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 indices or any 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.

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

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Insert</th>
<th>Remove</th>
<th>Search</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unsorted array</strong></td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>Sorted array</strong></td>
<td>O(log (n + n))</td>
<td>O(log (n + n))</td>
<td>O(log (n))</td>
</tr>
<tr>
<td><strong>Linked list</strong></td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(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" (numbered 0 to $b-1$)
- **hash function** maps elements to value in 0 to $b - 1$
- use hash to determine which bucket an element belongs in and only searches/modifies this one bucket
Hashing, hash functions

- The idea: We somehow 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 storing integers
  - Assume the following "hash function" $h$: Store int $i$ at index $i$ (a direct mapping)
    - if $i \geq array.length$, store $i$ at index $(i \mod array.length)$

  $h(i) = i \mod array.length$
Simple Integer Hash Functions

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

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

<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>18</td>
<td></td>
</tr>
</tbody>
</table>
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
# 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
Hash function example

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

\[ h(i) = i \mod 10 \]
- Does this function have the properties above?

Drawbacks?
- Lose all ordering information:
  - getMin, getMax, removeMin, removeMax
  - Ordered traversals; printing items in sorted order
Hash collisions

- Example: add 7, 18, 41, 34, then 21
  - 21 hashes into the same slot as 41!
  - Should 21 replace 41?
    - No!

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

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

<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>7</td>
<td></td>
<td>41</td>
<td></td>
<td></td>
<td>34</td>
<td></td>
<td></td>
<td>7</td>
<td>18</td>
<td></td>
</tr>
</tbody>
</table>
Hash function for strings

- elements = Strings
- How do we map a string into an integer index? (i.e., how do we "hash" it?)

- Let's view a string by its letters:
  - String $s: s_0, s_1, s_2, \ldots, s_{n-1}$

- One possible hash function:
  - Treat first character as an int, and hash on that
    - $h(s) = s_0 \% TableSize$
  - Is this a good hash function? When will strings “collide”?
  - What about $h(s) = s.length \% TableSize$?
Better string hash functions

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

- A third option (polynomial accumulation)
  - Perform a weighted sum of the letters, and hash on that
    \[ h(s) = \left( \sum_{i=0}^{k-1} s_i \cdot 37^i \right) \% \text{TableSize} \]

- Coming up with a great hash function is hard.
Chaining

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

- **load factor** \((\lambda)\): 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 average 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

- Runtime
  - insert: $O(1)$
  - remove: $O(1)$
  - search: $O(1)$
Implementing Set with Hash Table

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;
    }
}
StringHashSet class

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)]`
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;
}