HASH TABLES

cs2420 | Introduction to Algorithms and Data Structures | Spring 2015

administrivia…

-assignment 9 is due on Monday

-assignment 10 will go out on Thursday

-midterm on Thursday

last time…

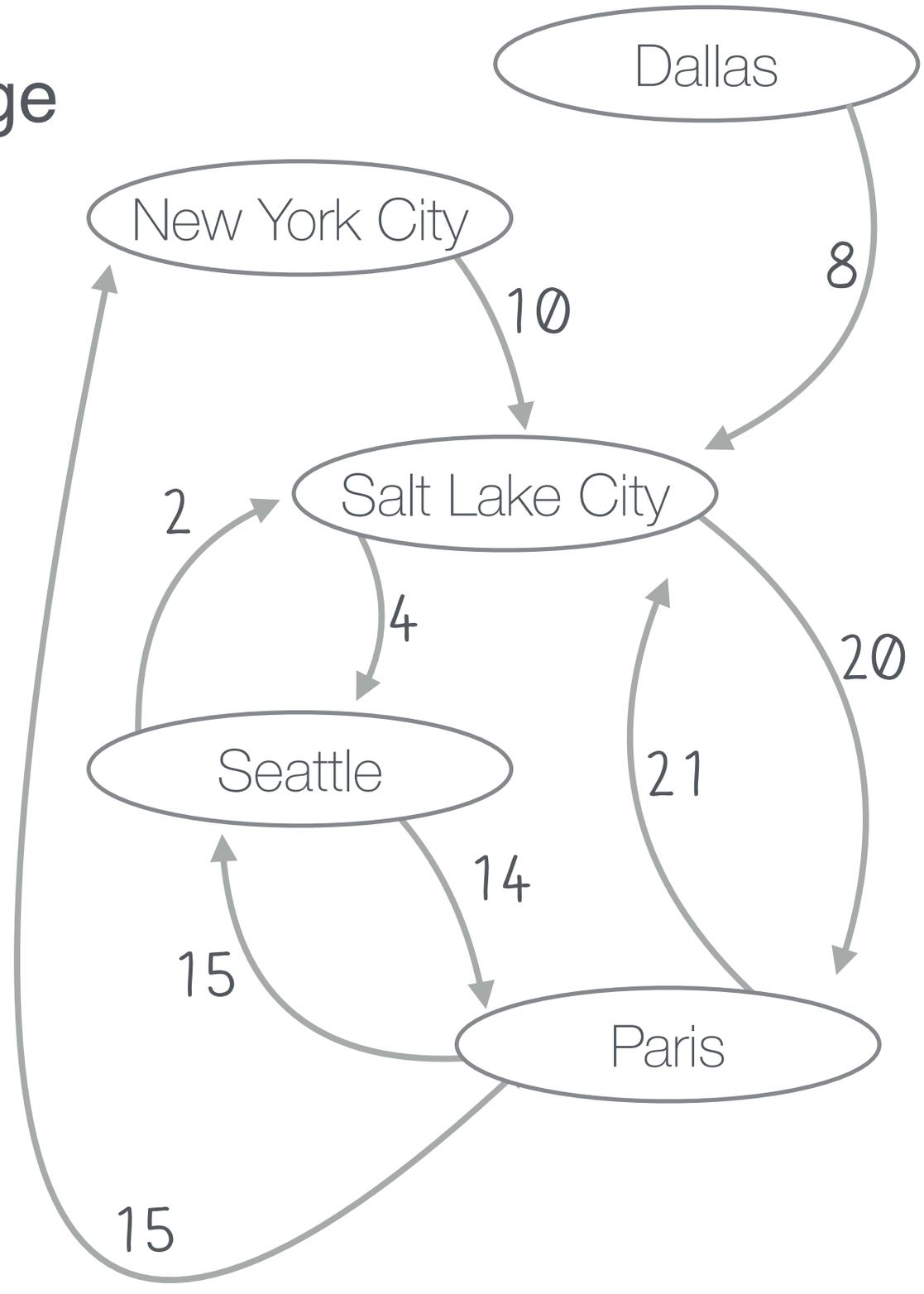
-we can add **weight** to each edge

-a higher weight indicates a more costly step

-**weighted path length** is the sum of all edge weights on a path

-cheapest is not always the shortest!

-will regular BFS find the cheapest path?



-**Dijkstra's algorithm** finds the *cheapest* path

-keep track of the total path cost from start node to the current node

-cost of path to next node is total cost so far plus weight of edge to next node

-instead of traversing nodes in the order they were encountered, traverse in order of cheapest total cost first

```
Dijkstra(Node start, Node goal)
{
  initialize all nodes' cost to infinity

  PQ.enqueue(start)
  while(!PQ.empty())
  {
    curr = PQ.dequeue()
    if(curr == goal) {return} \\done!
    curr.visited = true
    foreach unvisited neighbor n of curr:
    {
      if(n.cost > curr.cost + edgeweight
      {
        PQ.enqueue(n) || update n's position in PQ
        n.cameFrom = curr
        n.cost = curr.cost + edgeweight
      }
    }
  }
}
```

today…

-quick review

-quick exercise

-mapping

-hash table

-hash function

-linear probing

-quadratic probing

-chaining

-assignment 10 details

quick review

-arrays (and ArrayLists)
    -random access,
    -insert & delete: **O(N)**

-linked lists
    -linear access
    -insert & delete: **O(c)**

-binary search trees
    -everything: **O(logN)**
    -… must be balanced

-stacks
    -everything: **O(c)**
    -… limited to top item

-queues
    -everything: **O(c)**
    -… limited to front and back

	Access (search)	Insertion	Deletion	Notes
Arrays (ArrayLists)	Random access (constant)	Linear	Linear	Must know size ahead of time
Linked Lists	Linear	Linear, or O(1) on ends	Linear, or O(1) on ends	Can allocate new items on demand
Binar Search Trees	log N	log N	log N	Must be balanced
Stacks	Constant	Constant	Constant	Access limited to top
Queues	Constant	Constant	Constant	Access limited to front/back

quick exercise

WHAT IF WE WANT A DATA STRUCTURE THAT HOLDS INTEGERS, AND HAS CONSTANT TIME INSERTION & DELETION?

	Access (search)	Insertion	Deletion	Notes
Arrays (ArrayLists)	Random access	Linear	Linear	Must know size ahead of time
Linked Lists	Linear	Linear, or O(1) on ends	Linear, or O(1) on ends	Can allocate new items on demand
Binar Search Trees	log N	log N	log N	Must be balanced
Stacks	Constant	Constant	Constant	Access limited to top
Queues	Constant	Constant	Constant	Access limited to front/back

WHAT IF WE ALSO WANT CONSTANT TIME ACCESS TO **ANY** ITEM?

-constant time insertion, deletion, and random access

-we know:

  -possible range of integers is `[0...MAX_INT]`

-what is a naïve, brute-force solution?

  -hint: use an array

-create a gigantic array of size `MAX_INT`

-initialize everything to `-1`

-when inserting a number `n`, put it in the array at index `n`

-when searching for a number `n`, check if index `n` is equal to `-1` or not

-when deleting a number `n`, set array at index `n` to `-1`

DOES THIS FULFILL THE CONSTANT TIME INSERTION, DELETION, AND ACCESS REQUIREMENTS?

IS THIS REALISTIC???

mapping

-let's try using a smaller array, and mapping large indices to the range of the smaller array

-assume range of possible items is `[0...99]`

-and assume that we will have <<100 items

-assume array size is only 10

-how can we make this work for integers?

-use the mod operator, `%`

  -guaranteed to return a number in the range `[0…(denominator-1)]`

-mod the input index by the array size for the new index

INSERT:

12, 15, 17, 46, 89, 90

what about data without natural indices?

-how can we do this for non-integer items?

-integers have an obvious solution… use the integer itself as the index

-what index should use for, say, a String?

-one solution is to somehow generate an integer from a string

-length of string?

-sum of all characters?

-some combination of both?

-a method that generates an integer index given any object is called a ***hash function***

hash table

-a **hash table** is a general storage data structure

-insertion, deletion, and look-up are all **O(c)**

-like a stack, but not limited to top item

	Access	Insertion	Deletion	Notes
Hash Table	Constant	Constant	Constant	Magic?

-underlying data structure is just an array

-requires that all data types inserted have a hash function

-map the *hash value* to a valid index of the array using %

-empty spots in the array are set to null

-use hash value to instantly look-up the index of any item

  -insertion, deletion, and search: **O(1)**

    -assuming the hash function is **O(1)***!*

hash functions

-a **hash function** is a function that takes any item as input and produces an integer as output

-always returns the same number for the same object

-**if** `object1.equals(object2)`

-must return the same integer for both objects

-good hash functions return evenly distributed numbers for the input items

-it is not required that two non-equal objects have different hash values

Java's `hashCode`

-every `Object` in Java has a method `hashCode`

-returns an integer based on the object

-default for this method (if you don't override it) is to return the memory address of the object

-will not be very well-distributed if your items are contiguous in memory

linear probing

-remember: *it is NOT required that two non-equal object have different hash values*

-because of this, it is possible for two different objects to has to the same index

  -this is called a **collision**

INSERT:

12, 15, 17, 46, 89, 90, 92

resolving collisions

-there are multiple ways to resolve a collision, the first of which is called **linear probing**

-when *inserting*, if the spot is already taken, simply step forward one index at a time until an empty space is found

-and, then insert item in empty space

-when *searching*, start at the hashed index, and if this is not the item we are searching for, begin stepping forward until the item is found

-what is we hit an empty spot?

-wrap around to the beginning if at the end of the array

insert with linear probing

COLLISIONS ARE RESOLVED ON INSERTS BY SEQUENTIALLY SCANNING THE TABLE (WITH WRAPARAOUND) UNTIL AN EMPTY CELL IS FOUND

search with linear probing

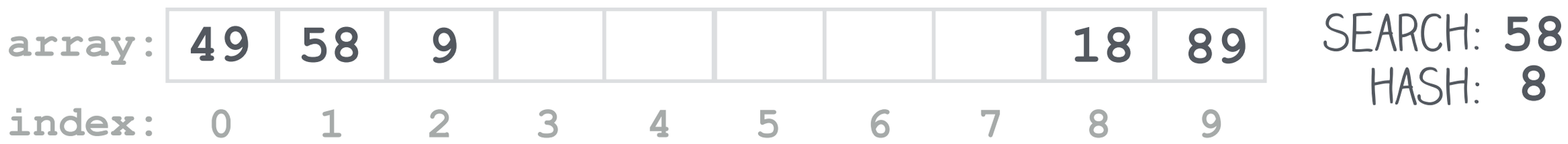
-if the table is not full, the item we seek, or an empty cell, will eventually be found

-cost?

  -recall that we are hoping of O(1)

-find operation follows the same path as insert… if empty cell reached, item not found

-how do we find 58?



delete with linear probing

-on a delete, the actual item cannot be deleted from the table because items serve as placeholders during collision resolution

HOW DO WE FIND 9?

array:	49	58	9						18	89
index:	0	1	2	3	4	5	6	7	8	9
deleted:	F	F	F						F	T

DELETE: 89
HASH: 9

-we must use **lazy deletion**, which marks items as deleted rather than actually removing them

performance

-if no collisions occur, performance of insert, delete, and search is **O(1)**

-to determine the real cost, define λ, the fraction of the table that is full

-called the **load factor**

-0 <= λ <= 1

-for each probe into the table, the probability that spot is occupied is λ

-assuming the above is correct, the average number of cells examined on an insert is 1/(1-λ)

-if λ = 0.5, average of two cells examined

clustering

-if an item's natural spot is taken, it goes in the next open spot, making a cluster for that hash

-**clustering** happens because once there is a collision, there is a high probability that there will be more

-this means that any item that hashes into the cluster will require several attempts to resolve the collision

-feedback loop:

-the bigger the clusters are, the more likely they are to be hit

-when a cluster gets hit, it gets bigger

quadratic probing

-quadratic probing attempts to deal with the clustering problem

-if `hash(item) = H`, and the cell at `H` is occupied:

  -try $H+1^2$

  -then $H+2^2$

  -then $H+3^2$

  -and so on…

  -wrap around to beginning of array if necessary

insert with quadratic probing

array:										89
index:	0	1	2	3	4	5	6	7	8	9

INSERT: 89
HASH: 9

array:									18	89
index:	0	1	2	3	4	5	6	7	8	9

INSERT: 18
HASH: 8

array:	49								18	89
index:	0	1	2	3	4	5	6	7	8	9

INSERT: 49
HASH: 9

array:	49		58						18	89
index:	0	1	2	3	4	5	6	7	8	9

INSERT: 58
HASH: 8

array:	49		58	9					18	89
index:	0	1	2	3	4	5	6	7	8	9

INSERT: 9
HASH: 9

concerns...

-is quadratic probing guaranteed to find an open spot? can it search the same spot twice?

-suppose the table size is 16, and `hash(item) = 0`

$0\%16 = 0$
$(0+1^2)\%16 = 1$
$(0+2^2)\%16 = 4$
$(0+3^2)\%16 = 9$
$(0+4^2)\%16 = 0$
$(0+5^2)\%16 = 9$
$(0+6^2)\%16 = 4$
$(0+7^2)\%16 = 1$

LIMITATION: AT MOST, HALF OF THE TABLE CAN BE USED TO RESOLVE COLLISIONS

ONCE TABLE IS HALF FULL IT IS DIFFICULT TO FIND AN EMPTY SPOT

...CALLED **SECONDARY CLUSTERING**

solution…

-the following two guidelines guarantee that every spot will be examined at least once

-ensure that the size of the array is a prime number

-mapping a hash value to an index will be modding by a prime number!

-ensure that the table is never more than 50% full

- $\lambda < 0.5$

-these guidelines also guarantee no cell is visited twice

-proof is the textbook

resizing the table

-since we now have the requirement that $\lambda < 0.5$, what do we do when we need to add another item?

-just like resizing an array, we resize the table to the next largest prime number

-instead of a simple copy-everything-over, all items must be rehashed

  -why?

-this is called **rehashing**

-quadratic probing does not eliminate the clustering problem

-but, secondary clustering is not as severe as primary clustering

-the only reason not to use quadratic probing is when maintaining a half-empty array is too costly

-can you think of an alternative for collision management?

separate chaining

-why not make each spot in the array capable of holding more than one item?

-use an array of linked lists

-hash function selects index into array

-called **separate chaining**

-for insertion, append the item to the end of the list

-insertion is **O(1)** if we have what?

-searching is a linear scan through the list

-fast if the list is short

performance

-different definition of the load factor λ

-λ = average length of linked lists

-therefore, search and delete operations scan λ items

-instead of rehashing when the table is half full, rehash when λ becomes large

  -analysis is required to find a good value

-rehashing is never *required* since lists can grow indefinitely, but it can be beneficial

assignment 10 details

-you will implement a quadratic probing hash table AND a separate chaining hash table for `Strings`

-the constructors for these hash tables takes a `HashFunctor` object

-recall that a functor is an object which encapsulates a method (just like `Comparator`)

-the `HashFunctor` defines a hash method, which implements a hash function

-you can create any number of different hash functions this way without changing any code in your hash table

-yay for encapsulation!

-start thinking about…

-what is a bad hash function for `Strings`?

-what is a good hash function for `Strings`?

-remember, `Strings` are just a sequence of `chars`

-and a `char` is just a smaller `int`

-we can perform any operation or combination of ops on the small numbers (`chars`) that make up the `String`

-an example `String` hash function is in the book

-there are also a bunch of good ones on the web

next time…

-midterm on Thursday in class

-reading for next week

    -chapter 21 in book

-homework

    -assignment 9 due Monday

    -assignment 10 out on Thursday, due next Thursday