Binary Heaps in Dynamic Arrays

Yufei Tao

ITEE
University of Queensland

We have already learned that the binary heap serves as an efficient implementation of a priority queue. Our previous discussion was based on pointers (for getting a parent node connected with its children). In this lecture, we will see a "pointerless" way to implement a binary heap, which in practice achieves much lower space consumption.

We will also see a way to build a heap from n integers in just $O(n)$ time, improving the obvious $O(n \log n)$ bound.

Recall:

Priority Queue

A priority queue stores a set S of n integers and supports the following operations:

- `Insert`(e): Adds a new integer to S.
- `Delete-min`: Removes the smallest integer in S, and returns it.

Recall:

Binary Heap

Let S be a set of n integers. A binary heap on S is a binary tree T satisfying:

1. T is complete.
2. Every node u in T corresponds to a distinct integer in S—the integer is called the key of u (and is stored at u).
3. If u is an internal node, the key of u is smaller than those of its child nodes.

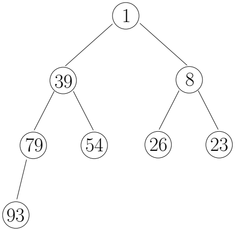
Storing a Complete Binary Tree Using an Array

Let T be any complete binary tree with n nodes. Let us linearize the nodes in the following manner:

- Put nodes at a higher level before those at a lower level.
- Within the same level, order the nodes from left to right.

Let us store the linearized sequence of nodes in an array A of length n.

Example



Stored as

1	39	8	79	54	26	23	93

Property 1

Let us refer to the i-th element of A as $A[i]$.

Lemma: Suppose that node u of T is stored at $A[i]$. Then, the left child of u is stored at $A[2i]$, and the right child at $A[2i + 1]$.

Observe this from the example of the previous slide.

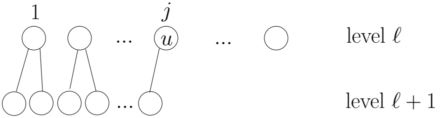
Property 1

Proof: Suppose that u is the j-th node at Level ℓ. This level must be full because u has a child node (which must be at Level $\ell + 1$). In other words, there are 2^ℓ nodes at level ℓ.

We will prove the lemma only for the left child (the right child is simply stored at the next position of the array). From the fact that u is the i-th node in the linearized order, we know:

$$\begin{aligned} i &= j + 2^0 + 2^1 + ... + 2^{\ell-1} \\ &= j + 2^\ell - 1. \end{aligned}$$

Property 1



Next we will prove that there are precisely $i - 1$ nodes in A after u but before its left child. These nodes include:

- Those at Level ℓ behind u: there are $2^\ell - j$ of them.
- Child nodes of the first $j - 1$ nodes at Level ℓ: there are $2(j - 1)$ of them.

Hence, in total, there are $2^\ell - j + 2(j - 1) = 2^\ell + j - 2 = i - 1$ such nodes. This completes the proof. $\square$

Property 1

The following is an immediate corollary of the previous lemma:

Corollary: Suppose that node u of T is stored at $A[i]$. Then, the parent of u is stored at $A[\lfloor i/2 \rfloor]$.

Property 2

The following is a simple yet useful fact:

Lemma: The rightmost leaf node at the bottom level is stored at $A[n]$.

Proof: Obvious. $\square$

Now we have got everything we need to implement the insertion and delete-min algorithms (discussed in the previous lecture) on the array representation of a binary heap.

Example

Inserting 15:

1	39	8	79	54	26	23	93	15

1	39	8	15	54	26	23	93	79

1	15	8	39	54	26	23	93	79

Performing a delete-min:

1	15	8	39	54	26	23	93	79

79	15	8	39	54	26	23	93

8	15	79	39	54	26	23	93

8	15	23	39	54	26	79	93

Performance Guarantees

Combining our analysis on (i) binary heaps and (ii) dynamic arrays, we obtain the following guarantees on a binary heap implemented with a dynamic array:

- Space consumption $O(n)$.
- Insertion: $O(\log n)$ time amortized.
- Delete-min: $O(\log n)$ time amortized.

Next, we consider the problem of creating a binary heap on a set S of n integers. Obviously, we can do so in $O(n \log n)$ time by doing n insertions. However, this is an overkill because the binary heap does not need to support any delete-min operations until all the n numbers have been inserted. This raises the question whether we can build the heap faster.

The answer is positive: we will see an algorithm that does so in $O(n)$ time.

Fixing a Messed-Up Root

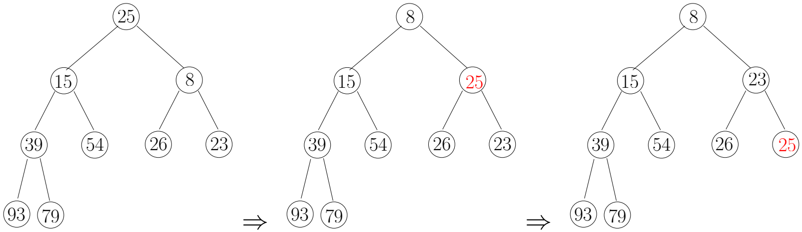
Let us first consider the following `root-fix` operation. We are given a complete binary tree T with root r. It is guaranteed that:

- The left subtree of r is a binary heap.
- The right subtree of r is a binary heap.

However, the key of r may not be smaller than the keys of its children. The operation fixes the issue, and makes T a binary heap.

This can be done in $O(\log n)$ time – in the same manner as the `delete-min` algorithm (by descending a path).

Example


25
15
8
39
54
26
23
93
79
⇒
8
15
25
39
54
26
23
93
79
⇒
8
15
23
39
54
26
25
93
79

Building a Heap

Given an array A that stores a set S of n integers, we can turn A into a binary heap on S using the following simple algorithm, which views A as a complete binary search tree T:

- For each $i = n$ downto 1
  - Perform `root-fix` on the subtree of T rooted at $A[i]$

> Think: Why are the conditions of `root-fix` always satisfied?

Example

							i
54	26	15	93	8	1	23	39

						i	
54	26	15	93	8	1	23	39

					i		
54	26	15	93	8	1	23	39

				i			
54	26	15	93	8	1	23	39

			i				
54	26	15	39	8	1	23	93

		i					
54	26	1	39	8	15	23	93

	i						
54	8	1	39	26	15	23	93

i							
1	8	15	39	26	54	23	93

Running Time

Now let us analyze the time of the building algorithm. Suppose that T has height h. Without loss of generality, assume that all the levels of T are full – namely, $n = 2^h - 1$ (why no generality is lost?).

Observe:

- A node at Level $h - 1$ incurs $O(1)$ time in `root-fix`; 2^{h-1} such nodes.
- A node at Level $h - 2$ incurs $O(2)$ time in `root-fix`; 2^{h-2} such nodes.
- A node at Level $h - 3$ incurs $O(3)$ time in `root-fix`; 2^{h-3} such nodes.
- ...
- A node at Level $h - h$ incurs $O(h)$ time in `root-fix`; 2^0 such nodes.

Running Time

Hence, the total time is bounded by

$$\sum_{i=1}^{h} O\left(i \cdot 2^{h-i}\right) = O\left(\sum_{i=1}^{h} i \cdot 2^{h-i}\right)$$

We will prove that the right hand side is $O(n)$ in the next slide.

Running Time

Suppose that

$$
\begin{aligned}
x &= 2^{h-1} + 2 \cdot 2^{h-2} + 3 \cdot 2^{h-3} + \ldots + h \cdot 2^{0} && (1)\\
\Rightarrow 2x &= 2^{h} + 2 \cdot 2^{h-1} + 3 \cdot 2^{h-2} + \ldots + h \cdot 2^{1} && (2)
\end{aligned}
$$

Subtracting (1) from (2) gives

$$
\begin{aligned}
x &= 2^{h} + 2^{h-1} + 2^{h-2} + \ldots + 2^{1} - h\\
&\leq 2^{h+1}\\
&= 2(n+1) = O(n).
\end{aligned}
$$