lecture21: Heaps and Sorting

Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

16th July, 2013
Announcements

- lab_avl due Thursday night (7/18)
- mp6.1 extra credit due Friday night (7/19)
Say we get an array of random elements. What is the easiest way to turn it into a heap so we can use it as a priority queue?

- Sort it?
- Something else?

Of course, we will have to consider the running time. We don’t want to do anything inefficient!
We know that a sorted array satisfies the heap property. So in order to write buildheap, we’ll just call our favorite sorting algorithm!

```cpp
template <class T>
void Heap<T>::buildHeap()
{
    mergeSort(_elems);
    // or...
    quickSort(_elems);
    // or...
    bogoSort(_elems);
}
```
Can we use functions that already exist in the heap class? We don’t need a fully-sorted array to be our heap.

```cpp
template <class T>
void Heap<T>::buildHeap()
{
}
```

This buildHeap uses repeated inserts into the heap. Running time: $O(n)$ heapifyUps, each taking $O(\log n)$ gives a $O(n \log n)$ running time, the same as mergesort!

But in addition... this is in-place! We don’t need to allocate any extra memory.
Can we make a similar buildHeap with heapifyDown?

```cpp
template <class T>
void Heap<T>::buildHeap()
{
}
```

What is the running time of this? We know that it is proportional to the height of the subtree that heapifyDown is being called on...
buildHeap: heapifyUp vs heapifyDown

The worst case for heapifyDown is when it’s called at the root; then, it has to do at worst $h$ operations. The worst case for heapifyUp is when it’s called on every single leaf and it has to travel to the top; this is about $\frac{n}{2} \cdot h$ operations.

We already know that using heapifyUp will run in $O(n \log n)$ time. The running time using heapifyDown is proportional to the sum of the heights of the subtrees it is called on.

Let’s model this as a recurrence: $S(h) = 2 \cdot S(h - 1) + h$, $S(0) = 0$. I claim the solution is $2^{h+1} - h - 2$. What is the running time of buildHeap if this is true?
Proving buildHeap is linear

\[ S(h) = 2 \cdot S(h - 1) + h, \quad S(0) = 0. \] Solution: \( 2^{h+1} - h - 2. \)

Proof of solution to recurrence. Consider an arbitrary \( h \geq 0. \)

- Case 1: \( h = 0, \) \( S(0) = 2^{0+1} - 0 - 2 = 0. \) \( \checkmark \)

- Case 2: \( h > 0, \) by an IH that says \( \forall j < h, S(j) = 2^{j+1} - j - 2, \) we know \( S(h - 1) = 2^h - h - 1. \) So

\[
S(h) = 2 \cdot S(h - 1) + h = 2(2^h - h - 1) + h = 2^{h+1} - 2 - h. \] \( \checkmark \)

The number of nodes \( n \) is proportional to \( 2^{h+1} - 2 - h \)
In terms of number of nodes

\[ h \leq \log n \]
\[ h + 1 \leq \log n + 1 \]
\[ 2^{h+1} \leq 2^{\log n + 1} \]
\[ 2^{h+1} - 2 - h \leq 2^{\log n + 1} - 2 - \log n \]

Simplifying,
\[ 2^{h+1} - 2 - h \leq 2^{\log n + 1} - 2 - \log n = n - 1 - \log n = O(n) \]
Sorting with a heap

How could you implement a sorting algorithm using a heap?

```cpp
template <class T>
Vector<T> heapsort(const Vector<T> & toSort) {
    Vector<T> sorted;
    Heap<T> h(toSort); // constructs heap from vector

    while(!h.empty())
        sorted.push_back(h.pop());

    return sorted;
}
```

What if we don’t want to allocate extra memory?
Sorting in-place within the heap class

// assume we have a valid heap

while(_size > 0)
{
}

reverse(_elems);  // is this in-place?
_size = elems.size();  // set correct value of _size
Runtime analysis of heapSort

1. buildHeap: $O(n)$
2. heapifyDown $n$ times: $O(n \log n)$
3. reverse: $O(n)$

Final running time: $O(n \log n)$. 
Why is heapsort interesting?

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>In-place?</th>
<th>Stable?</th>
<th>Average</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mergesort</td>
<td>No</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quicksort</td>
<td>Yes</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heapsort</td>
<td>Yes</td>
<td>No</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The top $k$ elements

Describe an algorithm that given an unsorted array of length $n$, returns the top $k$ elements, where $k \ll n$. It must run in time better than $O(n \log n)$.

Use a heap! This can give you either

- $O(n) + O(k \log n)$, or
- $O(n \log k)$

depending on your algorithm.

We can actually do this in linear time if we use an algorithm called quickselect (discussed more during mp6 and CS 473).