BINARY HEAP

cs2420 | Introduction to Algorithms and Data Structures | Fall 2016
administrivia...
- assignment 10 is due tonight

- assignment 11 is out

- midterm

- FYI: November 23rd class is canceled
last time...
- 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.

<table>
<thead>
<tr>
<th>Access</th>
<th>Insertion</th>
<th>Deletion</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash Table</td>
<td>Constant</td>
<td>Constant</td>
<td>Constant</td>
</tr>
</tbody>
</table>
- 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 \( \% \)

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

  - insertion, deletion, and search: \( \mathcal{O}(c) \)

  - with what assumption?
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
-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

array: [90, 12, 15, 46, 17, 89]
index: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
-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 have to the same index
  -this is called a collision

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

collision! where can we put 92?

array: 90 12 15 46 17 89
index: 0 1 2 3 4 5 6 7 8 9
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 $\text{hash}(\text{item}) \mod [\text{size of array}] = H$, 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
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 $O(c)$ if we have what?

-searching is a linear scan through the list
  -fast if the list is short
a bit more on hash functions...
- ints have an obvious hash value

<table>
<thead>
<tr>
<th>array:</th>
<th>90</th>
<th>12</th>
<th>15</th>
<th>46</th>
<th>17</th>
<th>89</th>
</tr>
</thead>
<tbody>
<tr>
<td>index:</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

- what about Strings? Books? Shapes?...

- we must not overlook the requirement of a *good* hash functions
remember...

- hash functions take any item as input and produce an integer as output
- given the same input the function always returns the same output
- two different inputs MAY have the same hash value
review

- **O(c)** for all major operations
  - assuming λ is managed

- **linear probing**
  - has clustering problems

- **quadratic probing**
  - has lesser clustering problems
  - requires λ < 0.5, and prime table size

- **separate chaining**
  - probably the easiest to implement, as well as the best performing
what is the load factor $\lambda$ for the following hash table?

A) 4  
B) 6  
C) 0.4  
D) 0.5  
E) 0.6
using linear probing, in what index will item 93 be added?

A) 1
B) 5
C) 6
D) 7
using quadratic probing, in what index will item 22 be added?

A) 1
B) 5
C) 6
D) 7
recap

- **i heart hash tables**
  - collection structure with $O(c)$ for major operations

- **but!...**
  - hash function must minimize collisions
    - *should evenly distribute values across all possible integers*
  - collisions must be carefully dealt with
  - hash function runtime must be fast

- no ordering
  - *how do we find the smallest item in a hash table?*
  - *in a BST?*
priority queues
- A priority queue is a data structure in which access is limited to the minimum item in the set
  - add
  - findMin
  - deleteMin

- Add location is unspecified, so long as the above is always enforced

- What are our options for implementing this?
option 1: a linked list
- add: $O(c)$
- findMin: $O(N)$
- deleteMin: $O(N)$ (including finding)

option 2: a sorted linked list
- add: $O(N)$
- findMin: $O(c)$
- deleteMin: $O(c)$
- option 1: a linked list
  - add: $O(c)$
  - findMin: $O(N)$
  - deleteMin: $O(N)$ (including finding)

- option 2: a sorted linked list
  - add: $O(N)$
  - findMin: $O(c)$
  - deleteMin: $O(c)$

- option 3: a binary heap
  - add: $O(c)$
  - findMin: $O(c)$
  - deleteMin: $O(\log N)$
complete trees
- A **complete binary tree** has its levels completely filled, with the possible exception of the bottom level.

- Bottom level is filled from left to right.

- Each level has twice as many nodes as the previous level.
what underlying data structure would you use to implement a complete tree?

- a complete binary tree has its levels completely filled, with the possible exception of the bottom level
- bottom level is filled from left to right
- each level has twice as many nodes as the previous level

A) array
B) linked list
C) binary tree
D) graph
complete trees as an array

-if we are guaranteed that tree is complete, we can implement it as an array instead of a linked structure

-the root goes at index 0, its left child at index 1, its right child at index 2

-for any node at index i, its two children are at index $(i*2) + 1$ and $(i*2) + 2$
-for example, d’s children start at (3*2) + 1

-how can we compute the index of any node’s parent?
-luckily, integer division automatically truncates

-any node’s parent is at index \((i-1) / 2\)
complete trees as an array

- keep track of a `currentSize` variable
  - holds the total number of nodes in the tree
  - the very last leaf of the bottom level will be at index `currentSize - 1`

- when computing the index of a child node, if that index is $\geq$ `currentSize`, then the child does not exist
traversal helper methods

```java
int leftChildIndex(int i) {
    return (i*2) + 1;
}

int rightChildIndex(int i) {
    return (i*2) + 2;
}

int parentIndex(int i) {
    return (i-1) / 2;
}
```
binary heap
-a binary heap is a binary tree with two special properties
  - *structure*: it is a complete tree
  - *order*: the data in any node is less than or equal to the data of its children

- this is also called a min-heap

- a max-heap would have the opposite property
-order of children does not matter, only that they are greater than their parent
is this a min-heap?
A) yes
B) no
is this a min-heap?
A) yes
B) no
adding to a heap
- we must be careful to maintain the two properties when adding to a heap
  - structure and order

- deal with the structure property first... where can the new item go to maintain a complete tree?

- then, *percolate* the item upward until the order property is restored
  - swap upwards until > parent
adding 14

put it at the end of the tree
adding 14

put it at the end of the tree
adding 14
put it at the end of the tree
percolate up the tree to fix the order
adding 14

put it at the end of the tree

percolate up the tree to fix the order
adding 14

put it at the end of the tree

percolate up the tree to fix the order
adding 14

put it at the end of the tree

percolate up the tree to fix the order
adding 14

put it at the end of the tree

percolate up the tree to fix the order
adding 14

put it at the end of the tree

percolate up the tree to fix the order
adding 14

put it at the end of the tree

percolate up the tree to fix the order
complexity of add?
cost of add

- percolate up until smaller than all nodes below it...

- how many nodes are there on each level (in terms of N)?
  - about half on the lowest level
  - about 3/4 in the lowest two levels
-if the new item is the smallest in the set, cost is \( O(\log N) \)
  -must percolate up every level to the root
  -complete trees have \( \log N \) levels
    -is this the worst, average, or best case?

-it has been shown that on average, 2.6 comparisons are needed for any \( N \)
  -thus, add terminates early, and average cost is \( O(c) \)
remove
let's remove the smallest item

take out 3
let's remove the smallest item

take out 3
let’s remove the smallest item

take out 3

fill with last item on last level.
why?
let's remove the smallest item

take out 3

fill with last item on last level. why?
let's remove the smallest item

take out 3

fill with last item on last level. why?
let's remove the smallest item

take out 3

fill with last item on last level. why?

percolate down
let's remove the smallest item

take out 3

fill with last item on last level. why?

percolate down
let's remove the smallest item

take out 3

fill with last item on last level. why?

percolate down
let’s remove the smallest item

take out 3

fill with last item on last level. why?

percolate down
let's remove the smallest item

take out 3

fill with last item on last level. why?

percolate down
let's remove the smallest item

take out 3

fill with last item on last level.
why?

percolate down
let's remove the smallest item

take out 3

fill with last item on last level.
why?

percolate down
complexity of remove?
cost of remove

-worst case is $O(\log N)$
  -percolating down to the bottom level

-average case is also $O(\log N)$
  -rarely terminates more than 1-2 levels from the bottom… why?
recap
-priority queues can be implemented any number of ways

-a binary heap’s main use is for implementing priority queues

-remember, the basic priority queue operations are:
  -add
  -findMin
  -deleteMin
-the average cases for a PQ implemented with a binary heap:
  - add
    - $O(c)$: percolate up (average of 2.6 compares)
  - findMin
    - $O(c)$: just return the root
  - deleteMin
    - $O(\log N)$: percolate down (rarely terminates before near the bottom of the tree)
- option 1: a linked list
  - add: $O(c)$
  - findMin: $O(N)$
  - deleteMin: $O(N)$ (including finding)

- option 2: a sorted linked list
  - add: $O(N)$
  - findMin: $O(c)$
  - deleteMin: $O(c)$

- option 3: a binary heap
  - add: $O(c)$
  - findMin: $O(c)$
  - deleteMin: $O(\log N)$
next time...
-reading
  - chapter 21 in book

-homework
  - assignment 10 due tonight