CSCI 136
Data Structures &
Advanced Programming

Lecture 23

Fall 2019

Instructor: B&S

Administrative Details

- Lab 8: Simulations
  - You will simulate two queuing strategies
  - You can work with a partner
  - Time spent on lab before Wed. is time well-spent!
- Problem Set 3 is online
  - Due this Friday at beginning of class

Last Time

Improving Huffman’s Algorithm

- Priority Queues & Heaps
  - A “somewhat-ordered” data structure
    - Conceptual structure
    - Efficient implementations

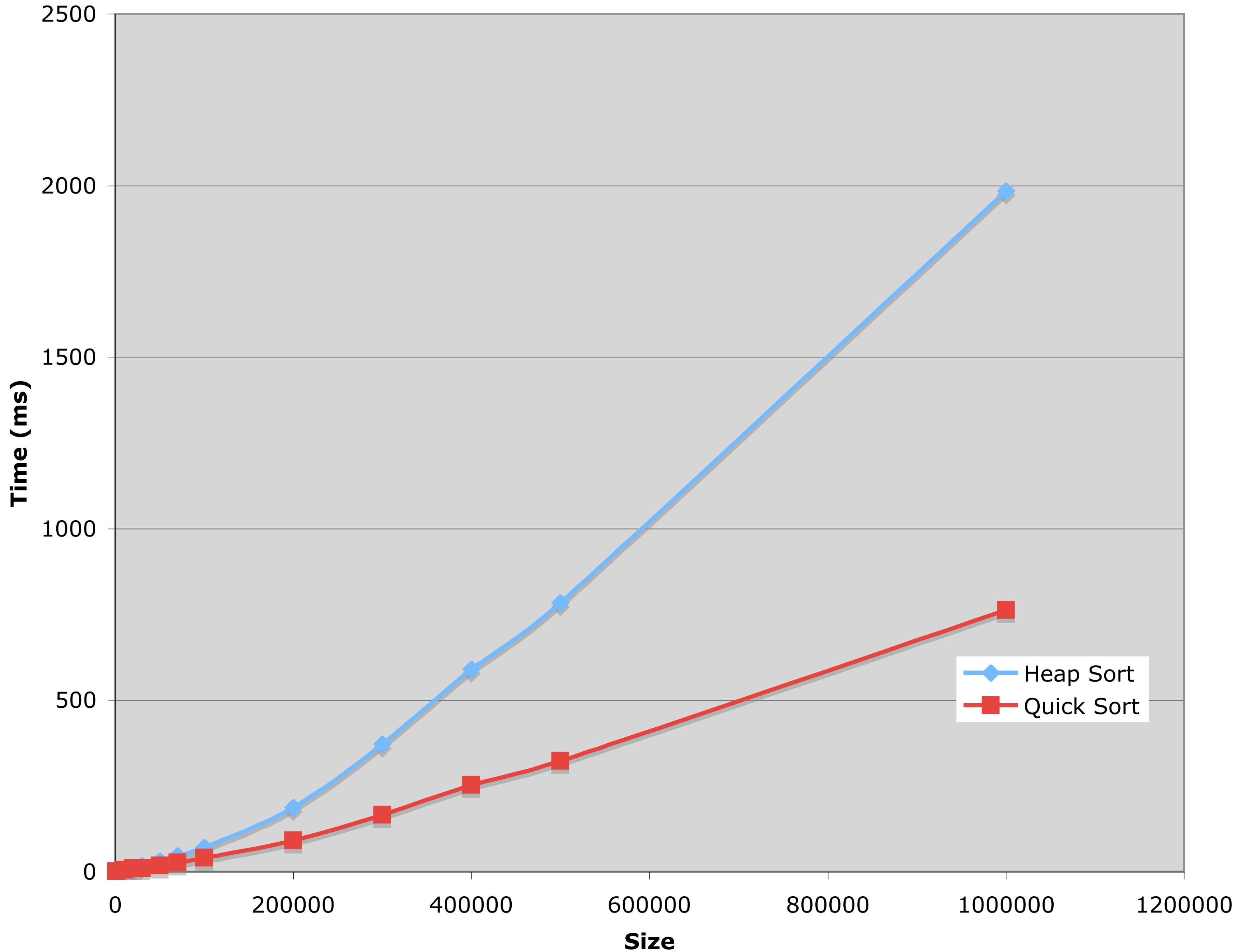
Today

- Finishing up with heaps
  - HeapSort
  - Alternative Heap Structures
- Binary Search Tree: A New Ordered Structure
  - Definitions
  - Implementation

HeapSort

- Heaps yield another O(n log n) sort method
- To HeapSort a Vector "in place"
  - Perform bottom-up heapify on the reverse ordering: that is: highest rank/lowest priority elements are near the root (low end of Vector)
  - Now repeatedly remove elements to fill in Vector from tail to head
    - For(int i = v.size() – 1; i > 0; i--)
      - RemoveMin from v[0..i] // v[i] is now not in heap
      - Put removed value in location v[i]

Heap Sort vs QuickSort



Why Heapsort?

- Heapsort is slower than Quicksort in general
- Any benefits to heapsort?
  - *Guaranteed* O(n log n) runtime
- Works well on mostly sorted data, unlike quicksort
- Good for incremental sorting

More on Heaps

- Set-up: We want to build a *large* heap. We have several processors available.
- We'd like to use them to build smaller heaps and then merge them together
- Suppose we can share the array holding the elements among the processors.
  - How long to merge two heaps?
  - How complicated is it?
- What if we use BinaryTrees for our heaps?

Mergeable Heaps

- We now want to support the additional *destructive* operation merge(heap1, heap2)
- Basic idea: heap with larger root somehow points into heap with smaller root
- Challenges
  - Points how? Where?
  - How much reheapifying is needed
  - How deep do trees get after many merges?

Skew Heap

- Don't force heaps to be complete BTs?
- Develop recursive merge algorithm that keeps tree shallow over time
- Theorem: Beginning with an empty SkewHeap, any set of m SkewHeap operations can be performed in O(m log n) time, where n is the total number of items in the SkewHeaps
  - So the *amortized* run-time of each operation is O(log n) !
- Let's sketch out merge operation....

Skew Heap: Merge Pseudocode

```
SkewHeap merge(SkewHeap S, SkewHeap T)
    if either S or T is empty, return the other
    if T.minValue < S.minValue
        swap S and T        (S now has minValue)
    if S has no left subtree, T becomes its left subtree
    else
        let temp point to right subtree of S
        left subtree of S becomes right subtree of S
        merge(temp, T) becomes left subtree of S
    return S
```

Skew Heaps

How would you implement add and remove?

Tree Summary

- Trees
  - Express hierarchical relationships
  - Tree structure captures relationship
    - i.e., ancestry, game boards, decisions, etc.
- Heap
  - Partially ordered tree based on item priority
  - Node invariants: parent has higher priority than each child
  - Provides efficient PriorityQueue implementation

Improving on OrderedVector

- The OrderedVector class provides O(log n) time searching for a group of n comparable objects
  - add() and remove(), though, take O(n) time in the worst case---and on average!
- Can we improve on those running times without sacrificing the O(log n) search time?
- Let's find out....

Binary Trees and Orders

- Binary trees impose multiple orderings on their elements (pre-/in-/post-/level-orders)
- In particular, in-order traversal suggests a natural way to hold comparable items
  - For each node v in tree
    - All values in left subtree of v are $\leq$ v
    - All values in right subtree of v are $\geq$ v
- This leads us to...

Binary Search Trees

- Binary search trees maintain a *total* ordering among elements
- Definition: A BST T is either:
  - Empty
  - Has root r with subtrees T_L and T_R such that
    - All nodes in T_L have smaller value than r
    - All nodes in T_R have larger value than r
    - T_L and T_R are also BSTs
- Examples….

BST Observations

- The same data can be represented by many BST shapes
- Searching for a value in a BST takes time proportional to the height of the tree
  - Reminder: trees have height, nodes have depth
- Additions to a BST happen at nodes missing at least one child (*a constraint!*)
- Removing from a BST can involve *any* node

BST Operations

- BSTs will implement the OrderedStructure Interface
  - `add(E item)`
  - `contains(E item)`
  - `get(E item)`
  - `remove(E item)`
  - `iterator()`
    - This will provide an in-order traversal
- Runtime of add, contains, get, remove: O(height)
- Goal: Keep the height to O(log n)
    - Duane’s BinarySearchTree class doesn’t achieve this…
    - But his RedBlackSearchTree does!

Application: Dictionary

- Create a BST of ComparableAssociations
  - Order BST by key
  - Two objects are equal if keys are equal

- Example: Symbol tables (PostScript lab) are Dictionaries
  - But would only use a BST if the set of possible symbols was very large

Application: Tree Sort

- Can we sort data using a BST?
  - Yes!
- Runtime?
  - To build a tree with n elements, we do n insertions: O(n*h), where h is the maximum height attained by the tree
  - In order traversal: O(n)
  - Total runtime: O(n*h)

BST Implementation

- The BST holds the following items
  - BinaryTree root: the root of the tree
  - BinaryTree EMPTY: a static empty BinaryTree
    - To use for all empty nodes of tree
  - int count: the number of nodes in the BST
  - Comparator<E> ordering: for comparing nodes
    - Note: E must implement Comparable
- Two constructors: One takes a Comparator
  - The other creates a NaturalComparator

BST Implementation: locate

- Several methods search the tree: add, remove, contains
- We factor out common code: locate method
- *protected* locate(BinaryTree<E> *node*, E *v*)
  - Returns a BinaryTree<E> in the subtree with root *node* such that either
    - *node* has its value equal to *v*, or
    - *v* is not in this subtree and *node* is where *v* would be added as a (left or right) child
- How would we implement locate()?

BST Implementation: locate

BinaryTree locate(BinaryTree root, E value)

if root's value equals value return root

child ← child of root that should hold value

if child is emptry tree, return root

// value not in subtree based at root

else //keep looking

return locate(child, value)