Priority-Queues II: Heaps

We restrict our discussion to binary tree implementation of heaps. We show the items of a heap to be capital Roman letters.

Binary Trees

There are several possible orderings of the items in a binary tree.

1. Alphabetic order, also called inorder. This is the order of the items of a binary search tree.
2. Preorder. The root holds the least item. Both the left and right subtrees are in preorder. Every item in the left subtree is less than or equal to every item in the right subtree.
3. Postorder. Dual to preorder. The root holds the greatest item. Both the left and right subtrees are in postorder. Every item in the left subtree is less than or equal to every item in the right subtree.
4. Level order. The root holds the least item. Each item in level ℓ is less than or equal to each item in level $\ell + 1$. The items in each level are increasing from left to right.
5. Heap order.
   (a) Max-heap order. The root holds the greatest item. Each item is less than or equal to its parent.
   (b) Min-heap order. The root holds the least item. Each item is greater than or equal to its parent.

   We will use only max-heap order.

Implementation of a Heap

We implement a heap H as an almost complete binary tree T whose items are the items of H is heap order. Each level of T is complete, except possibly for the bottom level, which is left-justified. Figure 1(a) shows an example.

Array Implementation of Binary Tree Implementation of Heap

An almost complete binary tree can be represented as an array $H[1 \ldots n]$[1] Hour rules are:

1. The items of the heap are stored in level order in the array.
2. The root is $H[1]$.
3. For any $i \geq 2$, the parent of $H[i]$ is $H\left[\, \lfloor \frac{i}{2} \rfloor \,\right]$.
4. $H[i]$ could have two, one, or no children. The indices of those children are $2i$ and $2i+1$ if $2i+1 \leq n$, just $2i$ if $2i = n$, and none if $2i > n$.

[1] Of course, you could start the array at 0, by making appropriate changes in the parent and child functions.

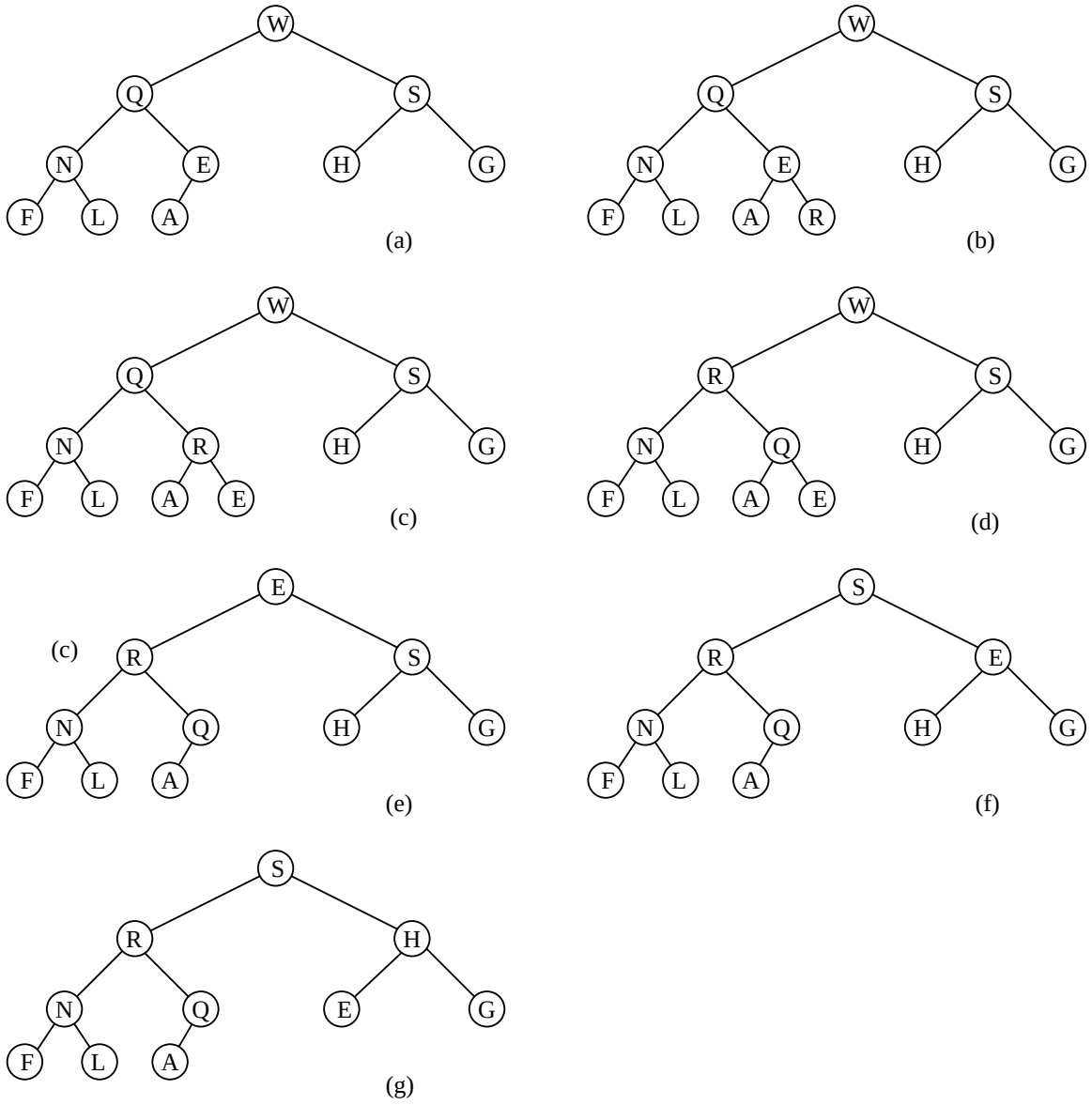

Figure 1

Figure (a) shows the initial heap, as a binary tree in heap order. In (b), we insert the item R. By the almost completeness property, R must be inserted in the next available space, as the right child of E. The tree is no longer in heap order. R must now *bubbleup* to its correct position. In (c), R switches with its parent E. In (d), R switches with its parent Q. Heap order is restored.

In (e), deletemax is executed. The maximum item W, which is at the root, is returned, and is deleted from the tree. The last item (in level order) E is moved to the root. The tree is no longer in heap order. E must now *bubbledown* to its correct position. In (f), E switches with its larger child S. In (g), E switches with its larger child G. Heap order is restored.

The heap shown in Figure 1 is implemented at the following array, and the evolution of the heap shown in that figure are indicated by evolution of the array. The first row of the array shows indices.

1	2	3	4	5	6	7	8	9	10	11
W	Q	S	N	E	H	G	F	L	A	
W	Q	S	N	E	H	G	F	L	A	R
W	Q	S	N	R	H	G	F	L	A	E
W	R	S	N	Q	H	G	F	L	A	E
E	R	S	N	Q	H	G	F	L	A	
S	R	E	N	Q	H	G	F	L	A	
S	R	H	N	Q	H	E	G	L	A	

Insertion

For a heap H and a item x, we first place x in the first, in level order, open place in the binary tree. We then *bubbleup* x, by exchanging it with its parent until it is smaller than its parent.

Deletemax

If our heap H is not empty, the function deletemax(H) returns the maximum item $H[1]$, and deletes it from H. The last (in level order) item in H is placed in the root. That item then executes *bubbledown*, exchanging with its larger child until it has no child larger than x.

There is no need to implement a binary tree in code, since all changes are recorded in the array.