CS 2110
Lecture 16

Hashing
• Dictionary ADT
• Hash functions
• Hash tables
• Collision resolution
Dictionaries

hashing, 221–252
chaining, 225–228, 250 pr.
The **dictionary** ADT

- Given a word, return its definition
- Given a contact’s name, return their phone number
- Given a student’s ID number, return their NetID

- All involve looking up a value associated with some key
- Also known as:
  - Map
  - Associative array
Dictionary interface

Generic on two type parameters

\[ \text{interface Map}<K, V> \{ ... \} \]

- K: Type of keys
- V: Type of values

- Like an array, but with objects (instead of ints) as “indices”
- Keys are unique (a key can only map to one value)
  - But value could be a list

- put\((key: K, value: V)\)
  Associate value with key

- get\((key : K) : V\)
  Return value associated with key

- remove\((key : K)\)
  Remove any association for key

- keySet(): Set\(<K>\)
  Allow iterating over keys

- containsKey(), size(), …
Map<String, LocalDate> bdays = ...;

bdays.put("Alan Turing", LocalDate.of(1912, 6, 23));
bdays.put("John von Neumann", LocalDate.of(1903, 12, 28));

println("Turing was born on " + bdays.get("Alan Turing"));
for (String name : bdays.keySet()) {
    println(name + " was born on " + bdays.get(name));
}
println("Do I know Katherine Johnson’s birthday? " +
    (bdays.containsKey("Katherine Johnson") ? "yes" : "no");
Data structures for implementing a Dictionary

- Unsorted list
  - Put: $O(1)$
  - Get: $O(N)$

- Sorted list
  - Put: $O(N)$
  - Get: $O(\log N)$
  - Keys must be comparable

- Binary search tree
  - Put: $\Omega(\log N) - O(N)$
  - Get: $\Omega(\log N) - O(N)$
  - Keys must be comparable

- Desired: array-like
  - Put: $O(1)$
  - Get: $O(1)$
  - Any key type
If keys were integers, could just use an array

• How hard could it be?
  • How to turn ordinary objects (keys) into integers?
  • How big an array will be needed?
  • What if two keys correspond to the same index?

• Hash: “to chop to pieces; to make a confused muddle of”
Example

- $h(\text{“Turing”}) \rightarrow 3$
- $h(\text{“von Neumann”}) \rightarrow 7$
- $h(\text{“Johnson”}) \rightarrow 5$

```java
class Entry<K, V> {
    K key;
    V value;
}
```

<table>
<thead>
<tr>
<th>Index</th>
<th>Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td>null</td>
</tr>
<tr>
<td>2</td>
<td>null</td>
</tr>
<tr>
<td>3</td>
<td>(Turing, 1912-06-23)</td>
</tr>
<tr>
<td>4</td>
<td>null</td>
</tr>
<tr>
<td>5</td>
<td>(Johnson, 1918-08-26)</td>
</tr>
<tr>
<td>6</td>
<td>null</td>
</tr>
<tr>
<td>7</td>
<td>(von Neumann, 1903-12-28)</td>
</tr>
</tbody>
</table>
Hash codes and indices

• Two step process:
  1. Hash a key into an int ("hash code")
  2. Turn a hash code into an array index ("index derivation")
   • Depends on array length!

• Object defines a hashCode() method
  • Any Java object can be used as a key
  • Implementer must ensure hash code is consistent with equality
    • If overriding equals(), must override hashCode() too!

• Keys should be immutable
  • If hash code changes, entries will be at the wrong index
Implementing `hashCode()`

- Goal: two non-equal objects should be unlikely to share a hash code
  - Should depend on all of an object’s state
  - Should depend on ordering of any sequential state (e.g. arrays)
  - Should span whole range of integers

- `Objects.hash()`, `Arrays.hashCode()` can help

- When analyzing performance, we will assume `hashCode()` is $O(1)$
  - Long strings, data tables would not make performant keys
Deriving an index

- \( h(\text{“Hopper”}) \rightarrow -95141326; \)
  
  Now what?

- Need an index between 0 and array length

- Solution: compute the remainder
  
  \[ \text{index} = \text{abs}(\text{hash} \mod \text{a.length}) \]

<table>
<thead>
<tr>
<th>Index</th>
<th>Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td>null</td>
</tr>
<tr>
<td>2</td>
<td>null</td>
</tr>
<tr>
<td>3</td>
<td>(Turing, 1912-06-23)</td>
</tr>
<tr>
<td>4</td>
<td>null</td>
</tr>
<tr>
<td>5</td>
<td>(Johnson, 1918-08-26)</td>
</tr>
<tr>
<td>6</td>
<td>(Hopper, 1906-12-09)</td>
</tr>
<tr>
<td>7</td>
<td>(von Neumann, 1903-12-28)</td>
</tr>
</tbody>
</table>
Collisions

• Multiple hash codes could have the same remainder, yielding the same index

• Multiple keys could have the same hash code
  • Only 4 billion ints, but many more possible object values

• Collisions are unavoidable
Poll: collision likelihood

How many students need to attend lecture before it is likely that two share a birthday? \((p > 0.5)\)

A. 365
B. 183
C. 92
D. 23
How many students need to attend lecture before it is likely that two share a birthday? (p > 0.5)

365: 0%
183: 0%
92: 0%
23: 0%
Collision resolution approaches

**Chaining**

- Treat array elements as “buckets” storing a *collection* of entries (e.g. a linked list)

- Finding the right bucket is O(1), but searching it will be slower

**Probing**

- Array elements point directly to entries

- If desired element is occupied, pick the next element to try according to a probing sequence
Chaining example

<table>
<thead>
<tr>
<th>Key</th>
<th>Hash code</th>
<th>Index (%8)</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jenny</td>
<td>126</td>
<td>6</td>
<td>x5309</td>
</tr>
<tr>
<td>Eddie</td>
<td>97</td>
<td>1</td>
<td>x7766</td>
</tr>
<tr>
<td>Brenda</td>
<td>86</td>
<td>6</td>
<td>x5635</td>
</tr>
<tr>
<td>Jack</td>
<td>255</td>
<td>7</td>
<td>x5555</td>
</tr>
<tr>
<td>Stacy</td>
<td>118</td>
<td>6</td>
<td>x7666</td>
</tr>
</tbody>
</table>
Complexity analysis: Get(k)

• Procedure
  1. Compute k.hashCode(): assume O(1)
  2. Derive index with %: O(1)
  3. Search bucket: TBD

• Best case: no collisions
  • Bucket is either empty or contains a list of length 1
  • Total cost: O(1)

• Worst case: everything collides
  • (either array is too small or hash function is very poor)
  • Bucket contains linked list of length $N$
  • Linear search: $O(N)$
Expected case

• Assume key indices are uniformly distributed
  • Requires good hash function or defensive index derivation
• Average cost to look up a random key = average length of chain

What is the average length of a chain in a hash table with $N$ keys and $M$ buckets?

A. $N$
B. $N/M$
C. $M/N$
D. $M*N$
What is the average length of a chain in a hash table with N keys and M buckets?

- N: 0%
- N/M: 0%
- M/N: 0%
- N*M: 0%
Load factor

- $\lambda = \frac{\text{number of elements}}{\text{number of buckets}}$

- May be >1 for chaining (but not for probing)

- Expected cost of lookup with chaining is $O(\lambda)$
  - For probing, see DSAJ

- Is that good?
  - If array size is fixed, then $\lambda$ is $O(N)$
  - If array size is proportional to $N$, then $\lambda$ is $O(1)$

- Must use a \textit{dynamic array} for good performance
Resizing

• Goal: keep $\lambda \sim 0.75$

• When load factor exceeds target, double size of array

• Cannot simply copy old array elements; must re-hash keys
  • Index derivation formula has changed with array size!

• Cost of resizing: $O(N)$

• But cost can be amortized over many “put” calls before resizing is needed again
  • If new array size is a multiple of the old size, the cost of resizing averages out to $O(1)$ per “put” call
## Linear probing example

<table>
<thead>
<tr>
<th>Key</th>
<th>Hash code</th>
<th>Index (%8)</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jenny</td>
<td>126</td>
<td>6</td>
<td>x5309</td>
</tr>
<tr>
<td>Eddie</td>
<td>97</td>
<td>1</td>
<td>x7766</td>
</tr>
<tr>
<td>Brenda</td>
<td>86</td>
<td>6</td>
<td>x5635</td>
</tr>
<tr>
<td>Jack</td>
<td>255</td>
<td>7</td>
<td>x5555</td>
</tr>
<tr>
<td>Stacy</td>
<td>118</td>
<td>6</td>
<td>x7666</td>
</tr>
</tbody>
</table>
Probing challenge: deletions

- Need a third state for each element: null, Entry, or “vacant”
  - Aka “tombstones”
- Tombstones count towards load factor, slowing things down (but can be cleared when resizing)
Concurrency

- Threads
- Race conditions
Threads
Concurrent execution

• A computer can run multiple programs at once
• A program can perform multiple tasks at once
  • Tasks are performed by threads

• How is concurrent execution achieved?
  How can programs take advantage?
Example: division of labor

• Vehicle simulator
  • Autopilot must periodically process navigation sensors and update controls
  • In the meantime, can simulate physics until the next autopilot period

• During each period, autopilot and physics are *independent*, may execute *concurrently*
Why break programs into concurrent tasks?

**Advantages**
- More responsive
  - Separate interactive code from long calculations
- Higher performance
  - Computers have multiple cores
  - Make progress when one task waits

**Disadvantages**
- New kinds of bugs
  - Race conditions
  - Deadlock
- Difficult to test, debug
How computers work

- Computer = Memory + CPU cores + I/O peripherals
Memory (RAM)

• Same memory is accessible to all CPU cores
• Different programs are not allowed to see each other’s memory
• But different threads in the same program can all access that program’s memory
  • Java Objects can be shared by threads working on different tasks
CPU cores

• Each core works on one sequence of instructions at a time
  • Knows which line of code to execute next
  • Knows stack of activation records that brought it to this point
Threads of execution

• A **thread** encapsulates work to be performed by a CPU core
  • Sequence of instructions
  • Stack of activation records

• Each program starts with one thread running **main()**
  • Bottom record on stack

• Additional threads can start running any function
  • Can be given references to existing objects when created – objects are now **shared** between threads
Demo
Java Thread API
Abstractions

• A “thread” is an abstraction for a sequence of instructions
  • To realize a thread on a computer requires low-level interaction with OS
  • Job of OS scheduler is to execute instructions on a CPU core

• A Java **Thread** is an object to manage threads of execution at a high level
  • Can get Thread object for main thread
  • Can create new Thread objects to start additional threads
Java’s Thread class

• Java represents threads with objects of class Thread
• Use constructor, methods to set up and manage the thread
  • Which code should it run? (may include references to existing objects)
  • When should it start running?
  • Is it done yet? Etc.
• Included in Java 1.0; mistakes were made
  • Several methods are deprecated
  • Several features (priorities, yielding) aren’t very useful in practice
What code to run

• Pass a **Runnable** as a constructor argument
  • Interface with one abstract method: `void run()`

• Sharing objects
  • Add fields to class implementing Runnable
  • Capture variables in anonymous function
  • Any object accessible through another object (e.g. elements in a list)

• (Alternative: extend `Thread`, override its `run()`; less flexible)
Demo

• Perform two tasks simultaneously
  • `main()` first starts another thread to do `task 2`
  • Then `main()` does `task 1` itself
  • Meanwhile, the second thread is working on `task2`
  • Program does not stop until both threads have finished their tasks
Starting and stopping threads

• Constructing a Thread does not start it running; must call start()
  • start() returns immediately (does not wait for run() to finish), so calling thread can proceed with other work

• When run() returns, thread stops
  • If run() throws an uncaught exception, thread stops
  • No return value, no one is notified
  • Other threads can wait for it to stop by calling join() on Thread object
    • join() is a blocking method

• When all non-daemon threads stop, program exits
  • “Daemon” thread: background worker; will not hold up program exit
Waiting for things to happen

Wait for time to pass
• `Thread.sleep()` – stop executing code for a specified amount of time

Waiting for other code to do something
• `Object.wait()` – stop executing code, and release locks, until notified by another thread
  • Topic of next lecture

Both can throw `InterruptedException`;
best practice is to abort work
Demo: SwingWorker

Creation thread

Constructor

execute()

Worker thread

doInBackground()

EDT

done()
Poll

Can the progress bar be updated directly by doInBackground()?

A. No – this would be prevented at compile-time
B. No – this would result in an exception at runtime
C. Technically yes, but it would violate the progress bar’s thread safety contract
D. Yes, this is fine
Multitasking

• With multiple cores, threads might actually run simultaneously on their own dedicated core
  • But typically computers are running more programs and threads than they have cores

• Reality: cores can be interrupted at any time and told to work on a different instruction sequence
  • “Context switch”
  • Program does not know this is happening

• OS uses this to rapidly switch between applications/threads

Concurrency bugs arise in either case!
How many threads?

• Some **oversubscription** (more threads than cores) is good
  • Lots of computing involves waiting around
• But too many threads is bad
  • Each Thread object consumes non-trivial system resources
  • Overhead for OS to manage
• What if many more tasks than cores?

• **Thread pools**
  • A fixed number of Thread objects executes a sequence of Runnables from a queue
  • Not all tasks can make progress at once
    • Tasks should be independent, or else use Fork/Join framework
• See **Executor**
Race conditions
Shared mutable state

- When a (mutable) object is shared between threads, its state (values of its fields) may change unexpectedly
  - May change between lines of code
  - May change while executing a single line of code

- Sequential logic no longer guarantees correctness
  - Invariants can be violated by someone else’s actions
Example: shared queue

```java
static Queue q = ...

• Thread 1:
  if (!q.isEmpty()) {
    q.poll();
  }

• Thread 2:
  if (!q.isEmpty()) {
    q.poll();
  }
```

• Suppose q.size() == 1

• Could this throw an exception?
  • A: No
  • B: Yes

  • Sequential logic says “no”
  • What if a thread is experiences a context switch between invoking isEmpty() and poll()?
Example: shared primitive

```
shared.x = 0;

• Thread 1:
  shared.x++;

• Thread 2:
  shared.x++;
```

• What value will shared.x have after both threads have completed?

• Even simple statements are not “atomic”

• This statement is really a sequence of 3 instructions:
  1. Load value of x from RAM into CPU register
  2. Increment value in register
  3. Store register value at x’s location in RAM
Example: shared primitive

Scenario 1
1. T1 LOAD \( (\text{reg}_1 \leftarrow 0) \)
2. T1 INC \( (\text{reg}_1 \leftarrow 1) \)
3. T1 STORE \( (x \leftarrow 1) \)
4. T2 LOAD \( (\text{reg}_2 \leftarrow 1) \)
5. T2 INC \( (\text{reg}_2 \leftarrow 2) \)
6. T2 STORE \( (x \leftarrow 2) \)

Scenario 2
1. T1 LOAD \( (\text{reg}_1 \leftarrow 0) \)
2. T2 LOAD \( (\text{reg}_2 \leftarrow 0) \)
3. T2 INC \( (\text{reg}_2 \leftarrow 1) \)
4. T2 STORE \( (x \leftarrow 1) \)
5. T1 INC \( (\text{reg}_1 \leftarrow 1) \)
6. T1 STORE \( (x \leftarrow 1) \)
Poll

What is the smallest possible final value of shared variable box.x (initially 0) if four threads run the following loop:

```java
for (int i=0; i<10; ++i) {
    shared.x++;
}
```

A. 0  
B. 1  
C. 2  
D. 10  
E. 40
Stopping subtleties

• Why is Thread.stop() deprecated?
  • Could be interrupted in middle of a method, before class invariant is restored
  • Object may be shared with other threads, who can observe the violation
  • Synchronization may have been protecting object, but goes away when thread stops

• How to safely stop?
  • Write run() as a loop, checking a shared signal each iteration
  • Break up long computations
Race conditions are serious bugs

- Linked to northeast power grid blackout in 2003
- Responsible for massive radiation overdose in Therac-25 radiation therapy machine
  - 3 patients died

- Preventing race conditions
  - Option 1: Avoid shared mutable state (e.g. one thread “owns” the state, others send messages to owner)
    - Swing assumes this approach
  - Option 2: Use synchronization, atomic instructions, to write correct concurrent data structures (topic of next lecture, 4410)
Next up: Synchronization

How to safely coordinate between interdependent threads?