

**CSCI 136**  
**Data Structures &**  
**Advanced Programming**

**Lecture 22**

**Fall 2017**

**Instructor: Bills**

# Announcement

Power outage (3-5am)

We'll be shutting down systems at 10pm tonight

Rebooting at 9am tomorrow

# Last Time

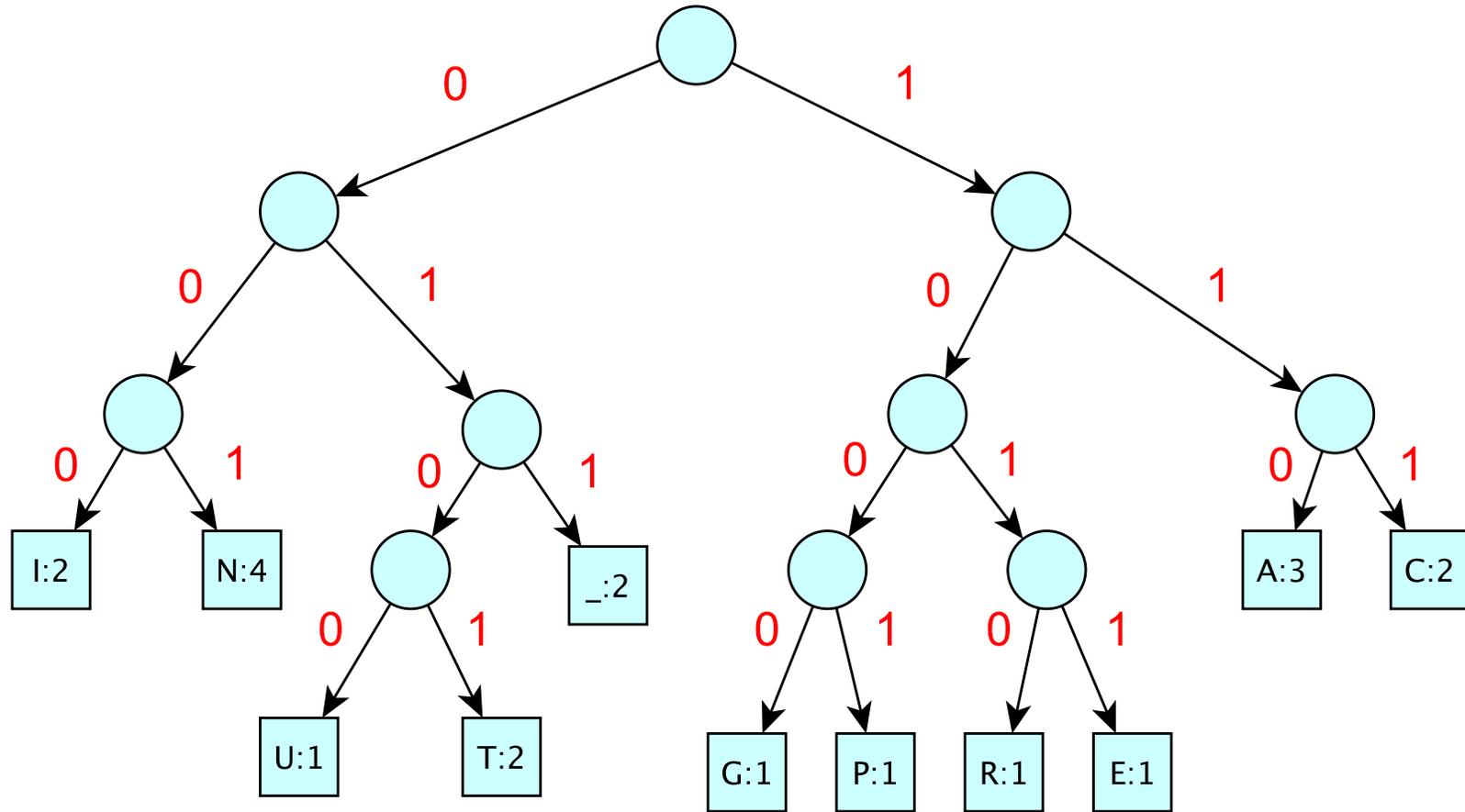
- Wrap up Binary Tree Iterators
- Breadth-First and Depth-First Search
- Array Representations of (Binary) Trees
- Application: Huffman Encoding

# Today

## Improving Huffman's Algorithm

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

# The Encoding Tree



Left = 0; Right = 1

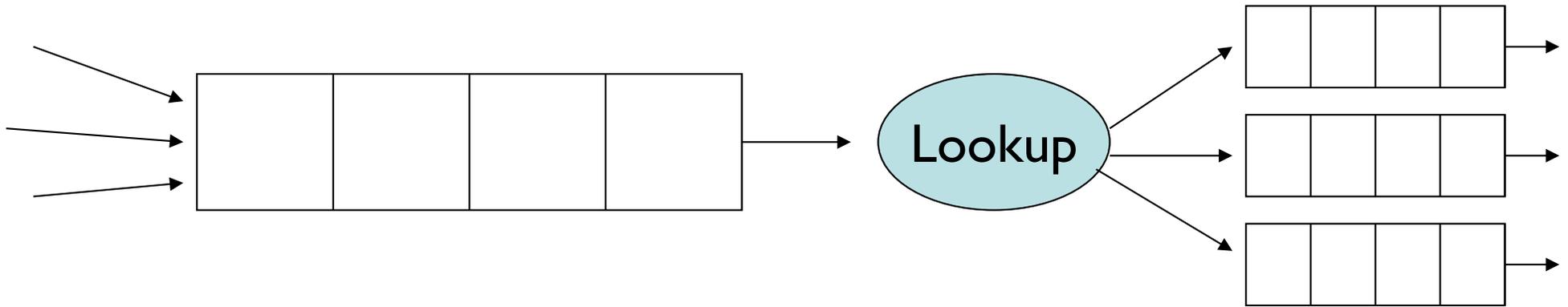
# Recall : Huffman Encoding Algorithm

- Keep a Vector of Binary Trees
- Sort them by decreasing frequency
  - Removing two smallest frequency trees is fast
- Insert merged tree into correct (sorted) location in Vector
- Running Time:
  - $O(n \log n)$  for initial sorting
  - $O(n^2)$  for rest:  $O(n)$  for each re-insertion
- Can we do better...?

# What Huffman Encoder Needs

- A structure  $S$  to hold items with *priorities*
- $S$  should support operations
  - `add(E item); // add an item`
  - `E removeMin(); // remove min priority item`
- $S$  should be designed to make these two operations fast
- If, say, they both ran in  $O(\log n)$  time, the Huffman while loop would take  $O(n \log n)$  time instead of  $O(n^2)$ !
- We've seen this situation before....

# Priority Queues



## Packet Sources May Be Ordered by Sender

sysnet.cs.williams.edu

bull.cs.williams.edu

yahoo.com

spammer.com

priority = 1 (best)

2

10

100 (worst)

# Priority Queues

- Priority queues are also used for:
  - Scheduling processes in an operating system
    - Priority is function of time lost + process priority
  - Order services on server
    - Backup is low priority, so don't do when high priority tasks need to happen
  - Scheduling future events in a simulation
  - Medical waiting room
  - Huffman codes - order by tree root "frequency"
  - A variety of graph/network algorithms
  - To roughly rank choices that are generated out of order

# Priority Queues

- Name is misleading: They are **not FIFO**
- Always dequeue object with **highest priority** (smallest rank) regardless of when it was enqueued
- Data can be received/inserted in any order, but it is always returned/removed according to priority
- Like ordered structures (i.e., OrderedVectors and OrderedLists), PQs require comparisons of values

# An Apology

- On behalf of computer scientists everywhere, I'd like to apologize for the confusion that inevitably results from the fact that  
Higher Priority  $\leftrightarrow$  Lower Rank
- The PQ removes the *lowest ranked* value in an ordering: that is, the *highest priority* value!

We're sorry!

# PQ Interface

```
public interface PriorityQueue<E extends Comparable<E>> {  
    public E getFirst(); // peeks at minimum element  
    public E remove(); // removes minimum element  
    public void add(E value); // adds an element  
    public boolean isEmpty();  
    public int size();  
    public void clear();  
}
```

# Notes on PQ Interface

- Unlike previous structures, we do not extend any other interfaces
  - Many reasons: For example, it's not clear that there's an obvious iteration order
- PriorityQueue uses Comparables: methods *consume* Comparable parameters and *return* Comparable values
  - Could be made to use Comparators instead...

# Implementing PQs

- Queue?
  - Wouldn't work so well because we can't insert and remove in the "right" way (i.e., keeping things ordered)
- OrderedVector?
  - Keep ordered vector of objects
  - $O(n)$  to add/remove from vector
  - Details in book...
  - Can we do better than  $O(n)$ ?
- Heap!
  - Partially ordered binary tree

# Heap

- A heap is a special type of tree
  - Root holds smallest (highest priority) value
  - Subtrees are also heaps (this is important!)
- So values increase in priority (decrease in rank) from leaves to root (from descendant to ancestor)
- *Invariant for nodes:* For each child of each node
  - `node.value() <= child.value()` // if child exists
- Several valid heaps for same data set (no unique representation)

# Inserting into a PQ

- Add new value as a leaf
- “Percolate” it up the tree
  - while (value < parent’s value) swap with parent
- This operation preserves the heap property since new value was the only one violating heap property
- Efficiency depends upon speed of
  - Finding a place to add new node
  - Finding parent
  - Tree height

# Removing From a PQ

- Find a leaf, delete it, put its *data* in the root
- “Push” *data* down through the tree
  - while ( *data.value* > value of (at least) one child )
    - Swap *data* with data of **smallest** child
- This operation preserves the heap property
- Efficiency depends upon speed of
  - Finding a leaf
  - Finding locations of children
  - Height of tree

# Implementing Heaps

- VectorHeap
  - Use conceptual array representation of BT (ArrayTree)
  - But use extensible Vector instead of array (makes adding elements easier)
  - Note:
    - Root of tree is location 0 of Vector
    - Children of node in location  $i$  are in locations  $2i+1$  (left) and  $2i+2$  (right)
    - Parent of node  $i$  is in location  $(i-1)/2$

# Implementing Heaps

- Features
  - No gaps in array (array is *complete*)-- why?
    - We always add in next available array slot (left-most available spot in binary tree);
    - We always remove using “final” leaf
  - *Heap Invariant becomes*
    - $\text{data}[i] \leq \text{data}[2i+1]; \text{data}[i] \leq \text{data}[2i+2]$  (or kids might be null)
  - When elements are added and removed, do small amount of work to “re-heapify”
    - How small? Note: finding a node’s child or parent takes constant time, as does finding “final” leaf or next slot for adding
    - Since this heap corresponds to a full binary tree, the depth of the tree is  $O(\log n)$ , so percolate/pushDown takes  $O(\log n)$  time!

# VectorHeap Summary

- Let's look at VectorHeap code....
- Add/remove are both  $O(\log n)$
- Data is not completely sorted
  - “Partial” order is maintained
- Note: `VectorHeap(Vector<E> v)`
  - Takes an unordered Vector and uses it to construct a heap
  - How?

# 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

# Top-Down vs Bottom-Up

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

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

- 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

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

- 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

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

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

$$\sum_{d=0}^{k} r^d = (r^{k+1} - 1) / (r - 1)$$

Try these to hone your skills

$$\sum_{d=1}^{k} d * 2^d = (k - 1) * 2^{k+1} + 2$$

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

$$\sum_{d=1}^{k} (k - d) * 2^d = 2^{k+1} - k - 2$$

# 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]

# Skew Heap

- What if heaps are not complete BTs?
- We can implement PQs using skew heaps instead of “regular” complete heaps
- Key differences:
  - Rather than use Vector as underlying data structure, use BT
  - Need a merge operation that merges two heaps together into one heap
- Details in book