Program Structure and Algorithms

Sid Nath

Lecture 9

Agenda

- Administrative
  - Midterm review
- Greedy
- Dijkstra
- Huffman’s prefix tree

Midterm Solutions

Greedy Algorithms

- A greedy algorithm always makes the choice that looks best ***at the moment***

- Put another way: Greed makes a ***locally*** optimal choice in the hope that this choice will lead to a ***globally*** optimal solution

- Greedy algorithms do **not** always yield optimal solutions, but for some problems they do
  – We'll study some problems where they do

Greedy Algorithms

- When do we use greedy algorithms?
  - When we need a heuristic for a hard problem
  - When the problem itself is “greedy”

- Examples of “Greedy” problems:
  - Minimum Spanning Tree
    - Prim’s and Kruskal’s algorithms follow from “cut property” and “cycle property”, which we’ll see next time
  - Minimum Weight Prefix Codes (Huffman coding)
  - Activity selection
  - Interval scheduling

Properties of Greedy Problems

- Greedy-choice property: A globally optimal solution can be arrived at by making a locally optimal (greedy) choice
  - Difficulty is in proving this…
- Optimal substructure property: An optimal solution to the problem contains ***within it*** optimal solutions to subproblems
  - Key ingredient of both DP and Greed

Examples of Greedy Approaches

- Traveling Salesperson Problem
  - What is a greedy approach? Start somewhere, always go to the nearest unvisited city
- **Knapsack Problem
  - What is a greedy approach? Use as much as possible of the highest value/weight ratio item (optimal for fractional knapsack problem*)
- **Coin Changing Problem
  - What is a greedy approach? Use as much as possible of the largest denominations first (optimal for US currency*)
- Graph Coloring, Vertex Cover, K-Center
  - Min #colors needed so that no edge has same-color endpoints. Use lowest unused color
  - Min #vertices to have at least one endpoint of each edge. Add highest-degree vertex
  - Pick k “centers” out of n points to minimize max point-to-center distance. Add new center that is farthest from all existing centers

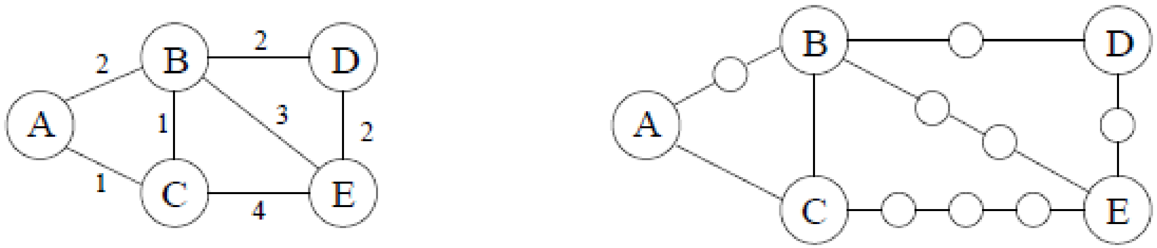
Shortest Path Problem Types

- Given a graph G=(V,E) and w: E → R
  - **(1 to 2) "s-t"** : Find a shortest path from s to t
  - **(1 to all) "single-source" or "SSSP"** : Find a shortest path from a source s to every other vertex v ∈ V
  - **(All to all) "all-pairs" or "APSP"** : Find a shortest path from every vertex to every other vertex

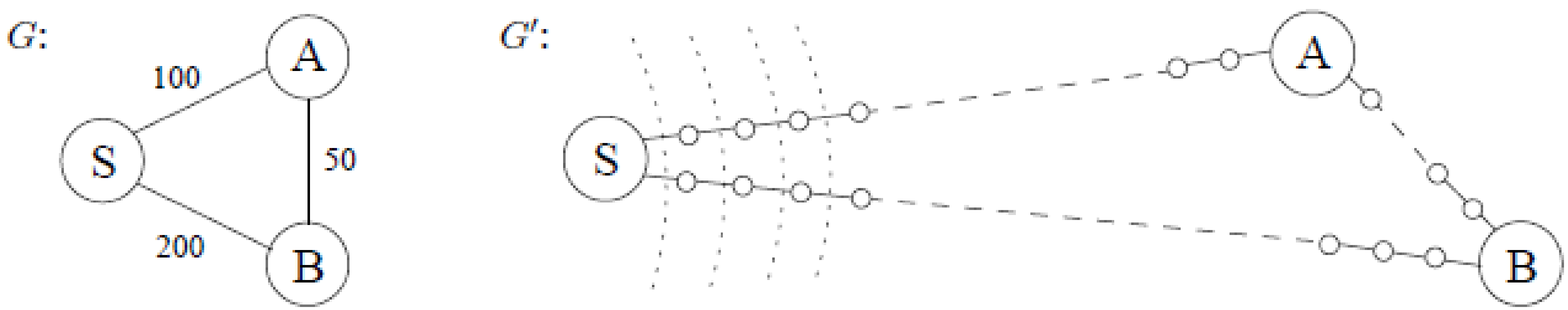
- The *weight* or *cost* of path $v_i, \ldots, v_k = \sum l(v_i, v_i + 1)$
  - **Sometimes: no negative edges.** Examples of "negative edges": travel cost incentives, exothermic chemical reactions, unprofitable transactions in arbitrage, …
    - Bellman-Ford for SP with negative edge weights
  - **Always: no negative cycles.** Otherwise, the shortest-path problem isn't well-defined

Extending BFS

- Suppose graph G has positive, integral edge lengths
- Simple trick: add dummy nodes to make G' with unit-length edges



  - For 'real nodes', distance in G = distances in G'
  - We know how to run BFS on G'
- But BFS on G' could waste a lot of time

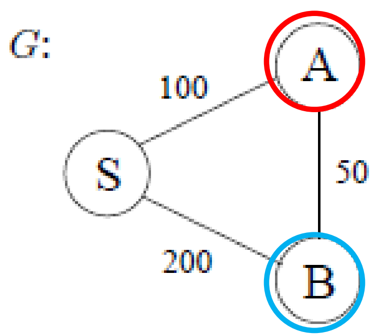


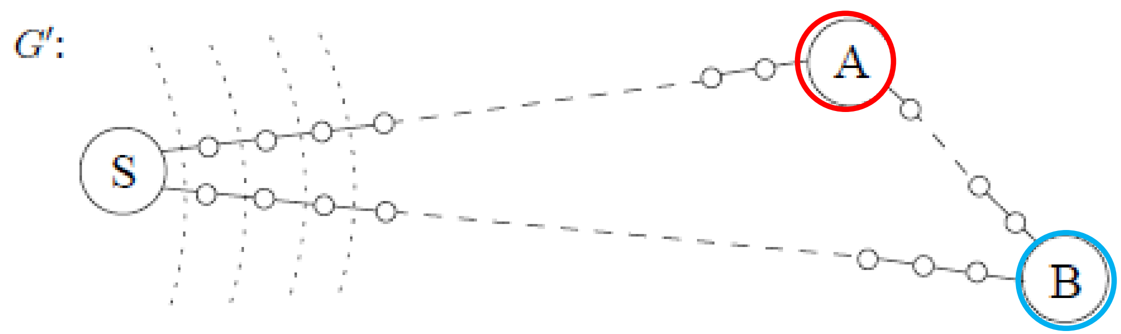
Can we adapt BFS to 'work' only when there is something interesting to do?

"Alarm Clocks"

- Idea
  - Snooze through visits to dummy nodes
  - 'Alarm' should wake us up whenever something interesting is happening, i.e., when BFS encounters a real node

- Alarm for each <u>real</u> node = estimated time of arrival based on edges currently being traversed
  - T = 0: set alarms for A (100), B (200); snooze
  - T = 100: wake up, BFS is at A; set alarm for B (150); snooze
  - T = 150: wake up; done.





“Alarm Clock” Algorithm

- Set an alarm for node s at time T = 0
- If the next alarm goes off at time T, for node u, then:
  - The shortest path distance to u is T
  - For each edge (u,w) ∈ E
    - If no alarm has been set for w, then set an alarm at time T + l(u,w)
    - If an alarm has been set, but at a time later than T + l(u,w), then move it to this earlier time

(1) Exactly simulates BFS on G’

(2) This is also known as Dijkstra’s algorithm

(3) What data structure helps implementation?

Priority Queue

A <u>Priority Queue</u> is a data structure that stores elements sorted by a <u>key</u> value.

<u>Operations:</u>

- Insert – adds a new element to the PQ.
- DecreaseKey – Changes the key of an element of the PQ to a specified *smaller* value.
- DeleteMin – Finds the element with the smallest key and removes it from the PQ.
- For n elements, after each of the above operations, O(log n) time to heapify / sort

Dijkstra's Algorithm

```
Dijkstra(G,s,ℓ)
  Initialize Priority Queue Q
  For v ∈ V
    dist(v) ← ∞
    Q.Insert(v)
  dist(s) ← 0
  While(Q not empty)
    v ← Q.DeleteMin()
    For (v,w) ∈ E
      If dist(v)+ℓ(v,w) < dist(w)
        dist(w) ← dist(v)+ℓ(v,w)
        Q.DecreaseKey(w)
```

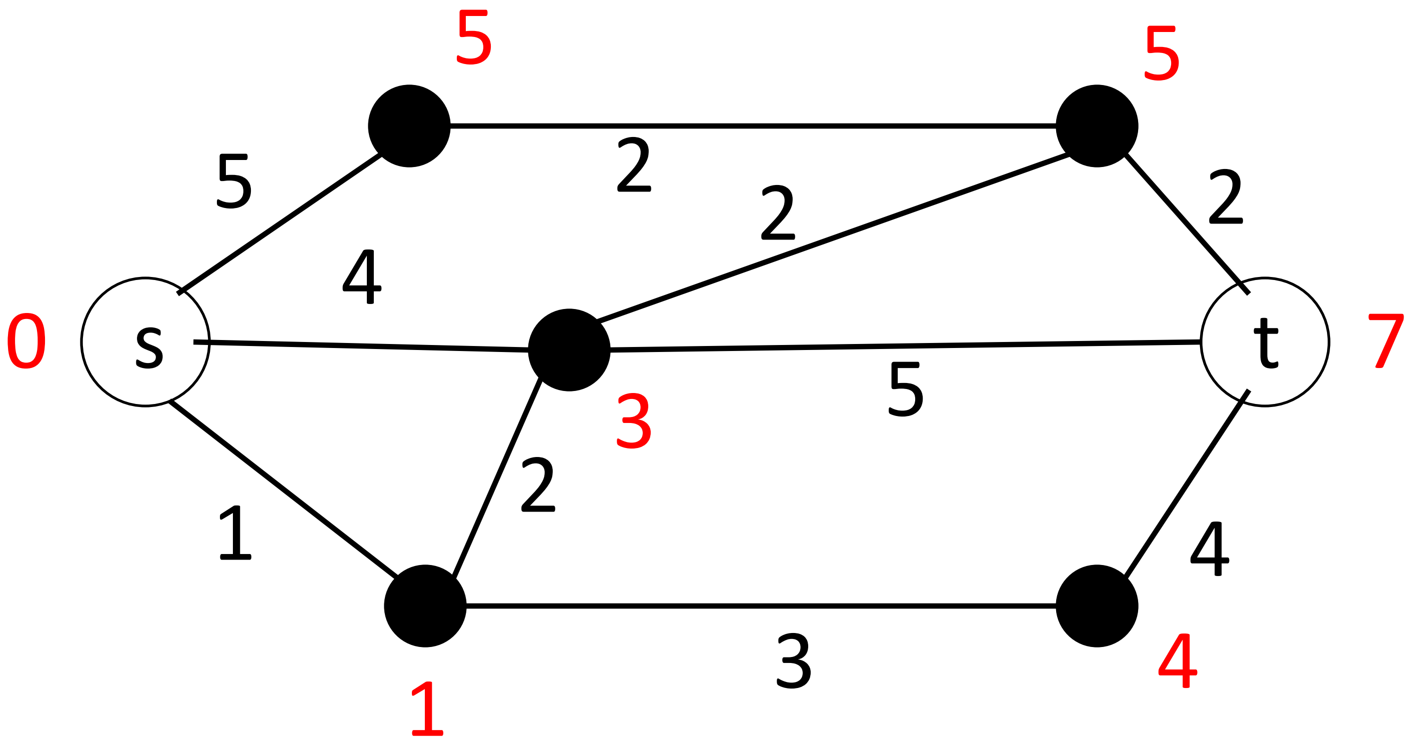
O(|V|) times

O(|V|) times

O(|E|) times

Runtime:
O(|V|) Inserts +
O(|V|) DeleteMins +
O(|E|) DecreaseKeys

Example



0+5=5

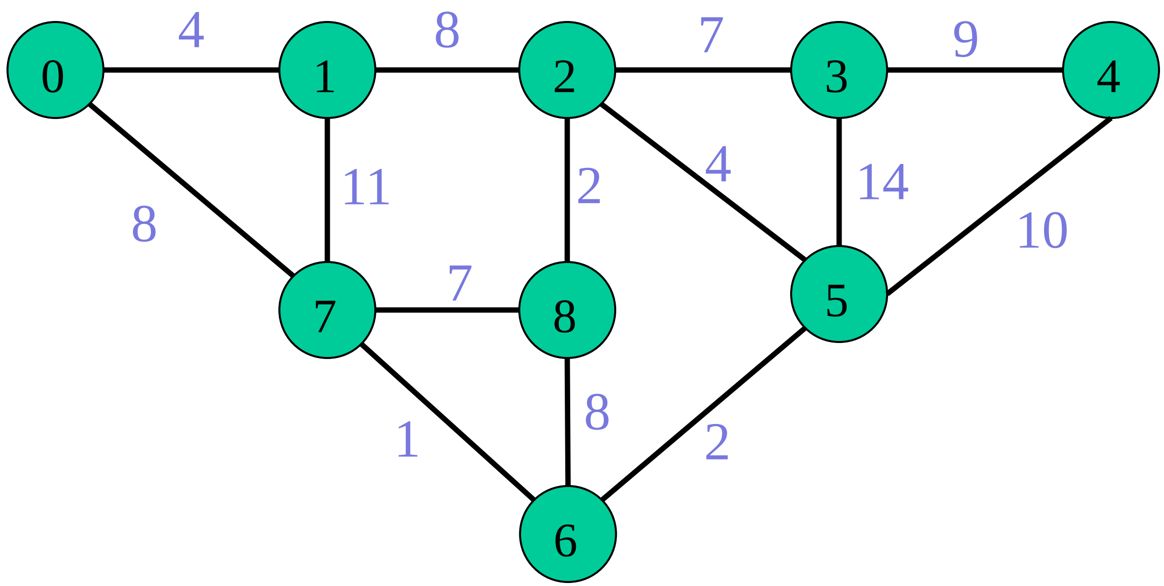
Why does this work?

Claim: Whenever the algorithm assigns a distance to a vertex v that is the length of the shortest path from s to v.

Proof by Induction:

- dist(s) = 0 [the empty path has length 0]
- When assigning distance to w, assume that all previously assigned distances are correct.

Code Demo



Iter	PQ (non-inf) nodes	d(0)	d(1)	d(2)	d(3)	d(4)	d(5)	d(6)	d(7)	d(8)
0	[0]	0	∞	∞	∞	∞	∞	∞	∞	∞
1	[1, 7]	0	4	∞	∞	∞	∞	∞	8	∞
2	[7, 2]	0	4	12	∞	∞	∞	∞	8	∞
3	[6, 2, 8]	0	4	12	∞	∞	∞	9	8	15
4	[5, 2, 8]	0	4	12	∞	∞	11	9	8	15
5	[2, 8, 4, 3]	0	4	12	25	21	11	9	8	15
6	[8, 3, 4]	0	4	12	19	21	11	9	8	14
7	[3, 4]	0	4	12	19	21	11	9	8	14
8	[4]	0	4	12	19	21	11	9	8	14
9	[]	0	4	12	19	21	11	9	8	14

Problem: Huffman Codes

- Given a file of characters from a character set $S = \{a_1, a_2, \dots, a_n\}$ together with the frequencies $\{f_1, f_2, \dots, f_n\}$ compress the file to optimal size:
  - Fixed-length bit-code
  - Variable-length bit-code

- Fixed-length bit-code is easier but achieves low compression ratio
- Variable-length may achieve higher compression ratio
  - Assign shortest bit-code to character with the highest frequency
  - Need to design scheme to avoid ambiguity

Huffman Codes

- **How to transmit English text using binary code?**
- 26 letters + space = alphabet has 27 characters
  - 5 bits per character suffices

- **Observation #1:** Not all characters occur with same frequency
  - Sherlock Holmes, “The Adventure of the Dancing Men” : ETAOIN SHRDLU
  - Suggests **variable-length encoding**

- **Observation #2:** Variable-length code should have prefix property
  - One code word per input symbol
  - No code word is a prefix of any other code word
  - Simplifies decoding process

Huffman Codes

- Prefix codes
  - An optimal code using full tree:
    - Assign bit 0 to left branch
    - Assign bit 1 to right branch
  - No ambiguity
  - Simplify encoding / decoding

Huffman Coding: Greedy Algorithm

- Huffman coding is based on probability with which symbols appear in a message
  - Goal is to minimize the expected code message length

- How it works
  - Create a tree root node for each nonzero symbol frequency, with the frequency as the value of the node
  - REPEAT
    - Find two root nodes with ***smallest*** value
    - Create a new root node with these two nodes as children, and value equal to the sum of the values of the two children
    - (Until there is only one root node remaining)

Example of Huffman Coding

- Example:

Symbol:	A	E	G	I	M	N	O	R	S	T	U	V	Y	Blank
Frequency:	1	3	2	2	1	2	2	2	2	1	1	1	1	3

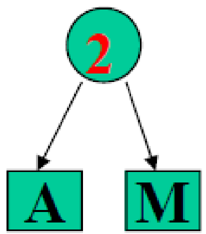
(Generic Implementation)

- Place the elements into a min heap (by frequency)
- Remove the first two elements from the heap
- Combine these two elements into one
- Insert the new element back into the heap

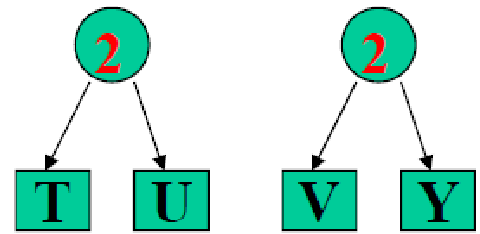
Example of Huffman Coding

Symbol:	A	E	G	I	M	N	O	R	S	T	U	V	Y	Blank
Frequency:	1	3	2	2	1	2	2	2	2	1	1	1	1	3

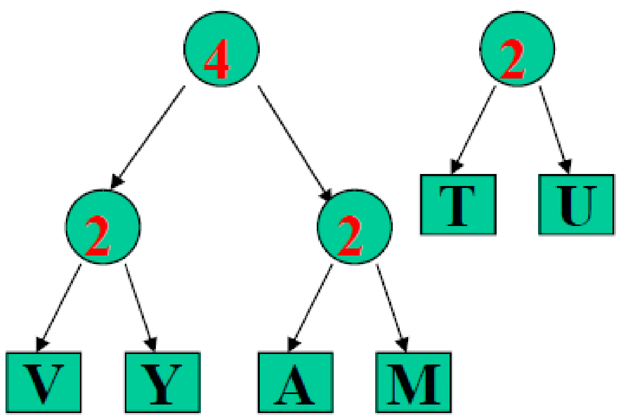
Step 1:



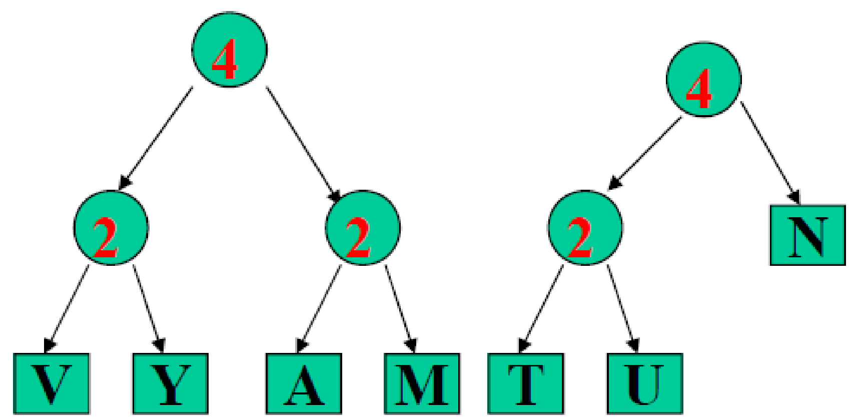
Step 2:



Step 3:



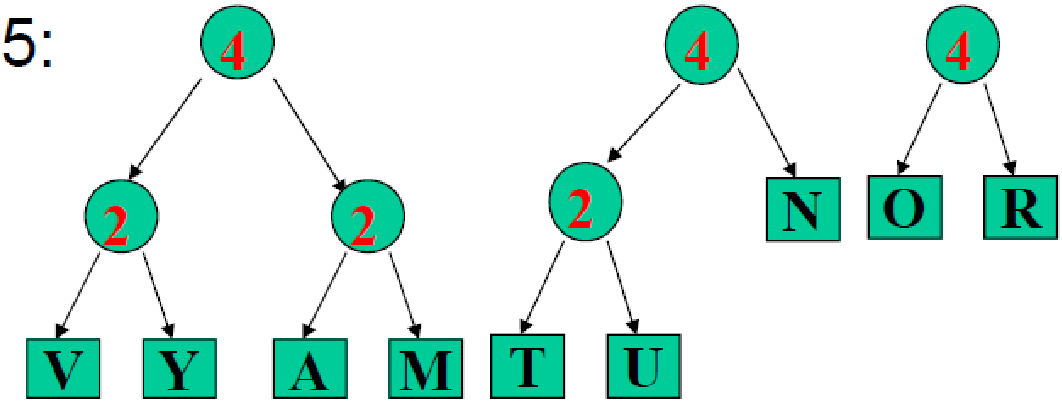
Step 4:



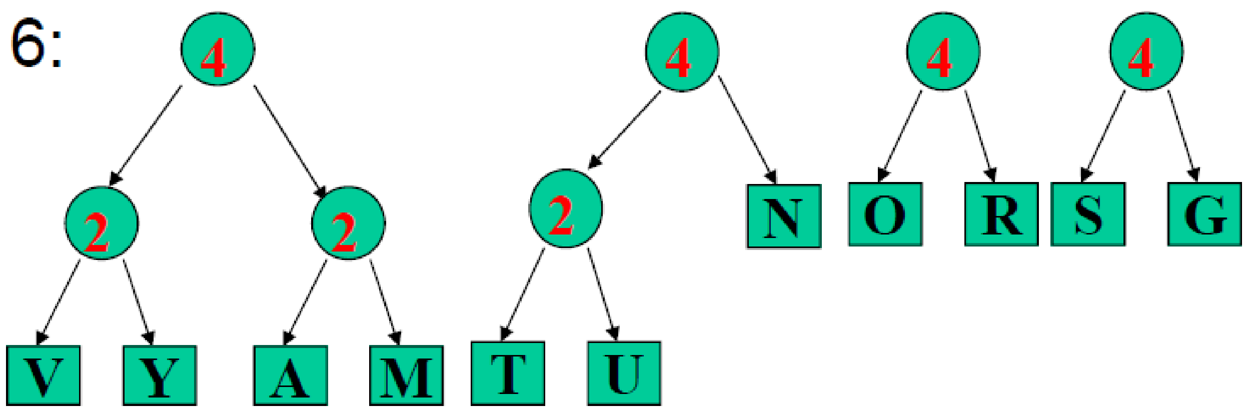
Example of Huffman Coding

Symbol:	A	E	G	I	M	N	O	R	S	T	U	V	Y	Blank
Frequency:	1	3	2	2	1	2	2	2	2	1	1	1	1	3

Step 5:



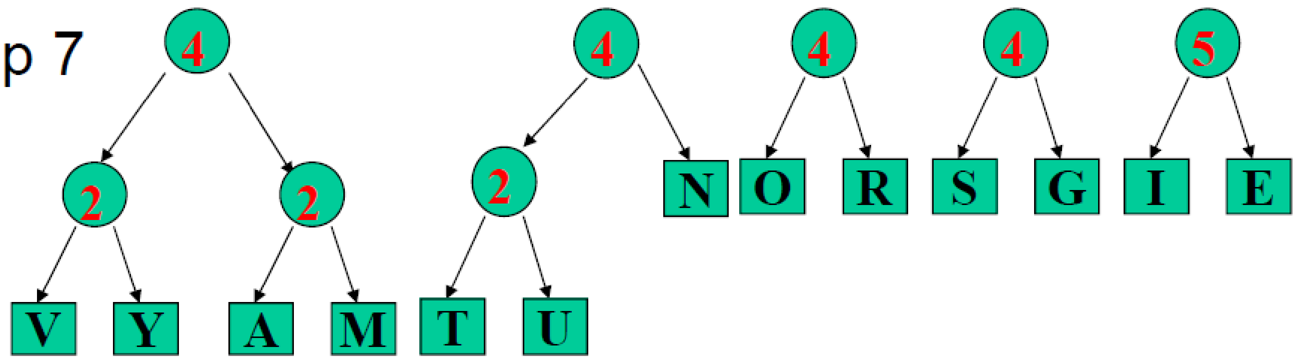
Step 6:



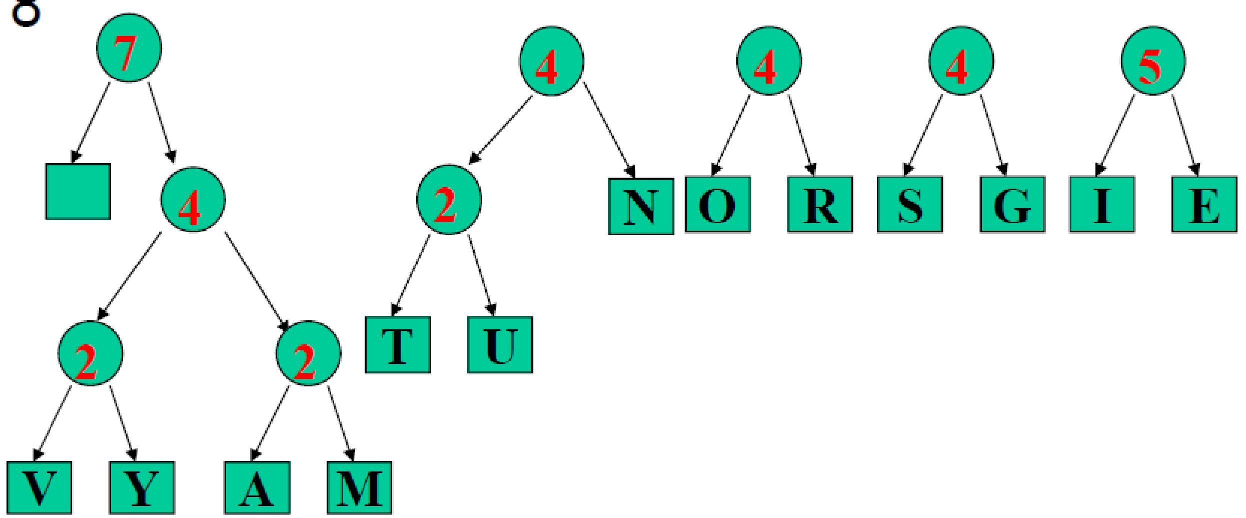
Example of Huffman Coding

Symbol:	A	E	G	I	M	N	O	R	S	T	U	V	Y	Blank
Frequency:	1	3	2	2	1	2	2	2	2	1	1	1	1	3

Step 7



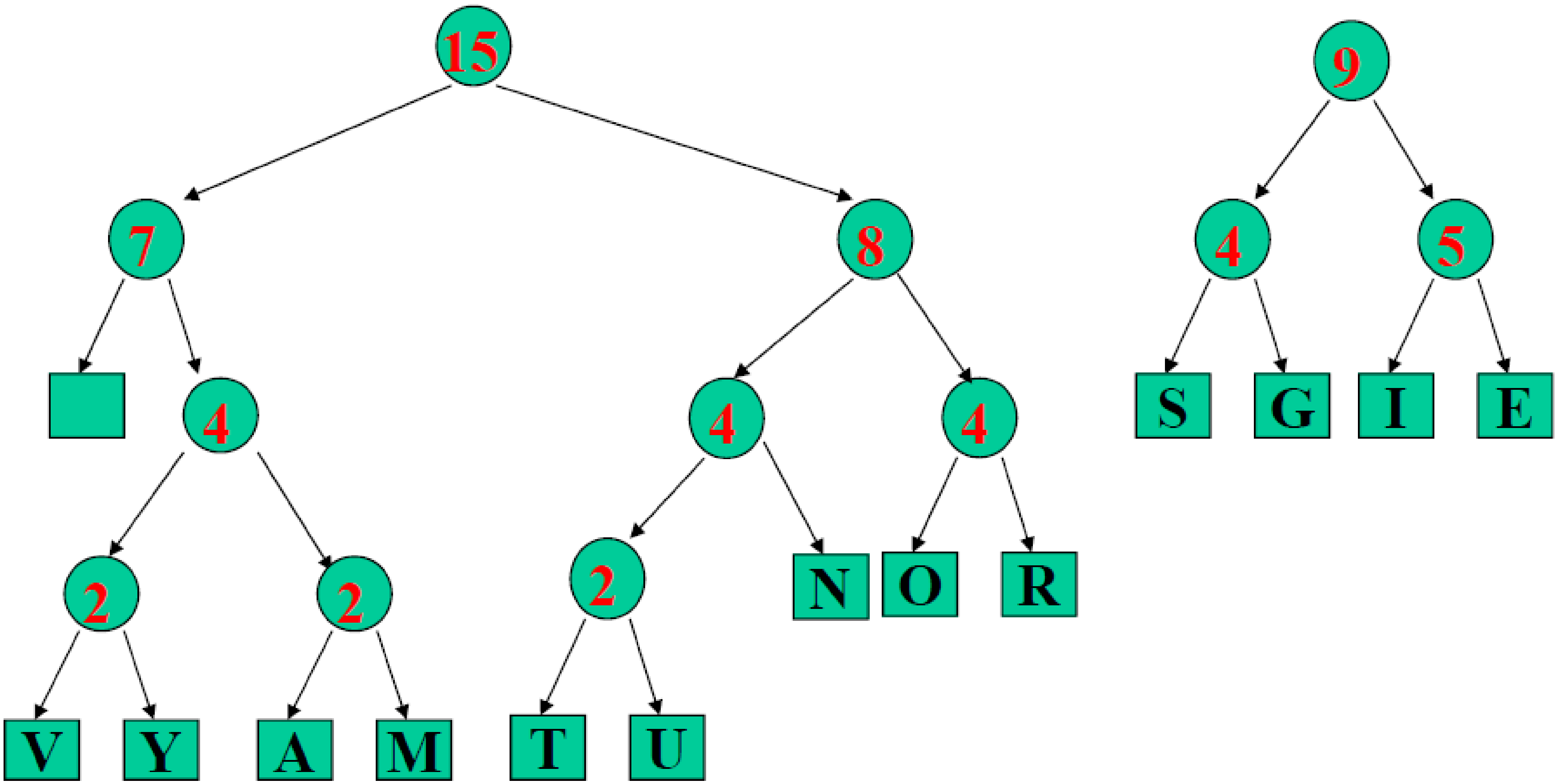
Step 8



Example of Huffman Coding

- Step 9

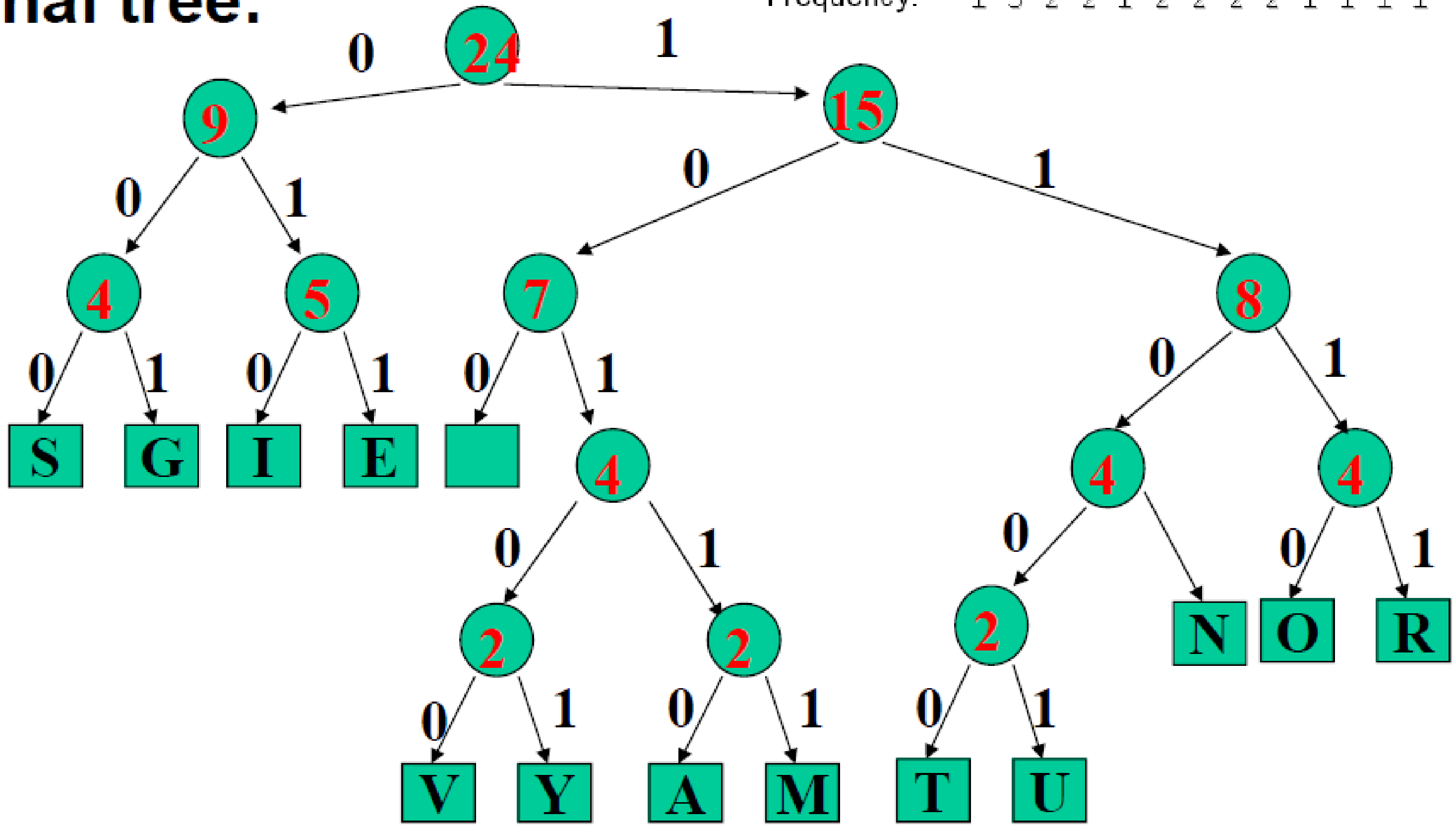
Symbol:	A	E	G	I	M	N	O	R	S	T	U	V	Y	Blank
Frequency:	1	3	2	2	1	2	2	2	2	1	1	1	1	3



Example of Huffman Coding

Symbol:	A	E	G	I	M	N	O	R	S	T	U	V	Y	Blank
Frequency:	1	3	2	2	1	2	2	2	2	1	1	1	1	3

Final tree:



E.g., code word for "Y" is "10101"

Sum of internal node values = total weighted pathlength of tree = $\Sigma W_i \cdot L_i$

= 4+5+9+2+2+4+7+2+4+4+8+15+24 = 90 (vs. $\Sigma W_i \cdot L_i$ = 120 in naïve 5 bit per symbol code)

PseudoCode


```
huffman(C, prob) {                           // C = chars, prob = probabilities
    for each (x in C) {
        add x to Q sorted by prob[x]         // add all to priority queue
    }
    for (i = 1 to |C| - 1) {                 // repeat until only 1 item in queue
        z = new internal tree node
        left[z]  = x = extract-min from Q // extract min probabilities
        right[z] = y = extract-min from Q
        prob[z]  = prob[x] + prob[y]         // z's probability is their sum
        insert z into Q                      // z replaces x and y
    }
    return the last element left in Q as the root
}
```

- Time complexity: $\theta(n \log n)$ // n is |C|
  - Loop of n-1 iterations
  - Heap operations: $\theta(\log n)$ // heapify

Lecture 9 summary

- Greedy
  - P1: Dijkstra’s shortest path
  - P2: Huffman’s min weight prefix tree