Hash tables

- **Average case** constant time contains, add, remove under certain conditions
- Underlying implementation for the Java `HashSet` class
- Array-based
- Hash function used to determine array index (Java calls this `hashCode`)

A very simple implementation to illustrate the basic ideas

- Let's say we want to have a set of integers
- Range: 0-999
- Use a boolean array of length 1000 (call it `hashTable`)
- add(i): set `hashTable[i]` to true
- remove(i): set `hashTable[i]` to false
- contains(i): check to see if `hashTable[i]` is true
- Complexity of these methods is...

Issues with this implementation

- What if we increase the range of integers to 0 ... $2^{31} - 1$ (the largest possible `int` in Java)
- It takes up a lot of space to have a boolean array of size $2^{31} - 1$ (approx 200 MBytes)
- Or, what if we want a set of another type of data?
  - Then we need a mapping between the possible data values and indices
- Introduce a **hash function**: maps data to an index in `hashTable` (again, in Java this is the `hashCode` method)
- Simple example hash function (for integers): `hashCode(x) = x % size`, where `size` is the size of the hash table
- But this introduces a problem: collisions
  - A **collision** occurs when two pieces of data are mapped to the same location in the hash table
  - We'll look at several possible ways to deal with collisions:
    - Separate chaining
    - Linear probing
    - Quadratic probing
    - Double hashing
• **Hash functions**
  
  o Properties:
    1. efficient
    2. evenly distributed across index range
  o Very important to choose a good hash function
  o Example: If hash table's size is 1000 and the data to be stored are ints, we could just define hashCode(x) as x\%1000. But what if all of the data end in 0? Then only 10% of the table is ever used.
  o A better idea:

    Make size of the hash table prime (e.g., 1009 = first prime number > 1000)

    define hashCode(x) as x\%1009

• **Collisions**
  
  o Hash function may (will) produce same position for different data items
  o Different strategies: separate chaining, linear probing, quadratic probing, double hashing

• **Load factor**
  
  o Defined as \( \lambda = n/t \), where \( n \) is the number of data items stored, and \( t \) is the table size
  o Used to determine when to "rehash" (to be discussed below)

• **Separate chaining**
  
  o Change `hashTable` to an array of linked lists
  o `hashCode()` tells us which list to search
  o \( \lambda \) may exceed 1
  o Worst-case complexity...
  o However, assuming the data distributes uniformly across the hash table, best-case and average-case complexity are ...

• **Separate chaining with rehashing**
  
  o Set a threshold for \( \lambda \)
  o When \( \lambda \) exceeds this threshold, expand the capacity of `hashTable`, and "rehash"

    - e.g., if data items are integers:
// assuming hashCode is a method of the Integer class
// not necessarily the best hash function in the world;
simply to illustrate the idea of rehashing
public int hashCode() {
    return intValue()%tableSize;
}

- This does not affect worst-case complexity of add, remove, or contains, but makes it very unlikely
- Average-case complexity...

- Implementation of separate chaining with rehashing
to be done in class

- Linear probing
  - If there is a collision when adding an item, simply search for next unoccupied space in `hashTable`, set its status to 1 (occupied) and store the item in the `data` field
    - If \( \lambda \) exceeds its threshold, then rehash
  - Example: shown in class
  - What about `remove` and `contains`?
  - A problem with linear probing is that collisions tend to form clusters of data in the array. This is called **primary clustering**.

- Implementation of **Linear Probing** to be done in class

- Quadratic probing
  - Let \( p(i) \) by the \( i \)-th probe for an item
    - \( p(0) = \text{hashCode}( ) \)
    - \( p(i) = (\text{hashCode}( ) + i^2)\%\text{tableSize} \)
  - Example to be shown in class
  - Less clustering than linear probing
  - Issues:
    - Will we probe enough different cells? make table size prime
    - Why? Because then, the first \( \frac{\text{tableSize}}{2} \) locations are distinct (proof to be given in class)
    - Can we generate next index efficiently?

- Double hashing
  - After a collision, compute a second hash function `hashCode2( )`
    - \( p(i) = (\text{hashCode}( ) + i\times\text{hashCode2}( ))\%\text{tableSize} \)
Must make sure that hashCode2() never evaluates to 0!!
One possible good choice: hashCode2( ) = R - hashCode(x)%R, where R is a prime number smaller than tableSize

- **Hash tables vs. linked lists and trees**
  - speed up add, remove, and contains to average-case constant time instead of worst-case $\Theta(\log(n))$ time (for trees)
    - Although the worst case is $\Theta(n)$
  - Lose ability to easily traverse elements in order
  - Can't compute max and min
  - Can't look for near matches (e.g., spelling correction)

- **jdk classes: HashSet, HashMap**

- **Summary of Java Collections Framework**
  In this table. Not an exhaustive list of classes or methods