CSCI 136
Data Structures & Advanced Programming

Lecture 21

Fall 2017

Instructor: Bills

Administrative Details

- Lab 8 today!
  - No partners this week
  - Review before lab; come to lab with design doc
    - Read over the supplied resources!

Last Time

- Trees with more than 2 children
  - Representations
  - Application: Lab 8: Hex-a-pawn!
- Binary Trees
  - Traversals
    - As methods taking a BinaryTree parameter
    - With Iterators

Today

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

Lexicon Lab Tips

Tasks (in order of implementation!):

- Review all lab materials (including .java files!)
- Implement LexiconNode
  - Add a single method, then test it: add main()
- Implement LexiconTrie
  - Same approach, but can also use Main.java to test
- Implement in an order that allows immediate testing!

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

Post-Order Iterator

```
public BTPostorderIterator(BinaryTree<E> root) {
      todo = new StackList<BinaryTree<E>>();
      this.root = root;
      reset();
}
public void reset() {
      todo.clear();
      BinaryTree<E> current = root;
      while (!current.isEmpty()) {
            todo.push(current);
            if (!current.left().isEmpty())
                current = current.left();
            else
                current = current.right();
       } // Top of stack is now left-most unvisited leaf
    }
```

Post-Order Iterator

```
public E next() {
        BinaryTree<E> current = todo.pop();
        E result = current.value();
        if (!todo.isEmpty()) {
            BinaryTree<E> parent = todo.get();
            if (current == parent.left()) {
                current = parent.right();
                while (!current.isEmpty()) {
                    todo.push(current);
                    if (!current.left().isEmpty())
                         current = current.left();
                    else current = current.right();
                }
            }
        }
        return result;
}
```

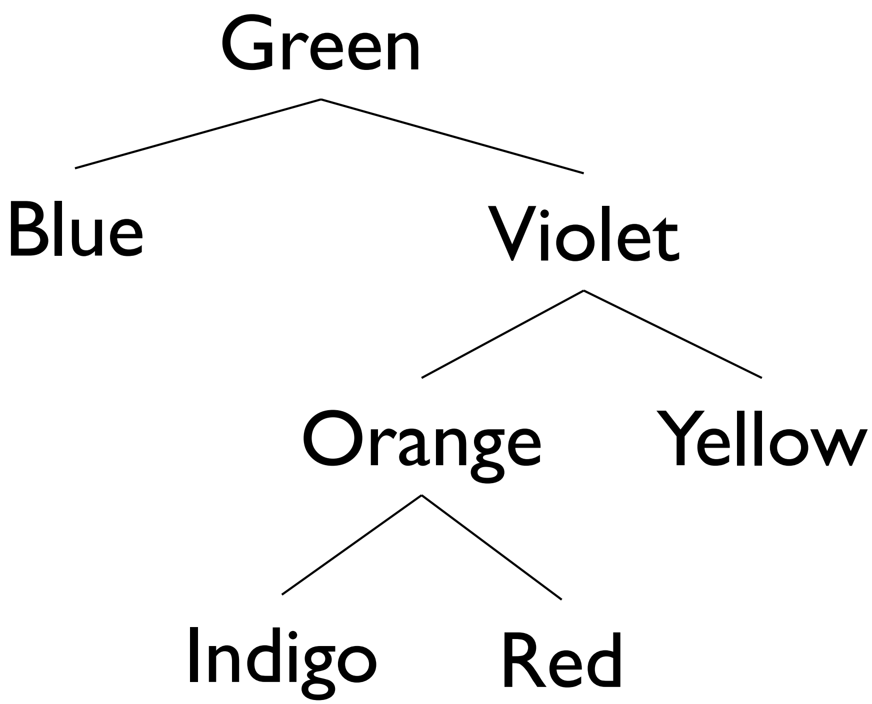
Traversals & Searching

- We can use traversals for searching trees
- How might we search a tree for a value?
  - Breadth-First: Explore nodes near the root before nodes far away (level order traversal)
    - Nearest gas station
  - Depth-First: Explore nodes deep in the tree first (post-order traversal)
    - Solution to a maze

Loose Ends – Really Big Trees!

- In some situations, the tree we need might be too big or expensive to build completely
  - Or parts of it might not be needed
- Example: Game Trees
  - Chess: you wouldn't build the entire tree, you would grow portions of it as needed (with some combination of depth/breadth first searching)

Alternative Tree Representations



- Total # "slots" = 4n
  - Since each BinaryTree maintains a reference to left, right, parent, value
- 2-4x more overhead than vector, SLL, array, …
- But trees capture successor and predecessor relationships that other data structures don't…

Array-Based Binary Trees

- Encode structure of tree in array indexes
  - Put root at index 0
- Where are children of node i?
  - Children of node i are at 2i+1 and 2i+2
  - Look at example
- Where is parent of node j?
  - Parent of node j is at (j-1)/2

ArrayTree Tradeoffs

- Why are ArrayTrees good?
  - Save space for links
  - No need for additional memory allocated/garbage collected
  - Works well for full or complete trees
    - Complete: All levels except last are full and all gaps are at right
    - "A *complete* binary tree of height h is a full binary tree with 0 or more of the rightmost leaves of level h removed"
- Why bad?
  - Could waste a lot of space
  - Tree of height of hrequires 2^{h+1}-1 array slots even if only O(h) elements

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]

Next up: Huffman Codes

- Goal: Encode a text as a sequence of bits
- Normally, use 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_)
  - Only need 4 bits per symbol (since $2^4 > 11$)!
    - 20*4 = 80 bits instead of 160!
  - Can we still do better??

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

Huffman Codes

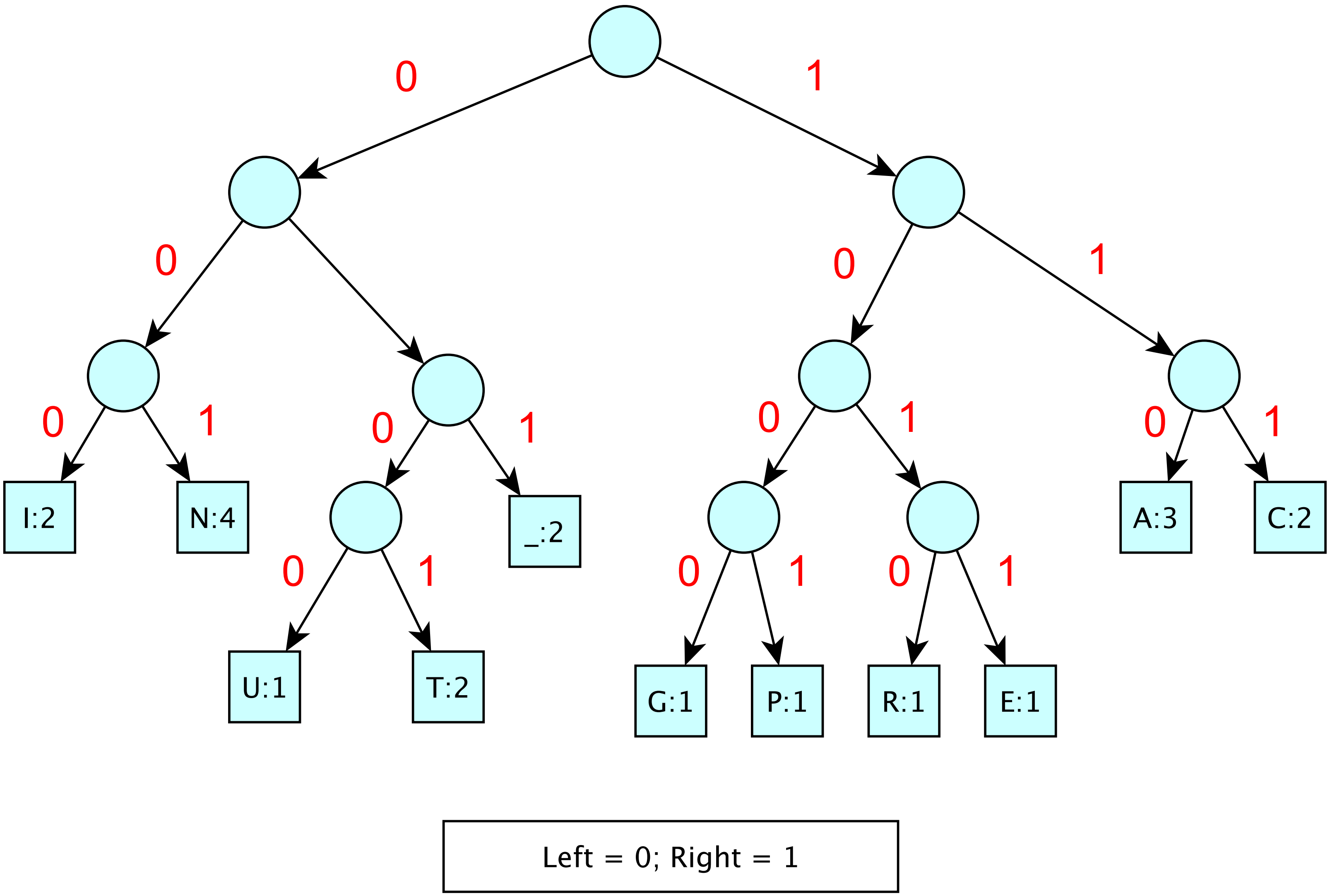
A	C	E	G	I	N	P	R	T	U	_
3	2	1	1	2	4	1	1	2	1	2
110	111	1011	1000	000	001	1001	1010	0101	0100	011

- Uses 67 bits to encode entire string
- Can we do better?

A	C	E	G	I	N	P	R	T	U	_
3	2	1	1	2	4	1	1	2	1	2
100	010	1100	1101	011	101	0001	0000	001	1110	1111

- Uses 67 bits to encode entire string

The Encoding Tree



Features of Good Encoding

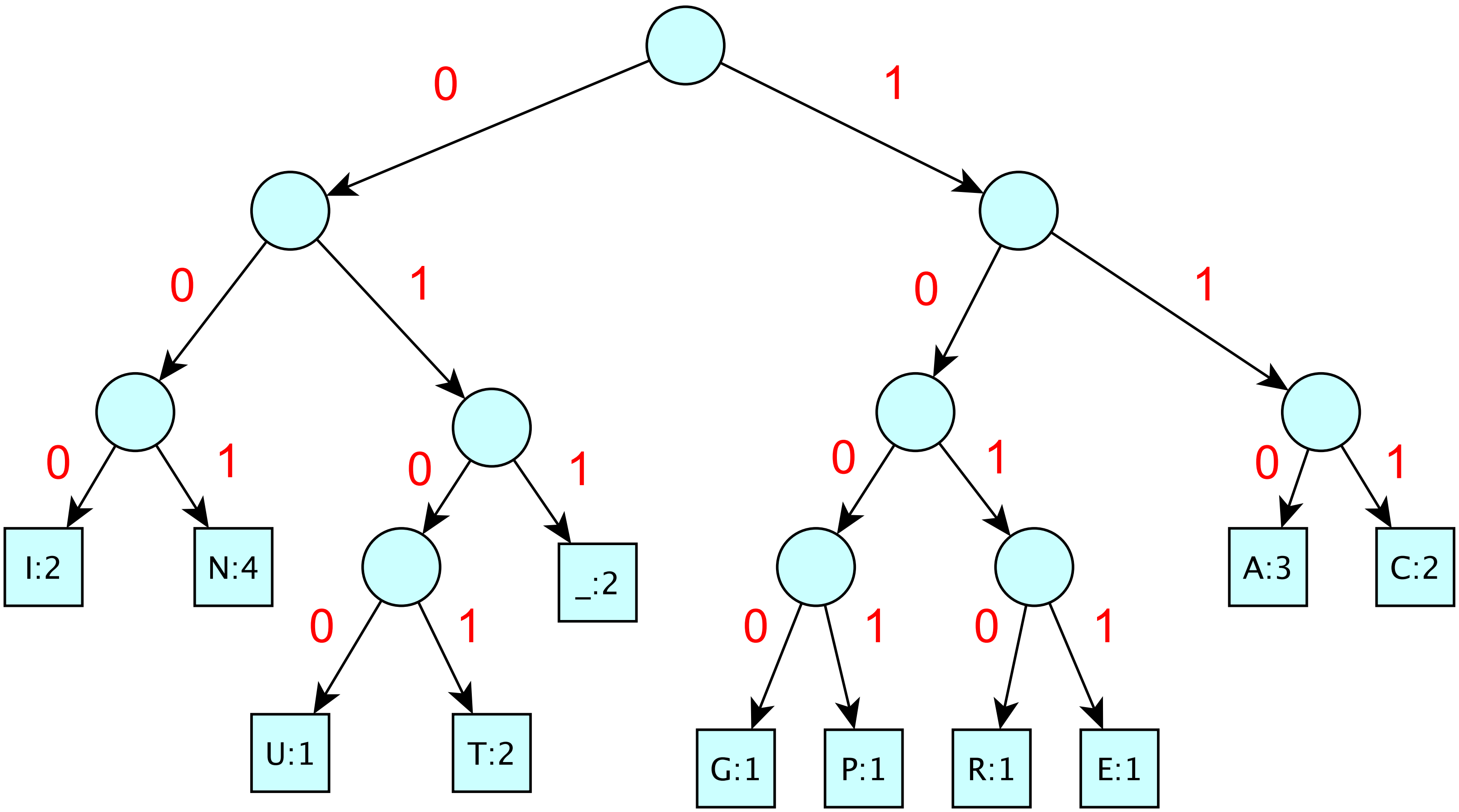
- Prefix property: No encoding is a prefix of another encoding (letters appear at leaves)
- No internal node has a single child
- Nodes with lower frequency have greater depth

- All optimal length unambiguous encodings have these features

Huffman Encoding

- Input: symbols of alphabet with frequencies
- Huffman encode as follows
  - Create a single-node tree for each symbol: key is frequency; value is letter
  - while there is more than one tree
    - Find two trees T1 and T2 with lowest keys
    - Merge them into new tree T with dummy value and key= T1.key+ T2.key
- Theorem: The tree computed by Huffman is an optimal encoding for given frequencies

The Encoding Tree



Left = 0; Right = 1

How To Implement Huffman

- 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) re-insertions of merged trees
- Can we do better...?