

Figure 1: Node with key value 13 is to be deleted

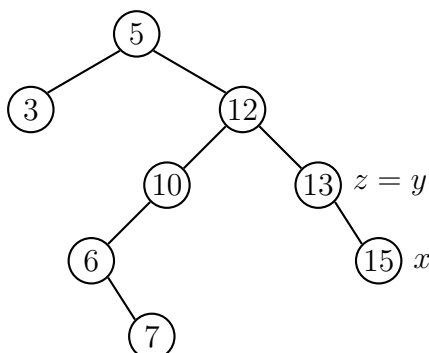


Figure 2: Node with key value 13 is to be deleted

1 Binary Search Trees (Cont'd)

The procedure `TreeDelete` is for deleting a given node z from a binary search tree. It takes as an argument a pointer to z and considers three cases.

- If z has no children, we modify its parent $p[z]$ to replace z with NIL as its child.

Example 1. See Figure 1.

- If the node has only a single child, we “splice out” z by making a new link between its child and its parent.

Example 2. See Figure 2.

- Finally, if the node has two children, we splice out z ’s successor y , which has no left child¹ and replace z ’s key and satellite data with y ’s key and satellite data.

Example 3. See Figure 3. y is the successor of z , since z has right subtree. y has at most one child (the right child), otherwise, the left child of y should be the successor of z instead.

In lines 1-3, the algorithm determines a node y to splice out. The node y is either the input node z (if z has at most 1 child) or the successor of z (if z has two children). Then, in lines 4 -

¹If a node in a BST has two children, then its successor has no left child and its predecessor has no right child.

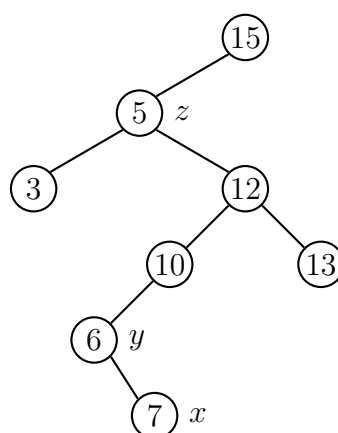


Figure 3: Node with key value 5 is to be deleted

6, x is set to the non-NIL child of y , or to NIL if y has no children. The node y is spliced out in lines 7 - 13 by modifying pointers in $p[y]$ and x . Splicing out y is somewhat complicated by the need for proper handling of the boundary conditions, which occur when $x = \text{NIL}$ or when y is the root. Finally, in lines 14 - 16, if the successor of z was the node spliced out, y 's key and satellite data are moved to z , overwriting the previous key and satellite data. The node y is returned in line 16 so that the calling procedure can recycle it via the free list. The procedure runs in $O(h)$ time.

```

TreeDelete(T, z)
1:  if left[z] = NIL or right[z] = NIL
2:      then y <- z
3:      else y <- TreeSuccessor(z)
4:  if left[y] != NIL
5:      then x <- left[y]
6:      else x <- right[y]
7:  if x != NIL
8:      then p[x] <- p[y]
9:  if p[y] = NIL
10: then root[T] <- x
11: else if y = left[p[y]]
12:     then left[p[y]] <- x
13:     else right[p[y]] <- x
14: if y != z
15:     then key[z] <- key[y]
16: return y
  
```

Note that this algorithm consider the following four cases:

	case 1	case 2	case 3	case 4
$p(y)$	NIL	some node $u = p(y)$	NIL	some node $u = p(y)$
y	y	y	y	y
x	NIL	NIL	x	x

For cases 3 and 4, where x is non-nil, Lines 7-8 decides the parent of x . Lines 9 - 13 decides the child of u for cases 2 and 4, decide the new value of $root[T]$ for cases 1 and 3.

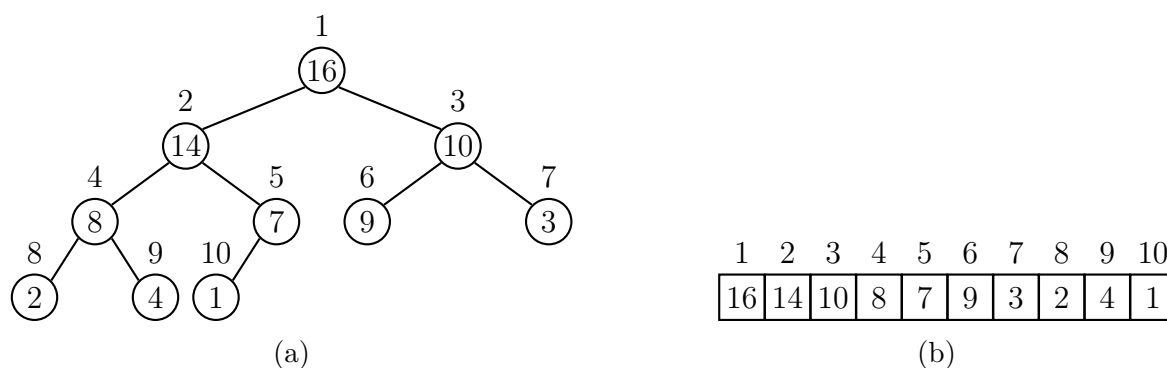


Figure 4: A max-heap viewed as (a) a complete binary tree and (b) an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array.

2 Heaps

The (binary) heap data structure is an array object that can be viewed as a nearly complete binary tree, as shown in Figure 4. Each node of the tree corresponds to an element of the array that stores the value in the node. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point. An array A that represents a heap is an object with two attributes:

- $length[A]$, which is the number of elements in the array, and
- $heap - size[A]$, the number of elements in the heap stored within array A .

That is, although $A[1 \dots length[A]]$ may contain valid numbers, no element past $A[heap - size[A]]$, where $heap - size[A] \leq length[A]$, is an element of the heap. The root of the tree is $A[1]$, and given the index i of a node, the indices of its parent $PARENT(i)$, left child $LEFT(i)$, and right child $RIGHT(i)$ can be computed simply: $\lfloor i/2 \rfloor$ (floor of $i/2$), $2i$, $2i + 1$, respectively. There are two kinds of binary heaps: max-heaps and min-heaps. In a max-heap, the max-heap property is that for every node i other than the root,

$$A[PARENT(i)] \geq A[i],$$

that is, the value of a node is at most the value of its parent. Thus, the largest element in a max-heap is stored at the root, and the subtree rooted at a node contains values no larger than that contained at the node itself.

A max-heap viewed as (a) a binary tree and (b) an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array.

MaxHeapify is an important subroutine for manipulating max-heaps. Its inputs are an array A and an index i into the array. When MaxHeapify is called, it is assumed that the binary trees rooted at $LEFT(i)$ and $RIGHT(i)$ are max-heaps, but that $A[i]$ may be smaller than its children, thus violating the max-heap property. The function of MaxHeapify is to let the value at $A[i]$ "float down" in the max-heap so that the subtree rooted at index i becomes a max-heap. It can be shown that the running time of MaxHeapify on a node of height h as $O(h)$.

```

MaxHeapify(A, i)
1: l ← LEFT(i)
2: r ← RIGHT(i)
3: if l ≤ heap-size[A] and A[l] > A[i]
4:   then largest ← l
5:   else largest ← i
6: if r ≤ heap-size[A] and A[r] > A[largest]
7:   then largest ← r
8: if largest ≠ i
9:   then exchange A[i] ↔ A[largest]
10:      MaxHeapify(A, largest)

```

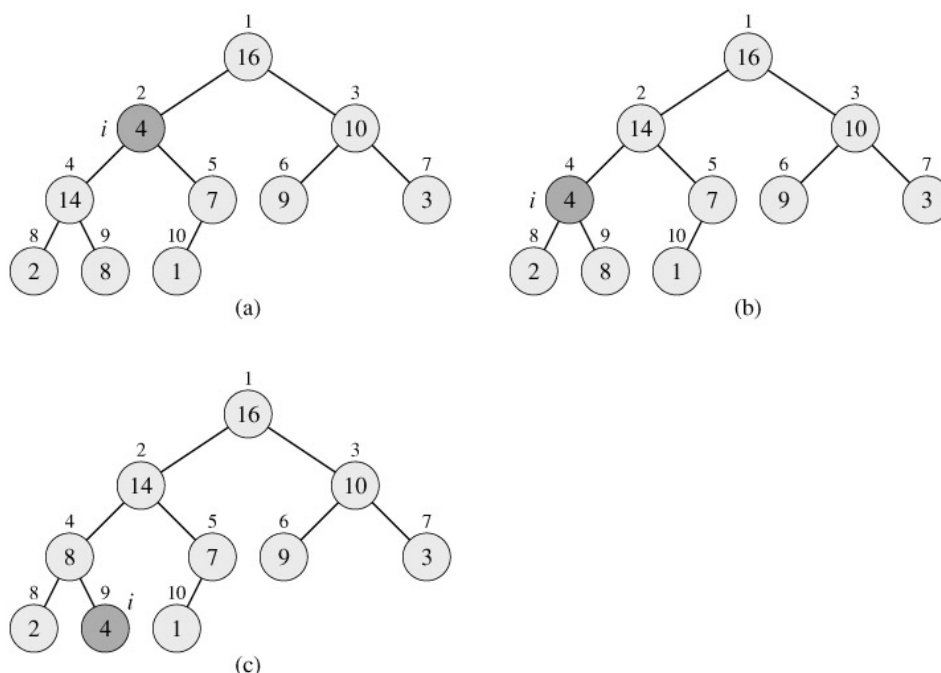


Figure 5: The action of $\text{MaxHeapify}(A, 2)$, where $\text{heap-size}[A] = 10$. (a) The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in (b) by exchanging $A[2]$ with $A[4]$, which destroys the max-heap property for node 4. The recursive call $\text{MaxHeapify}(A, 4)$ now has $i = 4$. After swapping $A[4]$ with $A[9]$, as shown in (c), node 4 is fixed up, and the recursive call $\text{MaxHeapify}(A, 9)$ yields no further change to the data structure.

The procedure **BuildMaxHeap** We can use the procedure MaxHeapify in a bottom-up manner to convert an array $A[1 \dots n]$, where $n = \text{length}[A]$, into a max-heap. The elements in the subarray $A[(\lfloor n/2 \rfloor + 1) \dots n]$ are all leaves of the tree, and so each is a 1-element heap to begin with. The procedure BuildMaxHeap goes through the remaining nodes of the tree and runs BuildMaxHeap on each one. It can be shown that the running time of BuildMaxHeap on a node of height h as $O(h)$.

```

BuildMaxHeap(A)
1: heap-size[A] <- length[A]
2: for i <- floorOf(length[A]/2) downto 1
3:   do MaxHeapify(A, i)

```

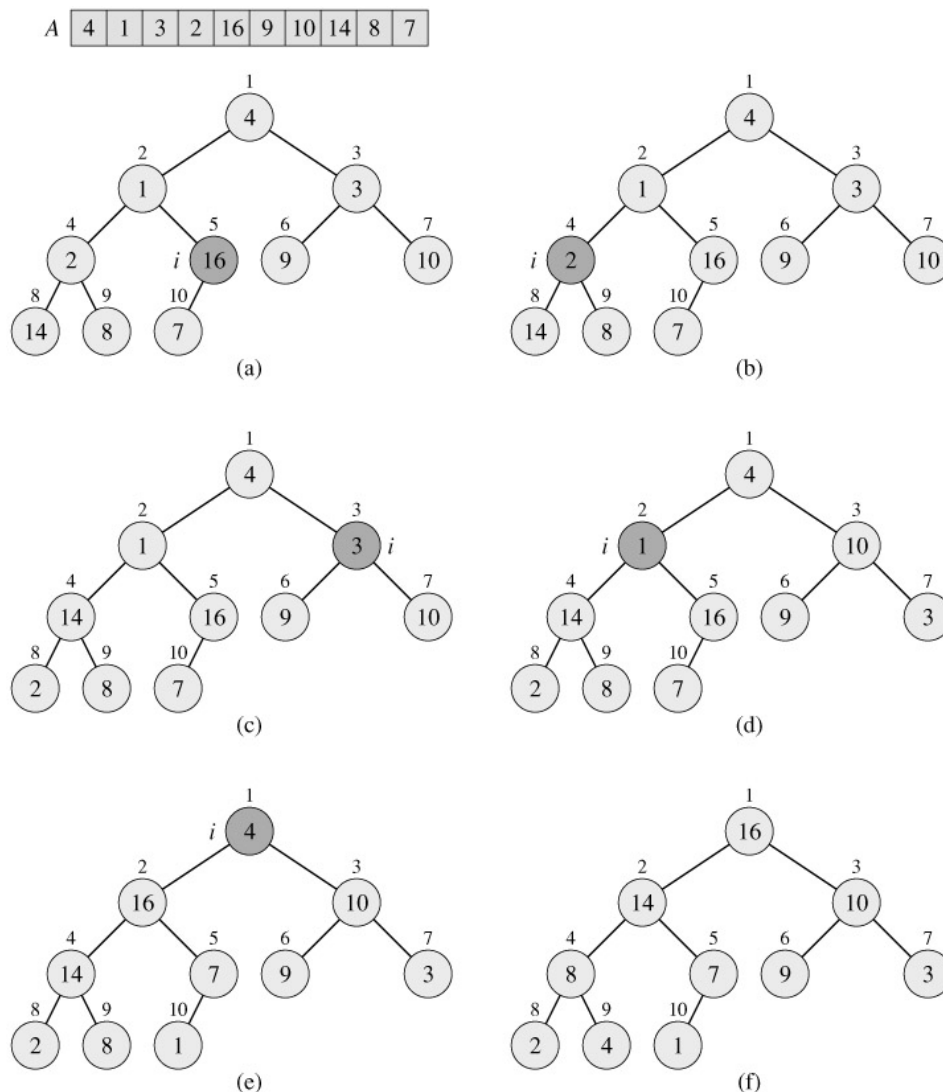


Figure 6: The operation of BuildMaxHeap, showing the data structure before the call to MaxHeapify in line 3 of BuildMaxHeap. (a) A 10-element input array A and the binary tree it represents. The figure shows that the loop index i refers to node 5 before the call MaxHeapify(A, i). (b) The data structure that results. The loop index i for the next iteration refers to node 4. (c)-(e) Subsequent iterations of the for loop in BuildMaxHeap. Observe that whenever MaxHeapify is called on a node, the two subtrees of that node are both max-heaps. (f) The max-heap after BuildMaxHeap finishes.

The **Heapsort** algorithm starts by using BuildMaxHeap to build a max-heap on the input array $A[1 \dots n]$, where $n = \text{length}[A]$. Since the maximum element of the array is stored at the root $A[1]$, it can be put into its correct final position by exchanging it with $A[n]$. If we now "discard" node n from the heap (by decrementing $\text{heap-size}[A]$), we observe

that $A[1 \dots (n - 1)]$ can easily be made into a max-heap. The children of the root remain max-heaps, but the new root element may violate the max-heap property. All that is needed to restore the max-heap property, however, is one call to $\text{MaxHeapify}(A, 1)$, which leaves a max-heap in $A[1 \dots (n - 1)]$. The Heapsort algorithm then repeats this process for the max-heap of size $n - 1$ down to a heap of size 2. The Heapsort procedure takes time $O(n \log n)$, since the call to BuildMaxHeap takes time $O(n)$ and each of the $n - 1$ calls to MaxHeapify takes time $O(\log n)$.

Heapsort(A)

```

1: BuildMaxHeap(A)
2: for i <- length[A] downto 2
3:   do exchange A[1] <-> A[i]
4:     heap-size[A] <- heap-size[A] - 1
5:     MaxHeapify(A, 1)

```

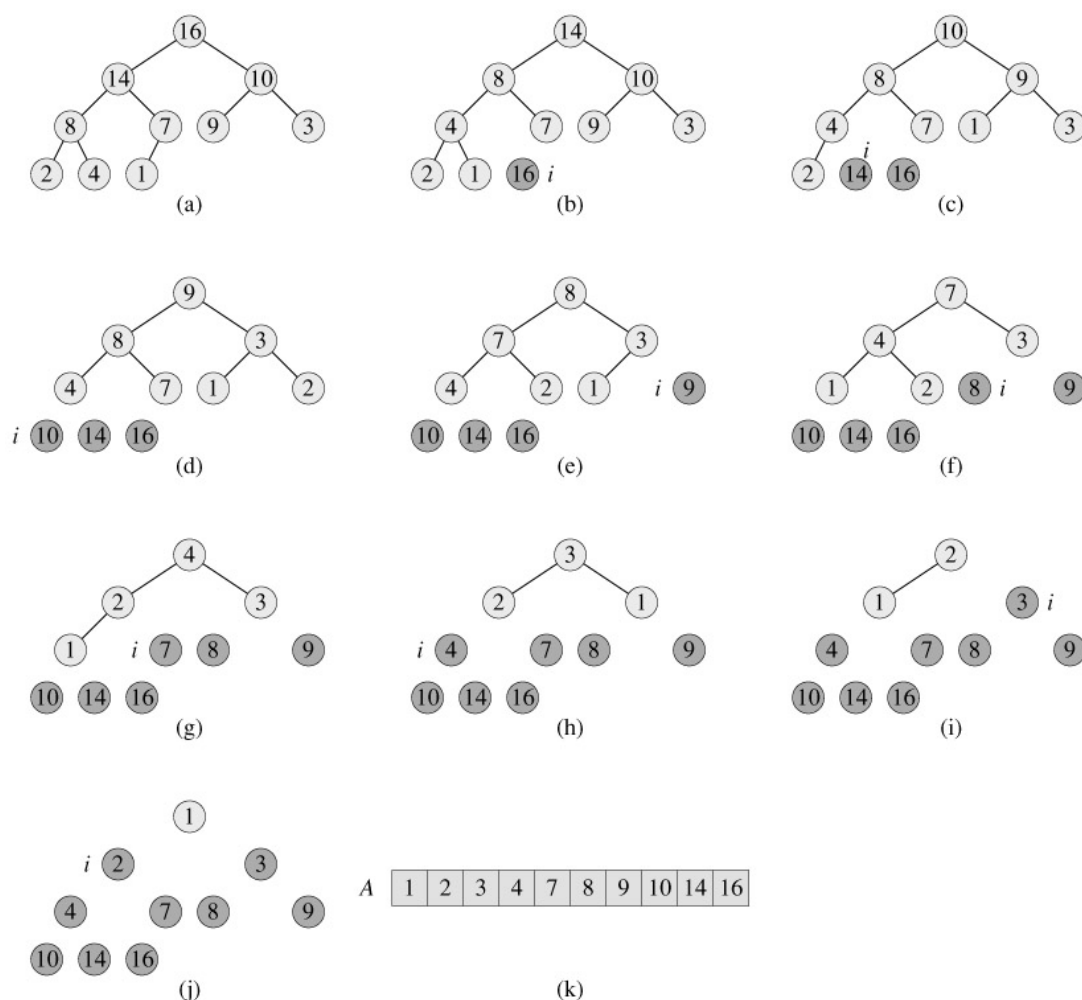


Figure 7: The operation of Heapsort. (a) The max-heap data structure just after it has been built by BuildMaxheap . (b)-(j) The max-heap just after each call of MaxHeapify in line 5. The value of i at that time is shown. Only lightly shaded nodes remain in the heap. (k) The resulting sorted array A .