CMPT 225

Lecture 26 – Heap Sort
Last Lectures

- We saw how to ...
  - Define minimum and maximum binary heaps
  - Demonstrate and trace their recursive operations
At the end of the next few lectures, a student will be able to:

- Define the following data structures:
  - Binary search tree
  - Balanced binary search tree (AVL)
  - Binary heap
  as well as demonstrate and trace their operations
- Implement the operations of binary search tree and binary heap
- Implement and analyze sorting algorithms: tree sort and heap sort
- Write recursive solutions to non-trivial problems, such as binary search tree traversals and heap sort
Today’s menu

- Our goal is to
  - Understand how heap sort works
  - Sort an array using heap sort
  - Analyze time/space efficiency of heap sort
Remember Selection Sort

- Sorting algorithm in which, at every iteration, we “selected” the smallest (or largest) element of the unsorted section of our array and “move” it into the sorted section of our array
Heap Sort

- Sorting algorithm in which, at every iteration, we “select” the smallest or min (largest or max) element of the unsorted section of our array and “move” it into the sorted section of our array.

- Use minimum binary heap in order to sort an array of data in descending order.

- Use maximum binary heap in order to sort data in ascending order.
Overview of Heap Sort Algorithm

To sort an array of n elements:
1. Interpret the array as a binary tree in contiguous storage
   - Such a tree is always complete
   - But it may not be a heap
2. Phase 1: Convert the binary tree (array) into a heap, i.e., heapify!
   Phase 2: Sort the heap

The heap sort algorithm consists of two phases
Heap Sort Algorithm

Phase 1
1. Let \( \text{index} \) be the index of the last parent node in the tree
2. While \( \text{index} \) >= 0
   
   reHeapDown( \( \text{index} \) )
   
   \( \text{index} \) --

Phase 2. (heap can be seen as the unsorted section of array)
1. Set counter unsorted to \( n \) (\( \text{unsorted} \) -> size of heap)
2. Store the last element of the heap into temporary storage \( \text{lastElement} \)
3. Move the root to the last position in the heap
4. Decrease counter unsorted to exclude the last entry from further sorting (can be seen as the first element in sorted section of array)
5. Insert \( \text{lastElement} \) into root (position now available)
6. reHeapDown( \( \text{indexOfRoot} \) ) between positions 0 and \( \text{unsorted} - 1 \)
7. Repeat steps 2 to 6 until \( \text{unsorted} \) is 1
ReHeapDown( )  (Min heap)

reHeapDown( indexOfRoot )
  if ( heap[indexOfRoot] is not a leaf )
    Set indexOfMinChild to index of smallest child of root
    if ( heap[indexOfRoot] > heap[indexOfMinChild] )
      Swap heap[indexOfRoot] with heap[indexOfMinChild]
    reHeapDown( indexOfMinChild )
ReHeapDown( )  (Max heap)

reHeapDown( indexOfRoot )

if ( heap[indexOfRoot] is not a leaf )
    Set indexOfMaxChild to index of largest child of root
    if ( heap[indexOfRoot] < heap[indexOfMaxChild] )
        Swap heap[indexOfRoot] with heap[indexOfMaxChild]
    reHeapDown( indexOfMaxChild )
Demo of Heap Sort - 1

- Phase 1

ARRAY: UNSORTED, HEAP
Demo of Heap Sort - 2

Phase 2

Iteration #1

Heap - Unsorted

Sorted

No longer a heap

So reheap "heap" part of array
Demo of Heap Sort - 3

iteration #2

HEAP - UNSORTED

2nd SMALLEST

last Element

ReHeapDown

DO UNTIL ARRAY IS SORTED!
Let’s try!
Time Complexity Analysis of Heap Sort

Phase 1
- $O(n)$

Phase 2
- $O(n \log n)$

Hence in the worst case the overall time complexity of the heap sort algorithm is:

$$\max[O(n), O(n \log n)] = O(n \log n)$$
Space Complexity Analysis of Heap Sort

- How much space (memory) does heap sort require to execute (aside from original data collection)?

Therefore, its space efficiency is ->
Summary

**Sorting Efficiency**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Comparisons</th>
<th>Swaps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble</td>
<td>n(n-1)</td>
<td>n(n-1)</td>
</tr>
<tr>
<td>Selection</td>
<td>n(n-1) / 2</td>
<td>n-1</td>
</tr>
<tr>
<td>Insertion</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sort</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Recursive:**
  - Worst: O(n^2)
  - Average: O(nlog_2 n)

- **Iterative:**
  - O(log_2 n)

- **Tree:**
  - Average: O(nlog_2 n)
  - Worst: O(n^2)

- **Merge:**
  - Recursive: O(nlog_2 n)
  - Iterative: O(nlog_2 n)

- **Quick:**
  - Recursive: O(nlog_2 n)
  - Iterative: O(n)

- **Heap:**
  - Recursive: O(nlog_2 n)
  - Iterative: O(1)

**Adv.: Simple**

- **time:** O(n^2)
- **Complexity:** O(n^2)
- **Space:** O(1)
- **“in-place”**

**Conclusions:**

- Good for small n
- O(n) for the tree
- Maybe out of memory if n is very large
- Good for large n

- Fastest
- Tree Sort
- Merge Sort
- Quick Sort
- Heap Sort
√ Learning Check

- We can now ...
  - Understand how heap sort works
  - Sort an array using heap sort
  - Analyze time/space efficiency of heap sort
Next Lectures

- Hashing