CSE 373: Data Structures and Algorithms

Lecture 16: Hashing
Set ADT

• **set**: A collection that does not allow duplicates
  – We don't think of a set as having indexes; we just add things to the set in general and don't worry about order

• **basic set operations:**
  – **insert**: Add an element to the set (order doesn't matter).
  – **remove**: Remove an element from the set.
  – **search**: Efficiently determine if an element is a member of the set.

```java
set.contains("to")  // true
set.contains("be")  // false
```
## Implementing Set ADT

<table>
<thead>
<tr>
<th></th>
<th>Insert</th>
<th>Remove</th>
<th>Search</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unsorted array</strong></td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td><strong>Sorted array</strong></td>
<td>$\Theta(\log(n) + n)$</td>
<td>$\Theta(\log(n) + n)$</td>
<td>$\Theta(\log(n))$</td>
</tr>
<tr>
<td><strong>Linked list</strong></td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td><strong>BST (if balanced)</strong></td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>
A different tactic

• How do you check to see if a word is in the dictionary?
  – linear search?
  – binary search?
  – A – Z tabs?
Hash tables

- table maintains \( b \) different "buckets"
- buckets are numbered 0 to \( b - 1 \)
- **hash function** maps elements to value in 0 to \( b - 1 \)
- operations use hash to determine which bucket an element belongs in and only searches/modifies this one bucket

![Diagram of hash table](image)

- elements (e.g., strings)
- hash func. \( h(\text{element}) \)
- hash table
Hashing, hash functions

- The idea: somehow we map every element into some index in the array ("hash" it); this is its one and only place that it should go
  - Lookup becomes constant-time: simply look at that one slot again later to see if the element is there
  - insert, remove, search all become O(1)!

- For now, let's look at integers (int)
  - a "hash function" $h$ for int is trivial: store int $i$ at index $i$ (a direct mapping)
    - if $i \geq \text{array.length}$, store $i$ at index $(i \mod \text{array.length})$
      - $h(i) = i \mod \text{array.length}$
Simple Integer Hash Functions

- elements = integers
- TableSize = 10
- $h(i) = i \mod 10$
- Insert: 7, 18, 41, 34
### Simple Integer Hash Functions

- **elements** = integers
- **TableSize** = 10
- \( h(i) = i \% 10 \)
- **Insert**: 7, 18, 41, 34

<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>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Simple Integer Hash Functions

• elements = integers
• TableSize = 10
• \( h(i) = i \% 10 \)
• Insert: 7, 18, 41, 34
Simple Integer Hash Functions

- elements = integers
- *TableSize* = 10
- \( h(i) = i \% 10 \)
- **Insert**: 7, 18, 41, 34

<table>
<thead>
<tr>
<th>Index</th>
<th>Hash Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>
Simple Integer Hash Functions

• elements = integers
• TableSize = 10

• $h(i) = i \% 10$

• Insert: 7, 18, 41, 34
Simple Integer Hash Functions

- elements = integers
- $TableSize = 10$
- $h(i) = i \% 10$
- **Insert:** 7, 18, 41, 34

<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>41</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>18</td>
<td></td>
</tr>
</tbody>
</table>
Simple Integer Hash Functions

- elements = integers
- TableSize = 10
- \( h(i) = i \% 10 \)
- **Insert**: 7, 18, 41, 34
Simple Integer Hash Functions

• elements = integers
• TableSize = 10

• \( h(i) = i \mod 10 \)

• Insert: 7, 18, 41, 34
Hash function example

• Desirable properties of a hash function
  – efficient computation
  – deterministic/stable result
  – uniformly distributes values over a range
  – distributes values over a range

• \( h(i) = i \mod 10 \)
  – does this function have the properties above?

• Drawback: lose all ordering information:
  – getMin, getMax, removeMin, removeMax
  – ordered traversals; printing items in sorted order
Hash function for strings

• elements = Strings
• let's view a string by its letters:
  – String \( s : s_0, s_1, s_2, \ldots, s_{n-1} \)
• how do we map a string into an integer index? (how do we "hash" it?)

• one possible hash function:
  – treat first character as an int, and hash on that
    • \( h(s) = s_0 \mod TableSize \)
    • Is this a good hash function? When will strings collide?
Better string hash functions

• view a string by its letters:
  – String \( s : s_0, s_1, s_2, \ldots, s_{n-1} \)

• another possible hash function:
  – treat each character as an int, sum them, and hash on that
    • \( h(s) = \left( \sum_{i=0}^{n-1} s_i \right) \mod TableSize \)
    • What's wrong with this hash function? When will strings collide?

• a third option (polynomial accumulation)
  – perform a \textit{weighted sum} of the letters, and hash on that
    – \( h(s) = \left( \sum_{i=0}^{k-1} s_i \cdot 37^i \right) \mod TableSize \)
Hash collisions

- **collision**: the event that two hash table elements map into the same slot in the array.

- **example**: add 7, 18, 41, 34, then 21
  - 21 hashes into the same slot as 41!
  - 21 should not replace 41 in the hash table; they should both be there.

**collision resolution**: means for fixing collisions in a hash table.
Chaining

- **chaining**: All keys that map to the same hash value are kept in a linked list.
Load factor

- **load factor**: ratio of elements to capacity
- load factor = size / capacity = 5 / 10 = 0.5
Analysis of hash table search

• analysis of search, with chaining:
  – unsuccessful: $\lambda$
    • the average length of a list at hash($i$)
  
  – successful: $1 + (\lambda/2)$
    • one node, plus half the avg. length of a list (not including the item)
Implementing Set with Hash Table

• Each Set entry adds an element to the table
  – hash function will tell us where to put the element in the hash table

• Hash table organized for constant time insert, remove, and search
Implementing Set with Hash table

```java
public interface StringSet {
    public boolean add(String value);

    public boolean contains(String value);

    public void print();

    public boolean remove(String value);

    public int size();
}
```
public class StringHashEntry {
    public String data;          // data stored at this node
    public StringHashEntry next; // reference to the next entry

    // Constructs a single hash entry.
    public StringHashEntry(String data) {
        this(data, null);
    }

    public StringHashEntry(String data, StringHashEntry next) {
        this.data = data;
        this.next = next;
    }
}
public class StringHashSet implements StringSet {
    private static final int DEFAULT_SIZE = 11;
    private StringHashEntry[] table;
    private int size;

    ...
}

- Client code talks to the StringHashSet, not to the entry objects stored in it

- The array (table) is of StringHashEntry — each element in the array is a linked list of elements that have the same hash
public boolean contains(String value) {
// figure out where value should be...
    int valuePosition = hash(value);

    // check to see if the value is in the set
    StringHashEntry temp = table[valuePosition];
    while (temp != null) {
        if (temp.data.equals(value)) {
            return true;
        }
        temp = temp.next;
    }

    // otherwise, the value was not found
    return false;
}
Set implementation: insert

• Similar structure to `contains`
  – Calculate hash of new element
  – Check if the element is already in the set

• Add the element to the front of the list that is at `table[hash(value)]`
Set implementation: insert

```java
public boolean add(String value) {
    int valuePosition = hash(value);

    // check to see if the value is already in the set
    StringHashEntry temp = table[valuePosition];
    while (temp != null) {
        if (temp.data.equals(value)) {
            return false;
        }
        temp = temp.next;
    }

    // add the value to the set
    StringHashEntry newEntry = new StringHashEntry(value, table[valuePosition]);
    table[valuePosition] = newEntry;
    size++;
    return true;
}
```