Chapter 6: Heapsort

Introduction

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

Heaps

- Definition: The **(binary) heap** data structure is an array object with the following two properties:
  - The heap can be viewed as 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.
  - The values in the heap satisfy a heap property. For a max-heap, the value at a node is $\geq$ the value at its child node(s). For a min-heap, the value at a node is $\leq$ the value(s) at these child node(s).
- We can map a heap to the indices of an array. An array $A[1:n]$ that represents a heap is an object with attribute $A.\textit{heap-size}$, which represents how many elements in the heap are stored in the array. That is, only the elements in $A[1:A.\textit{heap-size}]$, where $0 \leq A.\textit{heap-size} \leq n$, are valid elements of the heap. Then, the root of the heap is at $A[1]$.
- For a give node at index i, we can find its parent, left and right child with the following one-line procedures.

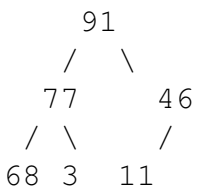
```
PARENT(i)
1. return i/2 // integer division

LEFT(i)
1. return 2i

RIGHT(i)
1. return 2i + 1
```

Advanced note In most languages, $PARENT(i) = i/2$ can be written as $i >> 1$ (shift right by 1 bit). Similarly, $LEFT(i) = 2i$ can be written as $i << 1$ (shit left by 1 bit) and $RIGHT(i) = 2i+1$ can be written as $i << 1 \mid 1$ (shift left by followed by bitwise or-ing 1 to add 1). This provides efficient implementations of these basic operations.

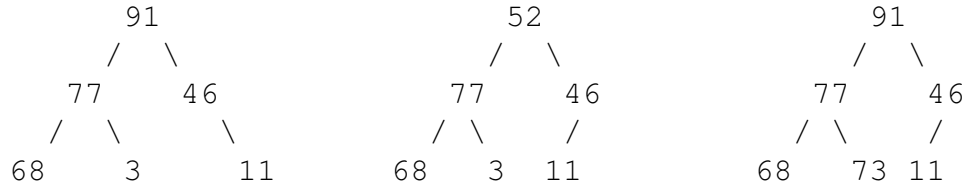
- Now we can restate the max-heap property as: For every node i other than the root, $A[\text{PARENT}(i)] \geq A[i]$. The largest element in a max-heap is at the root.
- Similarly, we can restate the min-heap property as: For every node i other than the root, $A[\text{PARENT}(i)] \leq A[i]$. The smallest element in a min-heap is at the root.
- Here is an example of a max-heap.



- The heap above represented as an array (root is $A[1]$)

91	77	46	68	3	11

- **In-class Exercise**. Which of the following are max-heaps and which are not?



- The height h of a heap with n nodes: $h = \Theta(\lg n)$.
- **Ex. 6.1-1 and 6.1-2** A heap with height h will have the minimum and maximum number of nodes n as follows.

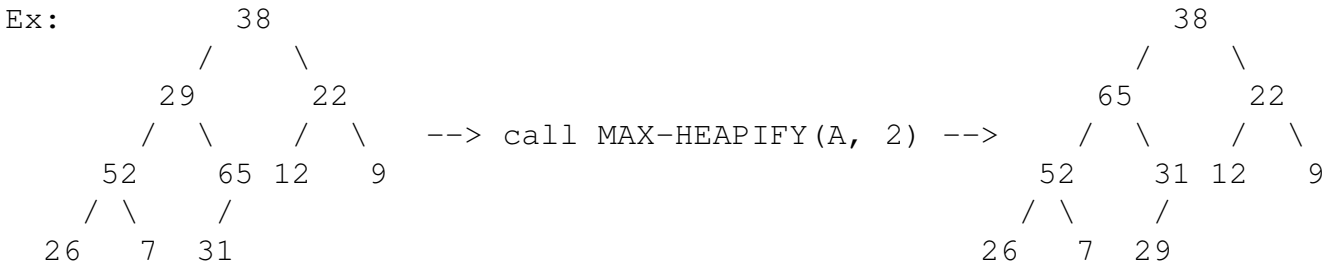
  Minimum $n = 1+2+2^2+\ldots+2^{h-1}+1 = 2^h$
  Maximum $n = 1+2+2^2+\ldots+2^h = 2^{h+1}-1$
  From the above two equations, we can derive $h = \Theta(\lg n)$.

- **Recommended Exercises**:
  - Ex 6.1-3: Show that in any subtree of a max-heap, the root of the subtree contains the largest element in that subtree.

- Ex 6.1-4: Where in a max-heap might the smallest element reside, assuming that all elements are distinct?
- Ex 6.1-5: Is an array that is sorted in increasing order a min-heap?
- Ex 6.1-8: Show that, with the array representation for storing an n-element heap, the leaves are the nodes indexed by $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$.

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 A.heap-size
   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)
```



- 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

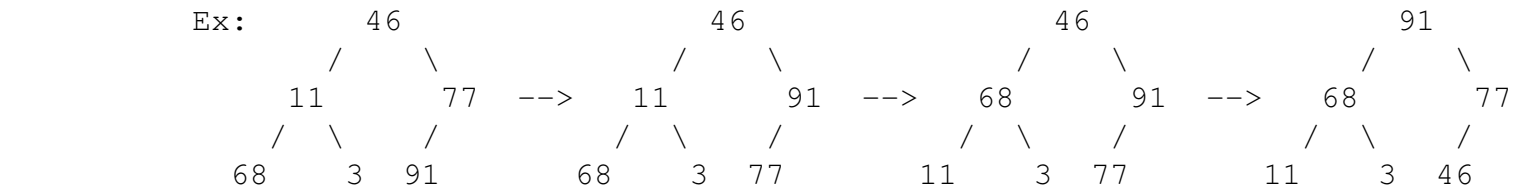
- **Recommended Exercises**:
  - Ex 6.2-1: Illustrate the operation of MAX-HEAPIFY(A,3) on the array A = (27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0).
  - Ex 6.2-3: What is the effect of calling MAX-HEAPIFY on a node i that is larger than its children?

- Ex 6.2-4: What is the effect of calling MAX-HEAPIFY on a leaf node?
- Ex 6.2-5: Write pseudocode for the procedure MIN-HEAPIFY, which performs the same operation as MAX-HEAPIFY but for min-heaps.

Building a 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 ending at the root (works backwards on the array).

```
BUILD-MAX-HEAP(A)
1. A.heap-size = n
2. for i = n/2 downto 1  //skip the leaves
3.     MAX-HEAPIFY(A, i)
```



- **Loop invariant**: At the start of each iteration of the **for** loop of lines 2–3, each node $i+1, i+2, \ldots, n$ is the root of a max-heap.
- **Initialization**: Prior to the first iteration, the subarray $A[\lfloor n/2 \rfloor + 1 \ldots n]$ contains all the leaves of the tree, each of which is a max-heap of size 1. Thus, the loop invariant holds prior to the first iteration.
- **Maintenance**: During the i-th iteration, the procedure MAX-HEAPIFY(A, i) is called. By the pre-condition of MAX-HEAPIFY, both the left and right subtrees of node i are max-heaps. Thus, after the call to MAX-HEAPIFY(A, i), the subtree rooted at node i is a max-heap. Therefore, at the end of the i-th iteration, each node $i, i+1, i+2, \ldots, n$ is the root of a max-heap, and the loop invariant holds.
- **Termination**: At termination, i = 0. Thus, by the loop invariant, each node $1, 2, \ldots, n$ is the root of a max-heap. In particular, node 1 is the root of the entire max-heap.
- Running time analysis for Build-Max-Heap (A):

  Call MAX-HEAPIFY about $n/2$ times $\Longrightarrow$ Build-Max-Heap (A) takes $O(n \lg n)$.

  Actually, $O(n \lg n)$ is an asymptotic upper bound but it is not tight. The tight upper bound is $O(n)$. The analysis is a bit complex and is in the textbook (Section 6.3) but we will skip it.

- **Recommended Exercises**:
  - Ex 6.3-1: Illustrate the operation of BUILD-MAX-HEAP on the array A = (5, 3, 17, 10, 84, 19, 6, 22, 9).
  - Ex 6.3-3: Why does the loop index i in line 2 of BUILD-MAX-HEAP start at $\lfloor n/2 \rfloor$ and not at n?

The Heapsort Algorithm

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

```
HEAPSORT(A)
// array A[1:n] is unsorted
1. BUILD-MAX-HEAP(A)
2. for i = n  downto 2
3.      exchange A[1] with A[i]
4.      A.heap-size = A.heap-size - 1
5.      MAX-HEAPIFY(A, i)
```

- **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.
- **Loop invariant**: At the start of each iteration of the **for** loop of lines 2–5, the subarray $A[1:i]$ is a max-heap containing the i smallest elements of $A[1:n]$, and the subarray $A[i+1:n]$ contains the $n-i$ largest elements of $A[1:n]$, sorted.
- **Recommended Exercises**: 6.4-1, 6.4-2.
  - **In-class Exercise 6.4-1**: Illustrate the operation of HEAPSORT on the array A = (4, 1, 3, 9, 7).
  - Ex 6.4-2: Argue the correctness of HEAPSORT using the loop invariant given above.

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, discrete event simulation, and many others

```
MAX-HEAP-MAXIMUM(A)
//O(1) time
1. if A.heap-size < 1
2.    error "heap underflow"
3. return A[1]

MAX-HEAP-EXTRACT-MAX(A)
//O(lg n) time
1. max = MAX-HEAP-MAXIMUM(A)
2. A[1] = A[A.heap-size]
3. A.heap-size = A.heap-size - 1
4. MAX-HEAPIFY(A, 1)
5. return max
```

Unlike the textbook, we will assume that the position of the element, whose key we want to increase, is known. In most common heap applications, this is the case. Thus we don't need to maintain an index mapping from elements to their positions in the heap.

```
MAX-HEAP-INCREASE-KEY(A, i, newKey)
//O(lg n) time
//Increase the key of A[i] to the newKey value
1. if newKey < A[i].key
2.    error "new key must be larger than current key"
3. A[i].key = newKey
```

```
4. while i > 1 and A[PARENT(i)].key < A[i].key
5.       exchange A[i] and A[PARENT(i)]
6.       i = PARENT(i)

MAX-HEAP-INSERT(A, x, n)
//O(log n) time
1. if A.heap-size == n
2.    error "heap overflow"
3. A.heap-size = A.heapsize + 1
4. k = x.key
5. x.key = -infinity //Integer.MIN_VALUE, for example
6. A[A.heap-size] = x
7. MAX-HEAP-INCREASE-KEY(A, A.heap-size, k)
```

- **Recommended Exercises**: Ex 6.5-1, 6.5-2, 6.5-3, 6.5-4, 6.5-8.
- Think-pair-share: Ex 6.5-9: 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. An array based implementation is typical.
  - In an actual application, we will have objects with key values and satellite data. The key values may be a combination of multiple attributes of an object so we can only use `compareTo` method calls to compare keys. That requires some careful refactoring of the pseudo-code above.
  - Review PriorityQueue class from Java docs.