CS 106X, Lecture 28
Hashing
Zachary Birnholz

reading:
Programming Abstractions in C++, Chapter 15

Inspired by slides created by Nick Troccoli, Marty Stepp, Keith Schwarz, Cynthia Lee, Chris Piech, Brahm Capoor, Anton Apostolatos, and others.
A thought exercise
Consider implementing a set with each of the following data structures:

<table>
<thead>
<tr>
<th>Collection</th>
<th>add(elem)</th>
<th>contains(elem)</th>
<th>remove(elem)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorted array</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If we store the elements in an array in **sorted** order:

```java
set.add(9);
set.add(23);
set.add(8);
set.add(-3);
set.add(49);
set.add(12);
```

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td></td>
<td>8</td>
<td>9</td>
<td>12</td>
<td>23</td>
<td>49</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Collection Performance

Consider implementing a set with each of the following data structures:

<table>
<thead>
<tr>
<th>Collection</th>
<th>add(elem)</th>
<th>contains(elem)</th>
<th>remove(elem)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorted array</td>
<td>O(N)</td>
<td>O(logN)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Unsorted array</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If we store elements in the **next available index**, like in a vector:

```
set.add(9);
set.add(23);
set.add(8);
...```

```
0 1 2 3 4 5 6 7 8 9
9 23 8 -3 49 12 / / / / /```
Consider implementing a set with each of the following data structures:

<table>
<thead>
<tr>
<th>Collection</th>
<th>add(elem)</th>
<th>contains(elem)</th>
<th>remove(elem)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorted array</td>
<td>O(N)</td>
<td>O(logN)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Unsorted array</td>
<td>O(1)</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>?????? array</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

What would this O(1) array look like? Where would we want to store each element?

```
set.add(9);
set.add(23);
set.add(8);
...```

```
0 1 2 3 4 5 6 7 8 9
```
Plan For Today

• O(1)?!?!?
• Hashing and hash functions
• Announcements
• HashSet implementation
  – Collision resolution
  – Coding demo
  – Load factor and efficiency
• Hash function properties

• **Learning Goal 1:** understand the hashing process and what makes a valid/good hash function.
• **Learning Goal 2:** understand how hashing is utilized to achieve O(1) performance in a HashSet.
Plan For Today

• O(1)?!?!?
• Hashing and hash functions
• Announcements
• HashSet implementation
  – Collision resolution
  – Coding demo
  – Load factor and efficiency
• Hash function properties
Where to store each element?

An idea: just store element \( n \) at index \( n \).

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
/ & 1 & / & 3 & / & / & / & 7 & / & /
\end{array}
\]

set.add(1);

set.add(3);

set.add(7);

• Benefits?
  – add, contains, and remove are all \( O(1) \)!

• Drawbacks?
  – What to do with set.add(11) or set.add(-5), for example?
  – Array might be sparse, leading to memory waste.
Hashing

- **Hashing**: process of storing each element at a particular predictable index
  - **Hash function**: maps a value to an integer.
    - `int hashCode(Type val);`
  - **Hash code**: the output of a value’s hash function.
    - Where the element would go in an infinitely large array.
  - **Hash table**: an array that uses hashing to store elements.
Hash Functions

• Our hash function before was `hashCode(n) \rightarrow n`.

  94305 → Integer Hasher → 94305

• To handle negative numbers, `hashCode(n) \rightarrow \text{abs}(n)`:  

  94305 → Integer Hasher++ → 94305  
  -1234 → Integer Hasher++ → 1234
Element placement process

• To handle large hashes, mod hash code by array capacity
  – “Wrap the array around”

```
set.add(37);
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
</tbody>
</table>

Capacity = 10
Element placement process

- To handle large hashes, mod by array capacity
  - “Wrap the array around”

```
set.add(37); // abs(37) % 10 == 7
```

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>37</td>
<td></td>
</tr>
</tbody>
</table>
```

Capacity = 10
Element placement process

• To handle large hashes, mod by array capacity
  – “Wrap the array around”

input → Hash Function → % capacity → index in array (0 to capacity - 1)

set.add(37); // abs(37) % 10 == 7
set.add(-2);

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>//</td>
<td>//</td>
<td>//</td>
<td>//</td>
<td>//</td>
<td>//</td>
<td>37</td>
</tr>
</tbody>
</table>

Capacity = 10
Element placement process

- To handle large hashes, mod by array capacity
  - “Wrap the array around”

```
set.add(37);  // abs(37) % 10 == 7
set.add(-2);  // abs(-2) % 10 == 2
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>37</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Capacity = 10
Element placement process

• To handle large hashes, mod by array capacity
  – “Wrap the array around”

Input → Hash Function → % capacity → index in array (0 to capacity - 1)

```
set.add(37);       // abs(37) % 10 == 7
set.add(-2);      // abs(-2) % 10 == 2
set.add(49);
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>/</td>
<td>-2</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>37</td>
<td>/</td>
</tr>
</tbody>
</table>
Element placement process

• To handle large hashes, mod by array capacity
  – “Wrap the array around”

```
set.add(37);  // abs(37) % 10 == 7
set.add(-2);  // abs(-2) % 10 == 2
set.add(49);  // abs(49) % 10 == 9
```

Not necessarily an int!

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>37</td>
<td></td>
<td>49</td>
</tr>
</tbody>
</table>

Capacity = 10
• **Collision**: when two distinct elements map to the same index in a hash table

```java
set.add(37);
set.add(-2);
set.add(49);
set.add(12);       // collides with -2...
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>-2</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>37</th>
<th></th>
<th>49</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

• **Collision resolution**: a method for resolving collisions
Plan For Today

• O(1)?!?!?
• Hashing and hash functions
• Announcements
• HashSet implementation
  – Collision resolution
  – Coding demo
  – Load factor and efficiency
• Hash function properties
Announcements

• Final exam review session is this **Wed. 12/5 7-8:30pm in Hewlett 103**

• Final exam info page is up on the website, more to come on Wednesday

• Zach’s office hours tomorrow are from 2-4pm
Plan For Today

• O(1)?!??!
• Hashing and hash functions
• Announcements
• **HashSet implementation**
  – Collision resolution
  – Coding demo
  – Load factor and efficiency
• Hash function properties
Separate Chaining

- **Separate chaining**: form a linked list at each index so multiple elements can share an index
  - Lists are short if the hash function is well-distributed
  - This is one of many different possible collision resolutions.

```java
set.add(37);
set.add(-2);
set.add(49);
```
Separate Chaining: add

- **Separate chaining**: form a linked list at each index so multiple elements can share an index
  - Just add new elements to the linked lists when adding to HashSet to resolve collisions

```java
set.add(37);
set.add(-2);
set.add(49);
set.add(12);
```
Separate Chaining: add

- **Separate chaining**: form a linked list at each index so multiple elements can share an index
  - Just add new elements to the linked lists when adding to HashSet to resolve collisions

```java
set.add(37);
set.add(-2);
set.add(49);
set.add(12);
```
Separate Chaining: add

- **Separate chaining**: form a linked list at each index so multiple elements can share an index
  - **For add**: just add new elements to the front of the linked lists when adding to HashSet to resolve collisions

```java
set.add(37);
set.add(-2);
set.add(49);
set.add(12);
set.add(-44);
```

```
0 1 2 3 4 5 6 7 8 9
/ / / / / / / / /
```

```
12 -44 37 49
```

```
-2
```

Separate Chaining: contains

- **Separate chaining**: form a linked list at each index so multiple elements can share an index
  - **For contains**: loop through appropriate linked list and see if you find the element you’re looking for

```java
set.contains(-2); // true
set.contains(7);  // false
```
Separate Chaining: contains

- **Separate chaining**: form a linked list at each index so multiple elements can share an index
  - For contains: loop through appropriate linked list and see if you find the element you’re looking for

```java
set.contains(-2); // true
set.contains(7);  // false
```
Separate Chaining: remove

- **Separate chaining**: form a linked list at each index so multiple elements can share an index
  - **For remove**: delete the element from the appropriate linked list if it’s there

```
set.remove(-2);
set.remove(-44);
```
Separate Chaining: remove

- **Separate chaining**: form a linked list at each index so multiple elements can share an index
  - **For remove**: delete the element from the appropriate linked list if it’s there

```cpp
set.remove(-2);
set.remove(-44);
```
Separate Chaining: remove

- **Separate chaining**: form a linked list at each index so multiple elements can share an index
  - **For remove**: delete the element from the appropriate linked list if it’s there

```java
set.remove(-2);
set.remove(-44);
```
• **Separate chaining:** form a linked list at each index so multiple elements can share an index
  
  – **For remove:** delete the element from the appropriate linked list if it’s there

```java
set.remove(-2);
set.remove(-44);
```
Plan For Today

• O(1)?!?!?
• Hashing and hash functions
• Announcements
• **HashSet implementation**
  – Collision resolution
  – **Coding demo**
  – Load factor and efficiency
• Hash function properties
Coding demo

• Let’s code a HashSet for integers using separate chaining!

• Note that we will use the following struct in our implementation:

```c
struct HashNode {
    int data;
    HashNode* next;
};
```
SeparateChainingHashSet::SeparateChainingHashSet(int capacity) {
    this->currSize = 0;
    this->capacity = capacity;
    elems = new HashNode*[capacity]();
}

SeparateChainingHashSet::~SeparateChainingHashSet() {
    clear();
    delete[] elems;
}

void SeparateChainingHashSet::add(int elem) { // note that this doesn’t account for re-hashing
    int index = hashCode(elem) % capacity;
    if (!contains(elem)) {
        HashNode* newElem = new HashNode(elem);
        newElem->next = elems[index];
        elems[index] = newElem;
        currSize++;
    }
}
bool SeparateChainingHashSet::contains(int elem) const {
    int index = hashCode(elem) % capacity;
    HashNode* curr = elems[index];
    while (curr != nullptr) {
        if (curr->data == elem) { return true; }
        curr = curr->next;
    }
    return false;
}
void SeparateChainingHashSet::remove(int elem) {
    int index = getIndex(elem);
    HashNode* curr = elems[index];
    if (curr != nullptr) {
        if (curr->data == elem) { // elem is at the front of the list
            eums[index] = curr->next;
            delete curr; currSize--;
        } else {
            /* loop through the list in this bucket until we find
            * elem and remove it from the list if it's there */
            while (curr->next != nullptr) {
                if (curr->next->data == elem) {
                    HashNode* trash = curr->next;
                    curr->next = trash->next;
                    delete trash;
                    currSize--;
                    break;
                }
                curr = curr->next;
            }
        }
    }
}
Plan For Today

• O(1)?!?!?
• Hashing and hash functions
• Announcements
• **HashSet implementation**
  – Collision resolution
  – Coding demo
  – **Load factor and efficiency**
• Hash function properties
Question: can a HashSet using separate chaining ever be “full”?

– It can never be “full”, but it slows down as its linked lists grow
Load Factor

• **Load factor**: the average number of values stored in a single index.

\[
\text{load factor} = \frac{\text{total # entries}}{\text{total # indices}}
\]

• A lower load factor means better runtime.

• Need to **rehash** after exceeding a certain load factor.
  – Generally after load factor >= 0.75.
• **Rehashing:** growing the hash table when the load factor gets too high.
  – Can’t just copy the old array to the first few indices of a larger one (why not?)

Capacity: 10
Load factor: 0.8 (high!)
Rehashing

- **Rehashing**: growing the hash table when the load factor gets too high.
  - Loop through lists and re-add elements into new hash table
  - Blue elements are ones that moved indices

Capacity: 20
Load factor: 0.4 (better!)
Plan For Today

- O(1)?!?!?
- Hashing and hash functions
- Announcements
- HashSet implementation
  - Collision resolution
  - Coding demo
  - Load factor and efficiency
- Hash function properties
• REQUIRED: a hash function must be **consistent**.
  – Consistent with itself:
    • `hashCode(A) == hashCode(A)` as long as A doesn’t change
  – Consistent with equality:
    • If A == B, then `hashCode(A) == hashCode(B)`
    • Note that A != B doesn’t necessarily mean that `hashCode(A) != hashCode(B)`

• DESIRABLE: a hash function should be **well-distributed**.
  – A good hash function minimizes collisions by returning mostly unique hash codes for different values.
Hash function properties

• Hash codes can be for any data type (not just for ints)
  – Need to somehow “add up” the object’s state.

• A well-distributed hashCode function for a string:

```java
int hashCode(String s) {
    int hash = 5381;
    for (int i = 0; i < (int) s.length(); i++) {
        hash = 31 * hash + (int) s[i];
    }
    return hash;
}
```

  – This function is used for hashing strings in Java
Question: Which of these two hash functions is better?

A.  
```java
int hashCode(string s) {
    return 42;
}
```

B.  
```java
int hashCode(string s) {
    return randomInteger(0, 9999999);
}
```

A! Because B is not a valid hash function (B is not consistent).
• Question: Which of these two hash functions is better?

A. 
```java
int hashCode(string s) {
    return (int) &s; // address of s
}
```

B. 
```java
int hashCode(string s) {
    return (int) s.length();
}
```

B! Because A is not valid (A is not consistent, since two equal strings might not be stored at the same memory address).
• Is the following hash function valid? Is it a good one? Could it have collisions?

```java
int hashCode(String s) {
    int hash = 0;
    for (int i = 0; i < (int) s.length(); i++) {
        hash += (int) s[i]; // ASCII value of char
    }
    return hash;
}
```

It’s valid, and it’s just okay (not as good as Java’s, e.g.). This has collisions for strings that are anagrams of each other.
• **Learning Goal 1:** understand the hashing process and what makes a valid/good hash function.

• **Learning Goal 2:** understand how hashing is utilized to achieve $O(1)$ performance in a HashSet.

• Take CS166 to learn a lot more about different kinds of hashing if you’re interested!