Administrative Details

• Lab 9: Simulations
  • You will simulate two queuing strategies
  • You can work with a partner
  • Time spent on lab before Wed. is time well-spent!
Last Time

• Finishing up with heaps
  • More on implementation
  • “Heapifying” constructor for VectorHeap
  • Alternate heapify approach
Today

• Finishing up with heaps
  • HeapSort
  • Alternative Heap Structures

• Binary Search Tree: A New Ordered Structure
  • Definitions
  • Implementation
Heapifying A Vector (or array)

• Method I: Top-Down
  • Assume $V[0..k]$ satisfies the heap property
  • Now call percolate on item in location $k+1$
  • Then $V[0..k+1]$ satisfies the heap property

• Method II: Bottom-up
  • Assume $V[k..n]$ satisfies the heap property
  • Now call pushDown on item in location $k-1$
  • Then $V[k-1..n]$ satisfies heap property

• Check out the demos at visualgo.net
Top-Down vs Bottom-Up

- Top-down heapify: elements at depth $d$ may be swapped $d$ times: Total # of swaps is at most

$$
\sum_{d=1}^{h} d2^d = (h - 1)2^{h+1} = (\log n - 1)2n + 2
$$

- This is $O(n \log n)$

- Some intuition: most of the elements are in the lowest levels of the tree, so each of them might have to move to root: $O(\log n)$ swaps per element
Top-Down vs Bottom-Up

- Bottom-up heapify: elements at depth $d$ may be swapped $h-d$ times: Total # of swaps is at most

$$\sum_{d=1}^{h} (h-d)2^d = 2^{h+1} - h - 2 = 2n - \log n + 2$$

- This is $O(n)$ --- beats top-down!

- Some intuition: most of the elements are in the lowest levels of the tree, so each of them will only be pushed down (swapped) a small number of times SO COOL!!!
Some Sums

\[ \sum_{d=0}^{d=k} 2^d = 2^{k+1} - 1 \]

\[ \sum_{d=0}^{d=k} r^d = \frac{(r^{k+1} - 1)}{(r - 1)} \]

\[ \sum_{d=1}^{d=k} d \cdot 2^d = (k - 1) \cdot 2^{k+1} + 2 \]

\[ \sum_{d=1}^{d=k} (k - d) \cdot 2^d = 2^{k+1} - k - 2 \]

All of these can be proven by (weak) induction.

Try these to hone your skills

The second sum is called a geometric series. It works for any \( r \neq 0 \)
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

- Time (ms) vs Size
- Heap Sort
- Quick Sort
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: 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
• Let’s sketch out merge operation....
Skew Heap: Merge Pseudocode

\( \text{SkewHeap merge}(\text{SkewHeap } S, \text{SkewHeap } T) \)

if either \( S \) or \( T \) is empty, return the other

if \( T.\text{minValue} < S.\text{minValue} \)

\( \text{swap } S \text{ and } T \) \hspace{1cm} (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 \( S \)
Tree Summary

• Trees
  • Express hierarchical relationships
  • Level ordering captures the 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: \( O(\text{height}) \)

- 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: $O(n^\ast h)$, where h is the maximum height attained by the tree
  • In order traversal: $O(n)$
  • Total runtime: $O(n^\ast h)$