Sorting and Heaps

Eugene Weinstein
New York University
Computer Science Department
Data Structures Fall 2008
Conceptual Merge Sort

• Let A[1..n] be the array to be sorted

• Merge(A1, A2) takes two sorted arrays and merges them into one sorted array

• MergeSort(A[1..n])

  • if (n > 1)

    • A1 ← MergeSort(A[1..(n/2)])

    • A2 ← MergeSort(A[(n/2+1)..n])

  • return Merge(A1, A2)
More Real Implementation

• Merge sort in place in array A

• Merge(A, tmpA, i, j, k) // Uses tmpA to merge A[i,j] and A[j+1, k] into A[i, k]

• MergeSort(A, left, right)
  • center ← (left + right)/2
  • MergeSort(A, tmpA, left, center)
  • MergeSort(A, tmpA, center+1, right)
  • Merge(A, tmpA, left, center+1, right)
Merge Sort Analysis

• Let us assume that \( n \) is a power of 2
• Assumption is without loss of generality (WLOG) - i.e., even if \( n \) is an arbitrary positive integer, this still works
• At each level of the recursion tree, we deal with \( n \) numbers
• For a MergeSort call of size \( n \), we make two calls of size \( n/2 \)
• A Merge of size \( n \) takes linear time, say \( n \)
• MergeSort of size 1 takes constant time
Recurrence Relation

• The running time for a MergeSort of size $n$ can be written recursively. This is called a recurrence relation:

• $T(n) = 2T(n/2) + n$ ; $T(1) = 1$

• Divide through by $n$: \[
\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + 1, \quad \frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + 1, ...
\]

• Now sum together:

\[
\frac{T(n)}{n} + \frac{T(n/2)}{n/2} + \cdots + \frac{T(2)}{2} = \frac{T(n/2)}{n/2} + \frac{T(n/4)}{n/4} + \cdots + \frac{T(1)}{1} + 1
\]

• Cancel terms and collect: \[
\frac{T(n)}{n} = \frac{T(1)}{1} + \log n
\]

• Worst-case MergeSort runtime:

\[
T(n) = n \log n + n = O(n \log n)
\]
Heap Review

• We are concerned with min-heaps
  • Complete binary trees obeying the heap property:
    • key(parent) \leq key(child)
• Heaps can be implemented efficiently with an array
• Array encoding of a binary tree:
  • index(root) = 1 (not 0)
  • index(left child) = 2 \times index(parent)
  • index(right child) = 2 \times index(parent) + 1
PercolateDown

- percolateDown(A, hole)
  - tmp ← A[hole]
  - while (hole*2 ≤ currentSize)
    - child ← hole * 2
      - child ← child + 1
    - else done
  - hole ← child
PercolateUp

• percolateUp(A, hole)

• \( \text{tmp} \leftarrow A[\text{hole}] \)

• while (\( \text{hold} > 1 \))
  • parent \( \leftarrow \) hole / 2
  • if (\( A[\text{child}] > \text{tmp} \))
    • \( A[\text{hole}] \leftarrow A[\text{parent}] \)
  • else done
  • hole \( \leftarrow \) parent
Heap Building

• Given n numbers, can we come up with an efficient algorithm to produce a heap of these n numbers?

• How about just making a new heap and calling insert n times?
  • Each insert is $O(\log n)$, so total complexity is $O(n \log n)$

• Can we do better? Perhaps $O(n)$?

• What if the n numbers are already given an array, can we efficiently rearrange the array to put it in heap order?
Array to Heap

• Here’s an idea:
• for i = 1..n
  • percolateUp(i)
• Let’s think about algorithm invariant of above algorithm
  • Invariant: Array A[1..i] is a heap
  • n percolateUp operations, each takes O(log n) time
  • So complexity is still O(n log n)
• How can we improve on this?
Better Efficiency

• Now, consider reversing the loop direction: for \( i = n \) to 1
• Think of the array as a collection of mini-heaps
  • \( A[n/2..n] \) are single-item heaps, no need to percolateDown
  • for \( i = n/2 \) to 1
    • percolateDown(i)
• Big realization: loop in percolateDown() only runs once because for any \( i \), all the trees in \( A[i+1..n] \) are valid heaps
Heap Building Analysis


• Invariant: for any i, all the trees in A[i+1..n] are valid heaps
• percolateDown(i) makes at most h(i) comparisons
  • h(i) is the height of ith node from the bottom of the tree
• Runtime on tree of size i is bounded by T(i), sum of the heights of all nodes in complete binary tree of height i

\[ T(i) = \sum_{j=i}^{n} h(j) \]

• Tree of height i consists of two trees of height i-1 and a root:

\[ T(i) = 2T(i-1) + i \]

• Solution to this recurrence: \( T(i) = 2^{i+1} - i - 2 \)
Heap Building Analysis

- The runtime complexity of heap building of a heap of height $h$ is at most
  \[ T(h) = 2^{h+1} - h - 2 \]

- In a complete binary tree of height $h$, there are $2^{h+1} - 1$ nodes

- So the runtime is $O(n)$