CSCI 136
Data Structures &
Advanced Programming

Lecture 20

Fall 2019

Instructor: B&S

Administrative Details

- Lab 7 is available online
  - No partners this week
  - Review before lab; come to lab with design doc
  - We'll give an overview shortly

Last Time

- Recursion/Induction on Trees
- Applications: Decision Trees
- Trees with more than 2 children
  - Representations
- Traversing Binary Trees
  - As methods taking a BinaryTree parameter

Today

- Binary Trees Traversals
  - With Iterators
- Big Trees
- Lab 7 Discussion
- Storing Trees in Arrays

In-Order Iterator

```
public BTInorderIterator(BinaryTree<E> root) {
      todo = new StackList<BinaryTree<E>>();
      this.root = root;
      reset();
}
 public void reset()
    {
       todo.clear();
       // stack is empty.  Push on nodes from root along
       // longest “left-only” path
       BinaryTree<E> current = root;
       while (!current.isEmpty()) {
           todo.push(current);
           current = current.left();
       }
    }
```

In-Order Iterator

```
public E next() {
       BinaryTree<E> old = todo.pop();
       E result = old.value();
       // we know this node has no unvisited left children;
       // if this node has a right child,
       //   we push right child and longest “left-only” path
       // else
       //   top element of stack is next node to be visited
       if (!old.right().isEmpty()) {
           BinaryTree<E> current = old.right();
           do {
               todo.push(current);
               current = current.left();
           } while (!current.isEmpty());
       }
       return result;
   }
```

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;
}
```

Tree Traversals

In summary:

- In-order: “left, node, right”
- Pre-order: “node, left, right”
- Post-order: “left, right, node”
- Level-order: visit all nodes at depth i before depth i+1

Stack (In-order, Pre-order, Post-order)

Queue (Level-order)

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: Search until leaves are reached
    - (post-order traversal; but halt when solution found)
    - 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)

Lab 7: Representing Numbers

- Humans usually think of numbers in base 10
- But even though we write `int x = 23;` the computer stores `x` as a sequence of `1`s and `0`s
- Recall Lab 3:

```
public static String printInBinary(int n) {
       if (n <= 1)
           return "" + n%2;

       return printInBinary(n/2)+n%2;
}
```

- 00000000 00000000 00000000 00010111

Bitwise Operations

- We can use *bitwise* operations to manipulate the 1s and 0s in the binary representation
  - Bitwise 'and': &
  - Bitwise 'or': |
- Also useful: bit shifts
  - Bit shift left: <<
  - Bit shift right: >>

& and |

- Given two integers `a` and `b`, the bitwise *or* expression `a | b` returns an integer s.t.
  - At each bit position, the result has a `1` if that bit position had a `1` in EITHER `a` OR `b` (or both)
  - `3 | 6 = ?`
- Given two integers `a` and `b`, the bitwise *and* expression `a & b` returns an integer s.t.
  - At each bit position, the result has a `1` if that bit position had a `1` in BOTH `a` AND `b`
  - `3 & 6 = ?`

>> and <<

- Given two integers a and `i`, the expression `(a << i)` returns `(a * 2`^{i}`)`
  - Why? It shifts all bits left by `i` positions
  - `1 << 4 = ?`
- Given two integers a and `i`, the expression `(a >> i)` returns `(a / 2`^{i}`)`
  - Why? It shifts all bits right by `i` positions
  - `1 >> 4 = ?`
  - `97 >> 3 = ?`   `(97 = 1100001)`
- Be careful about shifting left and “overflow”!!!

Revisiting printInBinary(int n)

- How would we rewrite a recursive `printInBinary` using bit shifts and bitwise operations?

```
public static String printInBinary(int n) {
    if (n <= 1) {
        return "" + n;
    return printInBinary(n >> 1) + (n & 1);
}
```

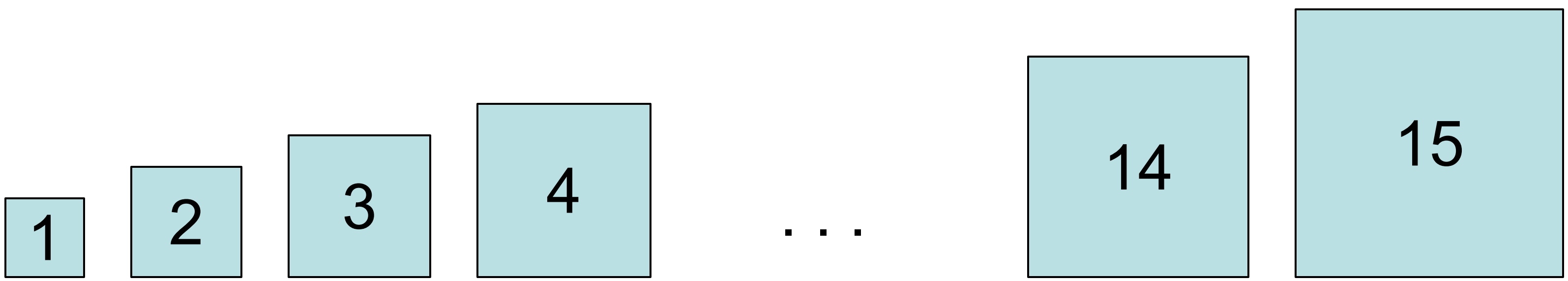
Revisiting printInBinary(int n)

- How would we write an iterative `printInBinary` using bit shifts and bitwise operations?

```
public static String printInBinary(int n,
                                    int width) {
    String result = "";
    for(int i = 0; i < width; i++)
        if ((n & (1<<i)) == 0)
            result = 0 + result;
        else
            result = 1 + result;
    return result;
}
```

Lab 7: Two Towers

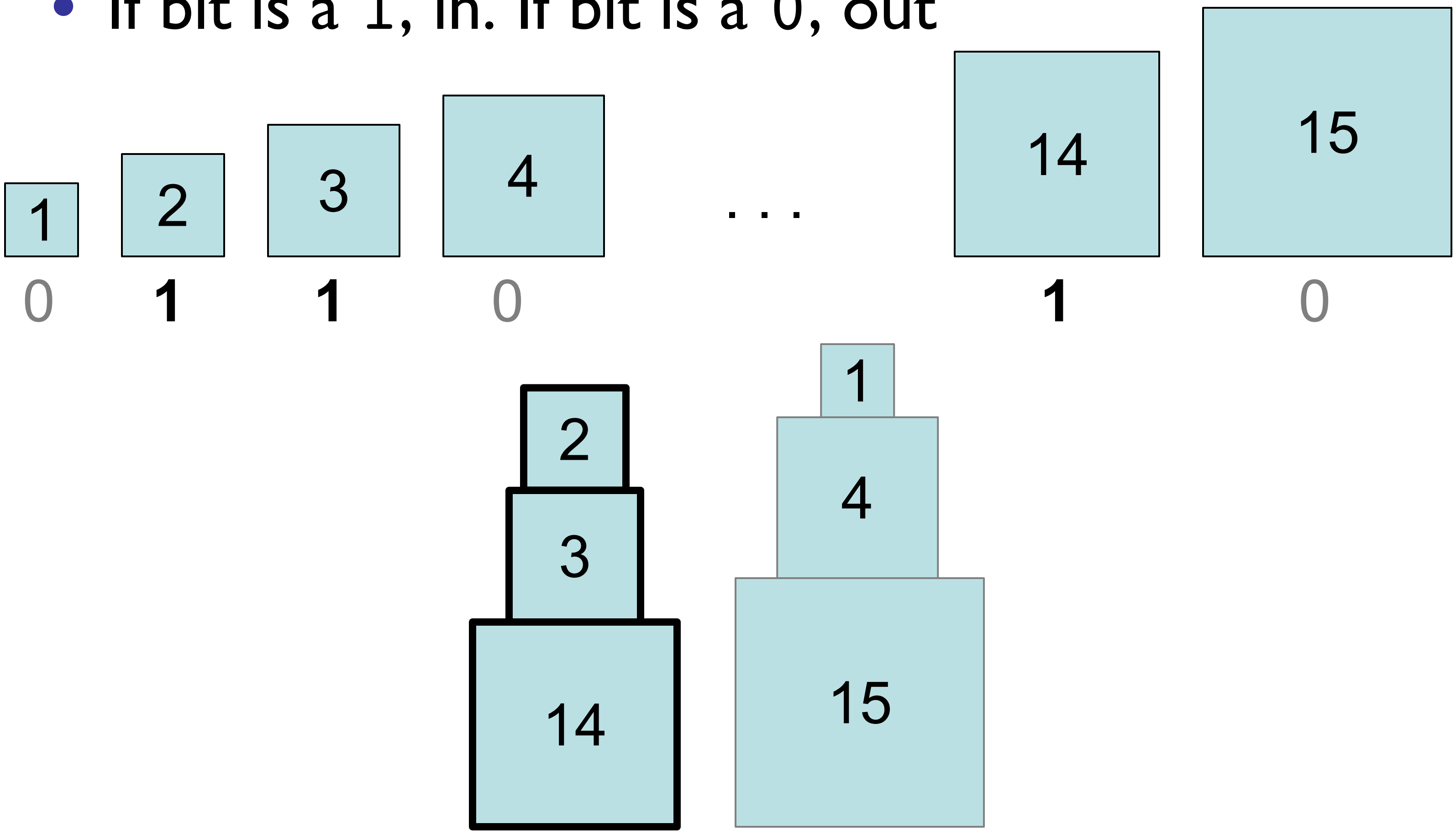
- Goal: given a set of blocks, iterate through all possible subsets to find the *best* set



- "Best" set produces the most balanced towers
- Strategy: create an iterator that uses the bits in a binary number to represent subsets

Lab 7: Two Towers

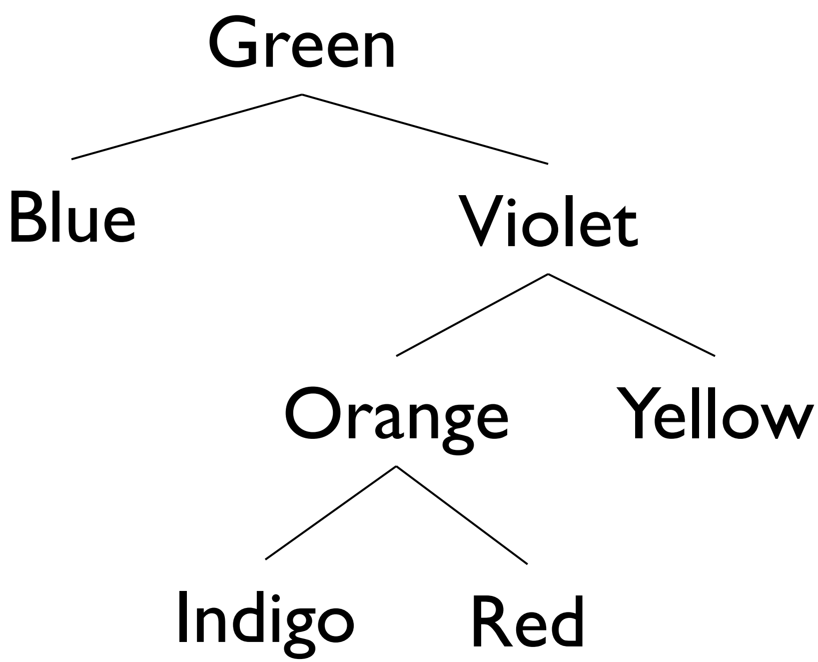
- A block can either be in the set or out
  - If bit is a 1, in. If bit is a 0, out



Questions?

- We will write a “SubsetIterator” to enumerate all possible subsets of a Vector<E>
- We will use SubsetIterator to solve this problem
- Can also be used to solve other problems
  - Identify all Subsequences of a String that are words
    - You just need a dictionary of legal words
    - Coming soon!

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 n requires $2^{n+1}-1$ array slots even if only O(n) elements

Application: Huffman Codes (a CS 256 Preview)

- 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]

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

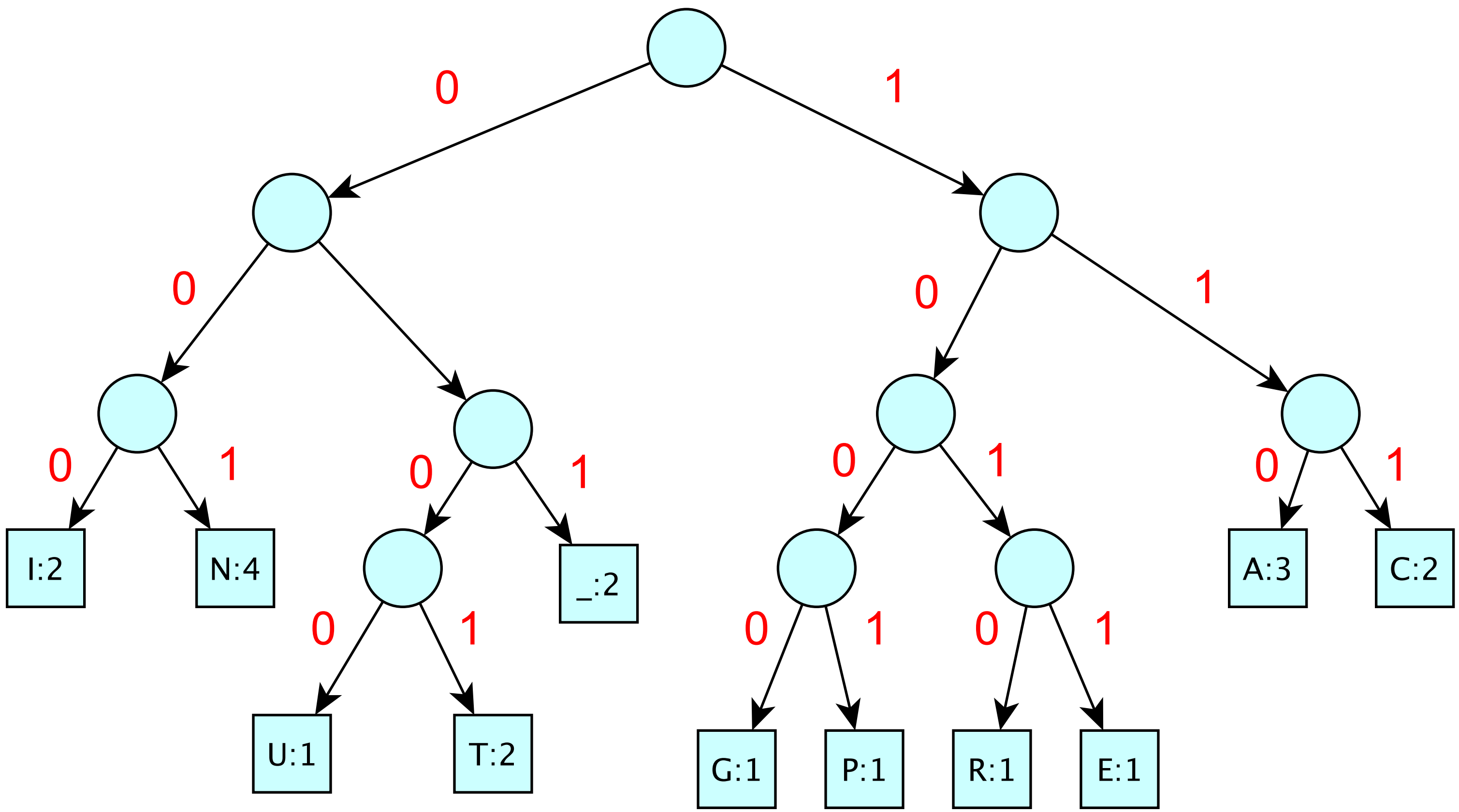
A	C	E	G	I	N	P	R	T	U	_
3	2	1	1	2	4	1	1	2	1	2
110	111	1011	1000	000	001	1001	1010	0101	0100	011

- 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



Left = 0; Right = 1

Features of Good Encoding

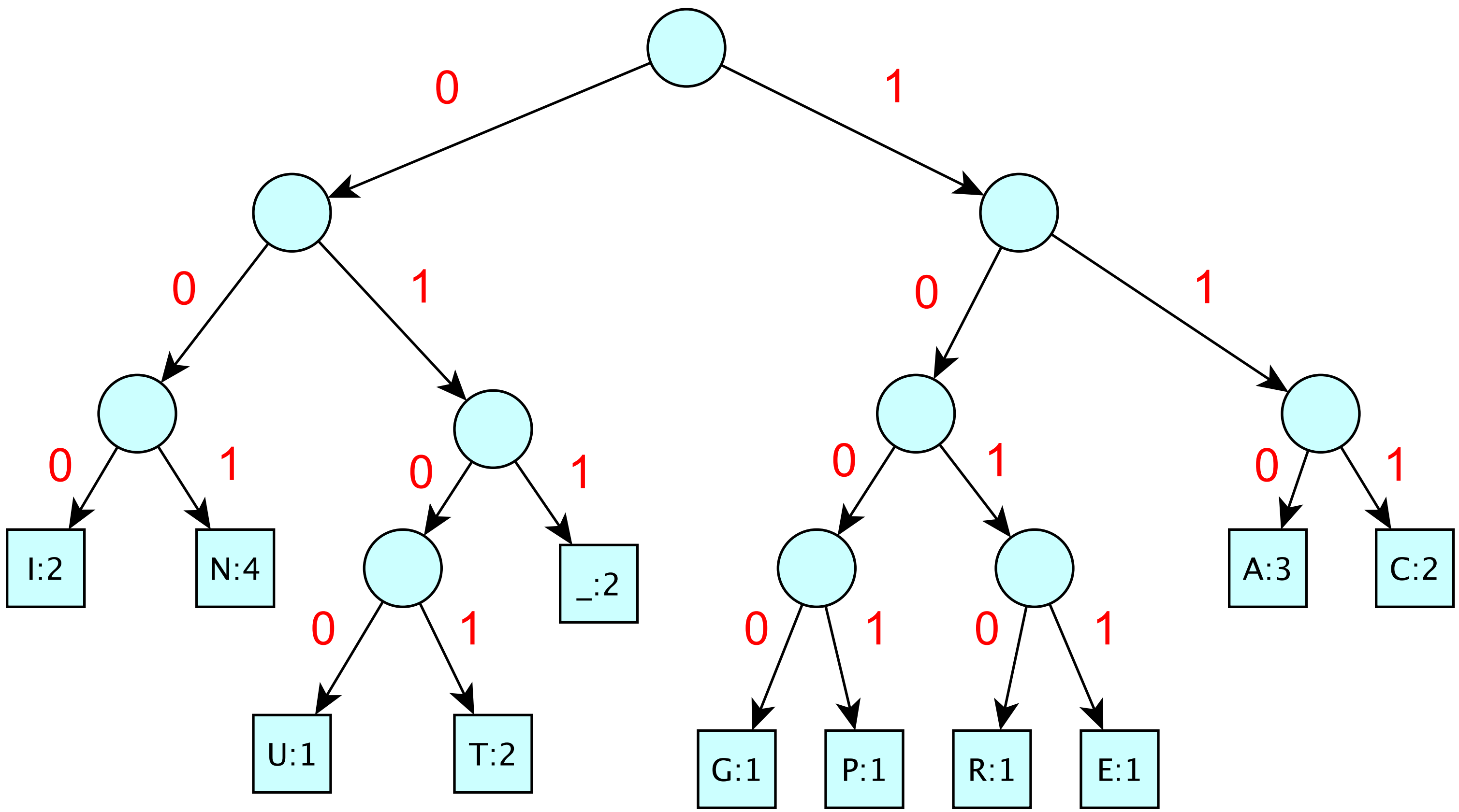
- Prefix property: No encoding is a prefix of another encoding (letters appear at leaves)
- No internal node has a single child
- Nodes with lower frequency have greater depth

- 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

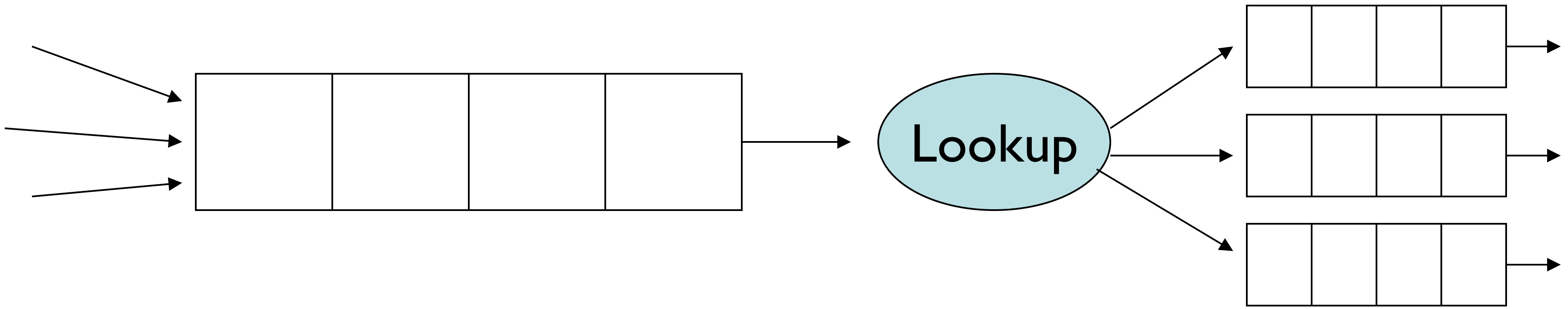
How To Implement Huffman

- 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...?

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 algorithm 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	priority = 1 (best)
bull.cs.williams.edu	2
yahoo.com	10
spammer.com	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 size/weight
  - A variety of graph/network algorithms
  - To roughly order 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   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
- A heap is a tree where:
  - 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*
  - node.value() >= node.parent.value()
    - Tree need not be binary….
- 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 **smaller** child
- This operation preserves the heap property
- Efficiency depends upon speed of
  - Finding a leaf
  - Finding locations of children
  - Height of tree