CSE 373

Set implementation; intro to hashing
read: Weiss 5.1 - 5.2, 5.4, 5.5

slides created by Marty Stepp
http://www.cs.washington.edu/373/

© University of Washington, all rights reserved.
Sets

- **set**: A collection of unique values (no duplicates allowed) that can perform the following operations efficiently:
  - add, remove, search (contains)

- The client doesn't think of a set as having indexes; we just add things to the set in general and don't worry about order.

```
set.contains("to")  // true
set.contains("be")  // false
```
Let's think about how to write our own implementation of a set.

- To simplify the problem, we only store ints in our set for now.
- As is (usually) done in the Java Collection Framework, we will define sets as an ADT by creating a Set interface.
- Core operations are: add, contains, remove.

```java
public interface IntSet {
    void add(int value);
    boolean contains(int value);
    void clear();
    boolean isEmpty();
    void remove(int value);
    int size();
}
```
Unfilled array set

• Consider storing a set in an unfilled array.
  ▪ It doesn't really matter what order the elements appear in a set, so long as they can be added and searched quickly.
  ▪ What would make a good ordering for the elements?

• If we store them in the next available index, as in a list, ...
  ▪ \(\text{set}.\text{add}(9);\)
    \(\text{set}.\text{add}(23);\)
    \(\text{set}.\text{add}(8);\)
    \(\text{set}.\text{add}(-3);\)
    \(\text{set}.\text{add}(49);\)
    \(\text{set}.\text{add}(12);\)
  ▪ How efficient is \(\text{add}\)? \(\text{contains}\)? \(\text{remove}\)?
    • \(O(1), O(N), O(N)\)
    • \(\text{contains must loop over the array}; \text{remove must shift elements.}\)
Sorted array set

- Suppose we store the elements in an unfilled array, but in *sorted* order rather than order of insertion.
  - `set.add(9);`
  - `set.add(23);`
  - `set.add(8);`
  - `set.add(-3);`
  - `set.add(49);`
  - `set.add(12);`

<table>
<thead>
<tr>
<th>index</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>value</td>
<td>-3</td>
<td>8</td>
<td>9</td>
<td>12</td>
<td>23</td>
<td>49</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>size</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- How efficient is `add? contains? remove?`
  - \(O(N), O(\log N), O(N)\)
  - (You can do an \(O(\log N)\) binary search to find elements in `contains`, and to find the proper index in `add/remove`; but `add/remove` still need to shift elements right/left to make room, which is \(O(N)\) on average.)
A strange idea

- **Silly idea:** When client adds value $i$, store it at index $i$ in the array.
  - Would this work?
  - Problems / drawbacks of this approach? How to work around them?

```java
set.add(7);
set.add(1);
set.add(9);
...
set.add(18);
set.add(12);
```

<table>
<thead>
<tr>
<th>index</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>value</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| value | 0 | 1 | 0 | 0 | 0 | 0 | 7 | 0 | 9 | 0 | 0 | 12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 18 |
| size  | 5 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
Hashing

• **hash**: To map a large domain of values to a smaller fixed domain.
  - Typically, mapping a set of elements to integer indexes in an array.
  - *Idea*: Store any given element value in a particular predictable index.
    • That way, adding / removing / looking for it are constant-time (O(1)).
  - **hash table**: An array that stores elements via hashing.

• **hash function**: An algorithm that maps values to indexes.
  - **hash code**: The output of a hash function for a given value.
  
  - In previous slide, our "hash function" was: \( \text{hash}(i) \rightarrow i \)
    • Potentially requires a large array \( \text{a.length} > i \).
    • Doesn't work for negative numbers.
    • Array could be very sparse, mostly empty (memory waste).
Improved hash function

• To deal with negative numbers:  hash(i) → abs(i)
• To deal with large numbers:  hash(i) → abs(i) % length

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

```
<table>
<thead>
<tr>
<th>index</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>value</td>
<td>0</td>
<td>0</td>
<td>-2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>37</td>
<td>0</td>
<td>49</td>
</tr>
<tr>
<td>size</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

// inside HashIntSet class
private int hash(int i) {
    return Math.abs(i) % elements.length;
}
public class HashIntSet implements IntSet {
    private int[] elements;
    ...
    public void add(int value) {
        elements[hash(value)] = value;
    }

    public boolean contains(int value) {
        return elements[hash(value)] == value;
    }

    public void remove(int value) {
        elements[hash(value)] = 0;
    }
}

- Runtime of add, contains, and remove: $O(1)$ !!
  
  • Are there any problems with this approach?
Collisions

- **collision**: When hash function maps 2 values to same index.

```java
set.add(11);
set.add(49);
set.add(24);
set.add(37);
set.add(54);  // collides with 24!
```

- **collision resolution**: An algorithm for fixing collisions.

```
<table>
<thead>
<tr>
<th>index</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>value</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>54</td>
<td>0</td>
<td>0</td>
<td>37</td>
<td>0</td>
<td>49</td>
</tr>
<tr>
<td>size</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Probing

• probing: Resolving a collision by moving to another index.
  ▪ linear probing: Moves to the next available index (wraps if needed).

```java
set.add(11);
set.add(49);
set.add(24);
set.add(37);
set.add(54);  // collides with 24; must probe
```

<table>
<thead>
<tr>
<th>index</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>value</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>24</td>
<td>54</td>
<td>0</td>
<td>37</td>
<td>0</td>
<td>49</td>
</tr>
<tr>
<td>size</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• variation: quadratic probing moves increasingly far away: +1, +4, +9, ...

Implementing HashIntSet

- Let's implement an int set using a hash table with linear probing.
  - For simplicity, assume that the set cannot store 0s for now.

```java
public class HashIntSet implements IntSet {
    private int[] elements;
    private int size;

    // constructs new empty set
    public HashIntSet() {
        elements = new int[10];
        size = 0;
    }

    // hash function maps values to indexes
    private int hash(int value) {
        return Math.abs(value) % elements.length;
    }

    ...
```
The add operation

- How do we add an element to the hash table?
  - Use the hash function to find the proper bucket index.
  - If we see a 0, put it there.
  - If not, move forward until we find an empty (0) index to store it.
  - If we see that the value is already in the table, don't re-add it.

```java
set.add(54);  // client code
set.add(14);
```

<table>
<thead>
<tr>
<th>index</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>value</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>24</td>
<td>54</td>
<td>14</td>
<td>37</td>
<td>0</td>
<td>49</td>
</tr>
<tr>
<td>size</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Implementing add

- How do we add an element to the hash table?

```java
public void add(int value) {
    int h = hash(value);
    while (elements[h] != 0 &&
        elements[h] != value) {
        h = (h + 1) % elements.length;  // for empty slot
    }
    if (elements[h] != value) { // avoid duplicates
        elements[h] = value;
        size++;
    }
}
```

<table>
<thead>
<tr>
<th>index</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>value</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>24</td>
<td>54</td>
<td>0</td>
<td>37</td>
<td>0</td>
<td>49</td>
</tr>
<tr>
<td>size</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The contains operation

- How do we search for an element in the hash table?
  - Use the hash function to find the proper bucket index.
  - Loop forward until we either find the value, or an empty index (0).
  - If find the value, it is contained (true). If we find 0, it is not (false).

- `set.contains(24)  // true`
- `set.contains(14)  // true`
- `set.contains(35)  // false`

<table>
<thead>
<tr>
<th>index</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>value</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>24</td>
<td>54</td>
<td>14</td>
<td>37</td>
<td>0</td>
<td>49</td>
</tr>
<tr>
<td>size</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
public boolean contains(int value) {
    int h = hash(value);
    while (elements[h] != 0) {
        if (elements[h] == value) {
            // linear probing
            return true;
            // to search
        }
        h = (h + 1) % elements.length;
    }
    return false;    // not found
}