CMPT 225
Heap Sort
Last Lecture

- Binary Heaps -> Min and Max
Unit Learning Outcomes

At the end of this unit, a student will be able to:

- **Abstract Data Types (ADTs):**
  - define binary heaps
    and demonstrate, trace and implement their operations
  - given a set of requirements, select the most appropriate data collection ADT class (and its most appropriate underlying data structure), taking into account its time and space efficiency
    - Binary heap may be selected as an ADT or as a data structure

- **Sorting:**
  - implement and analyze heap sort

- **Recursion:**
  - write recursive definitions of binary heap
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 “selected” 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
Heap Sort Algorithm

**Phase 1**
- Let \( index \) be the index of the last parent node in the tree
- While \( index \) >= zero
  - reHeapDown( \( index \) )
  - decrement \( index \)

**Phase 2** (heap can be seen as the unsorted section of array)

1. Set counter \( unsorted \) to \( n \) (\( unsorted \rightarrow \) size of heap)
2. Store the last element of the heap into temporary storage \( lastElement \)
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 \( lastElement \) into root (position now available)
6. reHeapDown(indexOfRoot) between positions 0 and \( unsorted - 1 \)
7. Repeat steps 2 to 6 until \( unsorted \) is 1
Demo of Heap Sort

- Phase 1

ARRAY: UNSORTED, 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

Phase 2

1. Unsorted = n

iteration #1

3. Smallest

4. Unsorted --

5. Last element

NO LONGER A HEAP

6. So reheap "heap" part of array

Heap - Unsorted

Sorted
Demo of Heap Sort

iteration #2

2nd SMALLEST
HEAP - UNSORTED

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

- Understood how heap sort works
- Sorted an array using heap sort
- Analyzed time/space efficiency of heap sort
Next Lecture

- Hashing