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.

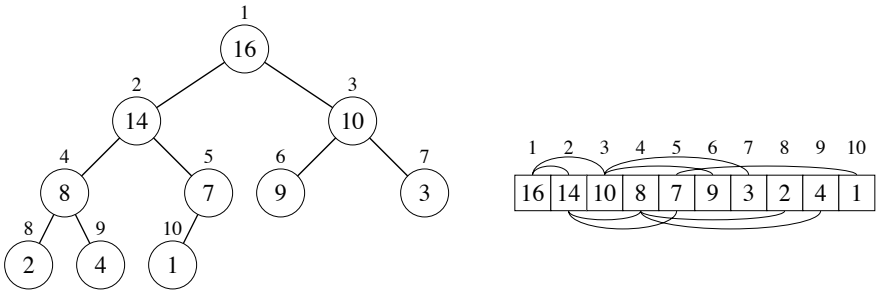
Heaps

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.

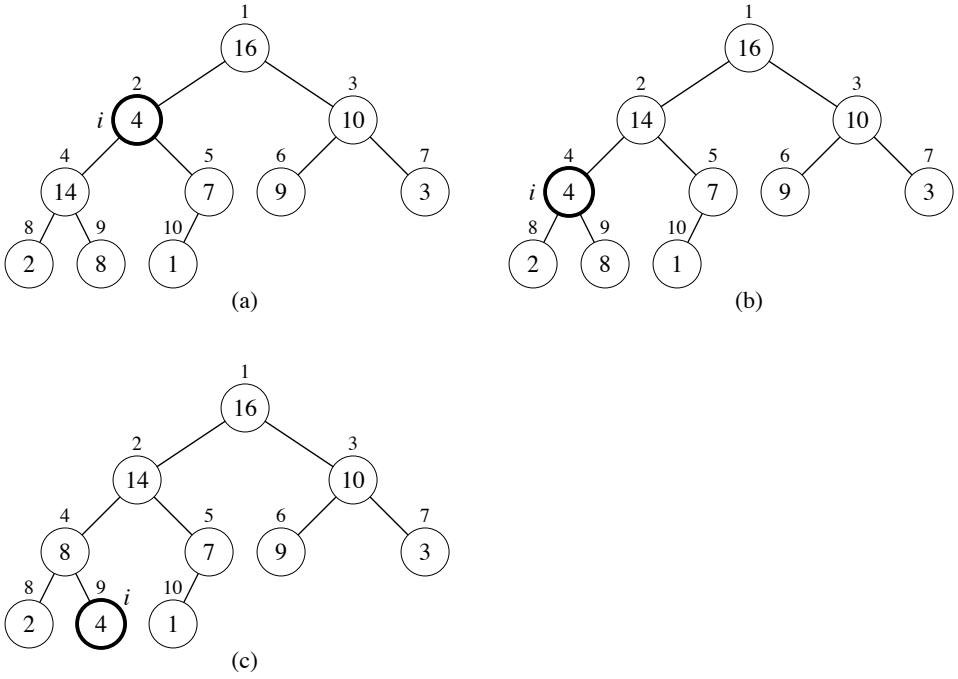
```
MAX-HEAPIFY(A, i, n)
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ n and A[l] > A[i]
  then largest ← l
  else largest ← i
if r ≤ n and A[r] > A[largest]
  then largest ← r
if largest ≠ i
  then exchange A[i] ↔ A[largest]
       MAX-HEAPIFY(A, largest, n)
```

[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.

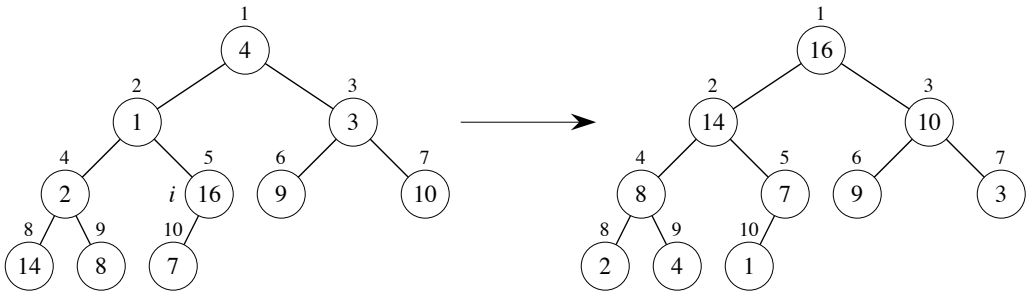
BUILD-MAX-HEAP(A, n)
for $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1
  do MAX-HEAPIFY(A, i, n)

[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.

	1	2	3	4	5	6	7	8	9	10
A	4	1	3	2	16	9	10	14	8	7



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

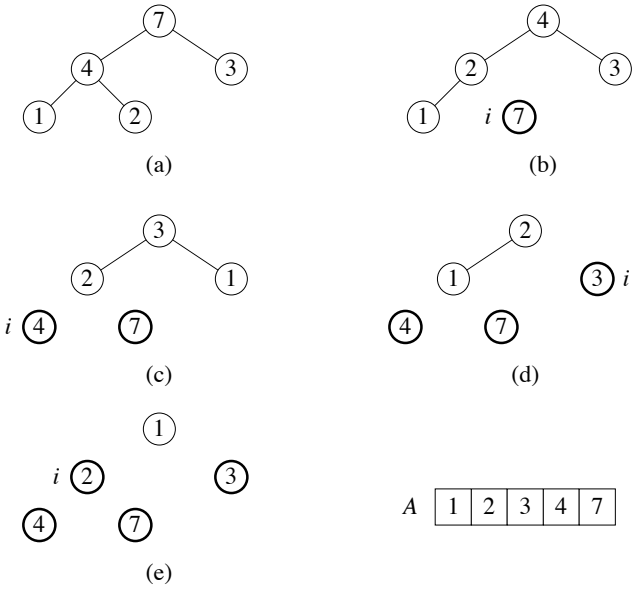
Given an input array, the heapsort algorithm acts as follows:

- 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.

Solutions for Chapter 6: Heapsort

Solution to Exercise 6.1-1

Since a heap is an almost-complete binary tree (complete at all levels except possibly the lowest), it has at most $2^{h+1} - 1$ elements (if it is complete) and at least $2^h - 1 + 1 = 2^h$ elements (if the lowest level has just 1 element and the other levels are complete).

Solution to Exercise 6.1-2

Given an n-element heap of height h, we know from Exercise 6.1-1 that

$2^h \le n \le 2^{h+1} - 1 < 2^{h+1}$.

Thus, $h \le \lg n < h + 1$. Since h is an integer, $h = \lfloor \lg n \rfloor$ (by definition of $\lfloor\ \rfloor$).

Solution to Exercise 6.1-3

Assume the claim is false—i.e., that there is a subtree whose root is not the largest element in the subtree. Then the maximum element is somewhere else in the subtree, possibly even at more than one location. Let m be the index at which the maximum appears (the lowest such index if the maximum appears more than once). Since the maximum is not at the root of the subtree, node m has a parent. Since the parent of a node has a lower index than the node, and m was chosen to be the smallest index of the maximum value, $A[\text{PARENT}(m)] < A[m]$. But by the max-heap property, we must have $A[\text{PARENT}(m)] \ge A[m]$. So our assumption is false, and the claim is true.

Solution to Exercise 6.2-6

If you put a value at the root that is less than every value in the left and right subtrees, then MAX-HEAPIFY will be called recursively until a leaf is reached. To

make the recursive calls traverse the longest path to a leaf, choose values that make MAX-HEAPIFY always recurse on the left child. It follows the left branch when the left child is $\geq$ the right child, so putting 0 at the root and 1 at all the other nodes, for example, will accomplish that. With such values, MAX-HEAPIFY will be called h times (where h is the heap height, which is the number of edges in the longest path from the root to a leaf), so its running time will be $\Theta(h)$ (since each call does $\Theta(1)$ work), which is $\Theta(\lg n)$. Since we have a case in which MAX-HEAPIFY's running time is $\Theta(\lg n)$, its worst-case running time is $\Omega(\lg n)$.

Solution to Exercise 6.3-3

Let H be the height of the heap.

Two subtleties to beware of:

- Be careful not to confuse the height of a node (longest distance from a leaf) with its depth (distance from the root).
- If the heap is not a complete binary tree (bottom level is not full), then the nodes at a given level (depth) don't all have the same height. For example, although all nodes at depth H have height 0, nodes at depth $H - 1$ can have either height 0 or height 1.

For a complete binary tree, it's easy to show that there are $\lceil n/2^{h+1} \rceil$ nodes of height h. But the proof for an incomplete tree is tricky and is not derived from the proof for a complete tree.

Proof By induction on h.

Basis: Show that it's true for $h = 0$ (i.e., that # of leaves $\leq \lceil n/2^{h+1} \rceil = \lceil n/2 \rceil$). In fact, we'll show that the # of leaves $= \lceil n/2 \rceil$.

The tree leaves (nodes at height 0) are at depths H and $H - 1$. They consist of

- all nodes at depth H, and
- the nodes at depth $H - 1$ that are not parents of depth-H nodes.

Let x be the number of nodes at depth H—that is, the number of nodes in the bottom (possibly incomplete) level.

Note that $n - x$ is odd, because the $n - x$ nodes above the bottom level form a complete binary tree, and a complete binary tree has an odd number of nodes (1 less than a power of 2). Thus if n is odd, x is even, and if n is even, x is odd.

To prove the base case, we must consider separately the case in which n is even (x is odd) and the case in which n is odd (x is even). Here are two ways to do this: The first requires more cleverness, and the second requires more algebraic manipulation.

1. First method of proving the base case:
   - If n is odd, then x is even, so all nodes have siblings—i.e., all internal nodes have 2 children. Thus (see Exercise B.5-3), # of internal nodes $=$ # of leaves $- 1$.

So, n = # of nodes = # of leaves + # of internal nodes = $2 \cdot$ # of leaves $- 1$. Thus, # of leaves $= (n+1)/2 = \lceil n/2 \rceil$. (The latter equality holds because n is odd.)

- If n is even, then x is odd, and some leaf doesn't have a sibling. If we gave it a sibling, we would have $n+1$ nodes, where $n+1$ is odd, so the case we analyzed above would apply. Observe that we would also increase the number of leaves by 1, since we added a node to a parent that already had a child. By the odd-node case above, # of leaves $+ 1 = \lceil (n+1)/2 \rceil = \lceil n/2 \rceil + 1$. (The latter equality holds because n is even.)

In either case, # of leaves $= \lceil n/2 \rceil$.

2. Second method of proving the base case:

   Note that at any depth $d < H$ there are 2^d nodes, because all such tree levels are complete.

   - If x is even, there are $x/2$ nodes at depth $H-1$ that are parents of depth H nodes, hence $2^{H-1} - x/2$ nodes at depth $H-1$ that are not parents of depth-H nodes. Thus,

     $$\begin{aligned} \text{total \# of height-0 nodes} &= x + 2^{H-1} - x/2 \\ &= 2^{H-1} + x/2 \\ &= (2^H + x)/2 \\ &= \lceil (2^H + x - 1)/2 \rceil \qquad \text{(because } x \text{ is even)} \\ &= \lceil n/2 \rceil \ . \end{aligned}$$

     ($n = 2^H + x - 1$ because the complete tree down to depth $H-1$ has $2^H - 1$ nodes and depth H has x nodes.)

   - If x is odd, by an argument similar to the even case, we see that

     $$\begin{aligned} \text{\# of height-0 nodes} &= x + 2^{H-1} - (x+1)/2 \\ &= 2^{H-1} + (x-1)/2 \\ &= (2^H + x - 1)/2 \\ &= n/2 \\ &= \lceil n/2 \rceil \qquad \text{(because } x \text{ odd} \Rightarrow n \text{ even)} \ . \end{aligned}$$

Inductive step: Show that if it's true for height $h-1$, it's true for h.

Let n_h be the number of nodes at height h in the n-node tree T.

Consider the tree T' formed by removing the leaves of T. It has $n' = n - n_0$ nodes. We know from the base case that $n_0 = \lceil n/2 \rceil$, so $n' = n - n_0 = n - \lceil n/2 \rceil = \lfloor n/2 \rfloor$.

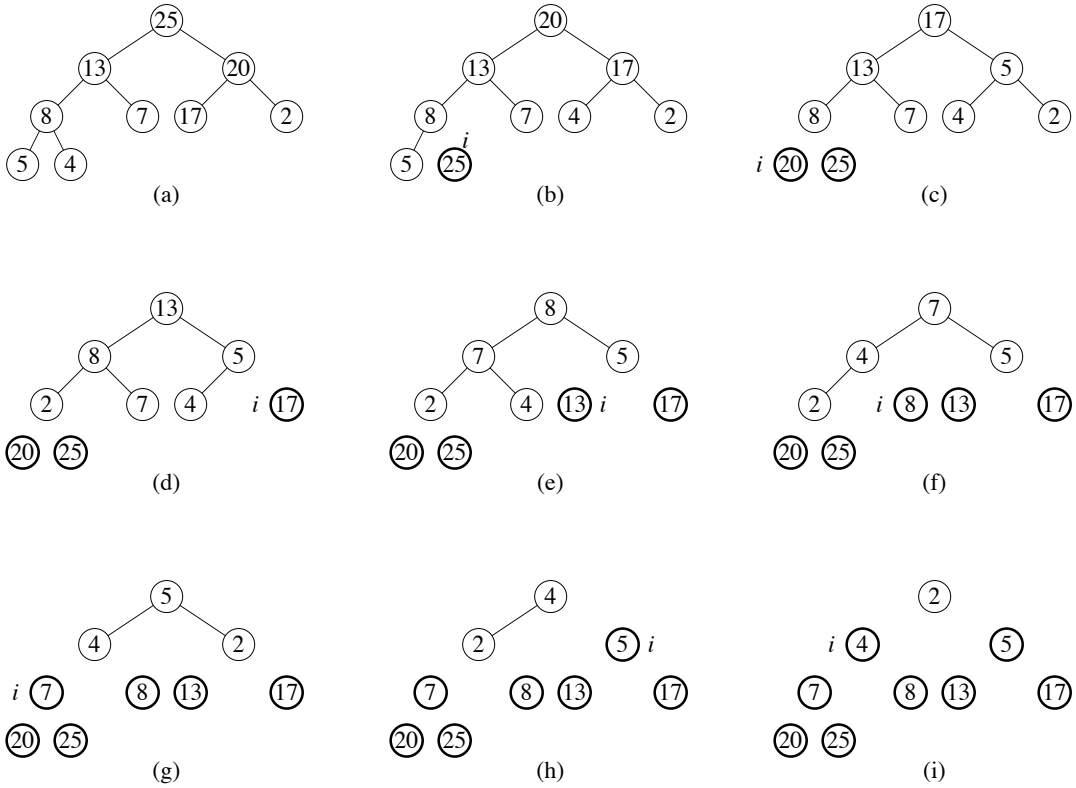
Note that the nodes at height h in T would be at height $h-1$ if the leaves of the tree were removed—that is, they are at height $h-1$ in T'. Letting n'_{h-1} denote the number of nodes at height $h-1$ in T', we have

$$n_h = n'_{h-1} \ .$$

By induction, we can bound n'_{h-1}:

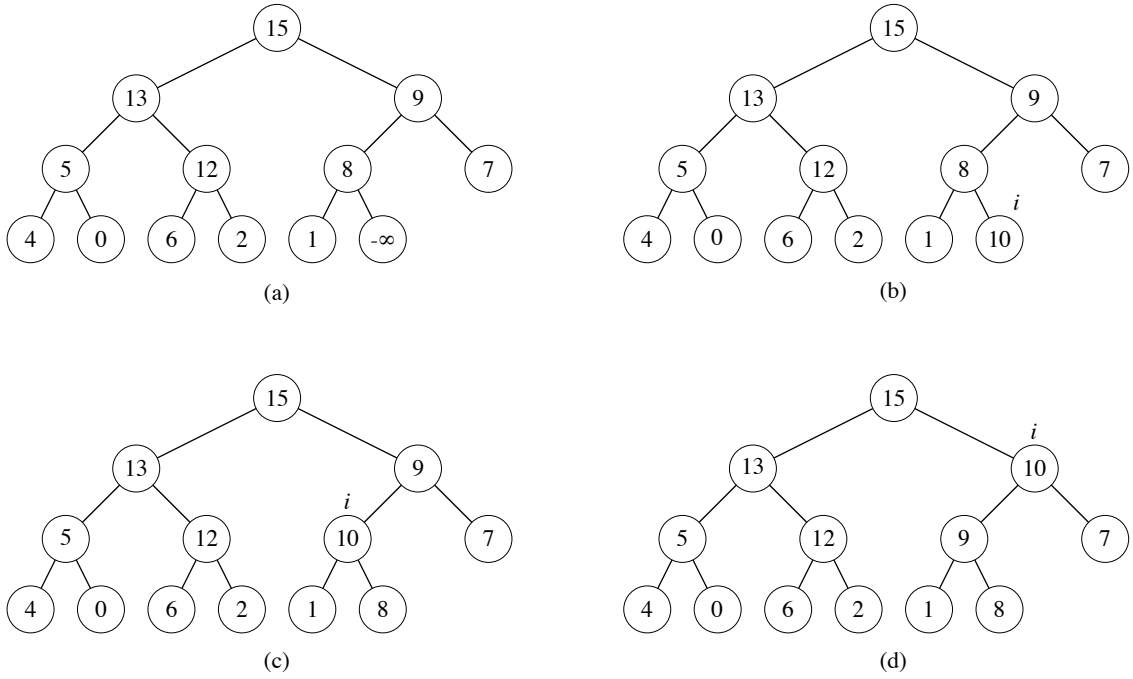
$$n_h = n'_{h-1} \le \lceil n'/2^h \rceil = \lceil \lfloor n/2 \rfloor / 2^h \rceil \le \lceil (n/2)/2^h \rceil = \lceil n/2^{h+1} \rceil \ . \qquad \blacksquare$$

Solution to Exercise 6.4-1



A	2	4	5	7	8	13	17	20	25

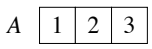
Solution to Exercise 6.5-2



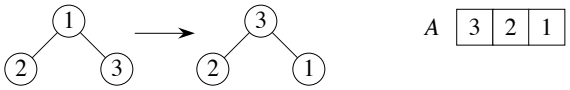
Solution to Problem 6-1

a. The procedures BUILD-MAX-HEAP and BUILD-MAX-HEAP′ do not always create the same heap when run on the same input array. Consider the following counterexample.

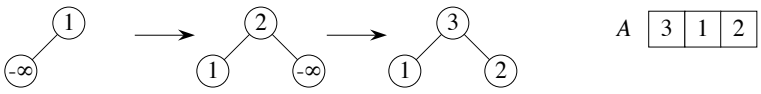
Input array A:



BUILD-MAX-HEAP(A):



BUILD-MAX-HEAP′(A):



b. An upper bound of $O(n \lg n)$ time follows immediately from there being $n - 1$ calls to MAX-HEAP-INSERT, each taking $O(\lg n)$ time. For a lower bound of

$\Omega(n \lg n)$, consider the case in which the input array is given in strictly increasing order. Each call to MAX-HEAP-INSERT causes HEAP-INCREASE-KEY to go all the way up to the root. Since the depth of node i is $\lfloor \lg i \rfloor$, the total time is

$$\begin{aligned}
\sum_{i=1}^{n} \Theta(\lfloor \lg i \rfloor) &\geq \sum_{i=\lceil n/2 \rceil}^{n} \Theta(\lfloor \lg \lceil n/2 \rceil \rfloor) \\
&\geq \sum_{i=\lceil n/2 \rceil}^{n} \Theta(\lfloor \lg(n/2) \rfloor) \\
&= \sum_{i=\lceil n/2 \rceil}^{n} \Theta(\lfloor \lg n - 1 \rfloor) \\
&\geq n/2 \cdot \Theta(\lg n) \\
&= \Omega(n \lg n) \; .
\end{aligned}$$

In the worst case, therefore, BUILD-MAX-HEAP′ requires $\Theta(n \lg n)$ time to build an n-element heap.

Solution to Problem 6-2

a. A d-ary heap can be represented in a 1-dimensional array as follows. The root is kept in $A[1]$, its d children are kept in order in $A[2]$ through $A[d+1]$, their children are kept in order in $A[d+2]$ through $A[d^2+d+1]$, and so on. The following two procedures map a node with index i to its parent and to its jth child (for $1 \leq j \leq d$), respectively.

```
D-ARY-PARENT(i)
return ⌊(i − 2)/d + 1⌋

D-ARY-CHILD(i, j)
return d(i − 1) + j + 1
```

To convince yourself that these procedures really work, verify that

$$\text{D-ARY-PARENT}(\text{D-ARY-CHILD}(i, j)) = i \; ,$$

for any $1 \leq j \leq d$. Notice that the binary heap procedures are a special case of the above procedures when $d = 2$.

b. Since each node has d children, the height of a d-ary heap with n nodes is $\Theta(\log_d n) = \Theta(\lg n / \lg d)$.

c. The procedure HEAP-EXTRACT-MAX given in the text for binary heaps works fine for d-ary heaps too. The change needed to support d-ary heaps is in MAX-HEAPIFY, which must compare the argument node to all d children instead of just 2 children. The running time of HEAP-EXTRACT-MAX is still the running time for MAX-HEAPIFY, but that now takes worst-case time proportional to the product of the height of the heap by the number of children examined at each node (at most d), namely $\Theta(d \log_d n) = \Theta(d \lg n / \lg d)$.

d. The procedure MAX-HEAP-INSERT given in the text for binary heaps works fine for d-ary heaps too. The worst-case running time is still $\Theta(h)$, where h is the height of the heap. (Since only parent pointers are followed, the number of children a node has is irrelevant.) For a d-ary heap, this is $\Theta(\log_d n) = \Theta(\lg n / \lg d)$.

e. D-ARY-HEAP-INCREASE-KEY can be implemented as a slight modification of MAX-HEAP-INSERT (only the first couple lines are different). Increasing an element may make it larger than its parent, in which case it must be moved higher up in the tree. This can be done just as for insertion, traversing a path from the increased node toward the root. In the worst case, the entire height of the tree must be traversed, so the worst-case running time is $\Theta(h) = \Theta(\log_d n) = \Theta(\lg n / \lg d)$.

```
D-ARY-HEAP-INCREASE-KEY(A, i, k)
A[i] ← max(A[i], k)
while i > 1 and A[PARENT(i)] < A[i]
    do exchange A[i] ↔ A[PARENT(i)]
       i ← PARENT(i)
```