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): Add a new integer e to S.
- `Delete-min`: Remove the smallest integer from 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 stores a distinct integer in S, which is called the **key** of u.
3. If u is an internal node, the key of u is smaller than those of its child nodes.

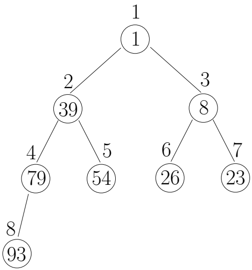
Storing a Complete Binary Tree in an Array

Let T be any complete binary tree with n nodes. We can **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.

The linearized node sequence can be stored in an array A of length n.

Example



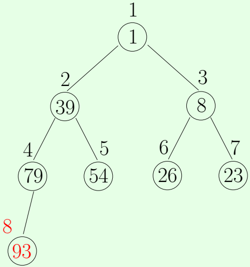
Stored as

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

A

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

Example:

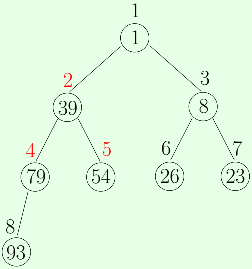


Index:	1	2	3	4	5	6	7	8
	1	39	8	79	54	26	23	93

A

Property 2: Suppose that node u of T is at $A[i]$. Then, the left child of u is at $A[2i]$ and its 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 $A[\lfloor i/2 \rfloor]$.

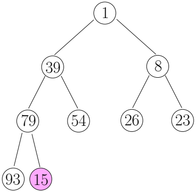
Now, we are ready to implement the insertion and delete-min algorithms using an array.

Insertion Example

Insert 15 and `swap-up`.

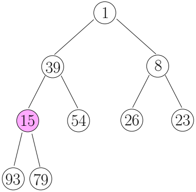
Index: 8

1	39	8	79	54	26	23	93	15



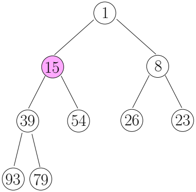
4

1	39	8	15	54	26	23	93	79



2

1	15	8	39	54	26	23	93	79



Delete-min Example

Replace 1 with 79 and `swap-down`.

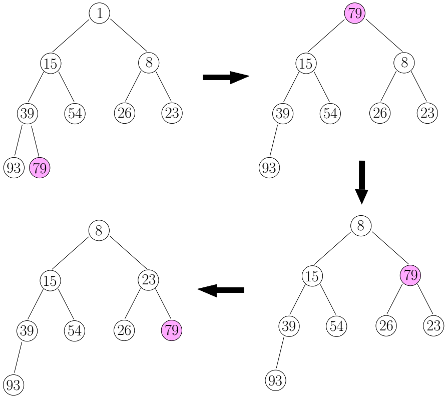
Index:

								9
1	15	8	39	54	26	23	93	79

1	2	3					
79	15	8	39	54	26	23	93

		3			6	7	
8	15	79	39	54	26	23	93

						7	
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 using a dynamic array:

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

How much time do we need to build a heap? Naively, this can be done in $O(n \log n)$ time by making n insertions. Next, we will see how to accomplish this in $O(n)$ time.

Fixing the Root

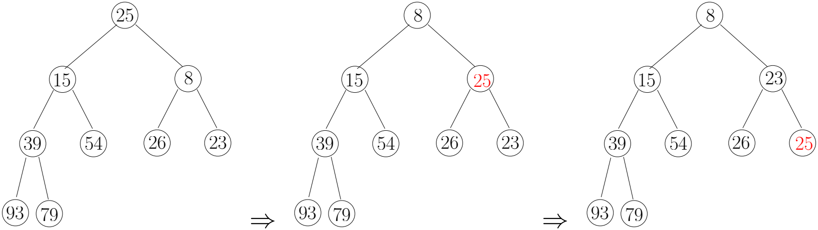
The Root Fixing Problem

Input: Let S be a set of integers and T be a complete binary tree where each node stores a distinct integer from S as its **key**. Denote by r the root of T. Both the left and right subtrees of r are binary heaps. However, the key of r may not be smaller than the keys of its children.

Output: Turn T into a binary heap on S.

This can be done in $O(\log n)$ time using the `swap-down` procedure 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 storing a set S of n integers. Alternatively, we can view A as a complete binary tree T. The following algorithm turns A into a binary heap on S.

- For each $i = \lfloor n/2 \rfloor$ downto 1
  - Solve the root-fixing problem on the subtree of $A[i]$

Example

				i				
after processing $i = 4$	54	26	15	39	8	1	23	93
			i					
after processing $i = 3$	54	26	1	39	8	15	23	93
		i						
after processing $i = 2$	54	8	1	39	26	15	23	93
	i							
after processing $i = 1$	1	8	15	39	26	54	23	93

Running Time

Next, we analyze the time of the heap-building algorithm. Let h be the height of T. Hence, $n \geq 2^{h-1}$ (why?).

- Each node at level $h-1$ incurs $O(1)$ time in `swap-down`. There are at most 2^{h-1} such nodes.
- Each node at level $h-2$ incurs $O(2)$ time in `swap-down`. There are exactly 2^{h-2} such nodes.
- Each node at level $h-3$ incurs $O(3)$ time in `swap-down`. There are exactly 2^{h-3} such nodes.
- ...
- A node at level 0 incurs $O(h)$ time in `swap-down`. There is $2^0 = 1$ such node.

Running Time

Hence, the total time is bounded by

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

The next slide proves that the right hand side of the above is $O(2^{h+1})$. Note that $O(2^{h+1}) = O(n)$

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

$$\begin{aligned} x &= 2^h + 2^{h-1} + 2^{h-2} + ... + 2^1 - h \\ &\leq 2^{h+1}. \end{aligned}$$