Data Structures in Java

Session 14

Instructor: Bert Huang

http://www1.cs.columbia.edu/~bert/courses/3134

Announcements

- Homework 3 Programming due
- Homework 4 on website

Review

- Lists, Stacks, Queues
- Trees, Binary Search Trees
  - AVL, Splay
- Priority Queues: Binary Heaps

Today’s Plan

- Hash Table ADT
- Array implementation
- Collision resolution strategies

Hash Table ADT

- **Search tree:**
  findMin, findMax, insert/delete, search
- **Priority Queue:**
  findMin (or max), insert/delete, no search
- **Hash Table:**
  insert/delete, search

Hash Table ADT

- **Search tree:**
  Stores complete order information
- **Priority Queue:**
  Stores <u>in</u>complete order information
- **Hash Table:**
  Stores <u>no</u> order information

Hash Table ADT

- Insert or delete objects by **key**
- Search for objects by **key**
- **No** order information whatsoever

- Ideally O(1) per operation

Implementation

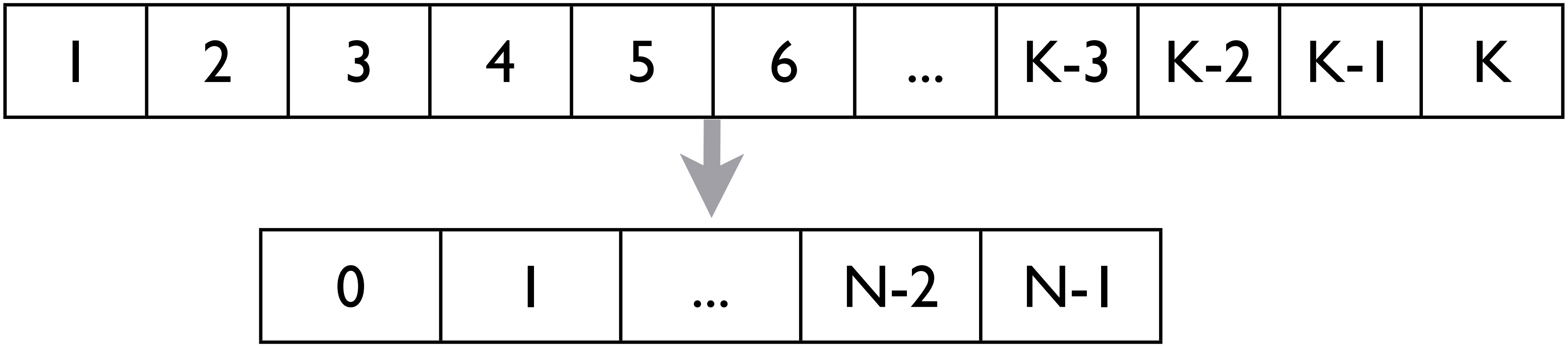
- Suppose we have keys between 1 and K
- Create an array with K entries
- Insert, delete, search are just array operations

1	2	3	4	5	6	...	K-3	K-2	K-1	K

- Obviously too expensive

Hash Functions

- A **hash function** maps any key to a valid array position
  - Array positions range from 0 to N-1
  - Key range possibly unlimited



1	2	3	4	5	6	...	K-3	K-2	K-1	K

0	1	...	N-2	N-1

Hash Functions

- For integer keys, (key mod N) is the simplest hash function
- In general, **any** function that maps from the space of keys to the space of array indices is valid
- but a good hash function spreads the data out evenly in the array
- A good hash function avoids **collisions**

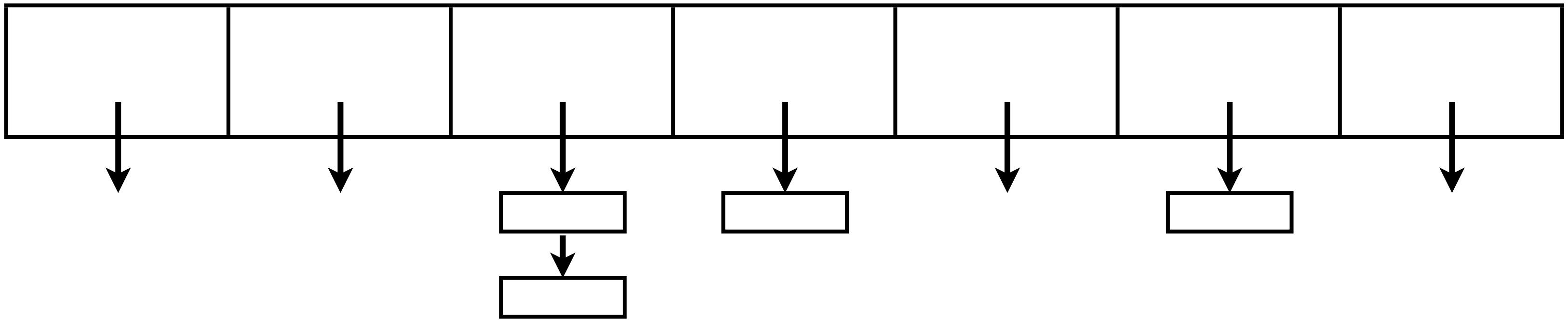
Collisions

- A **collision** is when two distinct keys map to the same array index
  - e.g., $h(x) = x \bmod 5$
    $h(7) = 2$, $h(12) = 2$
- Choose $h(x)$ to minimize collisions, but collisions are inevitable
- To implement a hash table, we must decide on collision resolution policy

Collision Resolution

- Two basic strategies
  - Strategy 1: Separate Chaining
  - Strategy 2: Probing; lots of variants

Strategy 1: Separate Chaining



- Keep a list at each array entry
  - Insert(x): find h(x), add to list at h(x)
  - Delete(x): find h(x), search list at h(x) for x, delete
  - Search(x): find h(x), search list at h(x)

Separate Chaining Average Case

- **Load Factor** λ = # objects / TableSize
- Average list length is λ
- Time to insert = constant, or constant + λ
- Time to search = constant + λ or constant + $\lambda/2$

Strategy 1: Advantages and Disadvantages

- Advantages:
  - Simple idea
  - Removals are clean *
- Disadvantages:
  - Need 2nd data structure, which causes extra overhead if the hash function is good

Strategy 2: Probing

- If h(x) is occupied, try **h(x)+f(i) mod N** for i = 1 until an empty slot is found
- Many ways to choose a good f(i)
- Simplest method: Linear Probing
    - f(i) = i

Linear Probing Example

- N = 5
- h(x) = x mod 5
- insert 7
- insert 12
- insert 2

		7		

		7	12	

		7	12	2

Primary Clustering

- If there are many collisions, blocks of occupied cells form: **primary clustering**
- Any hash value inside the cluster adds to the end of that cluster
- (a) it becomes more likely that the next hash value will collide with the cluster, and (b) collisions in the cluster get more expensive

Removals

- How do we delete when probing?
- Lazy-deletion: mark as deleted,
  - we can overwrite it if inserting,
  - but we know to keep looking if searching.

Quadratic Probing

- f(i) = i^2
- Avoids primary clustering
- Sometimes will never find an empty slot even if table isn't full!
- Luckily, if load factor $\lambda \leq \frac{1}{2}$,

  guaranteed to find empty slot

Quadratic Probing Example

- N = 7

- h(x) = x mod 7
- insert 9

		9				

- insert 16

		9	16			

- insert 2

		9	16			2

Double Hashing

- If $h_1(x)$ is occupied, probe according to

$$f(i) = i \times h_2(x)$$

- 2nd hash function must never map to 0

- Increments differently depending on the key

Double Hashing Example

- N = 7

- h1(x) = x mod 7,   h2(x) = 5-x mod 5
- insert 9

		9				

- insert 16

		9				16

- insert 2

		9			2	16

Hashing

- Indexing by the key needs too much memory
- Index into smaller size array, pray you don’t get collisions
- If collisions occur,
  - separate chaining, lists in array
  - probing, try different array locations

Reading

- Weiss Ch. 5