Transform and Conquer – representation change: Heapsort

- Reading: Chapter 6, Section 6.4
- Exercises:
  - 6.4: 1, 2, 3a, 6, 10

Next time: Space-time tradeoffs (Chapter 7)

Transform and Conquer

Solve problem by transforming into:
- a more convenient instance of the same problem (instance simplification)
  - presorting
  - Gaussian elimination
- a different representation of the same instance (representation change)
  - balanced search trees
  - heaps and heapsort
  - polynomial evaluation by Horner’s rule
  - Fast Fourier Transform
- a different problem altogether (problem reduction)
  - reductions to graph problems
  - linear programming

Heapsort

Definition:

A heap is a binary tree with the following conditions:

1. It is essentially complete:

2. The key at each node is ≥ keys at its children
Definition implies:

1. Given \( n \), there exists a unique binary tree with \( n \) nodes that is essentially complete, with \( h \leq \lceil \log n \rceil \)
2. The root has the largest key
3. The subtree rooted at any node of a heap is also a heap

Heapsort Algorithm:

1. Build heap
2. Remove root—exchange with last (rightmost) leaf
3. Fix up heap (excluding last leaf)

Repeat 2, 3 until heap contains just one node.

Heap construction

1. Insert elements in the order given breadth-first in a binary tree
2. Starting with the last (rightmost) parental node, fix the heap rooted at it, if it does not satisfy the heap condition:
   1. exchange it with its largest child
   2. fix the subtree rooted at it (now in the child’s position)

Example: 2 3 6 7 5 9
Root deletion

The root of a heap can be deleted and the heap fixed up as follows:

1. Exchange the root with the last leaf.
2. Compare the new root (formerly the leaf) with each of its children and, if one of them is larger than the root, exchange it with the larger of the two.
3. Continue the comparison/exchange with the children of the new root until it reaches a level of the tree where it is larger than both its children.

Representation

1. Use an array to store breadth-first traversal of heap tree:
2. Example:
   - Left child of node $j$ is at $2j$
   - Right child of node $j$ is at $2j + 1$
   - Parent of node $j$ is at $\lfloor j / 2 \rfloor$
3. Parental nodes are represented in the first $\lfloor n / 2 \rfloor$ locations.

Bottom-up heap construction algorithm

Algorithm: HeapBottomUp$[H[1..n]]$

1. Constructs a heap from the elements of a given array
2. By the bottom-up algorithm
3. Input: An array $H[1..n]$ of orderable items
4. Output: A heap $H[1..n]$
5. for $i \leftarrow \lfloor n / 2 \rfloor$ downto 1 do
6. $k \leftarrow i$
7. $v \leftarrow H[k]$
8. heap $\leftarrow$ false
9. while not heap and $2 \times k \leq n$ do
10. $j \leftarrow 2 \times k$
11. if $j < n$ then
12. if $H[j] < H[j + 1]$ then
13. $j \leftarrow j + 1$
14. else
15. heap $\leftarrow$ true
16. $H[k] \leftarrow H[j]$
17. $k \leftarrow j$
18. $H[k] \leftarrow v$
Analysis of Heapsort

See algorithm HeapBottomUp in section 6.4

1. Fix heap with “problem” at height $j$: $2^j$ comparisons
2. For subtree rooted at level $i$ it does $2(h-i)$ comparisons
3. Total for heap construction phase:
   \[ \sum_{i=0}^{h-1} 2(h-i) 2^i = 2( n - \lg(n + 1)) = \Theta(n) \]
   
   \# nodes at level $i$

Analysis of Heapsort (continued)

Recall algorithm:
\[ \Theta(n) \]
1. Build heap
2. Remove root—exchange with last (rightmost) leaf
\[ \Theta(\log n) \]
3. Fix up heap (excluding last leaf)
   
   Repeat 2, 3 until heap contains just one node.
   
   $n - 1$ times

Total: \[ \Theta(n) + \Theta(\log n) = \Theta(n \log n) \]

*Note: this is the worst case. Average case also \( \Theta(n \log n) \).

Priority queues

1. A priority queue is the ADT of an ordered set with the operations:
   * find element with highest priority
   * delete element with highest priority
   * insert element with assigned priority
2. Heaps are very good for implementing priority queues
Insertion of a new element

1. Insert element at last position in heap.
2. Compare with its parent and if it violates heap condition exchange them.
3. Continue comparing the new element with nodes up the tree until the heap condition is satisfied.

Example:

Efficiency:

Bottom-up vs. Top-down heap construction

1. **Top down**: Heaps can be constructed by successively inserting elements into an (initially) empty heap.
2. **Bottom-up**: Put everything in and then fix it.
3. Which one is better?