Lecture Notes for Chapter 6: Heapsort

Chapter 6 overview

Heapsort

- $O(n \lg n)$ worst case—like merge sort.
- Sorts in place—like insertion sort.
- Combines the best of both algorithms.

To understand heapsort, we'll cover heaps and heap operations, and then we'll take a look at priority queues.

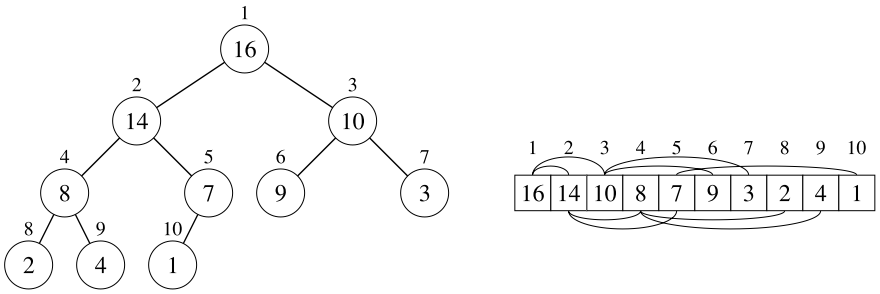
Heaps

Heap data structure

- Heap A (*not* garbage-collected storage) is a nearly complete binary tree.
  - ***Height*** of node = # of edges on a longest simple path from the node down to a leaf.
  - ***Height*** of heap = height of root = $\Theta(\lg n)$.
- A heap can be stored as an array A.
  - Root of tree is $A[1]$.
  - Parent of $A[i] = A[\lfloor i/2 \rfloor]$.
  - Left child of $A[i] = A[2i]$.
  - Right child of $A[i] = A[2i + 1]$.
  - Computing is fast with binary representation implementation.

[In book, have length *and* heap-size *attributes. Here, we bypass these attributes and use parameter values instead.]*

Example: of a max-heap. *[Arcs above and below the array on the right go between parents and children. There is no significance to whether an arc is drawn above or below the array.]*



Heap property

- For max-heaps (largest element at root), ***max-heap property:*** for all nodes i, excluding the root, $A[\text{PARENT}(i)] \geq A[i]$.
- For min-heaps (smallest element at root), ***min-heap property:*** for all nodes i, excluding the root, $A[\text{PARENT}(i)] \leq A[i]$.

By induction and transitivity of $\leq$, the max-heap property guarantees that the maximum element of a max-heap is at the root. Similar argument for min-heaps.

The heapsort algorithm we'll show uses max-heaps.

Note: In general, heaps can be k-ary tree instead of binary.

Maintaining the heap property

MAX-HEAPIFY is important for manipulating max-heaps. It is used to maintain the max-heap property.

- Before MAX-HEAPIFY, $A[i]$ may be smaller than its children.
- Assume left and right subtrees of i are max-heaps.
- After MAX-HEAPIFY, subtree rooted at i is a max-heap.

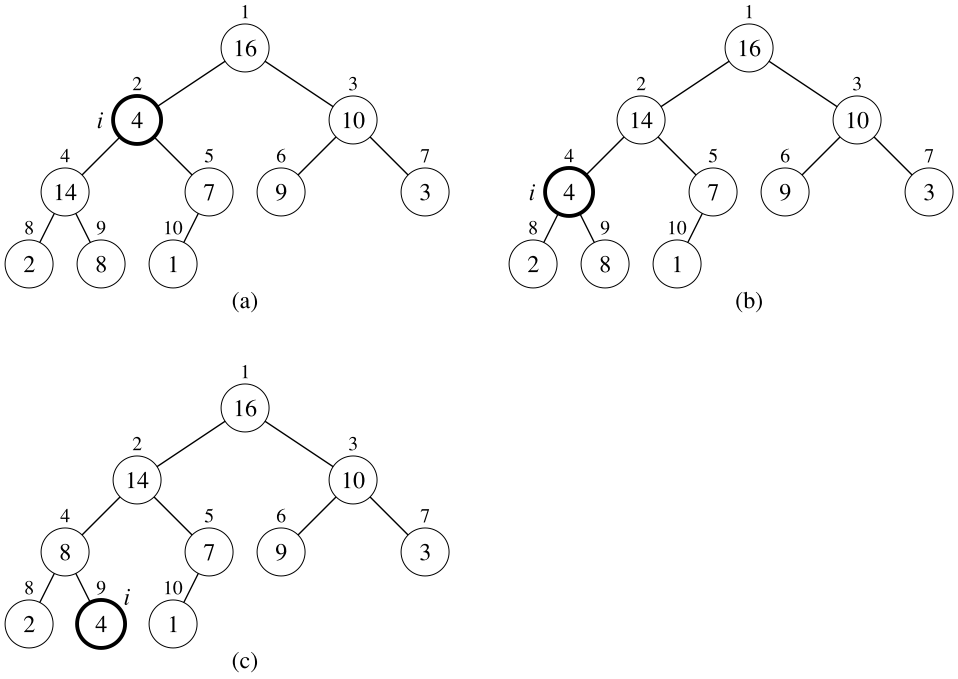
```
MAX-HEAPIFY(A, i, n)
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ n and A[l] > A[i]
  then largest ← l
  else largest ← i
if r ≤ n and A[r] > A[largest]
  then largest ← r
if largest ≠ i
  then exchange A[i] ↔ A[largest]
       MAX-HEAPIFY(A, largest, n)
```

[Parameter n replaces attribute $heap\text{-}size[A]$.]

The way MAX-HEAPIFY works:

- Compare $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$.
- If necessary, swap $A[i]$ with the larger of the two children to preserve heap property.
- Continue this process of comparing and swapping down the heap, until subtree rooted at i is max-heap. If we hit a leaf, then the subtree rooted at the leaf is trivially a max-heap.

Run MAX-HEAPIFY on the following heap example.



- Node 2 violates the max-heap property.
- Compare node 2 with its children, and then swap it with the larger of the two children.
- Continue down the tree, swapping until the value is properly placed at the root of a subtree that is a max-heap. In this case, the max-heap is a leaf.

Time: $O(\lg n)$.

Correctness: *[Instead of book's formal analysis with recurrence, just come up with $O(\lg n)$ intuitively.]* Heap is almost-complete binary tree, hence must process $O(\lg n)$ levels, with constant work at each level (comparing 3 items and maybe swapping 2).

Building a heap

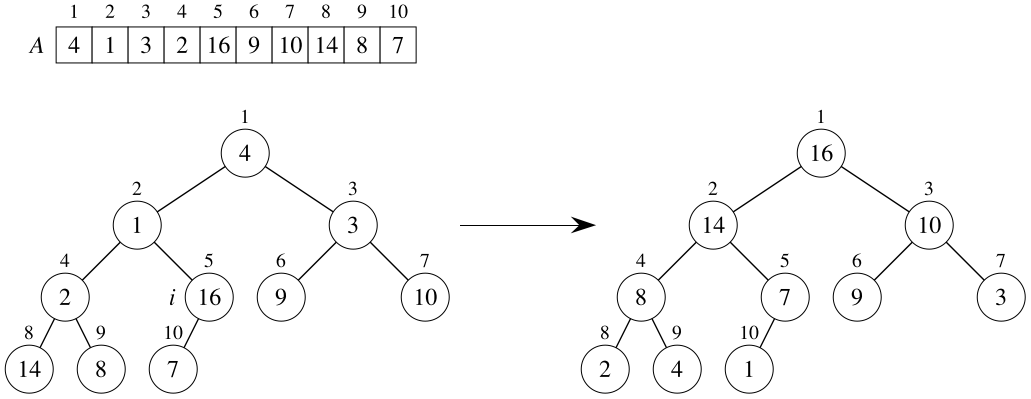
The following procedure, given an unordered array, will produce a max-heap.

BUILD-MAX-HEAP(A, n)
for $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1
  do MAX-HEAPIFY(A, i, n)

[Parameter n replaces both attributes $length[A]$ and $heap\text{-}size[A]$.]

Example: Building a max-heap from the following unsorted array results in the first heap example.

- i starts off as 5.
- MAX-HEAPIFY is applied to subtrees rooted at nodes (in order): 16, 2, 3, 1, 4.



Correctness

Loop invariant: At start of every iteration of **for** loop, each node $i+1$, $i+2, \ldots, n$ is root of a max-heap.

Initialization: By Exercise 6.1-7, we know that each node $\lfloor n/2 \rfloor + 1$, $\lfloor n/2 \rfloor + 2$, $\ldots, n$ is a leaf, which is the root of a trivial max-heap. Since $i = \lfloor n/2 \rfloor$ before the first iteration of the **for** loop, the invariant is initially true.

Maintenance: Children of node i are indexed higher than i, so by the loop invariant, they are both roots of max-heaps. Correctly assuming that $i+1, i+2, \ldots, n$ are all roots of max-heaps, MAX-HEAPIFY makes node i a max-heap root. Decrementing i reestablishes the loop invariant at each iteration.

Termination: When $i = 0$, the loop terminates. By the loop invariant, each node, notably node 1, is the root of a max-heap.

Analysis

- ***Simple bound:*** $O(n)$ calls to MAX-HEAPIFY, each of which takes $O(\lg n)$ time $\Rightarrow O(n \lg n)$. (Note: A good approach to analysis in general is to start by proving easy bound, then try to tighten it.)
- ***Tighter analysis:*** Observation: Time to run MAX-HEAPIFY is linear in the height of the node it's run on, and most nodes have small heights. Have $\leq \lceil n/2^{h+1} \rceil$ nodes of height h (see Exercise 6.3-3), and height of heap is $\lfloor \lg n \rfloor$ (Exercise 6.1-2).

The time required by MAX-HEAPIFY when called on a node of height h is $O(h)$, so the total cost of BUILD-MAX-HEAP is

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) .$$

Evaluate the last summation by substituting $x = 1/2$ in the formula (A.8) ($\sum_{k=0}^{\infty} kx^k$), which yields

$$\begin{aligned} \sum_{h=0}^{\infty} \frac{h}{2^h} &= \frac{1/2}{(1-1/2)^2} \\ &= 2 . \end{aligned}$$

Thus, the running time of BUILD-MAX-HEAP is $O(n)$.

Building a min-heap from an unordered array can be done by calling MIN-HEAPIFY instead of MAX-HEAPIFY, also taking linear time.

The heapsort algorithm

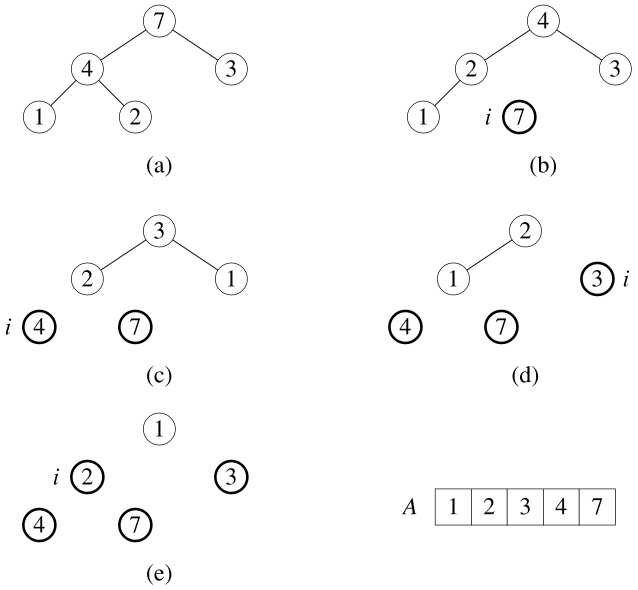
Given an input array, the heapsort algorithm acts as follows:

- Builds a max-heap from the array.
- Starting with the root (the maximum element), the algorithm places the maximum element into the correct place in the array by swapping it with the element in the last position in the array.
- "Discard" this last node (knowing that it is in its correct place) by decreasing the heap size, and calling MAX-HEAPIFY on the new (possibly incorrectly-placed) root.
- Repeat this "discarding" process until only one node (the smallest element) remains, and therefore is in the correct place in the array.

HEAPSORT(A, n)
BUILD-MAX-HEAP(A, n)
for $i \leftarrow n$ **downto** 2
  do exchange $A[1] \leftrightarrow A[i]$
    MAX-HEAPIFY(A, 1, $i - 1$)

[Parameter n replaces $length[A]$, and parameter value $i - 1$ in MAX-HEAPIFY *call replaces decrementing of heap-size$[A]$.]*

Example: Sort an example heap on the board. *[Nodes with heavy outline are no longer in the heap.]*



Analysis

- BUILD-MAX-HEAP: $O(n)$
- **for** loop: $n - 1$ times
- exchange elements: $O(1)$
- MAX-HEAPIFY: $O(\lg n)$

Total time: $O(n \lg n)$.

Though heapsort is a great algorithm, a well-implemented quicksort usually beats it in practice.

Heap implementation of priority queue

Heaps efficiently implement priority queues. These notes will deal with max-priority queues implemented with max-heaps. Min-priority queues are implemented with min-heaps similarly.

A heap gives a good compromise between fast insertion but slow extraction and vice versa. Both operations take $O(\lg n)$ time.

Priority queue

- Maintains a dynamic set S of elements.
- Each set element has a ***key***—an associated value.
- Max-priority queue supports dynamic-set operations:
  - INSERT(S, x): inserts element x into set S.
  - MAXIMUM(S): returns element of S with largest key.

  - EXTRACT-MAX(S): removes and returns element of S with largest key.
  - INCREASE-KEY(S, x, k): increases value of element x's key to k. Assume $k \geq x$'s current key value.
- Example max-priority queue application: schedule jobs on shared computer.
- Min-priority queue supports similar operations:
  - INSERT(S, x): inserts element x into set S.
  - MINIMUM(S): returns element of S with smallest key.
  - EXTRACT-MIN(S): removes and returns element of S with smallest key.
  - DECREASE-KEY(S, x, k): decreases value of element x's key to k. Assume $k \leq x$'s current key value.
- Example min-priority queue application: event-driven simulator.

Note: Actual implementations often have a ***handle*** in each heap element that allows access to an object in the application, and objects in the application often have a handle (likely an array index) to access the heap element.

Will examine how to implement max-priority queue operations.

Finding the maximum element

Getting the maximum element is easy: it's the root.

HEAP-MAXIMUM(A)
return $A[1]$

Time: $\Theta(1)$.

Extracting max element

Given the array A:

- Make sure heap is not empty.
- Make a copy of the maximum element (the root).
- Make the last node in the tree the new root.
- Re-heapify the heap, with one fewer node.
- Return the copy of the maximum element.

HEAP-EXTRACT-MAX(A, n)
if $n < 1$
  then error "heap underflow"
$max \leftarrow A[1]$
$A[1] \leftarrow A[n]$
MAX-HEAPIFY($A, 1, n - 1$) $\quad \triangleright$ remakes heap
return *max*

[Parameter n replaces heap-size$[A]$, and parameter value $n - 1$ in MAX-HEAPIFY *call replaces decrementing of heap-size$[A]$.]*

Analysis: constant time assignments plus time for MAX-HEAPIFY.

Time: $O(\lg n)$.

Example: Run HEAP-EXTRACT-MAX on first heap example.

- Take 16 out of node 1.
- Move 1 from node 10 to node 1.
- Erase node 10.
- MAX-HEAPIFY from the root to preserve max-heap property.
- Note that successive extractions will remove items in reverse sorted order.

Increasing key value

Given set S, element x, and new key value k:

- Make sure $k \geq x$'s current key.
- Update x's key value to k.
- Traverse the tree upward comparing x to its parent and swapping keys if necessary, until x's key is smaller than its parent's key.

HEAP-INCREASE-KEY(A, i, key)
if $key < A[i]$
  then error "new key is smaller than current key"
$A[i] \leftarrow key$
while $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
  do exchange $A[i] \leftrightarrow A[\text{PARENT}(i)]$
    $i \leftarrow \text{PARENT}(i)$

Analysis: Upward path from node i has length $O(\lg n)$ in an n-element heap.

Time: $O(\lg n)$.

Example: Increase key of node 9 in first heap example to have value 15. Exchange keys of nodes 4 and 9, then of nodes 2 and 4.

Inserting into the heap

Given a key k to insert into the heap:

- Insert a new node in the very last position in the tree with key $-\infty$.
- Increase the $-\infty$ key to k using the HEAP-INCREASE-KEY procedure defined above.

MAX-HEAP-INSERT(A, key, n)
$A[n+1] \leftarrow -\infty$
HEAP-INCREASE-KEY$(A, n+1, key)$

[Parameter n replaces $heap\text{-}size[A]$, and use of value $n+1$ replaces incrementing of $heap\text{-}size[A]$.]

Analysis: constant time assignments + time for HEAP-INCREASE-KEY.

Time: $O(\lg n)$.

Min-priority queue operations are implemented similarly with min-heaps.