Quadtrees
CSE 373 Winter 2020

Instructor: Hannah C. Tang

Teaching Assistants:
Aaron Johnston    Ethan Knutson    Nathan Lipiarski
Amanda Park      Farrell Fileas    Sam Long
Anish Velagapudi Howard Xiao    Yifan Bai
Brian Chan       Jade Watkins    Yuma Tou
Elena Spasova    Lea Quan
Announcements

❖ Homework 4: Heap is released and due *Wednesday*
  - Hint: you will need an additional data structure to improve the runtime for changePriority(). It does not affect the correctness of your PQ at all. Please use a built-in Java collection instead of implementing your own.
  - Hint: If you implemented a unittest that tested the exact thing the autograder described, you could run the autograder’s test in the debugger (and also not have to use your tokens).

❖ Please look at posted QuickCheck; we had a few corrections!
Lecture Outline

❖ Heaps, cont.: buildHeap

❖ Review: Set/Map data structures and logarithmic runtimes

❖ Multi-dimensional Data

❖ Uniform and Recursive Partitioning

❖ Quadtrees
Other Priority Queue Operations

❖ The two “primary” PQ operations are:
  ▪ removeMax()
  ▪ add()

❖ However, because PQs are used in so many algorithms there are three common-but-nonstandard operations:
  ▪ merge(): merge two PQs into a single PQ
  ▪ buildHeap(): reorder the elements of an array so that its contents can be interpreted as a valid binary heap
  ▪ changePriority(): change the priority of an item already in the heap
buildHeap: Naïve Implementation

- buildHeap() takes an array of size N and applies the heap-ordering principle to it

- Naïve implementation:
  - Start with an empty array (representing an empty binary heap)
  - Call add() N times
  - Runtime: \( \Theta(N \log N) \)

- Can we do better?
buildHeap: Clever Implementation

- $\approx \frac{1}{2}$ of all nodes in a complete binary tree are leaves
  - Remember that $2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1$

- Clever implementation:
  - Start with full array (representing a binary heap with lots of violations)
  - Call `percolateDown()` N/2 times starting from the rightmost leaf parent (i.e., middle of array)
  - Runtime: ??

This "clever implementation" is called Floyd’s Algorithm
What is buildHeap()’s runtime?

- Start with full array (representing a binary heap with lots of violations)
- Call percolateDown() N/2 times

A. Θ(1)
B. Θ(log N)
C. Θ(N)
D. Θ(N log N)
E. I’m not sure...
Lecture Outline

❖ Heaps, cont.: buildHeap

❖ **Review: Set/Map data structures and logarithmic runtimes**

❖ Multi-dimensional Data

❖ Uniform and Recursive Partitioning

❖ Quad-Trees
ADT / Data Structure Taxonomy

Maps and Sets

- Search Trees ("left is less-than, right is greater-than")
  - Binary Search Trees (branching factor == 2)
    - Plain BST (unbalanced)
      - Balanced BSTs: LLRB (other examples: "Classic" Red-Black, AVL, Splay, etc)
  - B-Trees (have a branching factor >2; balanced)
    - 2-3 Trees
    - 2-3-4 Trees

- Hash Tables (will cover later!)
Why Does Balance Matter?

- Balanced trees help us avoid considering all of the data all of the time

  - **Binary Search Tree**: Discarding approximately half of the remaining data at each recursive step leads to a logarithmic runtime

  - **Binary Heap**: Recursively percolating up one level approximately halves the number of potential positions to consider, again leading to a logarithmic runtime
Lecture Outline

- Heaps, cont.: buildHeap

- Review: Set/Map data structures and logarithmic runtimes

- Multi-dimensional Data

- Uniform and Recursive Partitioning

- Quad-Trees
Autocomplete as a 1-Dimensional Range Search

- Location names can be sorted 1-dimensionally (lexicographically aka dictionary ordering)

- Since the data is sorted, we could run two binary searches on the array
  - Range Search Runtime: \( O(\log N) \) (twice)
  - Insert Runtime: \( O(N) \)
Autocomplete as a 1-Dimensional Range Search

- Or we could do a range search in a balanced BST
  - Range Search Runtime: $O(\log N)$
  - Insert Runtime: $O(\log N)$

```java
void printRange(Node root, Key lo, Key hi) {
    if (root == null) return;

    if (lo < root->key)
        printRange(root->left, lo, hi);
    if (lo <= root->key && root->key >= hi)
        print(root->key);
    if (root->key > hi)
        printRange(root->right, lo, hi);
}
```
Geo-locating a Click on a 2D Map

❖ Why do some map clicks resolve to a lat/lng?
❖ And other clicks resolve to a point-of-interest?
2-d Range Search: Naïve Implementation

- Check every point for containment in the click target (ie, consider all of the data)

- **Range Search**
  - Scan through all the keys and collect matching results
  - Runtime: $O(N)$

- **Nearest Neighbour**
  - Range Search, hope for an non-empty result, iterate through results and choose nearest
  - Runtime: $O(N)$

- **Insert**
  - Put key anywhere
  - Runtime: $O(1)$
Lecture Outline

- Heaps, cont.: buildHeap
- Review: Set/Map data structures and logarithmic runtimes
- Multi-dimensional Data
- **Uniform and Recursive Partitioning**
- Quad-Trees
Uniform Partitioning

- Divide space into non-overlapping _subspaces_
  - Known as “spatial partitioning problem”

- **Uniform partitioning strategy**
  - Partition space into uniform rectangular buckets (“bins”)
  - Ex: 4x4 grid of such buckets.
What is the runtime to find the nearest neighbour to our blue point, assuming \( N \) points are evenly spread out across a 16-bin uniform partition?

A. \( \Theta(1) \)

B. \( \Theta(\log N) \)

C. \( \Theta(N) \)

D. \( \Theta(N^2) \)

E. I’m not sure ...

\( N/16 \in \Theta(N) \), so we are, essentially, still considering all of our data. We can do better!
Recursive Partitioning: An x-coordinate BST?

- Suppose we put points into a BST map ordered by x-coordinate.

```
A (-1, -1)
  E (-2, -2)
  F (-3, 2.5)
  C (0, 1)
  D (1, 0)

B (2, 2)
```

```
F (-3, 2.5)
(0, 1)
C
(2, 2)
D (1, 0)
E (-2, -2)
A (-1, -1)
```
Recursive Partitioning: An x-coordinate BST?

- Range Searching becomes: “What are all the points with x-coordinate less than -1.5?”
Recursive Partitioning: A y-coordinate BST?

But in a y-coordinate BST, we can’t prune anything!
Lecture Outline

❖ Heaps, cont.: buildHeap

❖ Review: Set/Map data structures and logarithmic runtimes

❖ Multi-dimensional Data

❖ Uniform and Recursive Partitioning

❖ Quad-Trees
Recursive Partitioning: Quadtree

- **1-dimensional data**
  - Keys are ordered on a line
  - Recursive decision: left or right

- **2-dimensional data**
  - Keys are located on a plane
  - Recursive decision: northwest, northeast, southwest, southeast.

---

Binary Search Tree

Quadtrees
Using Quadtrees to Recursively Partition

- Quadtrees produce recursive, hierarchical partitionings
  - Each point owns 4 subspaces
Quadtree: Insert

Demo: https://docs.google.com/presentation/d/1vqAJkvUxSh-Eq4iJZevjpY29nagNTjx-4N3HpDi0UQ/present?ueb=true&slide=id.g11ecaeaf56_0_0
Quadtree: Range Search

We can prune unnecessary subspaces!

Demo:
https://docs.google.com/presentation/d/1ZVvh_Q15Lh2D1_NnzZ4PR_aDsLBwvAU9JYQAwlSuXSM/present?ueb=true&slide=id.g52a9824549_0_129
3-dimensional Data and Beyond

- Oct-trees are generalization of quadtrees for 3D data
- Quadtree Applications:
tl;dr

❖ A Priority Queue’s core functionality is removeMax and add
  ▪ changePriority can be $\Theta(\log N)$ if you use an auxiliary data structure
  ▪ buildHeap can be $\Theta(N)$ if you percolate carefully

❖ Recursively subdividing input:
  ▪ allows you to find one piece data without examining all of it
  ▪ often yields logarithmic runtime

❖ Quadtrees allow you to recursively partition 2-dimensional data using a single 4-way question

<table>
<thead>
<tr>
<th>Range Search</th>
<th>Nearest Neighbour</th>
<th>Add</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta(\log N)$</td>
<td>$\Theta(\log N)$</td>
<td>$\Theta(\log N)$</td>
</tr>
</tbody>
</table>