CSCI 136
Data Structures &
Advanced Programming

Lecture 22

Fall 2019

Instructor: B&S

Administrative Details

- Problem Set 3 is available online
  - Due next Friday at *beginning of class!*

Last Time

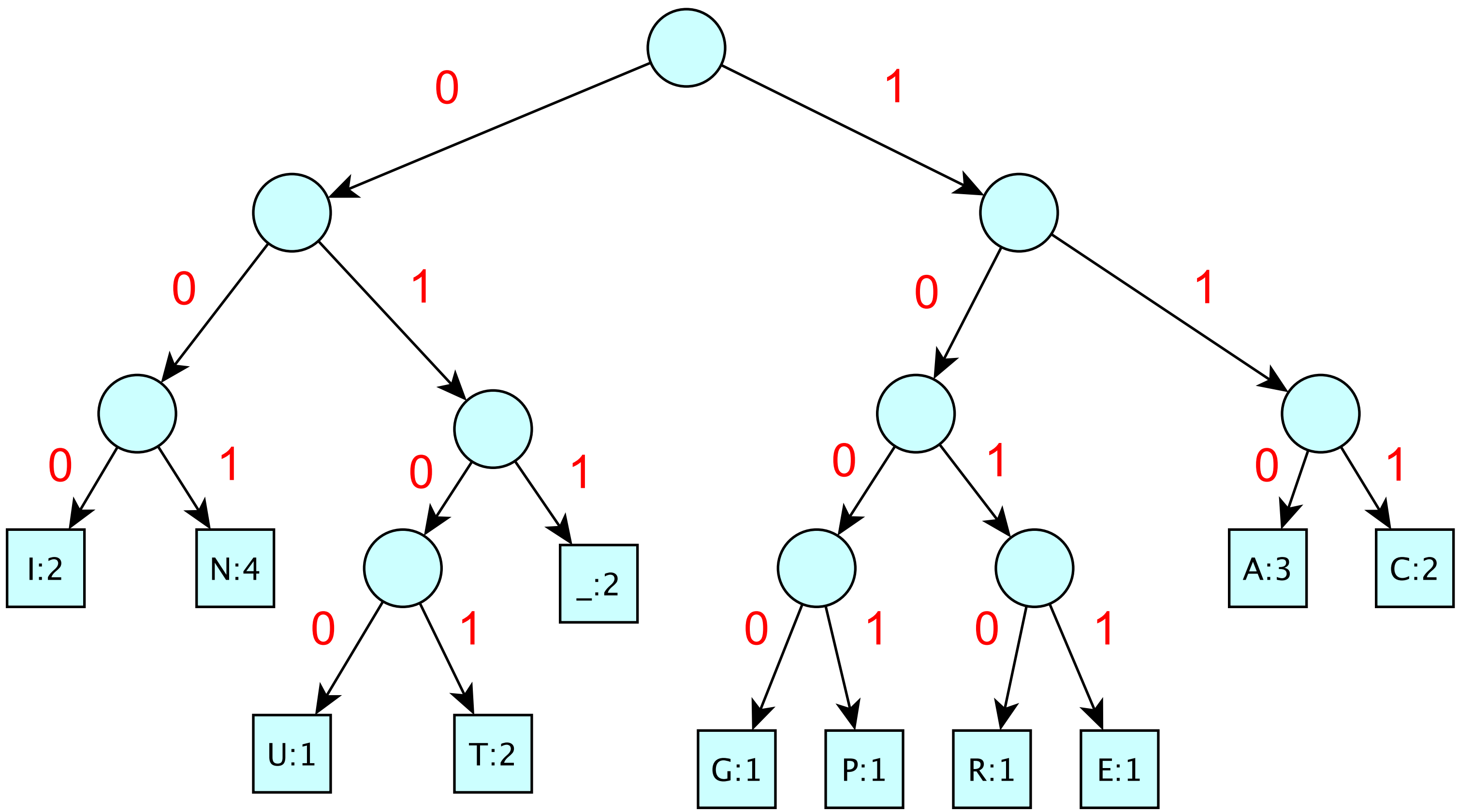
- Array Representations of (Binary) Trees
- Application: Huffman Encoding

Today

Improving Huffman's Algorithm

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

An 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...?

Optimality of Huffman Encoding

Measuring Quality of an Encoding

- Let T be an encoding tree for a variable-length binary encoding for $I = \{(a_i, f_i): 1 \le i \le n\}$
  - $a_1, \dots, a_n$ are letters, $f_1, \dots, f_n$ are frequencies
  - Let d_i be the depth of a_i in T
- Define E(T)–the encoding length of T–by

$$E(T) = \sum_{i=1}^{n} f_i \cdot d(a_i)$$

Theorem: The tree computed by Huffman minimizes $E(T)$ over all prefix-free encodings T

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

- A Priority Queue is a data structure that supports the operations
  - Add(E value) : Add value to PQ
  - removeMin() : remove and return item with minimum value from PQ
  - getMin() : return but don't remove item with minimum value
  - size() : return number of objects in PQ

- There are many possible implementations
- Goal: implement all operations to run in O(log n) time.

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();
}
```

Heap

- A heap is a special type of tree
  - Root holds smallest (highest priority) value
  - Subtrees are also heaps (recursive definition!)
- 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
  - Depth of newly added node

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*
    - data[i] <= data[2i+1]; data[i]<=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?

A Little Bit O' Math

Some facts about binary trees of height h

- Number n_k of nodes at level $k \leq h$

$$1 \leq n_k \leq 2^k$$

- Number F_h of nodes in *full* binary tree of height h:

$$\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

- Number N_h of nodes in tree of height h

$$h + 1 \leq N_h \leq 2^{h+1} - 1$$

Some Sums

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

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

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

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

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 $r \neq 0$

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 at most

$$\sum_{d=0}^{h} d2^{d} = (h-1)2^{h+1} + 2 = (\log n - 1)2n + 2$$

- 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

$$\sum_{d=0}^{h}(h-d)2^d = 2^{h+1} - h - 2 = 2n - \log n + 2$$

  - 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

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

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

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

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

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≠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]