CSCI 6620 / 2226 Spring, 2016

P6: Huffman Tree

1 Phase 1: Tallying the Text

We will start with a very short message in order to keep this process simple enough to diagram:

> Morals rule everything! (Or is it money?)

The first step is to count the letter frequencies in this text. We will use a simple algorithm:

- Create an array of 256 integers initialized to 0.
- Read the file one keystroke at a time and use the ASCII code of each char to index the array.
- Increment the counter at that array location.

The non-zero results of tallying our text are shown below in ASCII sequence order:

!	(	)	?	' '	M	O	a	e	g	h	i	l	m	n	o	r	s	t	u	v	y
1	1	1	1	6	1	1	1	4	1	1	3	2	1	2	2	4	2	2	1	1	2

2 Phase 2: Building the Heap

For each non-zero-count character, make a new Node using the character and its frequency. Put all of the nodes into an array, starting with slot 1. The result for our text will be:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
	!	(	)	?	' '	M	O	a	e	g	h	i	l	m	n	o	r	s	t	u	v	y
0	1	1	1	1	6	1	1	1	4	1	1	3	2	1	2	2	4	2	2	1	1	2

Figure 1: The heap array before heapify().

This array is compact (with no holes), as a heap must be, but it is not in heap order. A heap represents a complete binary tree in which every path from the root of the tree to a leaf is in sorted order. The root is stored in subscript 1; its left son is in subscript 2 and its right son is in subscript 3. Consider the node in subscript n; its sons are in subscripts $2n$ and $2n + 1$.

Now look at Figure 1. This array is not in heap order. For example, the path from the root to the node at subscript 21 goes through nodes 1, 2, 5, 10, and 21. This path is not sorted because node 5 has a higher frequency than node 10. To make this into a legal min-heap, we execute the $heapify()$ operation, as follows:

- Since there are 22 data items in the array, position 11 is the rightmost heap node that is a parent. The nodes in positions 12 through 22 are all leaf nodes. (Leaves are colored green in Figure 2.)

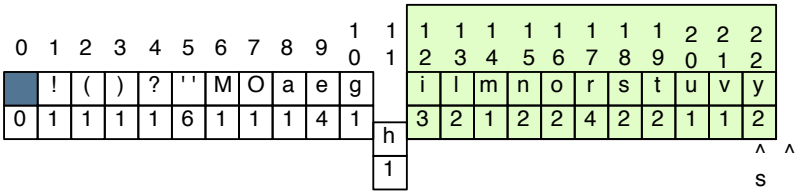

Figure 2: The first step of heapify().

To heapify, we start with $k = 11$ (the youngest parent) and call $downHeap(11)$, then $downHeap(10)$, etc. until we have finished $downHeap(1)$. Each execution of $downHeap(n)$ makes sure that all paths below n are in heap order, that is, the father node is smaller than (or equal to) both sons.

Doing the downHeap(). During the $downHeap(11)$ execution, we set $s = 22$ and $r = 23$. We note that 23 is off the end of the heap, so subscript 22 holds the last actual node. Now we compare the frequency at subscript 11 to s.frequency and find that they are in the right order already. So we are done with *downheap*(11).

Now we decrement k and execute $downHeap(10)$. We set $s = 20$ and $r = 21$ and compare s.frequency to r.frequency. Since they are the same, we do not change s. Now we compare k.frequency to s.frequency and find that they are the same, and thus, already in the right order. So we are done with *downheap*(10).

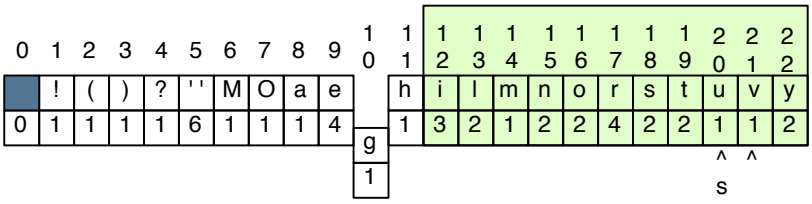

Figure 3: The second step of heapify().

Now we decrement k and execute $downHeap(9)$. We set $s = 18$ and $r = 19$ and compare s.frequency to r.frequency. Since they are the same, we do not change s. Now we compare k.frequency==4 to s.frequency==2 and find that they are in the wrong order. We swap the (e,4) with the (s,2). Since position 18 is a leaf, we are done with this pass and we decrement k.

The progress of $downHeap(8)$ and $downHeap(7)$ is almost identical to $downHeap(10)$, so we do not show the details. During $downHeap(6)$ there is one slight difference: the frequency of the right son (l, 2) is less than the frequency of the left son, so we set s to the subscript of the right son. Now we compare k.frequency to s.frequency. They are in the correct order and no swap takes place.

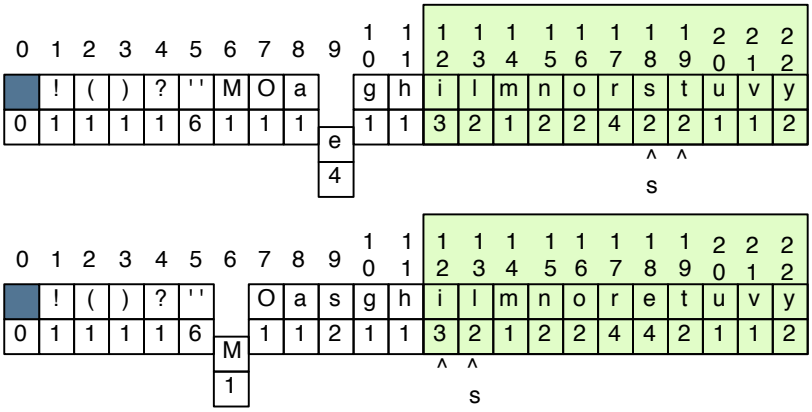

Figure 4: DownHeap(9) through downHeap(6).

When we execute $downHeap(5)$, new things start to happen. Subscript 5 is the first grandfather node in the array. We set $s = 10$ and $r = 11$ and compare s.frequency to r.frequency. Since they are the same, we do not change s. Now we compare k.frequency==6 to s.frequency==1 and find

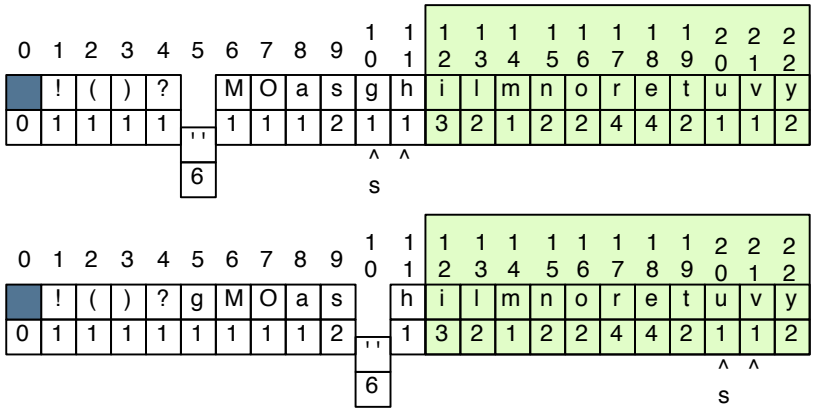

Figure 5: A 2-step operation at downHeap(5).

that they are in the wrong order. We swap the (space,6) with the (g,1). Now, unlike previous steps, we swapped into a position that is NOT a leaf, so the downHeap operation must continue. We set $k = 10$, $s = 20$ and $r = 21$ and compare s.frequency to r.frequency. Since they are the same, we do not change s. Now we compare k.frequency==6 to s.frequency==1 and find that they are in the wrong order. (See Figure 5) We swap the (space,6) with the (u,1). Since position 20 is a leaf, we are done with this pass. (See Figure 6)

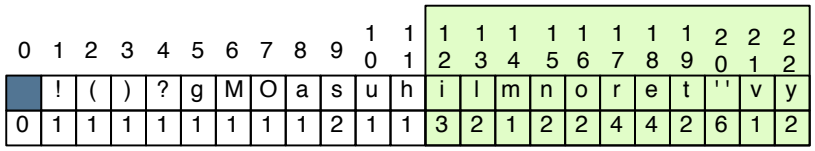

Figure 6: The result of heapify(): a legal heap.

All remaining passes terminate quickly, since the frequencies in slots 1 through 4 are all the lowest possible. No further swaps take place, and Figure 6 shows the final positions of all items in the heap.

3 Phase 3: Using the Heap

We have finished arranging the heap nodes so that every chain from root to leaf goes through nodes with increasing frequency. Now we are ready to build a Huffman tree.

- Remove (pop) the first Node from the heap. Figure 7 illustrates each of the steps of this algorithm. First, the top item (an exclamation point, frequency 1) is removed. Then the last item in the array (y, 2) is moved to slot 1. The diagrams show the progress of the *downHeap*() operation that moves the (y, 2) to its proper place in the tree. This job required 4 iterations – the maximum possible with a tree of this size.
  - In the top row, we see that positions 2 and 3 are the sons of position 1, and s is set to position 2 because position 3 has the same frequency. So we compare the frequencies in positions 1 and 2 and see that a swap is necessary.
  - The second row shows that (y, 2) has moved to position 2 and is ready to compare with position 4. Again, a swap is necessary.

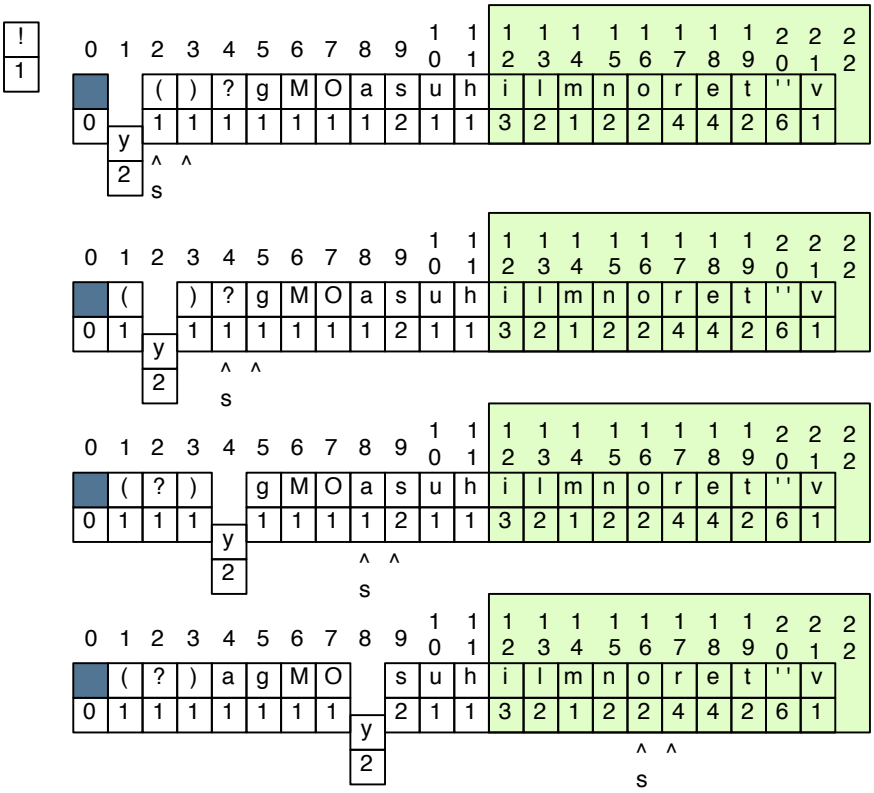

Figure 7: Removing the node with highest frequency.

  - The third row shows the (y, 2) in position 4, ready to compare to its sons. We compare y's frequency to position 8's frequency and see that a swap is necessary.
  - Row 4 shows the (y, 2) in position 8 ready to compare to its smaller son in postition 16. No swap is necessary this time, and we are done because position 16 is a leaf.

- Remove a second Node from the heap. The second node is a parenthesis with frequency 1. We replace it by the (v, 1) at the end of the heap and do a *downHeap*(1) operation. This terminates right away because the frequencies of both sons are no greater than the frequency of (v, 1). The result is shown in Figure 9.

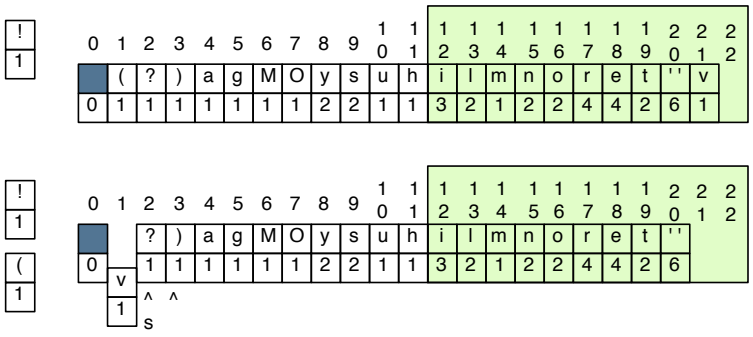

Figure 8: Deleting the second item.

- Create a new Node whose left and right sons are the two Nodes you removed and add the new Node to the heap. The frequency of this node equals the sum of the frequencies of the two sons. The diagram shows the heap immediately after the new node has been stored

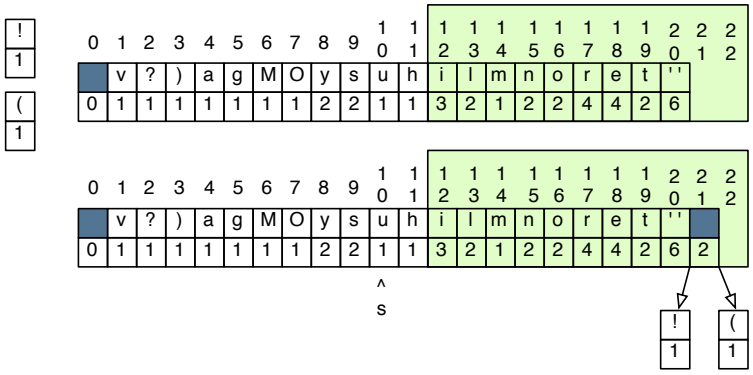

Figure 9: Putting the first combined Node into the heap.

there, and before the *upHeap*(21) operation. During *upHeap*(), the frequency of the parent node, position 10, is compared to the frequency of the new node in position 21. They are in the right order so no swap takes place. Figure 9 shows the final position of the data after finishing the *upHeap*(21).

- Repeat this process (pop, pop, push) until there is only one node left on the tree. On each iteration, the heap is shortened by 1 position. We diagram the heap during the second iteration, after deleting the first node, deleting the second, and finally inserting the combined node:

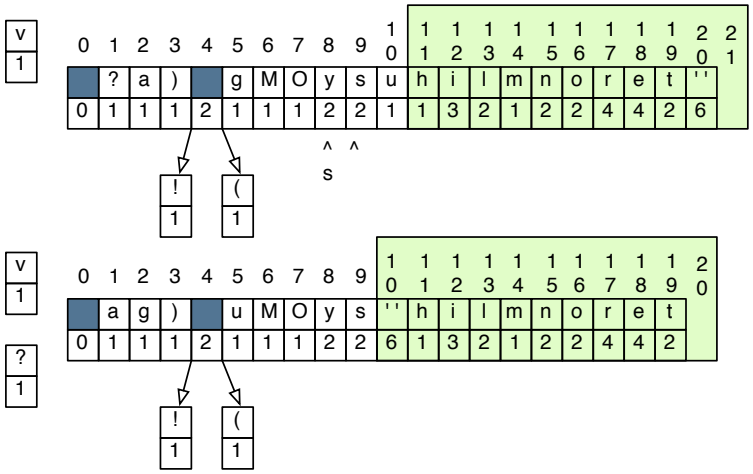

Figure 10: Removing two more nodes.

- After many iterations of pop-pop-push, the final node left in the heap is a Huffman code tree.

4 Future Work: Creating the Huffman Code

The properties of this Huffman tree are important: the most frequent characters are higher in the tree than the less frequent letters, and overall, the way the tree is constructed minimizes the number of bits needed to encode each letter.

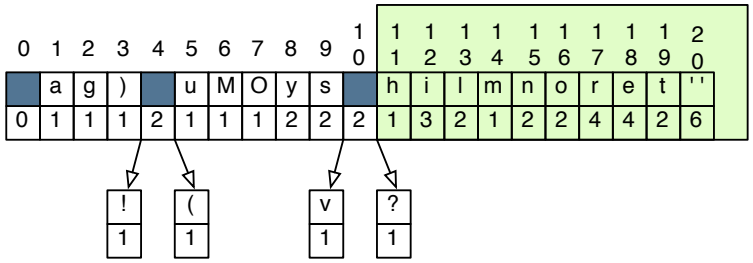

Figure 11: After adding the second combined Node to the heap.

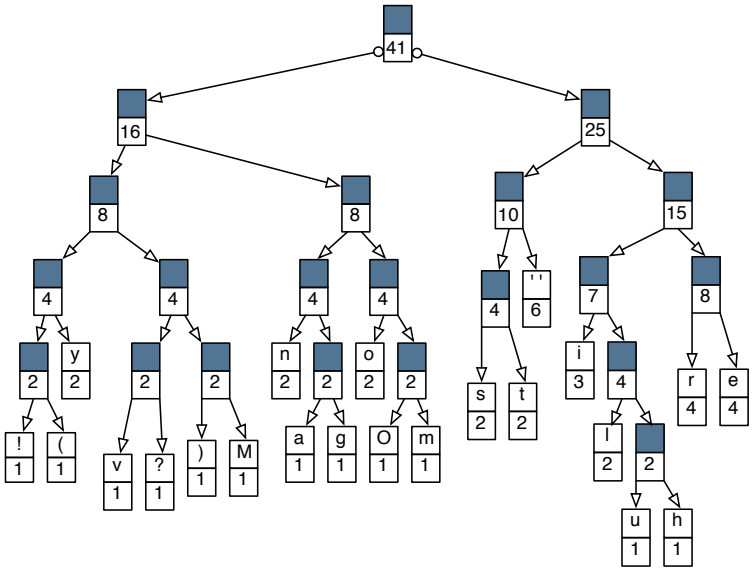

Figure 12: After all heap nodes have been combined into a single tree.

We create a Huffman code by traversing the tree. Every left-branch corresponds to a 0 in the code, and every right-branch corresponds to a 1. We traverse the tree in depth-first order, writing down the 0's and 1's as we go. When we reach a leaf node (which contains a letter) the current string of 0's and 1's becomes the code for that letter. This kind of code has some very special properties:

- It is a variable length code – common letters have shorter codes than uncommon ones.
- No code is a prefix of the code for any other letter. This enables us to decode messages.

!	5	0	0	0	0	0	O	5	0	1	1	1	0	
(	5	0	0	0	0	1	m	5	0	1	1	1	1	
y	4	0	0	0	1		s	4	1	0	0	0		
v	5	0	0	1	0	0	t	4	1	0	0	1		
?	5	0	0	1	0	1	space	3	1	0	1			
)	5	0	0	1	1	0	i	4	1	1	0	0		
M	5	0	0	1	1	1	l	5	1	1	0	1	0	
n	4	0	1	0	0		u	6	1	1	0	1	1	0
a	5	0	1	0	1	0	h	6	1	1	0	1	1	1
g	5	0	1	0	1	1	r	4	1	1	1	0		
o	4	0	1	1	0		e	4	1	1	1	1		