CS580 Algorithm Design, Analysis, and Implementation

Lecture notes, Jan 13, 2022
Wufei Ma

1 Heap

Binary tree. A binary tree of depth n is **balanced** if all the nodes at depths 0 through $n-2$ have two children. (The only depth that is not "full" is depth n.)

A balanced binary tree of depth n is **left-justified** if it has 2^n nodes at depth n or 2^k nodes at depth k, for all $k < n$, and the leaves at depth n are as far left as possible.

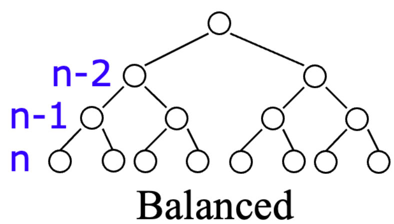

Figure 1: Binary tree.

Heap. A heap is a left-justified or complete binary tree with the property that no node has a value greater than the value in its parent.

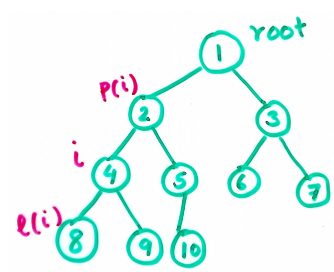

Figure 2: Heap.

Since the heap is a left-justified binary tree, we may implement the heap as an array. The embedding is defined by:

```
root = 1
l(i) = 2i
r(i) = 2i + 1
p(i) = floor(i/2)
```

and the heap property is given by `A[i] <= A[p(i)]` for all `i`.

The **height** of a node is the number of edges on the longest downward path starting at the node. The **height** of a heap is the height of the root. Since the heap is balanced,

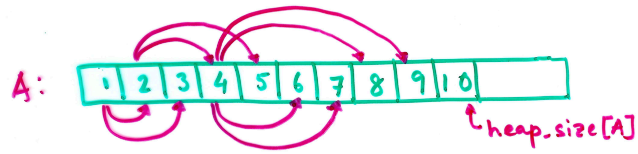

Figure 3: Heap as an array.

we have

$$
\begin{aligned}
\sum_{i=0}^{h-1} 2^i + 1 \leq & n \leq \sum_{i=0}^{h} 2^i \\
2^h \leq & n \leq 2^{h+1} - 1 \\
\log(n+1) - 1 \leq & h \leq \log n
\end{aligned}
$$

Maintain a heap. `DownHeap` extends the heap property by one more node.

```
Procedure DownHeap(i):
    max := i
    if l(i) <= heap_size[A] and A[max] < A[l(i)]
        max := l(i)
    if r(i) <= heap_size[A] and A[max] < A[r(i)]
        max := r(i)
    if max != i
        swap(A[i], A[max])
        Downheap(max)
```

The cost is $O(h)$. This procedure can also be written as an iterative algorithm.

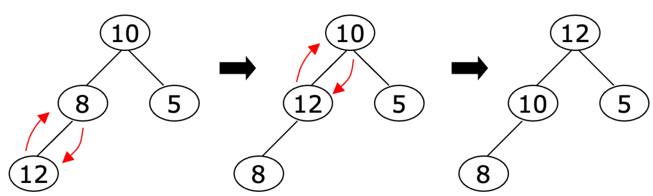

Figure 4: UpHeap.

Construct a heap. The idea is to construct the heap from bottom up.

```
Procedure BuildHeap(n):
    for i := n down to 1
        Downheap(i)
```

The cost is $O(n \log n)$, but a tighter analysis is possible. The amount of time to build the heap is at most

$$\begin{aligned}\sum_{i=0}^{h} 2^i O(h-i) &= O\left(h \sum_{i=0}^{h} 2^i - \sum_{i=0}^{h} 2^i \cdot i\right) \\ &= O\left(h \cdot 2^{h+1} - h - h \cdot 2^{h+1} + 2^{h+1} - 1\right) \\ &= O(2^{h+1} - h - 1) \\ &= O(n)\end{aligned}$$

Heap sort. The input is an unsorted array `A[1...n]`.

```
Procedure HeapSort(n):
   BuildHeap(n)
   heap_size[A] := n
   for i := n down to 2 do swap(A[1], A[i])
      heap_size[A] := i-1
      DownHeap(1)
```

The complexity is $O(n \log n)$.

Heap as a priority queue. A priority queue stores a multiset of S keys and support operations:

- `Insert(x)`: insert a new element.
- `Delete(i)`: delete element at location `i`.
- `Max`: return the largest key.
- `ExtractMax`: return the largest key and remove it.

and each operation takes $O(\log n)$ time.

```
Procedure Insert(x):
   heap_size[A] := heap_size[A] + 1
   i := heap_size[A]
   A[i] = x
   UpHeap(i)

Procedure UpHeap(i):
   while i > 1 and A[i] > A[p(i)]:
      swap(A[i], A[p(i)])
      i := p(i)
```

```
Procedure Delete(i):
   A[i] := A[heap_size[A]]
   heap_size[A] := heap_size[A] - 1
```

```
        if A[i] < A[p(i)]
            DownHeap(i)
        else
            UpHeap(i)
```

```
    Procedure ExtractMax:
        max := A[1]
        Delete(1)
        return max
```