Data Structures and Algorithms

Spring 2009-2010

Outline

Outline

Building the Heap: `build_heap()`

- Make n successive insertions to the heap
- Have seen that each insert takes $O(\log n)$ wost time; so n inserts can certainly be done in $O(n \log n)$-time
- Have mentioned that each insert takes $O(1)$ average time
- Thus, with no other intervening operations, we have $O(n)$ *average* time for the n inserts
- Can we get $O(n)$ worst-case time as well? Yes.
- Algorithm:
  - Place the n data in the array (in any order)
  - Perform following function:

```
for (i = n/2; i > 0; i--)// i > n/2 ==> i is leaf
  trickle_down(i);     //  trickle only non-leaves
```

Building the Heap: `build_heap()` (contd.)

- We will now show this algorithm gives a linear-time (worst case) solution to building a heap
- When the data are placed arbitrarily in the array/tree, the furthest distance that a datum will move is from its initial position to the lowest level
- A node on level i can be trickled down its height, $h - i$ times
- Therefore, the total no. of trickle-downs is at most the sum of the heights of all nodes in tree
- Consider the case if the tree was *perfect*
- We will patch up the deficiencies afterwards

Building the Heap: `build_heap()` (contd.)

Theorem

For a perfect binary tree of height h containing $n = 2^{h+1} - 1$ nodes, the sum of the heights of nodes is $2^{h+1} - 1 - (h+1)$

Proof:

A perfect binary tree has 2^i nodes on level i; a node on level i has depth i and height $h - i$. The sum of the heights of all these is

$$\begin{aligned} S &= \sum_{i=0}^{h} 2^i (h-i) \\ &= h + 2(h-1) + 4(h-2) + \cdots + 2^{h-1}(1) \\ 2S &= 2h + 4(h-1) + 8(h-2) + \cdots + 2^h(1) \end{aligned}$$

Subtracting these we get

Building the Heap: `build_heap()` (contd.)

$$\begin{aligned} 2S - S &= -h + 2 + 4 + \cdots + 2^h \\ S &= (2^{h+1} - 1) - (h + 1) \end{aligned}$$

Now,

$$n = 2^{h+1} - 1 \Rightarrow h + 1 = \log(n + 1)$$

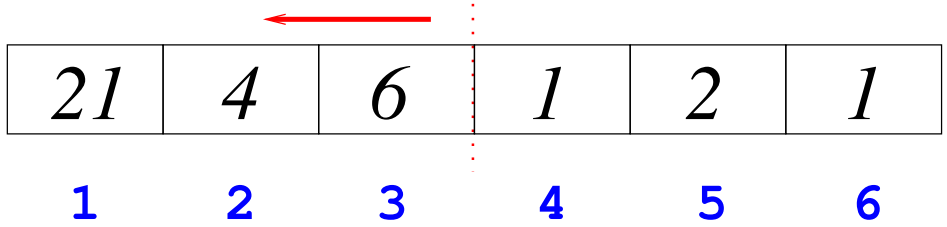
Thus, the sum of the heights for a perfect tree is

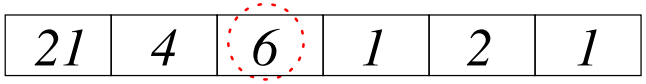
$$S \approx n - \log(n + 1) \approx n$$

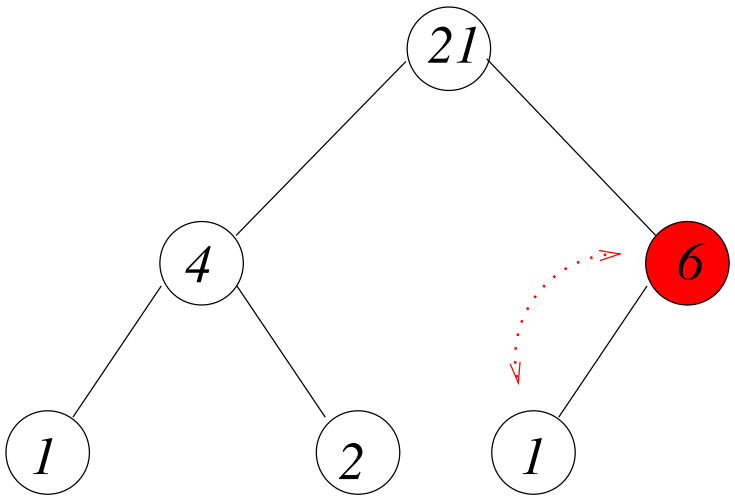
which is linear in the number of nodes in the tree. That is, $S = O(n)$.
In the general case any **complete** tree will have fewer nodes than a perfect tree – $2^h <= n < 2^{h+1}$.
So the result must hold for a complete tree, also.

Building the Heap: `build_heap()` (contd.)

Example

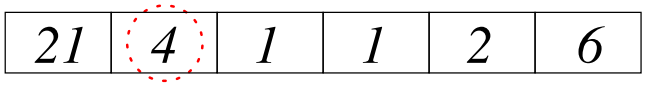


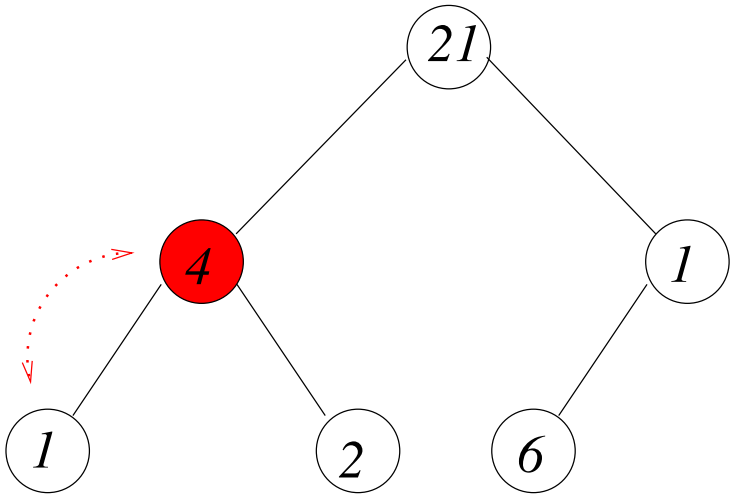




Building the Heap: `build_heap()` (contd.)

Example

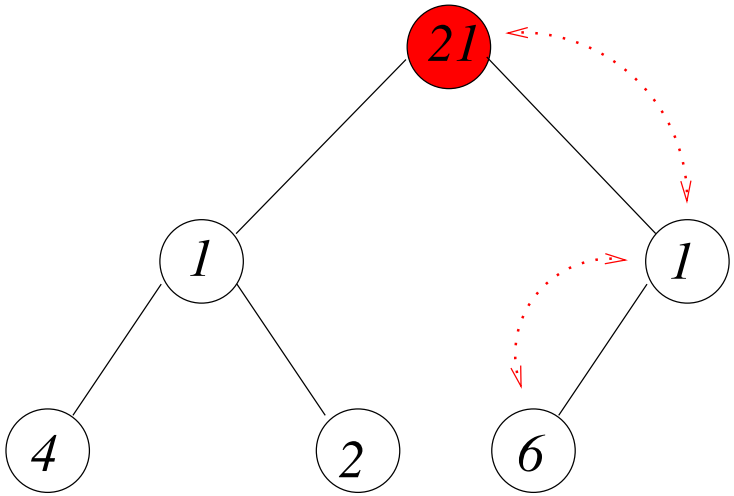




Building the Heap: `build_heap()` (contd.)

Example

21	1	1	4	2	6



1	1	6	4	2	21

Outline

d-Heaps

- In a *d*-heap, a node has *d* children instead of 2
- Since the tree is shallower, inserts are faster, but...
- ...since we now have to compare against *d* children at each level, deletions are more expensive: running time becomes $O(d \log_d n)$
- Also, fast multiplication and division by 2 (left shift or right shift) no longer is useful
- *d*-heaps are useful when the number of inserts hugely outweighs the number of deletions

Introduction

- How can we efficiently sort the elements in an array, $arr[1 \ldots n]$?
- We look at **quadratic** algorithms firstly and discover their inherent flaw

Outline

Insertion Sort

- comprises $n - 1$ passes
- at end of ith pass, elements $[1 \ldots i]$ will be sorted
- Insertion sort code here.
- ➜ note use of an insert at location $a[j]$ (last line of code) to avoid (many) three-assignment swaps

Insertion Sort: Running-Time Analysis

- In worst case the jth element will have to be moved all the way to position 1 for each iteration of i
- Number of iterations:

$$\sum_{i=2}^{n} i = 2 + \cdots + n = O(n^2)$$

- Alternative viewpoint: let $T(n)$ be time required to insertion-sort an n-element array

$$T(n) = T(n-1) + n, \qquad T(1) = 0$$

- This bound is *tight* (actually achieved) if input is sorted in *reverse* order; therefore, insertion sort is $\Theta(n^2)$
- If input is correctly sorted, running time is $O(n)$

Lower Bounds on (Simple) Sorting

- Any algorithm that relies on exchanges of adjacent elements (transpositions) will require $\Omega(n^2)$ time to sort its input
- Transpositions are closely linked to **inversions** in a list of numbers
- An inversion is a pair of elements that are out of sequence
- The list $L = (23, 12, 45, 8, 96, 7)$ has 9 inversions: 23/12, 23/8 23/7, 12/8, 12/7, 45/8, 45/7, 96/7, 8/7
- Since each of these inversions will have to be removed for the list to be sorted, the number of inversions will be exactly the number of swaps required to sort the list

Lower Bounds on (Simple) Sorting (contd.)

- If all of the elements of the list are *unique*, we can show that there are on average $O(n^2)$ inversions in the list:
  - Consider any pair of elements, l_i and l_j in a list, L
  - If l_i and l_j are inverted then in the *reversed* list L_r, l_i and l_j will not be inverted
  - Similarly, if the pair weren't inverted, in the reversed list they will be inverted
  - Thus, every pair of elements will be an inversion in either the list L or its reversal, L_r
  - There are $\binom{n}{2} = n(n-1)/2 \approx n^2$ pairs of elements so on average half are in the list L and half in the reversal, L_r
- Therefore, the number of swaps required to sort a list is $\Omega(n^2)$ since two adjacent elements get transposed only if they are inverted and the number of transpositions (swaps) required is the number of inversions in the initial list, L
- ➜ This argument is valid for *any* algorithm that works by swapping *adjacent* elements