EECS 3101 A: Design and Analysis of Algorithms

Suprakash Datta
Office: LAS 3043

Course page: `http://www.eecs.yorku.ca/course/3101A`
Also on Moodle

Sorting Algorithms

- Searching in databases: we can do binary search on sorted data
- A large number of computer graphics and computational geometry problems Closest pair, element uniqueness
- A large number of sorting algorithms are developed representing different algorithm design techniques.
- A lower bound for sorting $\Omega(n \log n)$ is used to prove lower bounds of other problems.

Sorting Algorithms - 2

- Insertion Sort: Worst-case running time $\Theta(n^2)$; in-place
- **Selection Sort**
  for $i = n$ downto 2
  A: Find the largest element among $A[1..i]$
  B: Exchange it with $A[i]$
  Running time: $\Theta(n^2)$, in place
- Merge Sort: Worst-case running time $\Theta(n \log n)$, but needs $\Theta(n)$ additional memory (WHY?)

Idea for improvement: use a data structure, to do both A and B in $O(\lg n)$ time, balancing the work, achieving a better trade-off, and a total running time $O(n \log n)$.

Heap Sort

Binary Max-Heap:

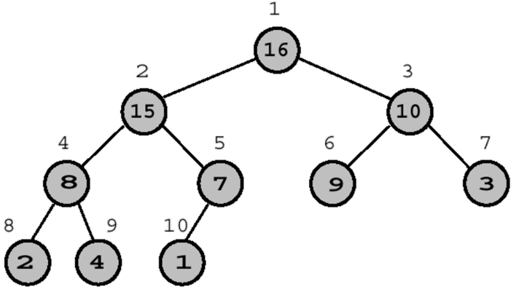
- A simple linear array
- Can be viewed as a nearly complete binary tree; All levels, except the lowest one are completely filled
- Two attributes: $length(A)$, $heapSize(A)$
- Max-Heap property: The key in root is greater or equal than all its children, and the left and right subtrees are again binary heaps
- Insertion, Deletion, Extract-max all take $O(\log n)$ time

Max Heaps

Parent(i): $\lfloor i/2 \rfloor$

LeftChild(i): $2i$

RightChild(i): $2i + 1$



1	2	3	4	5	6	7	8	9	10
16	15	10	8	7	9	3	2	4	1

Building Heaps - Part 1

Problem: Left, right subtrees are heaps, root violates heap property

```
n is total number of elements
HEAPIFY(A, i)
  ▷ Left & Right subtrees of i are heaps.
  ▷ Makes subtree rooted at i a heap.
  l ← LEFT(i)          ▷ l = 2i
  r ← RIGHT(i)         ▷ r = 2i + 1
  if l ≤ n and A[l] > A[i]
     then largest ← l
     else largest ← i
  if r ≤ n and A[r] > A[largest]
     then largest ← r
  if largest ≠ i
     then exchange A[i] ↔ A[largest]
          HEAPIFY(A, largest)
```

Heapify - Correctness

- Use induction on the height of the tree
- Base Case: $h = 1$
- Inductive Step: The root is exchanged with a node that is the largest among its descendants. The unaffected subtree is a heap. Inductively the other subtree is heapified.

Heapify - Running Time

- How large can a subtree be (in terms of number of nodes)?
  Claim: $\leq 2n/3$
- $T(n) \leq T(2n/3) + \Theta(1)$
- $T(n) = O(\log n)$ (WHY?)
- Alternatively, in the worst case one execution per level.... $O(\log n)$ time

Building Heaps - Part 2

Problem: Given any array $A[1..n]$, convert it to a heap

```
BUILD-HEAP(A)
1 for i ← ⌊n/2⌋ downto 1
2       do HEAPIFY(A, i)
```

- Elements in the subarray $A[(\lfloor n/2 \rfloor + 1)...n]$ are already 1-element heaps because they are leaf nodes
- Correctness: strong induction on i
- all trees rooted at $m > i$ are heaps, so heapify makes the subtree rooted at element i a heap
- Notice the reversal in direction of the induction

BuildHeap - Analysis

- Running time (less than n calls to Heapify):
  $\leq nO(\lg n) = O(n \lg n)$
- Good enough for an $O(n \lg n)$ bound on Heapsort, but sometimes we build heaps for other reasons, would be nice to have a tight bound
- Intuition: for most of the time Heapify works on smaller than n element heaps

BuildHeap - Tighter analysis

- Idea: Heapify runs in $O(h)$ time, where h is the height of a node
- How many nodes are there at height h?
  Answer: $2^{\lceil \lg n \rceil - h}$
- Assume a complete binary tree. Then the running time is

$$\sum_{h=1}^{\lfloor \lg n \rfloor} h 2^{\lceil \lg n \rceil - h} = 2^{\lceil \lg n \rceil} \sum_{h=1}^{\lfloor \lg n \rfloor} h 2^{-h}$$

- Claim: $\sum_{h=1}^{\lfloor \lg n \rfloor} h 2^{-h} = \Theta(1)$, and so running time of BuildHeap is $O(n)$

BuildHeap - Proof of Claim

$$
\begin{aligned}
\sum_{h=1}^{\infty} x^h &= \frac{1}{1-x} \text{ if } |x| < 1 \\
\sum_{h=1}^{\infty} h x^{h-1} &= \frac{1}{(1-x)^2} \text{ after differentiation wrt } x \\
\sum_{h=1}^{\infty} h x^{h} &= \frac{x}{(1-x)^2} \text{ multiplying by } x \\
\sum_{h=1}^{\infty} h 2^{-h} &= \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} \text{ substituting } x = \frac{1}{2} \\
&= 2
\end{aligned}
$$

Therefore the finite sum is $\Theta(1)$.

HeapSort

HEAPSORT(A)	Analysis
1 BUILD-HEAP(A)	??
2 **for** $i \leftarrow n$ **downto** 2	n times
3 **do** exchange $A[1] \leftrightarrow A[i]$	O(1)
4 $n \leftarrow n - 1$	O(1)
5 HEAPIFY($A, 1$)	O(lg n)

The total running time of heap sort is $O(n \lg n) +$ Build-Heap(A) time, which is $O(n)$, total $O(n \lg n)$

HeapSort: Correctness

LI: Before iteration i, $A[i+1..n]$ consists of the $n-i$ numbers originally in A but in sorted order and $A[1..i]$ is a heap, and consists of the rest of the numbers originally in A

- Initialization: Because we proved BuildHeap correct, after line 1, $A[1..n]$ is a valid heap
- Maintenance: in iteration i the max is exchanged with the last element of the heap and the heap length is shortened by 1.
  So $A[i..n]$ consists of the $n-i+1$ numbers originally in A but in sorted order.
  Meanwhile, in $A[1..i-1]$ the preconditions for Heapify are met – only the root violates the heap property.
  So heapify makes $A[1..i-1]$ a heap again

HeapSort: Correctness - 2

LI: Before iteration i, $A[i+1..n]$ consists of the $n-i$ numbers originally in A but in sorted order and $A[1..i]$ is a heap, and consists of the rest of the numbers originally in A

- Termination: The loop terminates at $i=1$.
- Plugging $i=1$ in the LI we get $A[2..n]$ consists of the $n-1$ numbers originally in A but in sorted order and $A[1..1]$ is a heap, and consists of the rest of the numbers originally in A
- This implies that the array is sorted

HeapSort: Observations

- Heap sort uses a heap data structure to improve selection sort and make the running time asymptotically optimal
- Running time is $O(n \log n)$ – like merge sort, but unlike selection, insertion, or bubble sorts
- Sorts in place – like insertion, selection or bubble sorts, but unlike merge sort

QuickSort

Characteristics

- sorts in place, i.e., does not require an additional array, like insertion sort
- Divide-and-conquer, like merge sort
- very practical, average sort performance $O(n \log n)$ (with small constant factors), but worst case $\Theta(n^2)$
- CAVEAT: this is true for the CLRS version

QuickSort: Strategy

Divide-and-conquer

- Divide: partition array into 2 subarrays such that elements in the lower part $\leq$ elements in the higher part
- Conquer: recursively sort the 2 subarrays
- Combine: trivial since sorting is done in place

QuickSort: Algorithm

```
Partition(A,p,r)
x←A[r]
i←p-1
j←r+1
while TRUE
    repeat j←j-1
        until A[j] ≤x
    repeat i←i+1
        until A[i] ≥x
    if i<j
        then exchange A[i]↔A[j]
        else return j
```

```
Quicksort(A,p,r)
if p<r
    then q←Partition(A,p,r)
         Quicksort(A,p,q)
         Quicksort(A,q+1,r)
```

QuickSort: Correctness

- Prove Partition correct using loop invariants
- Use induction to prove QuickSort correct

QuickSort: Analysis

- Assume that all input elements are distinct
- The running time depends on the distribution of splits
- Best case: Partition always splits the array evenly.
  $T(n) = 2T(n/2) + \Theta(n)$, implying $T(n) = \Theta(n \log n)$ using Case 2 of the Master Theorem
- Worst case: One side of the PARTITION has only one element.
  $T(n) = T(1) + T(n-1) + \Theta(n) = T(n-1) + \Theta(n)$.
  So $T(n) = \sum_{i=1}^{n} \Theta(i) = \Theta(n^2)$

QuickSort: Worst case

- When does the worst case appear?
- When the input is sorted!
- The running time depends on the distribution of splits
- Same recurrence for the worst case of insertion sort
- However, sorted input yields the best case for insertion sort!

Randomized QuickSort: Intuition

- Suppose the split is $1/10 : 9/10$
- $T(n) = T(n/10) + T(9n/10) + \Theta(n)$, so $T(n) = \Theta(n \log n)$
- How can we make sure that we are usually lucky? Partition around a random element (works well in practice)
- Randomized algorithms
  - running time is independent of the input ordering
  - no specific input triggers worst-case behavior
  - the worst-case is only determined by the output of the random-number generator

Randomized QuickSort: Steps

- Assume all elements are distinct
- Partition around a random element
- Randomization is a general tool to improve algorithms with bad worst-case but good average-case complexity