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:

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.
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 $\ell$. This level must be full because $u$ has a child node (which must be at Level $\ell + 1$). In other words, there are $2^\ell$ nodes at level $\ell$.

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:

$$i = j + 2^0 + 2^1 + \ldots + 2^{\ell-1}$$
$$= j + 2^\ell - 1.$$
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 $\ell$ behind $u$: there are $2^\ell - j$ of them.
- Child nodes of the first $j - 1$ nodes at Level $\ell$: there are $2j$ of them.

Hence, in total, there are $2^\ell - j + 2j = 2^\ell + j = i - 1$ such nodes. This completes the proof.
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 $A[\lfloor i/2 \rfloor]$. 
The following is a simple yet useful fact:

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

**Proof:** Obvious.
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:**

<table>
<thead>
<tr>
<th>1</th>
<th>39</th>
<th>8</th>
<th>79</th>
<th>54</th>
<th>26</th>
<th>23</th>
<th>93</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>39</td>
<td>8</td>
<td>15</td>
<td>54</td>
<td>26</td>
<td>23</td>
<td>93</td>
<td>79</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>8</td>
<td>39</td>
<td>54</td>
<td>26</td>
<td>23</td>
<td>93</td>
<td>79</td>
</tr>
</tbody>
</table>

**Performing a delete-min:**

<table>
<thead>
<tr>
<th>1</th>
<th>15</th>
<th>8</th>
<th>39</th>
<th>54</th>
<th>26</th>
<th>23</th>
<th>93</th>
<th>79</th>
</tr>
</thead>
<tbody>
<tr>
<td>79</td>
<td>15</td>
<td>8</td>
<td>39</td>
<td>54</td>
<td>26</td>
<td>23</td>
<td>93</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>15</td>
<td>79</td>
<td>39</td>
<td>54</td>
<td>26</td>
<td>23</td>
<td>93</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>15</td>
<td>23</td>
<td>39</td>
<td>54</td>
<td>26</td>
<td>79</td>
<td>93</td>
<td></td>
</tr>
</tbody>
</table>
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

Binary Heaps in Dynamic Arrays
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 $\text{root-fix}$ on the subtree of $T$ rooted at $A[i]$

Think: Why are the conditions of $\text{root-fix}$ always satisfied?
### Binary Heaps in Dynamic Arrays

An example of a binary heap in a dynamic array is shown below:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>26</td>
<td>15</td>
<td>93</td>
<td>8</td>
<td>1</td>
<td>23</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>26</td>
<td>15</td>
<td>93</td>
<td>8</td>
<td>1</td>
<td>23</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>26</td>
<td>15</td>
<td>93</td>
<td>8</td>
<td>1</td>
<td>23</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>26</td>
<td>15</td>
<td>39</td>
<td>8</td>
<td>1</td>
<td>23</td>
<td>93</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>26</td>
<td>1</td>
<td>39</td>
<td>8</td>
<td>15</td>
<td>23</td>
<td>93</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>8</td>
<td>1</td>
<td>39</td>
<td>26</td>
<td>15</td>
<td>23</td>
<td>93</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>15</td>
<td>39</td>
<td>26</td>
<td>54</td>
<td>23</td>
<td>93</td>
</tr>
</tbody>
</table>

The heap property is maintained by repeatedly swapping the value at index `i` with its largest child until the property is satisfied.
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.
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

\[ x = 2^{h-1} + 2 \cdot 2^{h-2} + 3 \cdot 2^{h-3} + \ldots + h \cdot 2^0 \]  

(1)

\[ \Rightarrow 2x = 2^h + 2 \cdot 2^{h-1} + 3 \cdot 2^{h-2} + \ldots + h \cdot 2^1 \]  

(2)

Subtracting (1) from (2) gives

\[ x = 2^h + 2^{h-1} + 2^{h-2} + \ldots + 2^1 - h \]

\[ \leq 2^{h+1} \]

\[ = 2(n + 1) = O(n). \]