Priority queue and heaps

Ordered dictionary ordinarily uses pointers for every key it stores and uses trees that are not “perfectly balanced”. We can do better if we use it in a restricted way. In a typical application, we store objects with priorities and in some applications, the priorities can change over time.

Set Operations:

- Initialize(Q), Insert(x, p, Q), DeleteMin(Q)

Other updates:

- IncreasePriority(i, p, Q), DecreasePriority(i, p, Q).

In the last operation, we decrease the priority of an object at position $i$. We can know this position if we use cross-referencing.

We implement priority queues using heaps. Heap is a tree in which parent node has smaller (or equal) priority to the node of the parent. This is called heap order or partial order. Because we do not have other limitations, we are quite flexible how we place the objects in the tree.

The simplest heap with $m$ objects is an array sector $(H, 1, m + 1)$ where position 1 is the root, and children of position $i$ are: $2i$ and $2i + 1$. Conversely, parent of $i$ is $i/2$ (rounded down).

Example

Example of a heap ($H[i]$ has number $i$ underneath):
Restoring heap property

We will define all operations in a heap using two restore operations. Assume that \(<H, 0, m>\) has heap property and that we want to change entry \(H[i]\) to \(x\). To restore the heap property we need to rearrange the entries. We have two cases.

The easier case is when \(x < H[i]\). We will call this task \(rad(H, i, x)\), for restore after decreasing. We can define \(rad(H, i, x)\) recursively.

\[
\text{rai}(H, i, x) \quad \{
\text{if (i == 1 || (p = i/2, x} \geq H[p]) \{ \\
H[i] = x; \\
\text{return;}
\}
\}
\]

\[
H[i] = H[p]; \\
\text{rai}(H, p, x);
\}
\]

This formulation is recursive so we can have an inductive proof of correctness. Because the recursive call is the last statement in this function, we can eliminate this call using tail recursion principle.

\[
\text{while (i > 1 && (p = i/2, x} < H[p])}) \}
\]

\[
H[i] = H[p], \\
i = p; \\
H[i] = x;
\]

The worst case running time is \(c \log_2 i\).
Example: \((H, 1, 13)\) forms a heap shown before and we execute \(\text{rad}(H, 9, 2)\). Location to be changed by \(\text{rad}\) will be colored blue.

2 and over-write the root in the last step
The more complicated case is when \( x > H[i] \). We will call this task \( rai(H, i, m, x) \), for **restore after increasing**; here \( m \) is the number of keys stored in the heap. We can define \( rai(H, i, m, x) \) recursively.

\[
rai(H, i, m, x) = \\
{\text{\{ } g = 2\times i; \quad \text{// start computing the good child of } i \\
\quad \text{if } (g > m) \{ \quad \text{// no children }\rightarrow \text{ finish} \\
\quad \quad H[i] = x; \\
\quad \quad \text{return;}
\}
\quad \text{if } (g+1 \leq m \&\& H[g] > H[g+1]) \\
\quad \quad g++; \quad \text{// good child has the smaller priority }
\quad \text{if } (x \leq H[g]) \{ \\
\quad \quad H[i] = x; \\
\quad \quad \text{return;}
\}
\quad H[i] = H[g];
\quad rai(H, g, m, x); \\
\})
\]

Again, we can eliminate the tail recursion:

\[
\text{while } (g = 2\times i, \ g < m) \{ \\
\quad \text{if } (g+1 \leq m \&\& H[g] > H[g+1]) \\
\quad \quad g++; \\
\quad \text{if } (x \leq H[g]) \\
\quad \quad \text{break;}
\quad H[i] = H[g];
\quad i = g;
\}
\quad H[i] = x;
\]

The worst case running time is \( c \log_2 m/i \).
Example: \( (H, 1, 13) \) we execute \( \text{rai}(H, 0, 12, 39) \). Location to be changed by \( \text{rai} \) are blue, the good child is yellow.
Heapify, impose the heap property

We can impose the heap property by pretending to restore it.

Version 1: pretend that all entries are $\infty$ and that you decrease them to their actual values. If you do it in order $A[0], A[1], \ldots$ then before you process $A[i]$ the fragment $< A, i, m >$ remains unchanged.

```c
for (i = 2; i <= m; i++)
    rad(A, i, A[i]);
```

The worst case running time is

$$c \sum_{i=1}^{m-1} \log_2 i \approx cm(log_2 m - log_2 e) = \Theta(m \log m).$$

Version 2: pretend that all entries are $-\infty$ and that you increase them to their actual values. If you do it in order $A[m-1], A[m-2], \ldots$ then before you process $A[i]$ the fragment $< A, 0, i >$ remains unchanged.

```c
for (i = m/2; i > 0; i--)
    rai(A, i, m, A[i]);
```

The worst case running time is

$$c \sum_{i=1}^{m-1} \log_2 m/i \approx cm \log_2 m - cm(log_2 m - log_2 e) = cm \log_2 e = \Theta(m).$$

Version 2 is much better!
Heap sort

We want to sort $(A, 0, n)$. We create heap $(H, 1, n + 1)$ and then we remove the minimum from the heap and place it at the beginning of the resulting sequence. At any stage, $(H, 1, m + 1) = (A, 0, m)$ contains the heap and $(A, m, n)$ stores the resulting sequence.

```
H = A-1;
for (i = n/2; i > 0; i--)
    rad(H,i,n,H[i]);
for (m = n; m > 1; m--)
    temp = H[m],
    H[m] = H[1],
    rai(H,1,m-1,temp);
```

This sorts in the reverse order. To sort in increasing order we need to modify the heap so it has maximum at the root rather then the minimum.
Priority queue

Abstract data type:

possible states: sets of elements of some comparable type;

operations:

MAKE_EMTPY, initialize empty set,
INSERT,
DELETE_MIN, remove and return the minimum.

Implementation with bounded capacity n, using array \( H[n] \) and \( m \):

MAKE_EMPTY()
{  \( m = 0; \)  }

INSERT(\( x \));
{  \( m++; \)
    rad(\( H,m,m,x \))
  }

DELETE_MIN()
{  \( x = H[1]; \)
    \( m--; \)
    rai(\( H,1,m,H[m+1] \));
    return \( x; \)
  }
Cross referencing

In graph algorithms we will see the following way to apply a priority queue:
- we have an array of all possible elements, nodes of a graph
- we may initialize all elements with priority $\infty$, which means, too high to make sense
  subsequent operations:
  - remove an element with the minimum priority
  - reduce priority of an element
We can use $\text{rad}$ function to reduce the priority of an element on the heap if we know where is it.
To use it, we need to have three arrays:
- $H[i]$ is the node in location $i$ of the heap
- $W[u]$ is the location of node $u$ on the heap
- $P[u]$ is the priority of node $u$.

We have $W[H[i]] = i$ and $H[W[u]] = u$ (if $u$ is in the priority queue). This is cross-referencing: arrays $H$ and $W$ refer to each other.

Now we need to re-write heap operations as follows:
- replace comparisons $H[i]$ versus $H[j]$ with $P[H[i]]$ versus $P[H[j]]$
- assignment $H[i] = u$ must be followed with $W[u] = i$
- if $u$ is returned by $\text{DeleteMin}$ we follow with $W[u] = 0$.
  $P[u]$ is the priority of node $u$.

Priority queue may be implemented differently, and it should be implemented differently if we have many more priority decreases than we have removals from the queue: Fibonacci heaps achieve $O(1)$ average cost of priority decreases.