CSCI 136
Data Structures & Advanced Programming

Lecture 23

Spring 2018

Profs Bill & Jon

Last Time

- Binary Tree Traversals
- Binary Tree Iterators
- Array representation of trees
    - Node i's children: 2i+1, 2i+2
    - Node i's parent: (i-1)/2
    - Good for full or complete trees
    - Wasted space if tree is sparse or unbalanced

Today

- Breadth-First and Depth-First Search
- Application: Huffman Encoding
- Priority Queues
- Heaps

Tree Traversals

Recall from last class:

- In-order: "left, node, right"
- Pre-order: "node, left, right"
- Post-order: "left, right, node"

Stack

- Level-order: visit all nodes at depth i before depth i+1

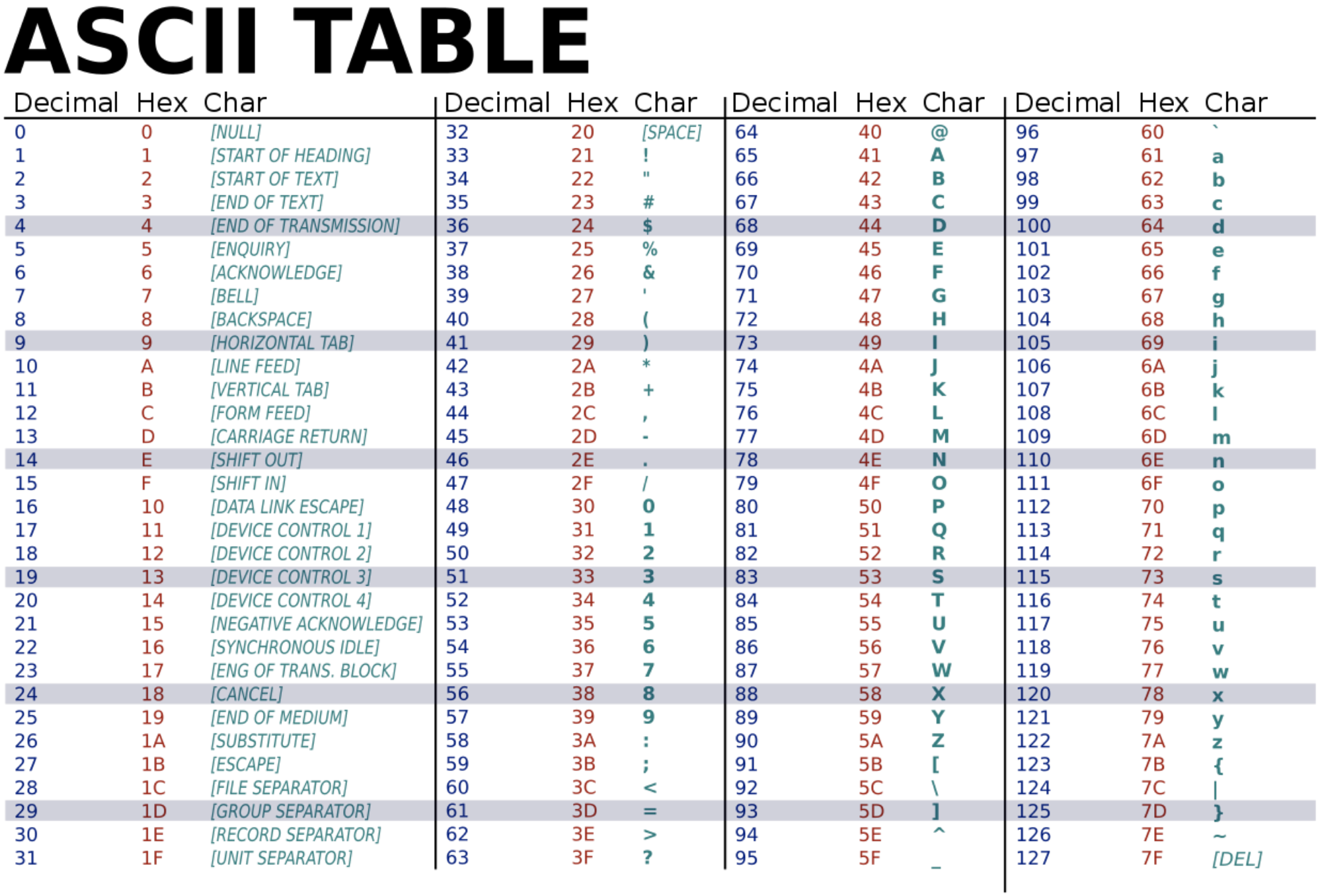
Queue

Traversals & Searching

- We can use traversals for searching unordered trees
- How might we search a tree for a value?
  - Breadth-First: Explore nodes near the root before nodes far away (level order traversal)
    - Find the nearest gas station
  - Depth-First: Explore nodes deep in the tree first (post-order traversal)
    - Solution to a maze
      - Go as far as you can until you hit a dead end, then choose a different branch (Maze video)

Next up: Huffman Codes

- Computers encode a text as a sequence of bits



ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

https://commons.wikimedia.org/wiki/File:ASCII-Table-wide.svg

Huffman Codes

- In ASCII: 1 character = 8 bits (1 byte)
    - Allows for 2^8 = 256 different characters
- 'A' = 01000001, 'B' = 01000010
- Space to store "AN_ANTARCTIC_PENGUIN"
    - 20 characters -> 20*8 bits = 160 bits

- Is there a better way?
    - Only 11 symbols are used (ANTRCIPEGU_)
    - "ASCII-lite" only needs 4 bits per symbol (since $2^4>11$)!
        - 20*4 = 80 bits instead of 160!

- Can we still do better??

Huffman Codes

- A Huffman code is an optimal prefix code for lossless compression
  - Compression: data is converted to a format that takes up less space than the original
  - Lossless: all of the information in the original data is preserved in the compressed version
  - Prefix code: a variable-length encoding where no codeword is a prefix of another codeword
- Our goal is to take a string and represent it using the smallest number of bits we can, without losing any information about the original string.

Huffman Codes

- Example
  - AN_ANTARCTIC_PENGUIN
  - Compute letter frequencies

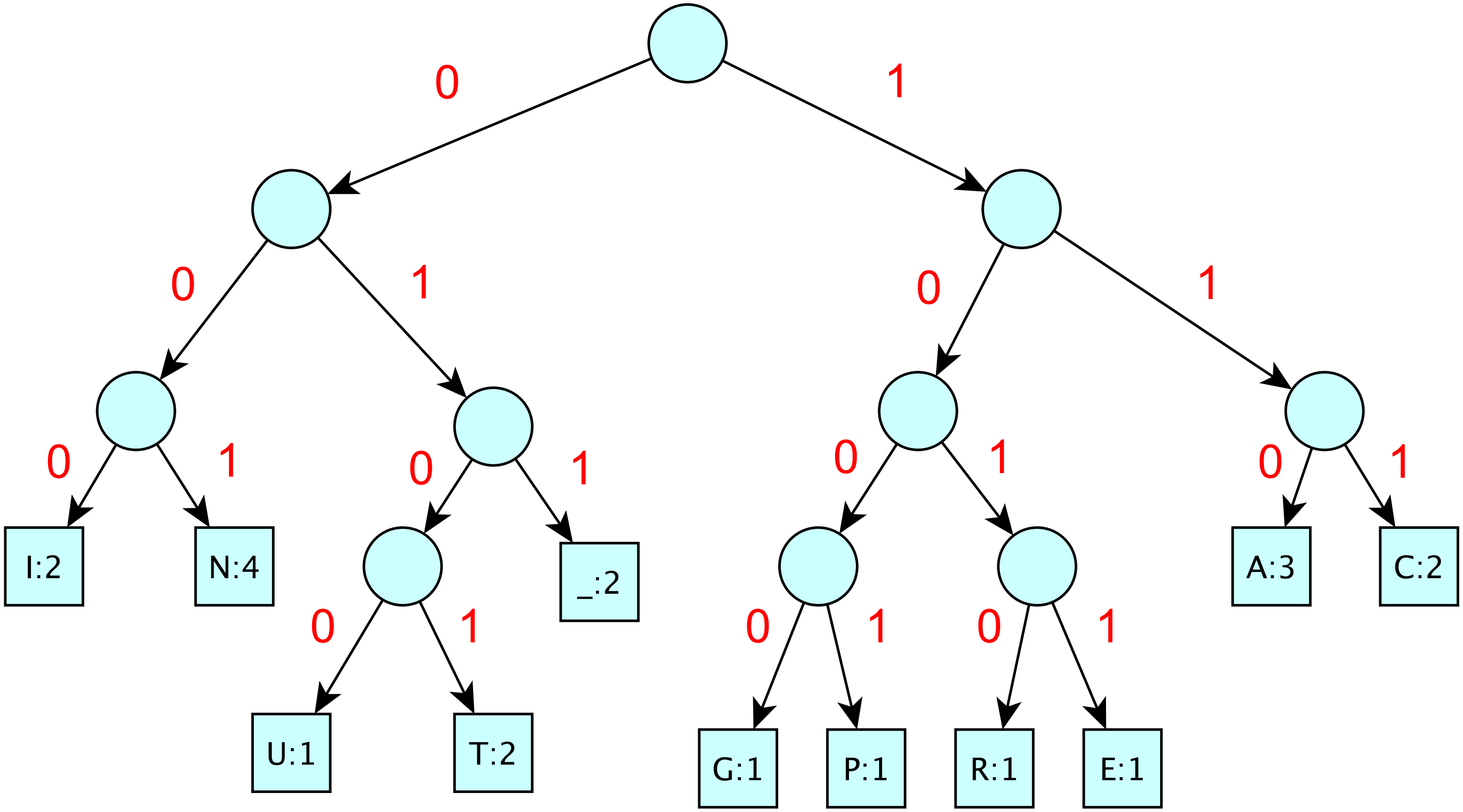
A	C	E	G	I	N	P	R	T	U	_
3	2	1	1	2	4	1	1	2	1	2

- **Key Idea:** Use fewer bits for most common letters

A	C	E	G	I	N	P	R	T	U	_
3	2	1	1	2	4	1	1	2	1	2
110	111	1011	1000	000	001	1001	1010	0101	0100	011

- Uses 67 bits to encode entire string

The Encoding Tree



Left = 0; Right = 1

Huffman Encoding Algorithm

Input: symbols of alphabet with frequencies

- Huffman encode algorithm is as follows:
  - Create a single-node tree for each symbol: key is frequency; weight is letter
  - while there is more than one tree:
    - Find two trees T_1 and T_2 with lowest weights
    - Merge them into new tree `T` with:

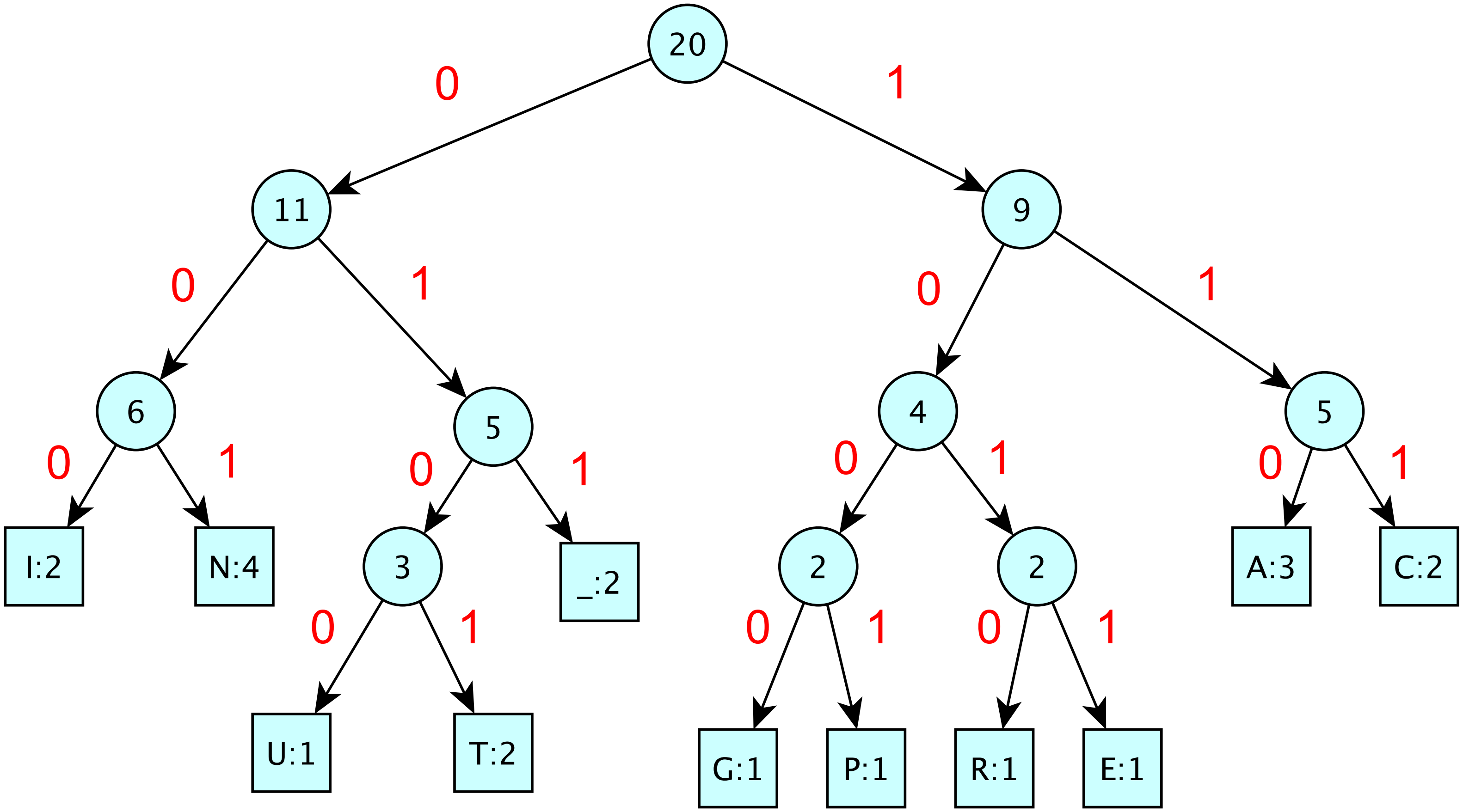
      $T.\texttt{weight} = T_1.\texttt{weight} + T_2.\texttt{weigth}$
- Theorem: The tree computed by Huffman is an optimal encoding for given frequencies

Demo

- To run the Huffman code demo found on course webpage:

```
java -jar huffman.jar
```

The Encoding Tree (With Weights)



Left = 0; Right = 1

*Each node’s value is the sum of the frequencies of all its children

Implementing the 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 while loop
- 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 $O(n^2)$!

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

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 size/weight
  - A variety of graph/network algorithms
  - To roughly rank choices that are generated out of order

An Apology

- On behalf of computer scientists everywhere, we'd like to apologize for the confusion that inevitably results from the fact that:

  Higher Priority == 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 + returns min 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 stores 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?
    - Like a normal `Vector`, but no `add(int i)`
        - Instead, `add(Object o)` places `o` at proper location according to the ordering of all objects in the `Vector`
    - `O(n)` to add/remove from vector
    - Details in book…
    - Can we do better than `O(n)`?
- Heap!
    - Partially ordered binary tree