Introduction to Algorithms and Data Structures

Tutorial 4

tutor name

University of Edinburgh

5th-8th November, 2024

Q1: binary search

```
binarySearch(A,key,i,j):
    if j−1 = i
        if A[i].key = key
            return A[i].value
        else FAIL
    else
        k = ⌊ i+j/2 ⌋
        if key < A[k].key
            return binarySearch(A,key,i,k)
        else return binarySearch(A,key,k,j)
```

Q1 (a): recurrence for binary search

Case $j - i = 1$: we execute exactly 3 lines (whether our search succeeds or fails)

Case $j - i > 1$: we do 5 line executions, *as well as* one of the recursive calls **binarySearch**(A,key,i,k) or **binarySearch**(A,key,k,j).

The subproblems are size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ respectively.
In the worst case the recursive call has size $\lceil n/2 \rceil$.
This leads us to the recurrence:

$$\begin{aligned} T(1) &= 3 \\ T(n) &= T(\lceil n/2 \rceil) + 5 \text{ when } n > 1 \end{aligned}$$

(Don't worry about fine details of what counts as a line execution)

Q1 (b): Master theorem for binary search

Simplifying to asymptotic terms, and forgetting the ceiling, we get

$$\begin{aligned} T(1) &= \Theta(1) \\ T(n) &= T(n/2) + \Theta(1) \quad \text{when } n > 1 \end{aligned}$$

Right format for the Master Theorem, with $a = 1$, $b = 2$, $k = 0$.

Since $b^k = 2^0 = 1 = a$, we are in the 'middle case' of the theorem.

Hence $T(n) = \Theta(n^k \lg n) = \Theta(\lg n)$

Q1 (c): Master theorem exercises

i. $T(n) = 2T(n/3) + \Theta(n)$

Here $a = 2, b = 3, k = 1$. So $b^k = 3$ and so $a < b^k$. Hence

$$T(n) = \Theta(n^k) = \Theta(n).$$

ii. $T(n) = 7T(n/2) + \Theta(n^2)$

Here $a = 7, b = 2, k = 2$.

So b^k is 4, giving $a > b^k$, and so

$$T(n) = \Theta(n^{\log_2 7}) = \Theta(n^{2.807\ldots}).$$

(note: Strassen's amazing algorithm for matrix mult. has this recurrence/asymptotics)

Q1 (c): Master theorem exercises

iii. $T(n) = 2T(n/4) + \Theta(\sqrt{n})$

Here $a = 2, b = 4, k = 1/2$.

Evaluate $\log_b a = \log_4 2 = 1/2$, and we see $\log_a b = k$, so

$$T(n) = \Theta(\sqrt{n} \lg n).$$

Discussion of differences between how we solved i. and ii.:

- Advantage for i., ii. that we didn't need to work-out a (possibly untidy) $\log$ calculation to determine which "case" we were in
- Advantage for iii., is that we can work out the "critical exponent" $c = \log_b a$ (which may be needed anyway) and directly compare to the "extra work" exponent k.

Q2: Heap worked exercise

Start with an empty Heap.

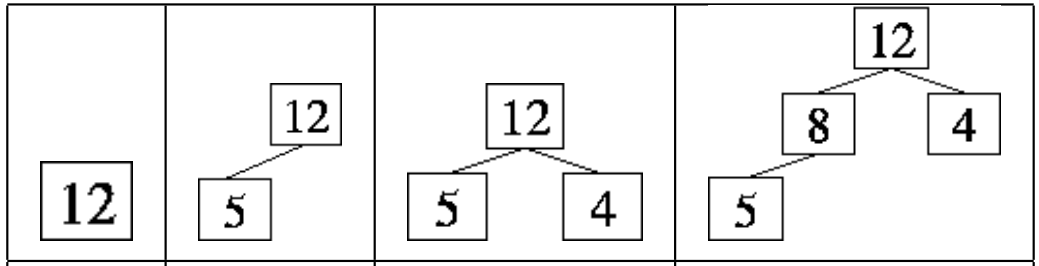
Sequence of insertions $\{12, 5, 4, 8, 9, 1, 16, 20, 7, 6\}$.

Take each item one-by-one, and add it via Max-Heap-Insert.

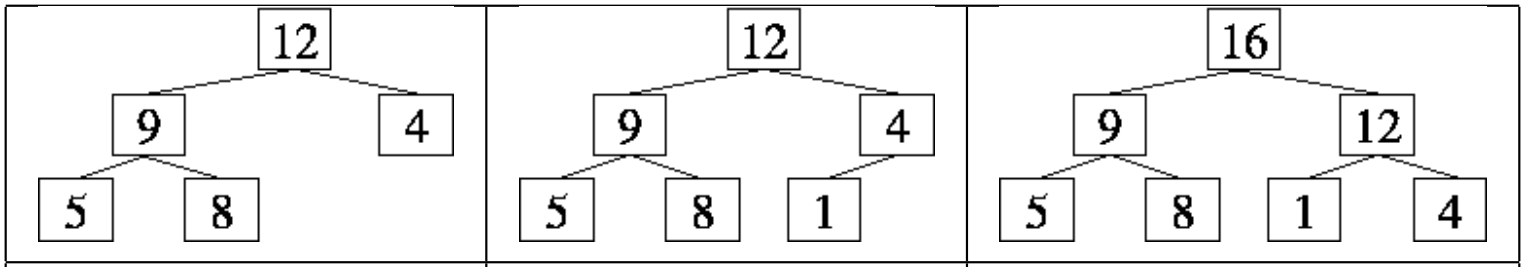
Pictures show the result *after* each Max-Heap-Insert.
Under each figure is written the array representation of the heap.

- For anyone with a tablet, or comfortable with online whiteboards, will be helpful to show a "bubbling-up" /
- For example, when 8 goes in, it first is added as the left child of 5, which at that point is the first available leaf node, and then it gets swapped with its parent 5.

Q2: Heap worked exercise

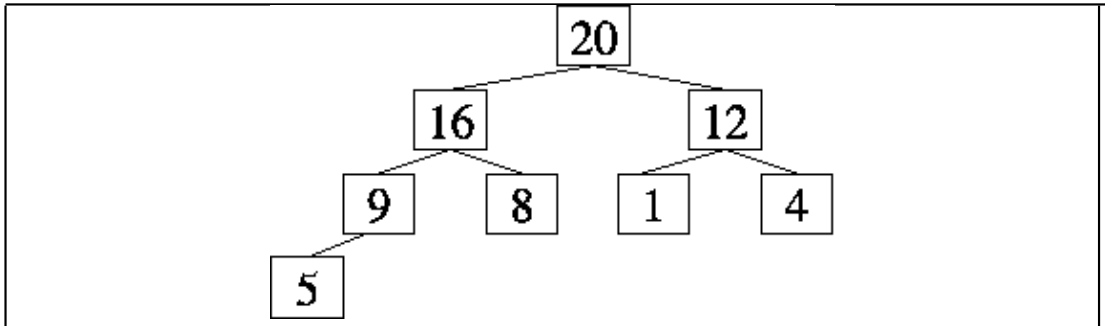


12	12,5	12,5,4	12,8,4,5

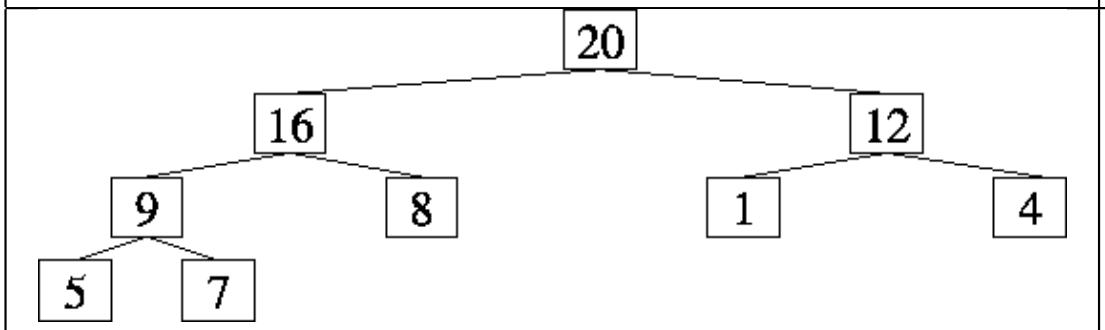


12,9,4,5,8	12,9,4,5,8,1	16,9,12,5,8,1,4

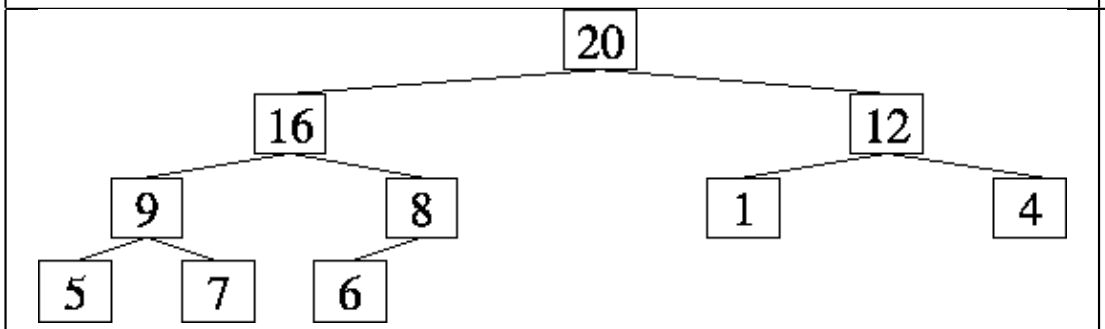
Q2: Heap worked exercise



20,16,12,9,8,1,4,5



20,16,12,9,8,1,4,5,7



20,16,12,9,8,1,4,5,7,6

Q3: Analysis of Insertion sequence

Show that when we consider a list of items in sorted order (smallest first) that it will take time $\Omega(n \log(n))$ to insert them into an initially empty heap.

Observation: If the new key being inserted exceeds the values of all items already stored in the heap, Max-Heap-Insert will take time $\Theta(h) = \Theta(\lg(m))$, where h is the current height of the heap, and m current number of items.

why? (because the new item will need to be swapped all the way up to the root of the heap).

Q3: Analysis of Insertion sequence

For an increasing sequence n items, "bigger than all items in the Heap" is true on *every* insertion.

Total time is:

$$\sum_{i=1}^{n} \Omega(\lg(i)) \geq \Omega\Big(\sum_{i=\lceil \frac{n}{2} \rceil}^{n} \lg(i) \Big) \qquad \text{drop the smallest } \log \left\lfloor \frac{n}{2} \right\rfloor \text{terms}$$

$$\geq \Omega\Big(\sum_{i=\lceil \frac{n}{2} \rceil}^{n} \lg\left(\frac{n}{2}\right) \Big) \qquad \text{remaining terms all at least } \frac{n}{2}$$

$$\geq \frac{n}{2} \Omega(\lg\left(\frac{n}{2}\right)) \in \Omega(n \lg n)\,,$$

so the time taken to insert the list $(1, 2, \ldots, n)$ into an initially empty heap is $\Omega(n \lg n)$.

Q3: comparison against BuildHeap

Why does this differ from the $\Theta(n)$ running-time for Build-Max-Heap on the input array?

Both Max-Heapify and Max-Heap-Insert have worst-case running-time proportional to height of the current Heap.

Difference is the number of times the operation is called for each heap size.

Let $h = \lg(n)$ for final heap size n.

Build-Max-Heap ...

height:	h	$h-1$	$h-2$	.....	1
♯calls	1	2	4	.....	$\sim n/4$

Max-Heap-Insert with $\{1, \ldots, n\}$ ...

height:	$\lg(n)$	$\lg(n-1)$	$\lg(n-2)$	.....	1
♯calls	1	1	1	.....	1

Q3: comparison against BuildHeap

Why does this differ from the $\Theta(n)$ running-time for Build-Max-Heap on the input array?

Note $\lg(n) - 1 = \lg(n/2)$.

Each of the values $\lg(n-1), \lg(n-2), \ldots \lg(n/2)$ are of height $\geq h - 1$

So the calls during the sequence of Insertions are distributed like ...

height:	h	$h-1$	$h-2$	.....	1
♯calls	1	n/2 -1	n/4	.....	1

(technically the headings should be $h, \geq h-1, \geq h-2 \ldots$)

Build-Max-Heap ...

height:	h	$h-1$	$h-2$	.....	1
♯calls	1	2	4	.....	$\sim n/4$

For the Insertions, calls are skewed to *large* heaps, with buildHeap, the opposite.

Q4: index methods

$$\begin{aligned} \text{Parent}(i) &= \left\lceil \frac{i}{2} \right\rceil - 1 \\ \text{Left}(i) &= 2i + 1 \\ \text{Right}(i) &= 2i + 2 \end{aligned}$$

(assuming the items exist of course. Have pre-conditions of $i > 0$, $2i + 1 <$ *heap_size* and $2i + 2 <$ *heap_size* respectively)

Q5: Python implementation of Heaps

This is a discussion question - check out the resource and notice things.

Download a recent release of Python, for example: `https://www.python.org/downloads/release/python-3130/`

Once unpacked, look for the file `heapq.py` in the subdirectory Lib, and open it (with any text viewer).