CS 620 / 226 November 28, 2011

Data Structures Making a Huffman Code

1 Program 7, 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 initialize 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 Program 7, Phase 2: Building the Heap

For each non-zero-count character, make a new Huffman 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().

Next, we must $heapify()$ this array. Since there are 22 data items in the array, position 11 is the rightmost heap node that is a parent. Everything after that is a leaf node (colored green in the diagram.) So we set $k = 11$ and call $downHeap(11)$.

During the $downHeap(11)$ execution, we set $s = 22$ and $r = 23$. We do not try to compare the priorities of the sons, since there is only one son. Now we compare k.priority to s.priority and find that they are in the right order already. So we are done with $downheap(11)$.

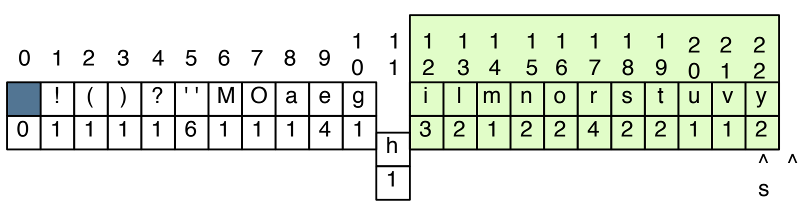

Figure 2: The first step of heapify().

Now we decrement k and execute $downHeap(10)$. We set $s = 20$ and $r = 21$ and compare s.priority to r.priority. Since they are the same, we do not change s. Now we compare k.priority==1 to s.priority==1 and find that they are in the right order already. So we are done with $downheap(10)$.

Now we decrement k and execute $downHeap(9)$. We set $s = 18$ and $r = 19$ and compare s.priority to r.priority. Since they are the same, we do not change s. Now we compare k.priority==4

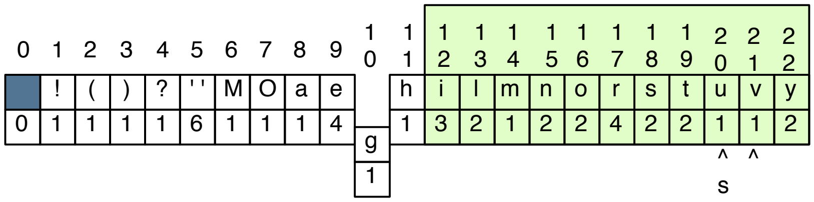

Figure 3: The second step of heapify().

to s.priority==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 priority of the right son (l, 2) is higher than the priority of the left son, so s is moved and we compare the priority of the right son to that of the parent. They are in the correct order and no swap takes place.

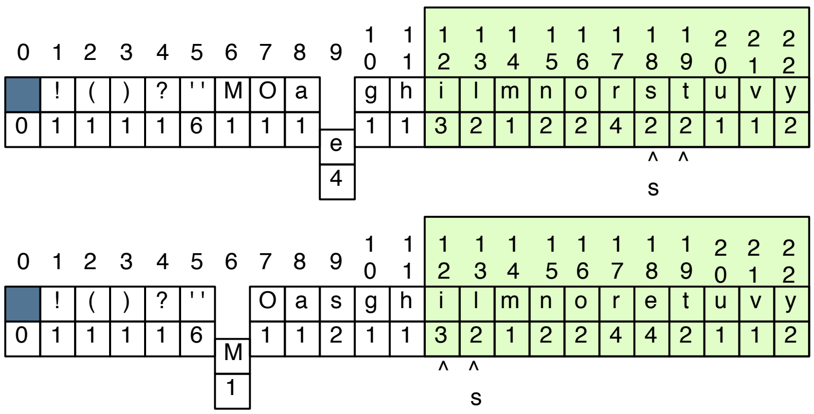

Figure 4: DownHeap(9) through downHeap(6).

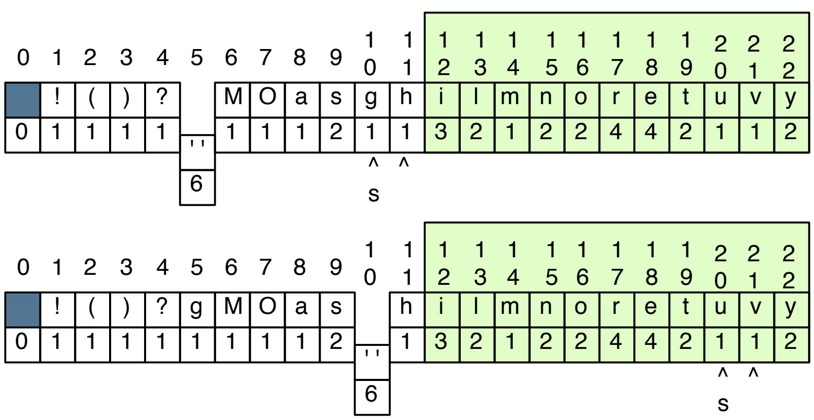

Figure 5: A 2-step operation at downHeap(5).

When we execute $downHeap(5)$, things start to happen. We set $s = 10$ and $r = 11$ and compare s.priority to r.priority. Since they are the same, we do not change s. Now we compare k.priority==6 to s.priority==1 and find 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 continues. We set $s = 20$ and $r = 21$ and compare s.priority to r.priority.

Since they are the same, we do not change s. Now we compare k.priority==6 to s.priority==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)

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

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

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

3 Program 7, Phase 3: Using the Heap

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

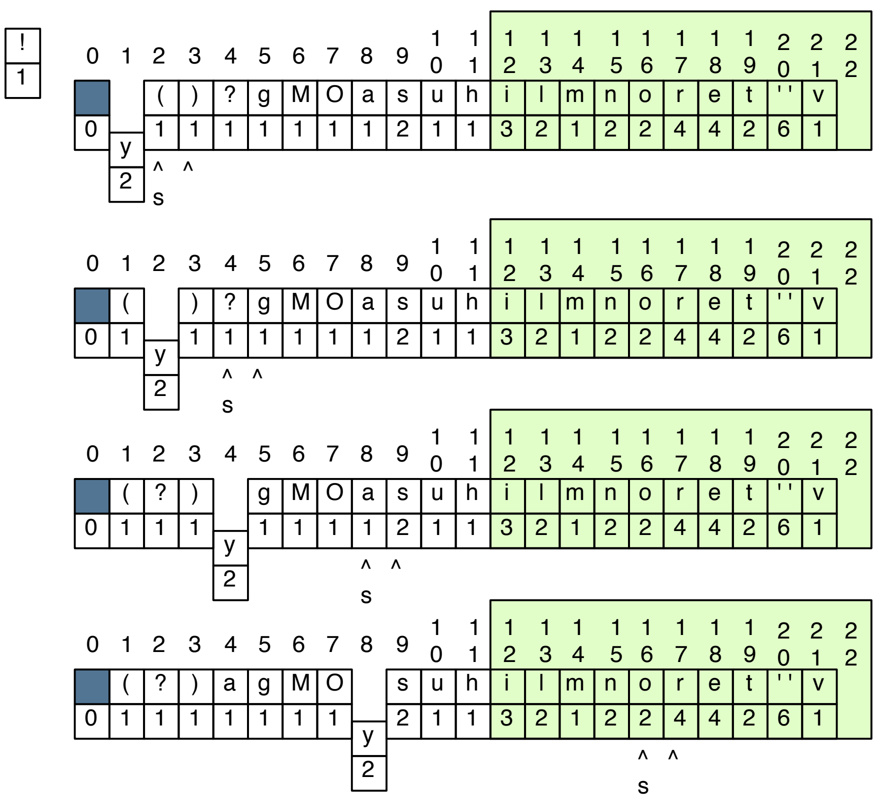

Figure 7: Removing the node with highest priority.

- Remove 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 passes – 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 does not have greater priority. So we compare the priorities 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.
- The third row shows the (y, 2) in position 4, ready to compare to its sons. We compare y's priority to position 9's priority and see that a swap is necessary.
- Row 4 shows the (y, 2) in position 8 ready to compare to its better son in postition 16. No swap is necessary this time, and we are done because position 16 is a leaf. Figure 8 shows the array after this swap takes place.

- 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 priorities of both sons are no greater than the priority of (v, 1). The result is shown in Figure 9.

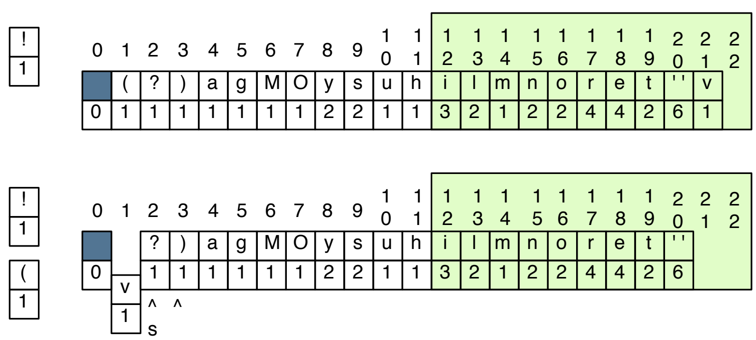

Figure 8: Deleting the second item.

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

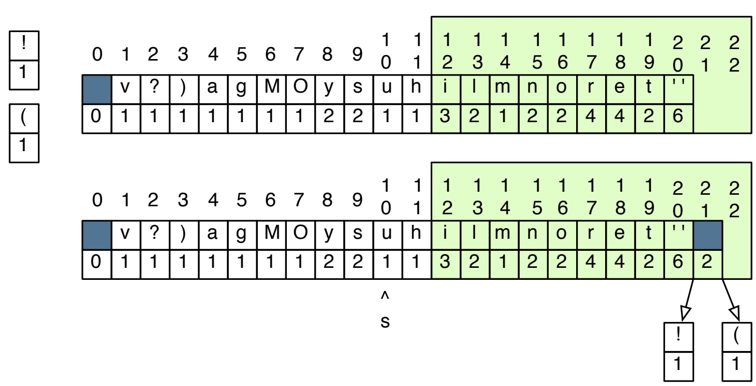

Figure 9: Putting the first combined Node into the heap.

and before the *upHeap*(21) operation. During *upHeap*(), the priority of the parent node, position 10, is compared to the priority 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 (remove, remove, add) until there is only one node left on the tree. Each time we do the process, 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:

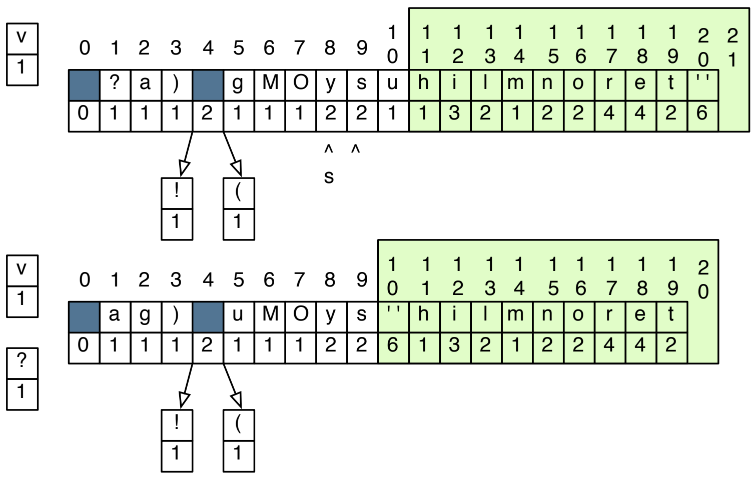

Figure 10: Removing two more nodes.

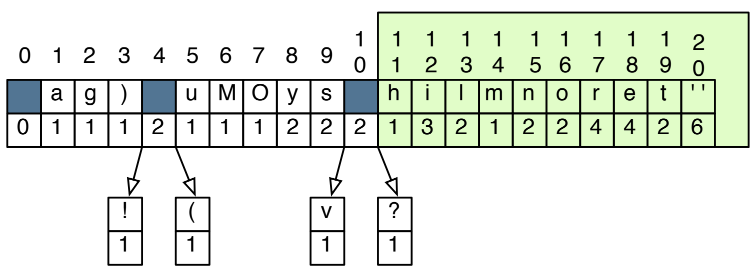

Figure 11: After adding the second combined Node to the heap.

- The final node left in the heap is a Huffman code tree. Do a recursive tree-walk to print the tree, then return the pointer to the tree for use in the next phase of the program.

4 Extra Credit: Phase 4: Creating the Huffman Code

We can create a 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. The table below shows the codes that would be generated from the tree above, in order of generation.

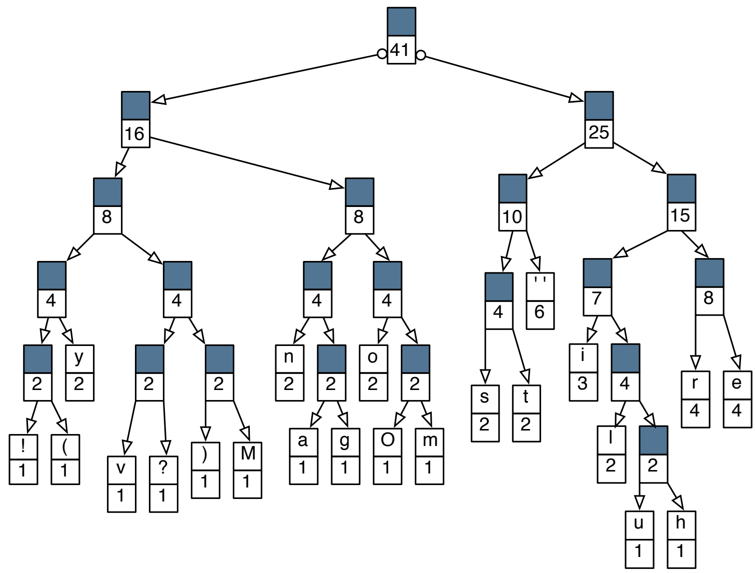

Figure 12: After adding the second combined Node to the heap.

!	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		

This kind of code has some very special properties:

- It is a variable length code – common letters have shorter representations than uncommon ones.
- No code is a prefix of the code for any other letter. This enables us to decode messages.
- When a message is encoded, the bits are packed into bytes. Every bit counts. The last byte is padded with 0 bits, if necessary.