Binary Heaps in Dynamic Arrays

CSCI 2100 Teaching Team

Department of Computer Science and Engineering
Chinese University of Hong Kong

Outline

1. An array-based implementation of the binary heap.
2. A heap building algorithm with $O(n)$ time complexity.

Review: 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 and returns the smallest integer in S.

Review: 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.

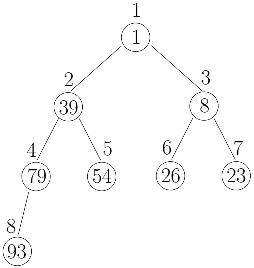
Storing a Complete Binary Tree Using an Array

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

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

Store the linearized node sequence in an array A of length n.

Example



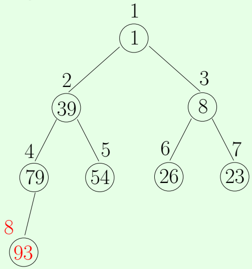
Stored as

Index:	1	2	3	4	5	6	7	8
	1	39	8	79	54	26	23	93

A

Property 1: The rightmost leaf node at the bottom level is stored at $A[n]$.

Example:

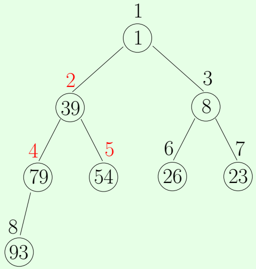


Index:	1	2	3	4	5	6	7	8
	1	39	8	79	54	26	23	93

A

Property 2: 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 $A[2i+1]$.

Example:



Index:	1	2	3	4	5	6	7	8
	1	39	8	79	54	26	23	93

A

Property 2 implies:

Property 3: Suppose that node u of T is stored at $A[i]$. Then, the parent of u is stored at $A[\lfloor i/2 \rfloor]$.

Now we are ready to implement the insertion and delete-min algorithms on the array representation of a binary heap.

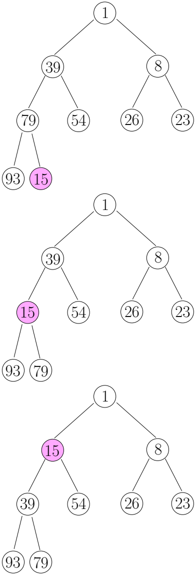
Insertion Example

Insert 15 and `swap-up`.

Index:	1	2	3	4	5	6	7	8	9
	1	39	8	79	54	26	23	93	15

1	2	3	4	5	6	7	8	9
1	39	8	15	54	26	23	93	79

1	2	3	4	5	6	7	8	9
1	15	8	39	54	26	23	93	79



Delete-min Example

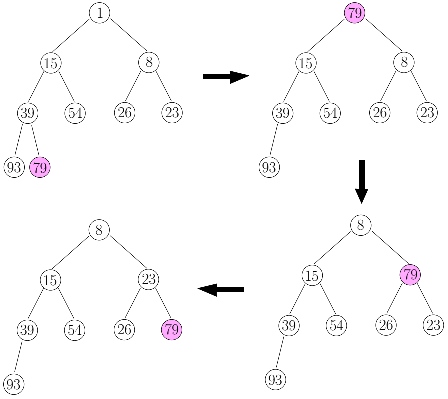
Replace 1 with 79 and `swap-down`.

Index:	1	2	3	4	5	6	7	8	9
	1	15	8	39	54	26	23	93	79

1	2	3	4	5	6	7	8
79	15	8	39	54	26	23	93

1	2	3	4	5	6	7	8
8	15	79	39	54	26	23	93

1	2	3	4	5	6	7	8
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 will see a heap building algorithm that runs in $O(n)$ time.

Fixing a Messed-Up Root

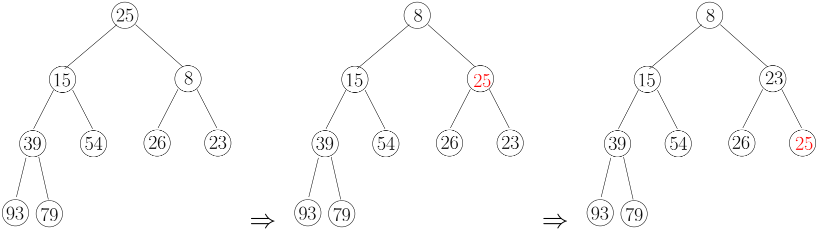
First, consider the following **root-fixing** problem. Suppose that we are given a complete binary tree T with root r such that

- 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. We need to fix the issue and makes T a binary heap.

This can be done in $O(\log n)$ time using the `swap-down` operation from the `delete-min` algorithm.

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 tree T).

- For each $i = \lfloor n/2 \rfloor$ downto 1
  - Apply `swap-down` to the subtree of T rooted at $A[i]$ to fix its root.

Think: Are the conditions of the root-fixing problem always satisfied?

Example

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

Running Time

Hence, the total time is bounded by

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

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

Running Time

Suppose that

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

Subtracting (1) from (2) gives

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