

05. Heaps, Heapsort, and the Sorting Lower Bound

CPSC 535 ~ Spring 2019

Kevin A. Wortman



CALIFORNIA STATE UNIVERSITY
FULLERTON

September 23, 2019



Heaps

“Heap” data structure in general

- ▶ like binary search tree, but weaker order condition
- ▶ find min/max fast, usually $\Theta(1)$
- ▶ insert/delete-min/max fast, $\Theta(\log n)$ or even $\Theta(1)$
- ▶ search slow, $\Theta(n)$

Applications

- ▶ operating system scheduler
- ▶ accelerate algorithms: selection sort (heapsort), Prim's MST algorithm, Dijkstra's shortest paths algorithm, others

Big Picture

Big idea in computer science: automata with simple local behavior can have amazing effects when assembled at scale

- ▶ CPU instructions → sophisticated software
- ▶ neural network nodes → teachable intelligent system
- ▶ hypertext pages → the web
- ▶ compromised network hosts → distributed denial of service
- ▶ (today): tree nodes w/ different order and balance invariant
→ organize large datasets efficiently in different ways

Kinds of Heaps

Like binary search trees and hash tables, there are many competing variations of heap

- ▶ binary heap (covered here)
- ▶ Fibonacci heap
- ▶ quake heap
- ▶ hollow heap

Heap memory is a disparate concept and unfortunate naming clash

Binary Heap Order Invariant

Let parent node p have children l, r

Recall **BST invariant**: $l.key < p.key < r.key$

Binary **max-heap invariant**: $p.key \geq l.key$ and $p.key \geq r.key$
(sketch)

Implies

1. BST is totally sorted, but loosely balanced
2. heap-order is “half-baked” sort; can tell parent comes before children, but siblings are unsorted relative to each other
3. root must be overall maximum; that one element is perfectly sorted
4. tradeoff: heap can be more tightly balanced than BST

Max-Heap vs. Min-Heap

Max-heap: maximum key at root; $p.key \geq l.key, p.key \geq r.key$

Min-heap: minimum key at root; $p.key \leq l.key, p.key \leq r.key$

No significant difference in implementation; swap $<$ for $>$ in **if** statements

Max-heap is more convenient for sorting in non-decreasing order, so we'll focus on that

(Min-heap is more convenient for non-increasing sort, Prim, Dijkstra)

Binary Heap Balance Invariant

Height of a node: distance from bottom (root is highest)

Depth of a node: distance from root (leaves are deepest)

Heap is nearly-perfectly-balanced

- ▶ every level except the bottom is completely full
- ▶ bottom level: full on the left side, may be missing elements on the right side (sketch)
- ▶ (N.B. insisting on a completely perfect tree is impractical because it requires $(n + 1)$ is a power of 2)
- ▶ \implies tree height $\Theta(\log n)$

Arrayed Binary Heap

Arrayed structure: packs something into an array

- ▶ locality of reference \implies good cache performance
- ▶ only one allocate/free per structure lifetime \implies fast constant factors

Arrayed max-heap:

- ▶ partially-filled array $A[1, \dots, n]$ stores k heap elements in $A[1, \dots, k]$ for $k \leq n$
- ▶ root/maximum always in $A[1]$
- ▶ $PARENT(i) = \lfloor i/2 \rfloor$
- ▶ $LEFT(i) = 2i$
- ▶ $RIGHT(i) = 2i + 1$
- ▶ $PARENT, LEFT, RIGHT$ are $\Theta(1)$ and ordinarily only 1-2 CPU instructions each

Create Heap Operation

1: **function** CREATE-MAX-HEAP(A)

Require: A is an array of size n that may become a heap

Ensure: A is a valid, empty, heap

2: $A.heapsize = 0$

3: **end function**

(Trivial pseudocode, but still worthwhile to write this down so that each operation is encapsulated and crystal clear.)

Clearly $\Theta(1)$ time

Find-Max Operation

1: **function** MAX-HEAP-MAXIMUM(A)

Require: A is a valid, non-empty heap

Ensure: returns the maximum key in A

2: **return** $A[1]$

3: **end function**

Again, straightforward and clearly $\Theta(1)$ time.

Max-Heapify Intro.

MAX – HEAPIFY(A, i)

- ▶ assuming *LEFT*(i) and *RIGHT*(i) obey the order invariant, ensure i obeys the invariant
- ▶ $A[i]$ might be OK, or might need to “float down” deeper
- ▶ Delete-max and build are easy once we have *MAX – HEAPIFY*
- ▶ $\Theta(\log n)$ time

Max-Heapify Pseudocode

```
1: function MAX-HEAPIFY( $A, i$ )  
Require:  $A$  is a heap,  $A[LEFT(i)]$  and  $A[RIGHT(i)]$  are heap-ordered  
Ensure:  $A[i]$  is heap-ordered  
2:    $l = LEFT(i), r = RIGHT(i)$   
3:   if  $l \leq k$  and  $A[l] > A[i]$  then  
4:      $largest = l$   
5:   else  
6:      $largest = i$   
7:   end if  
8:   if  $r \leq k$  and  $A[r] > A[largest]$  then  
9:      $largest = r$   
10:  end if  
11:  if  $largest \neq i$  then  
12:     $swap(A[i], A[largest])$   
13:     $MAX-HEAPIFY(A, largest)$   
14:  end if  
15: end function
```

Max-Heapify Analysis

- ▶ Suppose the subtree rooted at i contains n elements
- ▶ Everything except recursion takes $\Theta(1)$ time
- ▶ One recursive call on one child
- ▶ Worst-case: the child subtree we recurse into has more elements
- ▶ Balance invariant \implies at least $1/3$ elements on right side
 \implies at at most $2/3$ elements in worst case
- ▶ $T(n) = T(\frac{2}{3}n) + \Theta(1)$
- ▶ $T(n) \in \Theta(\log n)$ by master theorem case 2
- ▶ **Pushing the envelope:** for any fraction $f < 1$, including $f > \frac{1}{2}$, $T(fn) + \Theta(1) \in O(\log n)$

Delete-Max Operation

Idea: grab rightmost node on bottom level, move it to root, heapify root (sketch)

1: **function** MAX-HEAP-DELETE-MAX(A)

Require: A is a valid non-empty heap

Ensure: the maximum key in A is removed and then returned

2: $max = A[1]$

3: $A[1] = A[A.heapsize]$

4: $A.heapsize = A.heapsize - 1$

5: $MAX - HEAPIFY(A, 1)$

6: **return** max

7: **end function**

Analysis: $\Theta(1)$ plus $MAX - HEAPIFY$ so $\Theta(\log n)$

Increase Key Operation

$A[i]$ “floats up” until it is either in heap-order, or becomes the root

1: **function** MAX-HEAP-INCREASE-KEY(A, i, key)

Require: A is a valid heap, $1 \leq i \leq n$, $\text{key} > A[i]$

2: $A[i] = \text{key}$

3: **while** $i > 1$ and $A[\text{PARENT}(i)] < A[i]$ **do**

4: $\text{swap}(A[i], A[\text{PARENT}(i)])$

5: $i = \text{PARENT}(i)$

6: **end while**

7: **end function**

$\Theta(\text{depth of } A[i]) = \Theta(\log n)$ time

Insert

1: **function** MAX-HEAP-INSERT(A , key)

Require: A is a valid heap, $A.heapsize < n$

2: $A.heapsize = A.heapsize + 1$

3: $A[A.heapsize] = -\infty$

4: MAX-HEAP-INCREASE-KEY(A , $A.heapsize$, key)

5: **end function**

Analysis: $\Theta(1)$ plus MAX-HEAP-INCREASE-KEY, so
 $\Theta(\log n)$

Build-Heap

Online/incremental construction: build structure one element at a time

Offline construction/build: given n elements all at once, build valid data structure

Offline is often faster, by constant factors or even asymptotically

Leaves are trivially in heap order and live in $A[\lfloor n/2 \rfloor + 1, \dots, n]$

Parents might be out of heap-order and live in $A[1, \dots, \lfloor n/2 \rfloor]$

Just heapify all the parents!

Build-Heap Pseudocode

```
1: function BUILD-MAX-HEAP( $A$ )  
Require:  $A[1, \dots, n]$  may or may not be in heap-order  
Ensure:  $A[1, \dots, n]$  is in heap-order  
2:    $A.heapsize = n$   
3:   for  $i$  from  $\lfloor n/2 \rfloor$  down to 1 do  
4:      $MAX - HEAPIFY(A, i)$   
5:   end for  
6: end function
```

Build-Heap Analysis

Loose analysis: if $A[i]$ is at height h , $MAX - HEAPIFY$ takes $\Theta(h)$; $h \in O(\log n)$, so each call is $O(\log n)$; n calls; so $O(n \log n)$
(Correct but loose upper bound.)

Fact about balanced binary trees:

- ▶ Sum of all node **depths** is $\Theta(n \log n)$
- ▶ Sum of all node **heights** is $\Theta(n)$
- ▶ Observe that $1/2$ of nodes are at height 0, $1/4$ at height 1, $1/8$ at height 2, etc.
- ▶ (sketch)

n calls to $MAX - HEAPIFY$ take time

$$\Theta(\sum_i (\text{height of } i)) = \Theta(n)$$

$\therefore BUILD - MAX - HEAP$ takes $\Theta(n)$ time

Arrayed Binary Max-Heap Summary

Operation

Create empty heap

Find maximum element

Insert one element

Delete and return maximum element

Increase key of previously-inserted element

Build n -element heap offline

Time Compl.

$\Theta(1)$

$\Theta(1)$

$\Theta(\log n)$

$\Theta(\log n)$

$\Theta(\log n)$

$\Theta(n)$

Heapsort Intro

- ▶ Reduction to max-heap operations
- ▶ Selection sort: find maximum unsorted element, place at back of sorted array, repeat until done
- ▶ Use max-heap to accelerate "find maximum" step
- ▶ Convenient: if heap holds $k \leq n$ elements, it occupies $A[1, \dots, k]$
- ▶ Remaining $n - k$ elements $A[k + 1, \dots, n]$ are free to hold sorted elements

High-level heapsort

1. Build-heap; all n elements hold a valid heap
2. Delete-max; removes maximum element from heap zone, vacates one element in array
3. Move the old max into the vacancy
4. Repeat until done

Heapsort Pseudocode

```
1: function HEAPSORT( $A[1, \dots, n]$ )  
Ensure:  $A$  is in non-decreasing order  
2:   BUILD – MAX – HEAP( $A$ )  
3:   for  $i$  from  $n$  down to 2 do  
4:      $A[i] = \text{MAX – HEAP – DELETE – MAX}(A)$   
5:   end for  
6: end function
```

(Observe: no need for $i = 1$ iteration.)

Analysis

- ▶ build-heap = $\Theta(n)$
- ▶ $(n - 1)$ iterations of loop $\times \Theta(\log n)$ each
- ▶ = $\Theta(n \log n)$ total

Sorting Lower Bound

So far all our sorts have compared elements to each other, e.g.

$$A[i] < A[j]$$

(insertion, selection, merge, heap sort; also quick sort)

Q: what is the minimum number of comparisons adequate to sort?

A: enough to decide which of the $n!$ permutations of A would be in order

Binary search through a set of N things takes $\lceil \log_2 N \rceil$ steps

\implies correct sort makes $\geq \lceil \log_2 n! \rceil$ comparison operations

$= \Omega(n \log n)$ comparisons

\therefore every comparison-based sorting algorithm takes $\Omega(n \log n)$ time
(**sorting lower bound**)

Optimal Algorithms

Optimal Algorithm: time complexity matches problem's lower bound

E.g.

- ▶ Proven $\Omega(n \log n)$ lower bound for sorting
- ▶ Merge sort, heap sort take $\Theta(n \log n)$ time
- ▶ \implies merge sort, heap sort are **optimal**

(Heap sort is theoretically superior because it is in-place. Practical constant factors depends on whether allocation (mergesort) or cache misses (heapsort) are costlier.)

Optimal algorithms are the end-goal of algorithm design and lower-bound analysis.

Epilogue — Sorted Order vs. Heap Order

We can sort by building a heap/BST and then retrieving elements in order.

Sorting lower bound \implies that process **must** be $\Omega(n \log n)$

Phase of sorting	BST	heap
Build structure	$\Theta(n \log n)$	$\Theta(n)$
Retrieve in order	$\Theta(n)$	$\Theta(n \log n)$

Whack-a-mole: a $\Omega(n \log n)$ phase inevitably pops up somewhere!

Heap-order is not organized enough to be asymptotically significant, which is why it can be $o(n \log n)$, and maintain a stronger balance invariant than BSTs.