COMP 250

Lecture 24

heaps 3

Nov. 4, 2016
RECALL: min Heap (definition)

Complete binary tree with (unique) comparable elements, such that each node’s element is less than its children’s element(s).
Heap index relations

\[
\text{parent} = \text{child} / 2 \\
\text{left} = 2 \times \text{parent} \\
\text{right} = 2 \times \text{parent} + 1
\]
Given a list with size elements:

```plaintext
buildHeap(list){
    create new heap array  // length > list.size
    for (k = 0; k < list.size; k++)
        add( list[k] )  // add to heap[ ]
}
```
$2^{level} \leq i < 2^{level} + 1$

$level \leq \log_2 i < level + 1$

Thus, $level = \text{floor}(\log_2 i)$
Worse case of buildHeap

\[
t(n) = \sum_{i=1}^{n} \text{number of swaps for node } i
\]

= \sum_{i=1}^{n} \text{level of node } i

= \sum_{i=1}^{n} \text{floor}(\log_2 i)
$t(n) = \sum_{i=1}^{n} floor(\log_2 i)$

Area under the dashed curve is the total number of swaps (worst case) of buildHeap.
\[
\frac{1}{2} n \log_2 n \leq t(n) \leq n \log_2 n
\]
Thus, worst case: \(\text{buildHeap} \text{ is } O(n \log_2 n)\)

This is a tight \(O(\ )\) bound.

Today, I will show you a \(O(n)\) algorithm.
Given a list with size elements:

```java
buildHeapFast(list){
    copy list into a heap array
    for (k = size/2; k >= 1; k--)
        downHeap( k, size )
}
```
downHeap(3, 6)
downHeap(3, 6)
downHeap( 2, 6 )
downHeap( 1, 6 )
**buildHeapFast** (list){
    copy list into a heap array
    for (k = size/2;  k >= 1;  k--)
        downHeap( k, size )
}

Claim: this algorithm is O(n).

Intuition for why this algorithm is so fast?
buildheap algorithms

last lecture
upHeap based

today
downHeap based
We tend to draw binary trees like this:

But the number of nodes doubles at each level. So we should draw trees like this:
buildheap algorithms

last lecture

Most nodes swap $h$ times in worst case.

today

Few nodes swap $h$ times in worst case.
How to show buildHeapFast is $O(n)$?

Worst case number of swaps needed to add node $i$ is the height of that node.

(Recall the height of a node is the length of the longest path from that node to a leaf.)

$$t(n) = \sum_{i=1}^{n} \text{height of node } i$$
Worse case of buildHeapFast?

How many elements at level $l$? $(l \in 0, \ldots, h)$

What is the height of each level $l$ node?
Worse case of buildHeapFast?

level $l$ has $2^l$ elements, $l \in 0, \ldots, h$

level $l$ nodes have height $h - l$.

t(n) = \sum_{i=1}^{n} \text{height of node } i

= ?
Worse case of buildHeapFast?

level \( l \) has \( 2^l \) elements, \( l \in 0, \ldots, h \)

level \( l \) nodes have height \( h - l \).

\[
t(n) = \sum_{i=1}^{n} \text{height of node } i
\]

\[
= \sum_{l=0}^{h} (h - l) \cdot 2^l
\]
\[ t_{\text{worst case}}(h) = \sum_{l=0}^{h} (h - l) \ 2^l \]

\[ = h \sum_{l=0}^{h} 2^l - \sum_{l=0}^{h} l \ 2^l \]

Easy \quad \text{(number of nodes)} \quad \text{(sum of node depths)}
I have removed the next two slides which derived a closed form expression for the second summation (the difficult one). Please see lecture notes for a slightly different derivation, which is simpler.
\[ t_{\text{worst case}}(h) = \sum_{l=0}^{h} (h - l) \ 2^l \]

\[ = h \sum_{l=0}^{h} 2^l - \sum_{l=0}^{h} l \ 2^l \]

\[ = h(2^{h+1} - 1) - (h - 1)2^{h+1} - 2 \]

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

In terms of \( n \), we have

\[ t(n) = n - \log(n + 1) \]
Summary: buildheap algorithms

last lecture

\[ O(n \log_2 n) \]

today

\[ O(n) \]