Chapter 6: Heapsort

Introduction

- Heapsort is an $\Theta(n \lg n)$ worst-case sorting algorithm that runs in-place, so it requires only $\Theta(1)$additional memory to run. It combines the best features of insertion sort and unlike merge sort.
- Algorithm design technique: using a data structure to manage information. Heapsort works by using a heap data structure. This data structure is useful for managing priority queues as well for several other algorithms.

Max-Heaps

- Definition:
  To be a binary max-heap, two conditions need to be satisfied.

  1. It should be a **nearly complete binary** tree, that is a binary tree where all levels, except the last level, must be full and all nodes in the last level need to be as far left as possible. This is to map it easily to an array
  2. The value of a node should be greater than or equal to its children.

```
Ex:      91                  52                  91
        /  \                /  \                /  \
      77    46            77    46            77    46
     /  \     \          / \    /            / \    /
   69    3     11      69   3  11          69   3  11
   not a max-heap      not a max-heap       a max-heap
```

- Array representation for a max-heap:
  Assume array index starts at 1. Let heap-size[A] stands for the number of elements in the heap stored in the array A.
  That is, A[1...heap-size[A]] stores the heap and the root of the heap is stored in A[1].
  The parent-child relationship between two nodes are represented by the following formulas.

  Given a node at array index i, Parent(i) = $\lfloor i/2 \rfloor$
  Left(i) = $2i$
  Right(i) = 2i + 1

  The example max-heap in this page can be represented in an array as

91	77	46	69	3	11

- The height h of a heap with n nodes: $h = \Theta(\lg n)$.

- **Ex. 6.1-1** A heap with height h will have the minimum and maximum of nodes as follows.
  Minimum of $n = 1 + 2 + 2^2 + \ldots + 2^{h-1} + 1 = 2^h$
  Maximum of $n = 1 + 2 + 2^2 + \ldots + 2^h = 2^{h+1} - 1$
  From the above two equations, we can derive $h = \Theta(\lg n)$.

- **Recommended Exercises**: Ex. 6.1-3, 6.1-4, 6.1-5, 6.1-6

Maintaining the Heap Property

```
Max-Heapify(A, i) // heapification downward
   Pre-condition: Both the left and right subtrees of node i are max-heaps
                   and i is less than or equal to heap-size[A]
   Post-condition: The subtree rooted at node i is a max-heap
1. l = Left(i)
2. r = Right(i)
3. if l <= A.heap-size and A[l] > A[i]
4.    largest = l
5. else largest = i
6. if r <= A.heap-size and A[r] > A[largest]
7.    largest = r
8. if largest != i
9.    exchange A[i] with A[largest]
10.   Max-Heapify(A, largest)
```

```
Ex:         38                                               38
          /    \                                           /    \
        29      22                                       65      22
       /  \    /  \   --> call Max-Heapify(A, 2) -->    /  \    /  \
     52    65 12   9                                  52    31 12   9
    / \    /                                         / \    /
  26   7  31                                       26   7  29
```

- Running time analysis for Max-Heapify(A, i):
  The element A[i] will be swapped down along a tree path. Thus, the running time for the procedure is $O(h)$, where $h = \Theta(\lg n)$ is the tree height

- **Recommended Exercises**: Ex. 6.2-1, 6.2-2, 6.2-3, 6.2-4.

Building a Max-Heap:

Using bottom-up approach to convert an array A[1...n] to a max-heap by calling a sequence of Max-Heapify procedures, starting at the last non-leaf node and ended at the root (backward).

```
Build-Max-Heap(A)
1. A.heap-size = A.length
2. for i = A.length/2 downto 1  //sip the leaves
3.     do Max-Heapify(A, i)
```

```
Ex:      46                46                46                91
       /    \            /    \            /    \            /    \
     11      77  -->   11      91  -->   69      91  -->   69      77
    /  \    /         /  \    /         /  \    /         /  \    /
  69    3  91       69    3  77       11    3  77       11    3  46
```

- Running time analysis for Build-Max-Heap (A):
  Call Max-Heapify about $n/2$ times $\implies$ Build-Max-Heap (A) takes $O(n \lg n)$.
  Actually, $O(n \lg n)$ is an asymptotically upper bound but not tight. The tight upper bound is $O(n)$.
- **Recommended Exercise**: Ex. 6.3-1, 6.3-2.

The Heapsort Algorithm

1. Make the input array A to a max-heap by calling Build-Max-Heap(A) procedure. We know A[1] stored the largest element.
2. Exchange A[1] $\leftrightarrow$ A[heap-size[A]] and then decrement heap-size[A] by one.
3. Call Max-Heapify(A, 1) to re-heapify A[1..heap-size[A]].
4. Repeatedly perform Step 2 and Step 3 until heap-size[A] = 1.

```
Heapsort(A)
1. Build-Max-Heap(A)
2. for i = length[A] downto 2
3.     do exchange A[1] with A[i]
4.        A.heap-size = A.heap-size - 1
5.        Max-Heapify(A, 1)
```

- **Run-time analysis of Heapsort**: Heapsort calls Build-Max-Heap(A) once and calls Max-Heapify(A) $n-1$ times. Thus, the running time is $O(n \lg n)$.
- Heapsort uses a special data structure to solve a problem.
- Heapsort is very similar to selection sort - in each iteration, pick the largest element in the remaining set of elements and put it to the correction position (the "last" position). The difference is that they use different ways to pick the largest element.

Priority Queues

- A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key. It allows us to access elements in order of the priority represented by the key value.
- A max-priority queue supports the insert, maximum, extract-max and increase-key operations efficiently.
- A min-priority queue supports the insert, minimum, extract-min and decrease-key operations efficiently.
- The root of a max-heap contains the largest value. If we build a priority queue using a max-heap based on the priority values of elements, then the next element to be extracted from the queue is always located at the root. Similarly we can use a min-heap to build a min-priority queue.
- *Applications*: job scheduler, event-driven simulator.

```
Heap-Maximum(A)
//O(1) time
1. return A[1]

Heap-Extract-Max(A)
//O(lg n) time
1. if A.heap-size < 1
2.    then error -- heap underflow
3. max = A[1]
4. A[1] = A[A.heap-size]
5. A.heap-size--
6. Max-Heapify(A, 1)
7. return max
```

```
Heap-Increase-Key(A, i, key)
//O(lg n) time
1. if key < A[i]
2.    then error -- new key must be larger than current key
3. A[i] = key
4. while i > 1 and A[Parent(i)] < A[i]
5.       exchange A[i] and A[Parent(i)]
6.       i = Parent(i)

Max-Heap-Insert(A, key)
//O(log n) time
1. heap-size[A]++
2. A[A.heap-size] = negative-infinity
3. Heap-Increase-Key(A, A.heap-size, key)
```

- **Recommended Exercises**: Ex 6.5-1, 6.5-2, 6.5-3, 6.5-4, 6.5-7, 6.5-8.
- Think-pair-share: Ex 6.5-7: Show how to implement a FIFO queue with a priority queue. Show how to implement a stack using a priority queue.
- **Implementation issues**
  - A priority queue can be implemented by extending a heap implementation based on an array or based on a binary tree structure using pointers.
  - In an actual application, we will have objects with key values and satellite data. That requires some careful refactoring of the pseudo-code above.
  - Review PriorityQueue class from Java docs.