Queue

A *queue* data structure supports the following four operations.

1. empty (Q): decides if queue Q is empty;
2. insert (Q, x): add element x to Q;
3. find-earliest (Q): return the earliest added element in Q;
4. delete-earliest (Q): remove the earliest added element in Q.

To implement above operations, we can use a (dynamic) array S to stores all elements, and use two pointers, *head* and *tail*, where *head* pointer always points to the first available space in S, and *tail* pointer always points to the earliest added element in S. When we add an element to S we can directly add it to the place *head* points to, and when we delete the earliest added element, we can directly remove the one *tail* points to.

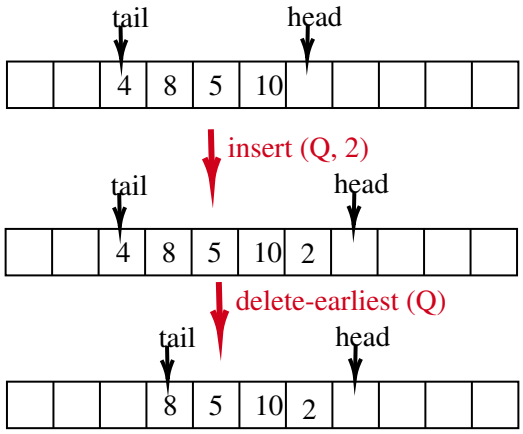

Figure 1: An example of queue.

The pseudo-code for implementing above functions are given below. Note that the input Q represents above 3 data structures, the array S, the two pointers *head* and *tail*.

```
function empty(Q)
    if head = tail: return true;
    else: return false;
end function;

function insert(Q, x)
    S[head] = x;
    head = head + 1;
end function;

function find-earliest(Q)
    return S[tail];
end function;
```

```
function delete-earliest(Q)
    if empty(Q) = true: return;
    tail = tail + 1;
end function;
```

Note, a queue data structure exhibits a first-in-first-out property. This is because delete-earliest is the only function that removes element and it always removes the earliest added element.

Priority Queue

In a priority queue, each element is labeled with a *priority* (also called key). The priority/key serves as the identify of the element. Additional data can be associated. A *priority queue* data structure supports the following operations.

1. empty (PQ): decides if priority queue PQ is empty;
2. insert (PQ, x): add element x to PQ;
3. find-min (PQ): return the element in PQ with smallest key (i.e., highest priority);
4. delete-min (PQ): remove the element in PQ with smallest key (i.e., highest priority).
5. decrease-key (PQ, pointer-to-an-element, new-key): set the key of the specified element as the given new-key.

Note that a queue can be regarded as a special case of priority queue, for which the priority is the time an element is added to the queue (hence, minimum-priority = minimum adding-time = earliest added).

There are numerous different implementations for priority queue (check wikipedia). Here we introduce one of them, *binary heap*. To do it, let's first formally introduce *heap*.

A *heap* is a (rooted) tree data structure satisfies the *heap property*. A heap is either a *min-heap* if it satisfies the *min-heap property*: for any edge (u,v) in the (rooted) tree T, the key of u (i.e., the parent) is smaller than that of v (i.e., the child), or a *max-heap* if it satisfies the *max-heap property*: for any edge (u,v) in the (rooted) tree T, the key of u is larger than that of v. Here we always consider a heap as a min-heap, unless otherwise specified.

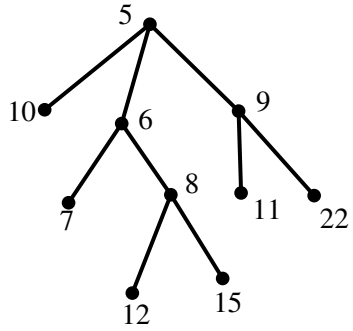

Figure 2: An example of heap. The key of an element is next to each vertex.

Because of the heap property, the root of the tree always has the smallest key. Hence, to find the element with the smallest key (i.e., find-min) one can simply return the root. This is the main reason of using a heap to implement priority queue.

A *binary heap* is a heap with the rooted tree being the *complete binary tree*. A *complete binary tree* is a binary tree (i.e., each vertex has at most 2 children) and that in every layer of the tree, except possibly the last, is completely filled, and all vertices in the last layer are placed from left to right.

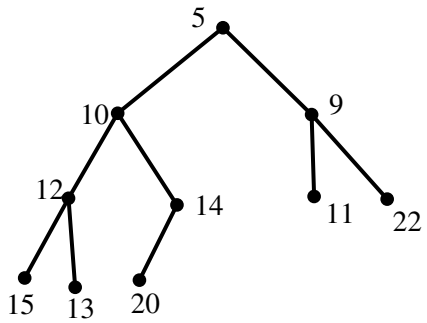

Figure 3: An example of binary heap.

Since a binary heap T is so regular, we can use an array S to store its elements (rather than maintaining an actual tree). The root (i.e., layer 1) of T is placed in $S[1]$ (we assume that the index of S starts from 1), the first element of the layer 2 is placed in $S[2]$, and so on. Generally, the j-th element in the i-th layer of T will be placed in $S[2^{i-1}+j-1]$. We can also easily access the parent and children of an element:

1. the left child of $S[k]$ is $S[2k]$;
2. the right child of $S[k]$ is $S[2k+1]$;
3. the parent element of $S[k]$ is $S[\lfloor k/2 \rfloor]$.

We now introduce two common procedures used in implementing a binary heap. These procedures apply when one vertex violates the heap property, and they can adjust the heap to make it satisfy the heap property.

The *bubble-up* function applies when a vertex has a smaller key than its parent.

```
function bubble-up (S, k)
    if k ≤ 1: return;
    p = ⌊k/2⌋;
    if (S[k].key < S[p].key);
        swap S[p] and S[k];
        bubble-up (S, p);
    end if;
end function;
```

The *sift-down* function applies when a vertex has a larger key than its children. We use n to denote the size of array S, i.e., the number of elements stored in S.

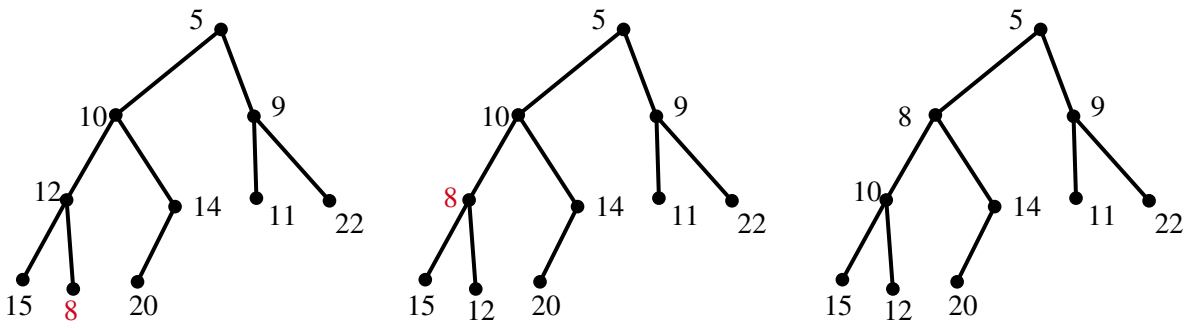

Figure 4: Illustrating bubble-up procedure.

function sift-down (S, k)
    if $2k > n$: return;
    if $2k = n$: $c = 2k$;
    else $c = \arg\min_{t \in \{2k, 2k+1\}} S[t].key$ be the index of the child of $S[k]$ with smaller key;
    if ($S[k].key > S[c].key$);
        swap $S[c]$ and $S[k]$;
        sift-down (S, c);
    end if;
end function;

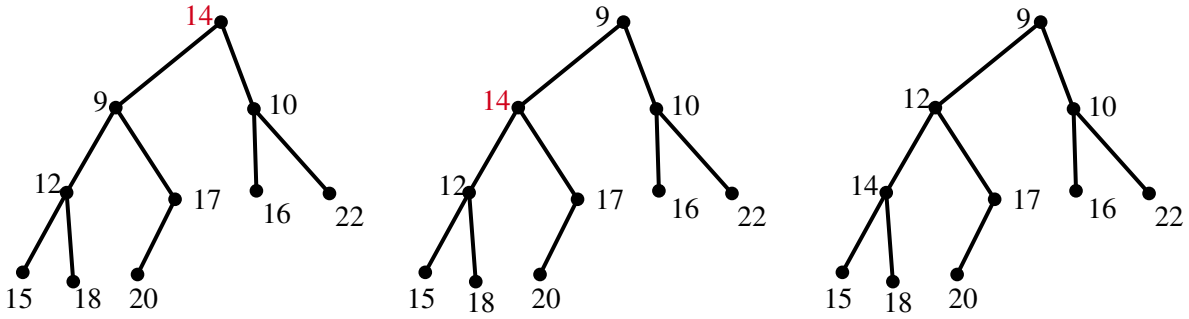

Figure 5: Illustrating sift-down procedure.

Both bubble-up and sift-down procedures runs in $O(\log n)$ time. This is because, a complete binary tree with n vertices has a height of $\log n$, while the worst case of either procedure traverses along a path between the root and a leaf. Formally, in each recursive call, k is either halved or doubled, and hence the number of recursive calls is $\log n$.

We finally give the pseudo-code for implementing the binary heap. In them the input PQ means the array S and its size n.

function empty(PQ)
    if $n = 0$: return true;
    else: return true;
end function;

```
function insert(PQ, x)
    n = n + 1;
    S[n] = x;
    bubble-up (S, n);
end function;

function find-min(PQ)
    return S[1];
end function;

function delete-min(PQ)
    S[1] = S[n];
    n = n − 1;
    sift-down (S, 1);
end function;

function decrease-key(PQ, k, new-key)
    S[k].key = new-key;
    bubble-up (S, k);
end function;
```

The empty and find-min procedures takes $\Theta(1)$ time; the other 3 procedures takes $O(\log n)$ time, as they are dominated by either bubble-up or sift-down.