

---

MX4002 ALGORITHMS  
SHEET 3: ABSTRACT DATA TYPES

---

1. queue-as-stack Show how a queue can be implemented using two stacks. How many different stack operations are needed to implement each `add` or `next` operation of the queue?
2. queue-as-array Show how to implement a queue using an array. Recall the command `i rem N`, which returns the remainder when  $i$  is divided by  $N$ .
3. three-flags Assume now that at least half the tokens in the 'three coloured flags' array are red. Describe an algorithm which uses fewer calls to the `swap` routine.
4. flattening The operation of "flattening" a list is the removal of any sublist structure, while keeping all the elements. This the flattening of `[A [B C [[D] E] F] G]` is just `[A B C D E F G]`. Show that the flattening of the list representation of a tree is one of the preorder, inorder or postorder traversal. Which one?

Compute each of the traversals of the tree whose list representation is

`[A [[B [[E] [F] [G] [H]]] [C [I]] [D]]]`.

5. binary-tree Show that the definitions of a binary tree in terms of terminal nodes and in terms of left and right children, are equivalent.
6. priority-queue Show that the priority queue representation shown in Fig.?? is the one obtained by using the "add" operation described above on the first 13 letters of the string "A SORTING EXAMPLE" one by one in the given order to an empty priority queue. Continue the process to give the priority queue using all 15 letters.
7. string-sorting Verify the construction used to produce the Huffman encoding tree for the string "A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS".
8. huffman Describe a *Huffman* code and give examples of circumstances when it (or a variant) might be used. Illustrate your answer by constructing a binary Huffman code for the string

### A TEST EXAMINATION ANSWER

As well as deriving the coding tree, you should give your code for the word ANSWER. Note that the string contains 12 distinct symbols, include the "space" symbol, and 25 symbols in all.

In what sense is a Huffman code optimal?

9. heapsort Give a brief description of a priority queue and a complete binary tree. What does it mean to say that a complete binary tree satisfies the *heap condition*?

Describe the use of Heapsort to sort the string

### HEAPSORT

Give the full construction of the heap in detail, and give details of the first step in its use.

Does Heapsort have any advantages over Quicksort?

---

MX4002 ALGORITHMS  
SOLUTIONS TO SHEET 3: ABSTRACT DATA TYPES

---

1. queue-as-stack Pop the stack and push onto an auxiliary stack until the first stack is empty, and you can get at the bottom element. Then push them all back again, noting that at least they come out in the right order. So if there are  $n$  elements on the stack, it takes  $2n$  operations to remove the next element from the queue: not a very bright idea!
2. queue-as-array Add at the end of the array, remove at the front, and roll round to avoid running out of space before the array is full. This means there is need for two points to be kept current all the time, say  $f$ , pointing to the front of the queue, and  $b$ , pointing to the back, where new elements will be added. If  $f < b$  things are as you expect, while if  $f > b$ , the queue “really” stretches from  $f$  to  $b + N$ , where  $N$  is the size of the array.
3. three-flags Since there are more red tokens than any other, we re-arrange the algorithm to avoid performing a swap when we meet a red token. Here is the new algorithm:

```
b = w = 1; r = N; // initialise as before
while (w < r + 1) begin
  if (R(r)) r = r - 1 // it was a red token
  else if (W(r)) begin // it was a white token
    swap(w, r); w = w + 1
  end
  else begin // it was a blue token
    swap(b, w); swap(r, b); b = b + 1; w = w + 1
  end
end // while
```

The action on seeing a blue token is first to move region  $W$  up by swapping it's first member with the unknown token at the bottom of region  $X$ , and then to swap that unknown token with the one at the bottom of region  $R$ , which has just proved to be blue.

The total number of calls to `swap` is thus  $|W| + 2|B|$ . This is faster than the original algorithm provided

$$|W| + 2|B| \leq |B| + |R|$$

and this occurs iff  $|W| + |B| \leq |R|$ ; in other words, when at least half of the tokens are red.

4. flattening It is the preorder traversal.

```
Preorder:-      [A B E F G H C I D]
Inorder:-       [E B F G H A I C D]
Postorder:-     [E F G H B I C D A]
```

5. binary-tree To pass from a “left-right children” description to a “terminal node” description, replace the empty tree by one consisting of a single terminal node. Next replace every child node which is a leaf node by one having two children, each of which is a terminal node. Finally, if a node has a left child, add a terminal right node etc.

To pass from the “terminal node” description to the “left-right children” description, simply delete all terminal nodes.

6. priority-queue Try it - going to Sedgewick if necessary for more help.
7. string-sorting Note that it is not unique; where two symbols have the same frequency, it is necessary to make exactly the same choice as shown in the diagram in order to exactly reproduce the tree given in the notes.
8. huffman A Huffman code is used when the input is from a limited character set, and contains information that is redundant; a typical example is English text, in which the letter “e” occurs significantly more often than random. One such application is in the “gzip” utility which is based on the Huffman code.

We describe how to build the binary Huffman code by building the corresponding binary tree. We start by analysing the message to find the frequencies of each symbol that occurs in it. Our basic strategy will be to assign short codes to symbols that occur frequently, while still insisting that the code has the prefix property. Our example will be build around the message

#### A TEST EXAMINATION ANSWER

The corresponding frequencies are given in Table 1; we write the space symbol “ ”, in the table as  $\square$ .

A	T	E	S	X	M	I	N	O	W	R	$\square$
4	3	3	2	1	1	2	3	1	1	1	3

Table 1: Symbol frequencies used to build a Huffman Code.

Now begin with a collection (a forest) of very simple trees, one for each symbol to be coded, with each consisting of a single node, labelled by that symbol, and the frequency with which it occurs in the string. The construction is recursive: at each stage the two trees which account for the least total frequency in their root nodes are selected, and used to produce a new binary tree. This has, as its children the two trees just chosen: the root is then labelled with the total frequency accounted for by both subtrees, and the original subtrees are removed from the forest. The construction continues in this way until only one tree remains; that is then the Huffman encoding tree.

[NB The particular order in which the symbols are displayed was determined by trial and error in order to present the result in a tabular form in which only adjacent columns were merged. It is not necessary to give a presentation in this form; crossing are perfectly acceptable.]

We now code the word ANSWER using this coding tree. Although this is a prefix code and so spaces are *not* required, we include spaces between letters for clarity. We read up from the bottom of the table, assigning 0 to a choice of a left hand branch, and 1 to a choice of the right hand branch. The resulting code is then

N	□	A	X	M	O	W	T	E	R	S	I
3	3	4	1	1	1	1	3	3	1	2	2
				2		2		6		3	2
	6			8						5	
			14						11		
				25							

Table 2: Symbol frequencies used to build a Huffman Code.

010□000□1101□01111□101□1100

The Huffman code has the prefix property; no character code is the prefix, or start of the the code for another character. In the same way that the Huffman code was constructed, we can associate a binary code with the prefix property to any binary tree. Given any binary tree with the same set of leaf nodes, and thus able to code the same symbols, the Huffman code is *optimal* in the sense that the corresponding coded message length is at least as short as that from the given tree.

9. heapsort A priority queue is a collection of elements or items, each of which has an associated priority. The operations available are:-

**create** creates an empty priority queue;

**add(item)** adds the given *item* to the priority queue; and

**remove** removes the item with the highest priority from the queue.

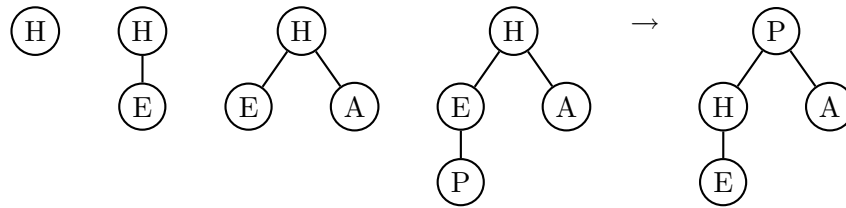
A complete binary tree is a tree which is either empty, or one in which every node:

- has no children; or
- has just a left child; or
- has both a left and a right child.

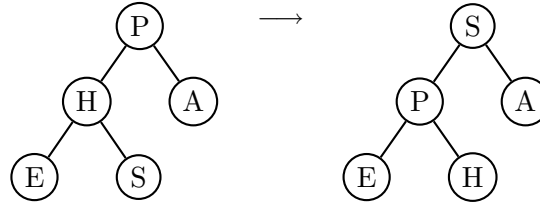
In addition, we require that all the levels, except perhaps the last, has both a left and a right child; while on the last level, any missing nodes are to the right of all the nodes that are present.

If we have a representation of a priority queue as a complete binary tree in which each node contains an element and its associated key or priority, it satisfies the *heap condition* if at each node, the associated key is larger than the keys associated with either child of that node.

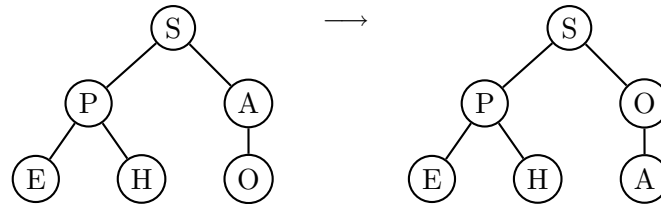
We now create a priority queue, represented as a complete binary tree, in which letters at the end of the alphabet have priority. When the first three characters are added as additional nodes in the binary tree, the new tree already satisfies the heap condition. Adding “P” violates that condition; “P” thus has to rise until it is again satisfied.



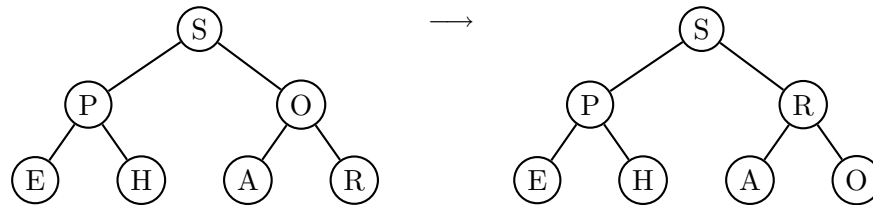
When “S” is added, it rises to the top of the tree before the heap condition holds.



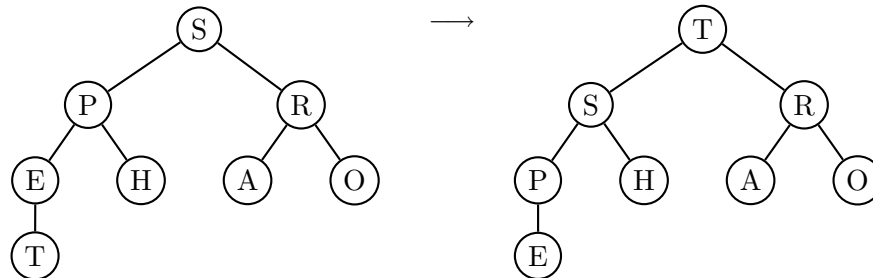
Now add “O”; it has to rise to be above “A”



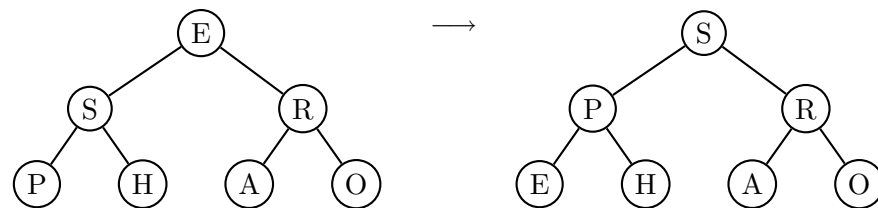
Now add “R”; this replaces “O” and fills the tree at level 2.



The final addition of “T” completes the tree, when it has been re-heaped.



The string is now sorted by removing the root node — this is the item with the highest priority remaining in the tree — and replacing it with the “last” node in the tree. The tree no longer satisfies the heap condition, and so the node that has just been promoted is allowed to fall again until the condition holds.



The root of the tree again contains the next character in order and is removed. This continues until the whole heap has been destroyed.

Finally, note that although in general Heapsort is slower than Quicksort by a factor of about 2, the “worst case” behaviour remains  $O(n \log n)$ , whereas Quicksort can become  $O(n^2)$ .