COMP 250
Lecture 26
maps
Nov. 8/9, 2017
A map is a set of pairs \( \{(x, f(x))\} \).

Each \( x \) in domain maps to exactly one \( f(x) \) in codomain, but it can happen that \( f(x_1) = f(x_2) \) for different \( x_1, x_2 \), i.e. many-to-one.
Familiar examples

Calculus 1 and 2 ("functions"):

\[ f : \text{real numbers} \rightarrow \text{real numbers} \]

Asymptotic complexity in CS:

\[ t : \text{input size} \rightarrow \text{number of steps in a algorithm.} \]
A map is a set of (key, value) pairs. For each key, there is at most one value.
The black dots here indicate objects (or potential objects) of type K or V that are *not* in the map.
<table>
<thead>
<tr>
<th>Map</th>
<th>Keys</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address book</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Map</td>
<td>Keys</td>
<td>Values</td>
</tr>
<tr>
<td>-----------</td>
<td>------</td>
<td>-----------------</td>
</tr>
<tr>
<td>Address book</td>
<td>Name</td>
<td>Address, email..</td>
</tr>
<tr>
<td>Caller ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Map</td>
<td>Keys</td>
<td>Values</td>
</tr>
<tr>
<td>------------------</td>
<td>----------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>Address book</td>
<td>Name</td>
<td>Address, email..</td>
</tr>
<tr>
<td>Caller ID</td>
<td>Phone #</td>
<td>Name</td>
</tr>
<tr>
<td>Student file</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Map</td>
<td>Keys</td>
<td>Values</td>
</tr>
<tr>
<td>-------------------------</td>
<td>--------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>Address book</td>
<td>Name</td>
<td>Address, email..</td>
</tr>
<tr>
<td>Caller ID</td>
<td>Phone #</td>
<td>Name</td>
</tr>
<tr>
<td>Student file</td>
<td>ID or Name</td>
<td>Student record</td>
</tr>
<tr>
<td>Index at back of book</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Map</td>
<td>Keys</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>Address book</td>
<td>Name</td>
<td>Address, email..</td>
</tr>
<tr>
<td>Caller ID</td>
<td>Phone #</td>
<td>Name</td>
</tr>
<tr>
<td>Student file</td>
<td>ID or Name</td>
<td>Student record</td>
</tr>
<tr>
<td>Index at back of book</td>
<td>keyword</td>
<td>List of book pages</td>
</tr>
</tbody>
</table>
Each (key, value) pair is called an entry.

In this example, there are four entries.
In COMP 250 this semester, the above mapping has ~600 entries. Most McGill students are not taking COMP 250 this semester.

Student ID also happens to be part of the student record.
Map ADT

• put( key, value )  // add

• get(key)  // why not get(key, value) ?

• remove(key)

• ...

13
Data Structures for Maps

How to organize a set of (key, value) pairs, i.e. entries?
Array list

```plaintext
put(key, value)
get(key)
remove(key)
```
Singly (or Doubly) linked list

put( key, value )

get(key)

remove(key)
Important property of keys

Two different keys *can* have *(map to)* the same value.

One key *cannot* have *(map to)* two values.
Special case #1: what if keys are *comparable*?
Array list (sorted by key)

```
put(key, value)
get(key)
remove(key)
```
Binary Search Tree (sorted by key)

- put(key, value)
- get(key)
- remove(key)
minHeap (priority defined by key)

put(key, value)
get(key)
remove(key)
Special case #1: what if keys are comparable?

Special case #2: what if keys are unique positive integers in small range?

Then, we could use an array of type $V$ (value) and have $O(1)$ access.

This would not work well if keys are 9 digit student IDs.
Special case #1: what if keys are *comparable*?

Special case #2: what if keys are unique positive integers in small range?

General Case:

Keys might not be comparable.

Keys might be not be positive integers.

e.g. Keys might be strings or some other type.
Strategy for the General Case (Hash Maps – next lecture):

Try to define a map from keys to *small* range of positive integers (array index), and then store the corresponding values in the array.

Recall notation: black dots are not part of the map.
Rest of today:

Define a map from keys to *large* range of positive integers.

Such a map is called a *hash code*. 
“default” hashcode() map in Java

By default, “obj1 == obj2” means “obj1.hashcode() == obj2.hashcode()”
String.hashCode() in Java

For each String, define an integer.
Example hash code for Strings (not used in Java)

\[ h(s) \equiv \sum_{i=0}^{s.length-1} s[i] \]

e.g.

\[ h("eat") = h("ate") = h("tea") \]

ASCII values of ‘a’, ‘e’, ‘t’ are 97, 101, 116.
String.hashCode() in Java

\[ s.\text{hashCode}() \equiv \sum_{i=0}^{s.\text{length}-1} s[i] x^{s.\text{length} - 1 - i} \]

where \( x = 31 \).
String.hashCode() in Java

\[ \text{s.hashCode()} \equiv \sum_{i=0}^{s.length-1} s[i] \times x^{s.length-1-i} \]

where \( x = 31 \).

e.g. \( s = \text{“eat”} \) then \( \text{s.hashCode()} = 101 \times 31^2 + 97 \times 31 + 116 \)

\begin{align*}
\text{‘e’} & \quad \text{‘a’} & \quad \text{‘t’} \\
\text{s[0]} & \quad \text{s[1]} & \quad \text{s[2]} \\
\text{s.length = 3} & \quad & \end{align*}
String.hashCode() in Java

\[ s.\text{hashCode}() \equiv \sum_{i=0}^{s.\text{length}-1} s[i] \times s.\text{length} - 1 - i \]

where \( x = 31 \).

e.g. \( s = \text{“ate”} \) then \( s.\text{hashCode}() = 97 \times 31^2 + 116 \times 31 + 101 \)

\begin{align*}
\text{‘a’} & \quad \text{‘t’} & \quad \text{‘e’} \\
s[0] & \quad s[1] & \quad s[2]
\end{align*}

s.length = 3
String.hashCode() in Java

\[ s\text{.hashCode()} \equiv \sum_{i=0}^{s\text{.length}-1} s[i] \times (31)^{s\text{.length} - 1 - i} \]

If \ s1\text{.hashCode}\ () == s2\text{.hashCode}\ () \ then \ ... \ ?
String.hashCode() in Java

\[ s.\text{hashCode}() \equiv \sum_{i=0}^{s.\text{length}-1} s[i] \times (31)^{s.\text{length} - 1 - i} \]

If \( s1.\text{hashCode}() == s2.\text{hashCode}() \) then ... ?

\( s1 \) may or may not be the same string as \( s2 \).
String.hashCode() in Java

\[
s.\text{hashCode}() \equiv \sum_{i=0}^{s.\text{length}-1} s[i] \times (31)^{s.\text{length} - 1 - i}
\]

If \( s1.\text{hashCode}() == s2.\text{hashCode}() \) then … ?

\( s1 \) may or may not be the same string as \( s2 \).

If \( s1.\text{hashCode}() != s2.\text{hashCode}() \) then … ?
String.hashCode() in Java

\[ s.\text{hashCode}() \equiv \sum_{i=0}^{s.\text{length}-1} s[i] \times (31)^{s.\text{length} - 1 - i} \]

If \( s1.\text{hashCode}() == s2.\text{hashCode}() \) then ...

\( s1 \) may or may not be the same string as \( s2 \).

If \( s1.\text{hashCode}() != s2.\text{hashCode}() \) then ...

\( s1 \) is a different string than \( s2 \).
ASIDE: Use Horner’s rule for efficient polynomial evaluation

\[ s[0] \times x^3 + s[1] \times x^2 + s[2] \times x + s[3] \]

There is no need to compute each \( x^i \) separately.
ASIDE: Use Horner’s rule for efficient polynomial evaluation

\[ s[0] \times 31^3 + s[1] \times 31^2 + s[2] \times 31 + s[3] \]

\[ = ( s[0] \times 31^2 + s[1] \times 31^1 + s[2] ) \times 31 + s[3] \]

\[ = ( ( s[0] \times 31^1 + s[1] ) \times 31 + s[2] ) \times 31 + s[3] \]

\[
\begin{align*}
h &= 0 \\
\text{for} & \ (i = 0; \ i < s. length; \ i++) \\
& \ h = h \times 31 + s[i]
\end{align*}
\]

For a degree \( n \) polynomial, Horner’s rule uses \( O(n) \) multiplications, not \( O(n^2) \).
We want to map the keys to a small range of positive integers.

How?