JCF hashed collections

Java collections are in the java.util package: Use HashMap for P1

Hashed collections

<K> k    non null Object implements hashCode() and equals()
<V> v    non null Object that K "maps to"

collection's default construction parameters:
    float loadFactor default 0.75    // jdk 7.*
    int initialCapacity default 16    // jdk 7.*

void clear() removes all <K><V> mapping, sets size to 0
<V> put (<K> k, <V> v) returns null if no previous mapped value
    else returns previously mapped value
<V> get (<K> k) returns value mapped to k
<V> remove (<K> k) returns value mapped to k, removes mapping
int size() number of mapped values
Collection<V> values() returns collection of values
    Collection can get an iterator for a HashMap
HashMap w/ Iterator example

// This is a partial, "pseudocode", example
import java.util.HashMap;
import java.util.Iterator;

public class HashIterator { // keys don't have to be Integer
    private static final int capacity = n;
    HashMap <Integer, DataObject> hashMap;
    Integer key;

    public static void main (String[] arg) {
        HashIterator hi = new HashIterator();
    }

    public HashIterator() {
        hashMap = new HashMap<Integer, DataObject>();
        for(int i = 0; i < capacity; i++) {
            // create aDataObject and get its key
            hashMap.put(key, aDataObject);
        }
        // get iterator to "visit" all values in HashMap
        Iterator<DataObject> iterator = hashMap.values().iterator(); // uses the collection
        while (iterator.hasNext()) {
            aDataObject = iterator.next();
        }
    }
}
HashMap <K, V> Collection <V> returned by values()
    Iterator not fail-safe, must explicitly synchronize.  // no worries …
    iterator is not ordered wrt <K> or <V>
    permits null values to be mapped
    Recommended JCF hashed ADT

HashTable <K,V> like a HashMap, "legacy hashed" collection.
    can also returned an Enumeration with keys() and elements(),
    fail-safe.

HashSet <E> implements a set backed by a hashed ADT

LinkedHashMap <K, V> maintains a doubly-linked list of values
    predictable iteration order – slower operations (faster than TreeMap)
    order of insertion (not keys) default constructor
    access order: true– most recently used, false – least recently used
Array accessing

Consider the primitive array type as a collection typed, sequential allocation, direct access.

1D array
int [] v = new int [N];
base address of v[0] and an offset of N * typeSizeOf
v[i] value stored at memory location v[0] + (typeSizeOf * i )

2D array
int [][] m = new int [J][K]
base address of m[0] and an offset of J * K * typeSizeOf
v[j][k] value stored at memory location
v[0] + ( j * K + k) * typeSizeOf

Java does not have a sizeOf method/operator like C…
# include <stdio.h>
int main (int argc, char ** argv) {

    const int J = 3;
    const int K = 4;
    int m[J][K];

    for(int x = 0; x < J; x++)
        for (int y = 0; y < K; y++)
            m[x][y] = x*y + y;  // values

    printf("sizeof(int) = %d \n", sizeof(int));

    // print values and addresses
    for(int x = 0; x < J; x++)
        for (int y = 0; y < K; y++)
            // next line breaks -- "\t memory" -- for slide display
            printf("m[%d][%d] = %d at %u index offset %u \t memory offset %u\n",
                   x, y, m[x][y], &m[x][y], &m[x][y] - &m[0][0],
                   (&m[x][y] - &m[0][0]) * sizeof(int));
}
$ g++ -o arrayAllocation arrayAllocation.c

$ arrayAllocation

sizeOf(int) = 4

m[0][0] = 0 at 2293496 index offset 0 memory offset 0
m[0][1] = 1 at 2293500 index offset 1 memory offset 4
m[0][2] = 2 at 2293504 index offset 2 memory offset 8
m[0][3] = 3 at 2293508 index offset 3 memory offset 12
m[1][0] = 0 at 2293512 index offset 4 memory offset 16
m[1][1] = 2 at 2293516 index offset 5 memory offset 20
m[1][2] = 4 at 2293520 index offset 6 memory offset 24
m[1][3] = 6 at 2293524 index offset 7 memory offset 28
m[2][0] = 0 at 2293528 index offset 8 memory offset 32
m[2][1] = 3 at 2293532 index offset 9 memory offset 36
m[2][2] = 6 at 2293536 index offset 10 memory offset 40
m[2][3] = 9 at 2293540 index offset 11 memory offset 44

$
Hashing

Access data (table) with minimum search time
   ideally direct access to key    O(1)

Use hash function to return location of value in a "map collection" of
   <key><value> held in a store

\[
\text{value} = \text{store}.\text{get}(\ \text{hf(arg)}) \quad \// \quad \text{key} \Leftarrow \text{hf(arg)}
\]

Hash function's 2 desired properties

1. easy and fast to compute
   mod function       \text{key} \ % \ \text{capacity}
   store is a contiguous allocation: array

2. place data uniformly across store
   store capacity is prime number > expected collection size
   Caldwell, C.K., The first 50 million primes,
   \url{http://primes.utm.edu/lists/small/millions}
   if load factor \approx 0.75 then 25 % store space is open (unused)
   capacity \approx 1.33 * expected collection size
WayPoint's <K> key ??

Consider a Class WayPoint that holds information about navigational places on a terrain that will be stored in HashMap<K, WayPoint> wayPoint.

<table>
<thead>
<tr>
<th>WayPoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>- x : int</td>
</tr>
<tr>
<td>- y : int</td>
</tr>
<tr>
<td>- height : int</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

+ WayPoint( int xPos, yPos, heightValue, ...) |
| ... |

What are possible key values to use with HashMap ?

aHashMap.put(key, aWayPoint);

or

aWayPoint = aHashMap.get(key)
Object’s `hashCode()`

Java Object documentation

```java
public int `hashCode()`
Returns a hash code value for the object.
```

```java
key ← anObject.hashCode()
```

```java
HashMap<Integer, WayPoint> wayPoint;
...
```

```java
aWayPoint = new WayPoint(x, y, height);
```

```java
wayPoint.put(aWayPoint.hashCode(), aWayPoint)
```

------------- output from program  ------------------

```java
HashMap wayPoint created, size = 463
last aWayPoint created had values:
x = 300 y = 220 height = 146 hashCode key = 225912260
set aWayPoint to null, and then use aKey to get it back
```

```java
aWayPoint = null
aWayPoint = wapPoint.get(225912260)
aWayPoint values = 300 220 146
```
Point key

key ← Point object

key = new Point(x, y)
x = key.getX() y = key.getY()

HashMap<Integer, WayPoint> wayPoint;
...
wayPoint = new HashMap<Point, WayPoint> ();
...
aWayPoint = new WayPoint(x, y, height);
pKey = new Point(x, y);
if (wayPoint.put(pKey, aWayPoint) != null) // test for error

------------------ output from program ------------------- -------------------

HashMap wayPoint created, size = 463

last aWayPoint created had values:
x = 300 y = 220 height = 146 key = java.awt.Point[x=300,y=220]
set aWayPoint to null, and then use key to get it back
aWayPoint = null
aWayPoint = wapPoint.get(java.awt.Point[x=300,y=220])
aWayPoint values = 300 220 146
encoding an int key

key \leftarrow \text{int or Integer}

key = x + (y - 20) \times 512 \quad // \text{20 offset in Y 512 X dim}
x = \text{key} \mod 512 \quad y = y/512 + 20 \quad // \text{integer division}

\text{HashMap<Integer, WayPoint> wayPoint;}

\ldots

\text{wayPoint} = \text{new HashMap<Integer, WayPoint> ()} ;

\ldots

aWayPoint = \text{new WayPoint}(x, y, \text{height});
aKey = x + (y - 20) \times 512;
if (\text{wayPoint}.\text{put}(aKey, aWayPoint) \neq \text{null}) \quad // \text{test for error}

------------------ output from program -------------------
\text{HashMap wayPoint created, size} = 463

\text{last aWayPoint created had values:}
x = 300 \quad y = 220 \quad \text{height} = 146 \quad \text{key} = 102700
\text{set aWayPoint to null, and then use key to get it back}
aWayPoint = \text{null}
aWayPoint = \text{wayPoint}.\text{get}(102700)
aWayPoint values = 300 220 146
encoding a String key

key ← String

Concatenate object’s fields to make a String key

```
HashMap<String, WayPoint> wayPoint;
...
wayPoint = new HashMap<String, WayPoint>();
...
sKey = String.format("%3d::%3d", x, y);
if (wayPoint.put(sKey, aWayPoint) != null)  // test for error
```

------------------ output from program -------------------

HashMap wayPoint created, size = 463

last aWayPoint created had values:
x = 300  y = 220  height = 146  key = 300::220
set aWayPoint to null, and then use key to get it back
aWayPoint = null
aWayPoint = wayPoint.get(300::220)
aWayPoint values = 300 220 146
String to int key

Convert key to number ... 

```java
// Horner's rule for polynomial function (of 37)
int hash(String key, int capacity)
    hashValue = 0
    for (i = 0; i < key.length; i++)
        hashValue = 37 * hashValue + key.charAt(i)
    hashValue = hashValue % capacity
    if (hashValue < 0) hashValue += capacity
    return hashValue
```

$ java StringHash "Mike Barnes" 97
String: Mike Barnes = 92

$ java StringHash "Comp 282 Advanced Data Structures" 97
String: Comp 282 Advanced Data Structures = 56
Hash functions & Social Security number

Lets examine a “near universal” USA identification number -- SSN

40 out of 280 million SSN are assigned to more than 1 person ...

AAA-GG-SSSS  area – group – serial
area code        derived from zip code of person requesting a number
group code       4 orders for odd/even group number assignment
serial code      strict serial assignment (next number)

After 6/25/2011 Social Security numbers were assigned randomly.

mod  hf = SSN % size   hf = SSSS % size
selection hf = SSSS     hf = SSSS – selects substring as K
folding  hf = A_0 + A_1 + A_2 + G_0 + G_1 + S_0 + S_1 + S_2 + S_3


collisions, open addressing. linear probing

Hash store has a capacity of buckets
Buckets can also have a capacity = values they can hold (assume 1)

Collision occurs when hash function maps key into a full bucket.

Open addressing finds another bucket in store for value

linear probing check next bucket until space is found

\[
\text{pos} = \text{hf(key)} \\
\text{while } (\text{store[pos] isFull()}) \\
\quad \text{pos} = (\text{pos} + 1) \mod \text{capacity}
\]

clusters of full buckets grow and merge causing long probe iterations (primary clustering)

probe after deletion – target further in probe sequence?
buckets have deletedCount if deletedCount != 0 probe isFull() true if capacity == storedCount
insertions decrement deletedCount to 0
**double hashing**

Use second hash function ($hf_2$) to determine probe offset

**beware:** non-linear probes can have endless cycle through store ...

\[
\begin{align*}
hf_1(key) & \quad key \mod \text{capacity} \quad // \text{or capacity is } i\text{th prime number} \\
hf_2(key) & \quad n - (key \mod n) \quad // \text{or } n = (i\text{th} - 1) \text{ prime number}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Key</th>
<th>hf1 (key)</th>
<th>hf2 (key)</th>
</tr>
</thead>
<tbody>
<tr>
<td>83</td>
<td>83</td>
<td>48</td>
</tr>
<tr>
<td>17</td>
<td>17</td>
<td>30</td>
</tr>
<tr>
<td>44</td>
<td>44</td>
<td>48</td>
</tr>
<tr>
<td>83</td>
<td>83</td>
<td>30</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
<td>48</td>
</tr>
<tr>
<td>48</td>
<td>48</td>
<td>30</td>
</tr>
</tbody>
</table>

- **capacity** = 13
- **hf1** = $k \mod \text{capacity}$
- **hf2** = capacity $- (k \mod \text{capacity})$

<table>
<thead>
<tr>
<th>Keys</th>
<th>Linear Probes</th>
<th>Double Probes</th>
<th>HF2 Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>44</td>
<td>5, 4</td>
<td>4, 0, 9</td>
<td>8</td>
</tr>
<tr>
<td>17</td>
<td>4, 5, 6</td>
<td>9, 0, 4, 8</td>
<td>9</td>
</tr>
<tr>
<td>83</td>
<td>4, 5, 6, 7</td>
<td>4, 0, 9</td>
<td>8</td>
</tr>
<tr>
<td>30</td>
<td>4, 5, 6, 7</td>
<td>9, 0, 4, 8</td>
<td>4</td>
</tr>
<tr>
<td>48</td>
<td>4, 5, 6, 7</td>
<td>4, 0, 9</td>
<td>8</td>
</tr>
</tbody>
</table>
7,000 unique randomly ordered keys are inserted into a 10,000 store with open addressing hashing. \( hf() = \text{key} \% 9,973 \)

3 experiments with 10,000 trials

- \( \text{address} = (\text{address} + 1) \% \text{capacity} \)
- \( \text{address} = (\text{address} + \text{offset}) \% \text{capacity} \)
- \( \text{address} = (\text{address} + (\text{key} \% \text{offset}) \% \text{capacity}) \) // double hashing
rehash, bucket capacity

Increase size of the table and rehash collection
  increase to next prime number?
rehash time consuming growth overhead
shrink on deletions?

Restructure hash collection
  Increase bucket capacity – postpones overflow resolution
store becomes collection of references to buckets + 1 access
buckets have "internal" load size (internal unused space)
decrease bucket capacity on deletions?
Chaining

Chaining buckets are reference to link list of overflow buckets, buckets could be ArrayLists.

Hash for initial entry to hash store. If key not found search bucket's "chain" for target.
load factor \((lf)\) = (size of hashed collection) / (capacity of hash store)

linear probe expected comparisons – function of \(lf\) and capacity

\[
\text{success} = \frac{1}{2} \left[ 1 + \frac{1}{(1 - lf)} \right]
\]

When \(lf = 2/3\)

success comparisons = 2

\[
\text{unsuccessful} = \frac{1}{2} \left[ 1 + \frac{1}{(1 - lf)^2} \right]
\]

unsuccessful comparisons = 5

chaining performance – search is a function of number of chains

expected \(lf\) = chain length = size / chains

\[
\text{success} = 1 + \frac{lf}{2}
\]

performance can have a wide range.

\[
\text{unsuccessful} = lf
\]

How could you experimentally verify these expectations?