CSCI 136
Data Structures &
Advanced Programming

Lecture 22

Fall 2019

Instructor: Bill & Sam

Administration

- PS3 out
- Lab 5 back

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
- Sometimes, 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

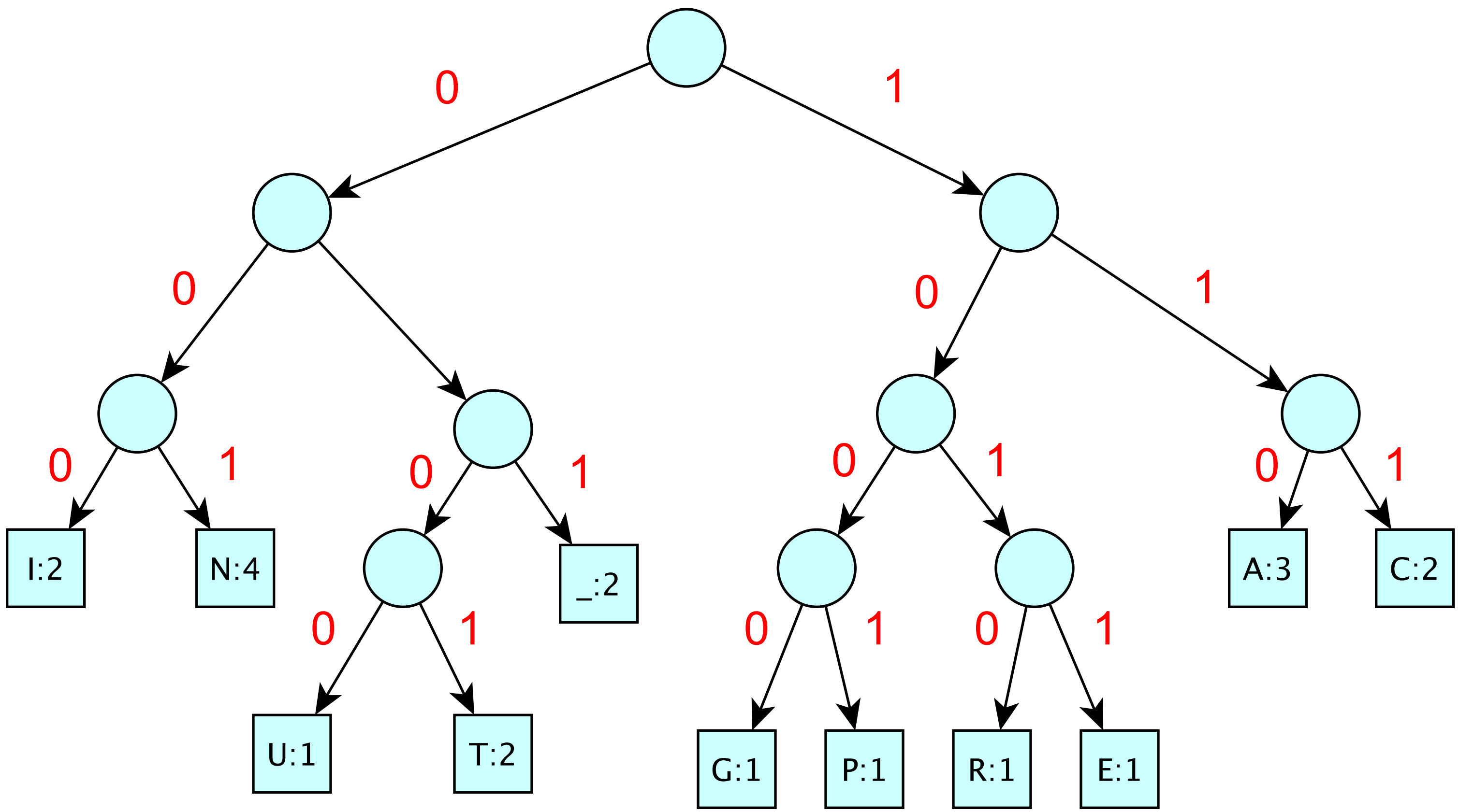
A	C	E	G	I	N	P	R	T	U	_
3	2	1	1	2	4	1	1	2	1	2

- 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

The Encoding Tree



Left = 0; Right = 1

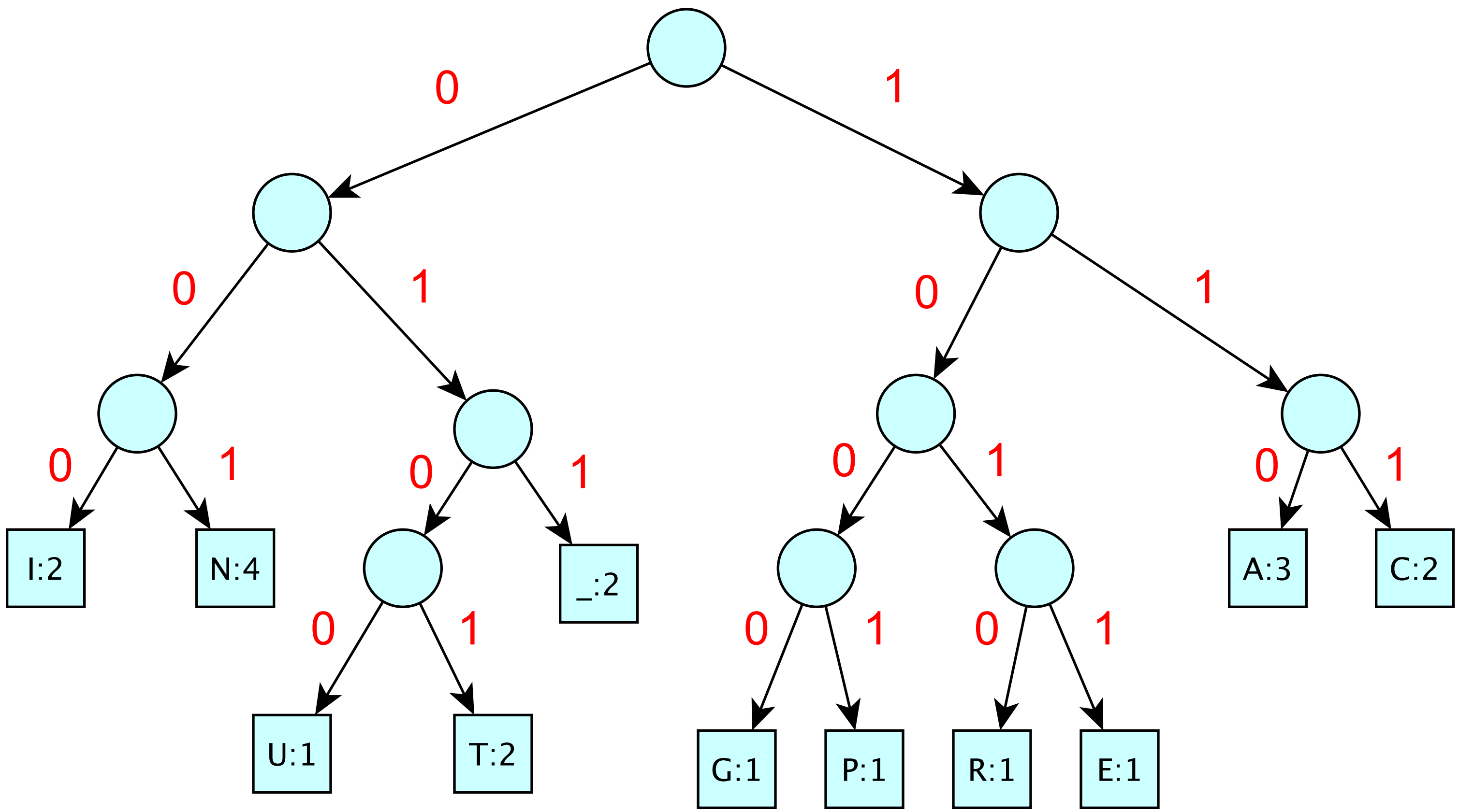
Features of Good Encoding

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

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

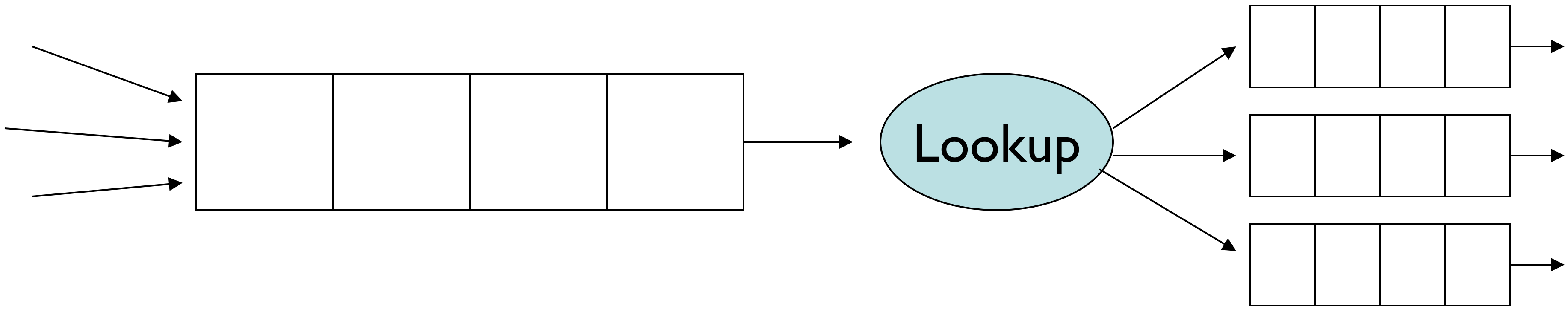
How To Implement Huffman

- 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

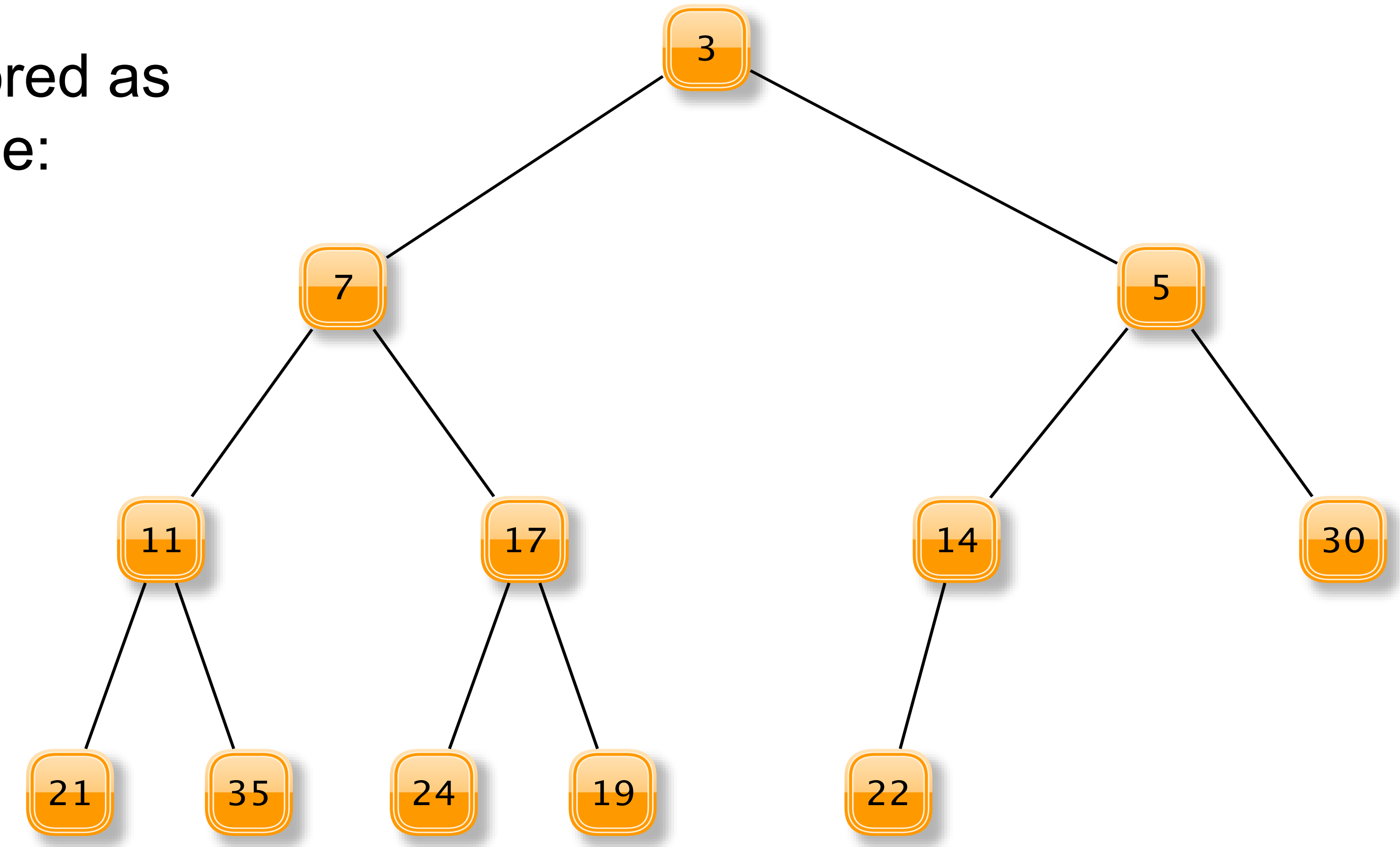
- Find a leaf, delete it, put its *data* in the root
- "Push" *data* down through the tree
  - while (*data.value* > value of (at least) one child)
    - Swap *data* with data of **smaller** child
- This operation preserves the heap property
- Efficiency depends upon speed of
  - Finding a leaf
  - Finding locations of children
  - Height of tree

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

Heap

Stored as Tree:



Stored as Vector:

3	**7**	**5**	**11**	**17**	**14**	**30**	**21**	**35**	**24**	**19**	**22**
0	1	2	3	4	5	6	7	8	9	10	11

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?

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]

Mergeable Heaps

- We now want to support the additional operation merge(heap1, heap2)
- Basic idea: heap with larger root somehow points into heap with smaller root
- Challenges
  - Points how? Where?
  - How much reheapifying is needed
  - How deep do trees get after many merges?

Skew Heap

- What if heaps are not complete BTs?
- We can implement PQs using skew heaps instead of “regular” complete heaps
- Key differences:
  - Rather than use Vector as underlying data structure, use BT
  - Need a merge operation that merges two heaps together into one heap
- Details in book

Skew Heap: Merge Pseudocode

```
SkewHeap merge(SkewHeap S, SkewHeap T)
    if either S or T is empty, return the other
    if T.minValue < S.minValue
        swap S and T          (S now has minValue)
    if S has no left subtree, T becomes left subtree
    else
        let temp point to right subtree of S
        left subtree of S becomes right subtree of S
        merge(temp, T) becomes left subtree of S
    return S
```

Tree Summary

- Trees
  - Express hierarchical relationships
  - Tree structure captures relationship
    - i.e., ancestry, game boards, decisions, etc.
- Heap
  - Partially ordered tree based on item priority
  - Node invariants: parent has higher priority than each child
  - Provides efficient PriorityQueue implementation

Improving on OrderedVector

- The OrderedVector class provides O(log n) time searching for a group of n comparable objects
  - add() and remove(), though, take O(n) time in the worst case---and on average!
- Can we improve on those running times without sacrificing the O(log n) search time?
- Let’s find out....

Binary Trees and Orders

- Binary trees impose multiple orderings on their elements (pre-/in-/post-/level-orders)
- In particular, in-order traversal suggests a natural way to hold comparable items
  - For each node v in tree
    - All values in left subtree of v are at most v
    - All values in right subtree of v are at least v
- This leads us to...

Binary Search Trees

- Binary search trees maintain a *total* ordering among elements
- Definition: A BST T is either:
  - Empty
  - Has root r with subtrees T_L and T_R such that
    - All nodes in T_L have smaller value than r
    - All nodes in T_R have larger value than r
    - T_L and T_R are also BSTs
- Examples

BST Observations

- The same data can be represented by many BST shapes
- Searching for a value in a BST takes time proportional to the height of the tree
  - Reminder: trees have height, nodes have depth
- Additions to a BST happen at nodes missing at least one child (*a constraint!*)
- Removing from a BST can involve *any* node

BST Operations

- BSTs will implement the OrderedStructure Interface
  - `add(E item)`
  - `contains(E item)`
  - `get(E item)`
  - `remove(E item)`
  - `iterator()`
    - This will provide an in-order traversal
- Runtime of add, contains, get, remove: O(height)
- Goal: Keep the height to O(log n)
    - Duane's BinarySearchTree class doesn't achieve this…
    - But his RedBlackSearchTree does!

Application: Dictionary

- Create a BST of ComparableAssociations
  - Order BST by key
  - Two objects are equal if keys are equal
- Example: Symbol tables (PostScript lab) are Dictionaries
  - But would only use a BST if the set of possible symbols was very large

Application: Tree Sort

- Can we sort data using a BST?
  - Yes!
- Runtime?
  - To build a tree with n elements, we do n insertions: O(n*h), where h is the maximum height attained by the tree
  - In order traversal: O(n)
  - Total runtime: O(n*h)

BST Implementation

- The BST holds the following items
  - BinaryTree root: the root of the tree
  - BinaryTree EMPTY: a static empty BinaryTree
    - To use for all empty nodes of tree
  - int count: the number of nodes in the BST
  - Comparator<E> ordering: for comparing nodes
    - Note: E must implement Comparable
- Two constructors: One takes a Comparator
  - The other creates a NaturalComparator

BST Implementation: locate

- Several methods search the tree: add, remove, contains
- We factor out common code: locate method
- *protected* locate(BinaryTree<E> *node*, E *v*)
  - Returns a BinaryTree<E> in the subtree with root *node* such that either
    - *node* has its value equal to *v*, or
    - *v* is not in this subtree and *node* is where *v* would be added as a (left or right) child
- How would we implement locate()?

BST Implementation: locate

```
BinaryTree locate(BinaryTree root, E value)
    if root's value equals value return root
    child ← child of root that should hold value
    if child is empty tree, return root
        // value not in subtree based at root
    else //keep looking
        return locate(child, value)
```