table.

A number of theoretic patterns may emerge depending on the choice of \( M \).

Programmers often choose their hash table size \( M \) to be a large prime number to avoid collisions.
A popular hashing function is called the multiplication method:

\[ H(x) = (\text{int}) M \times (A \times x - (\text{int}) (A \times x)) \]

where \( 0 < A < 1 \) and \( M = \text{HASH}\_\text{SIZE} \) are constants.

In words, we multiply \( M \) by the fractional part of \( A \times x \) and then round down.

Invented by Donald Knuth.

The choice of \( A \) depends on the data. Knuth recommends:

\[ A = (\sqrt{5} - 1)/2 \]

If the keys \( x \) are uniformly distributed, this choice of \( A \) minimizes the collisions.
Hashing non-int values

Object.equals():

```java
public boolean equals(Object obj) {
    return (this == obj);
}
```

Integer.equals():

```java
public boolean equals(Object obj) {
    return (this == obj);
}
```

Long.hashCode():

```java
public int hashCode() {
    return (int) (value ^ (value >>> 32));
}
```
Hashing non-primitive values

**JKeyValue.equals():**

```java
public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof JKeyValue)) return false;
    JKeyValue jKeyValue = (JKeyValue) o;
    if (!string.equals(jKeyValue.string)) return false;
    return value.equals(jKeyValue.value);
}
```

**JKeyValue.hashCode():**

```java
public int hashCode() {
    int result = string.hashCode();
    result = 31 * result + value.hashCode();
    return result;
}
```
java.util.Hashtable<K, V>

```java
public class Hashtable<K, V>
    extends Dictionary<K, V>
    implements Map<K, V>, Cloneable, java.io.Serializable

public Hashtable(int initialCapacity, float loadFactor)
```

Any non-null object can be used as a key or as a value.

keys must implement the `hashCode` method and the `equals` method

java.util.Hashmap is similar, allows null for key or value, and isn’t synchronized