Sorting (and an introduction to recurrences)

Sorting in $O(n^2)$ Time

Insertion Sort.

- At each step k, ensure the first k elements of the list are sorted.
- Requires finding the right spot for k^{th} element, shifting others.

Bubble Sort.

- Each iteration steps through the list, comparing adjacent elements and swapping as necessary
- After each iteration k, the largest k numbers have bubbled to the top of the list and are in order.
- Continue while at least one swap occurs in a given iteration.

Selection Sort.

- At each step k, find the k^{th} smallest number (i.e. the smallest remaining unsorted number) and put it in its proper location.
- Swap with element currently there if necessary.

Sorting in O(n log n) using Divide-and-Conquer

Divide-and-conquer.

- Break up problem into several parts.
- Solve each part recursively.
- Combine solutions to sub-problems into overall solution.

Most common usage.

- Break up problem of size n into two equal parts of size $\frac{1}{2}$n.
- Solve two parts recursively.
- Combine two solutions into overall solution in linear time.

Consequence.

- Divide-and-conquer: n log n.

Mergesort

Mergesort.

- Divide array into two halves.
- Recursively sort each half.
- Merge two halves to make sorted whole.

```
MERGESORT (X, n)
1.  if (n == 1) RETURN X
2.  middle = n/2 (round down)
3.  A = {x_1, x_2, …, x_middle}
4.  B = {x_middle+1, x_middle+2, …, x_n}
5.  As = MERGESORT (A, middle)
6.  Bs = MERGESORT (B, n-middle)
7.  RETURN MERGE (As, Bs)
```

A Useful Recurrence Relation

Def. T(n) = number of comparisons to mergesort an input of size n.

Mergesort recurrence.

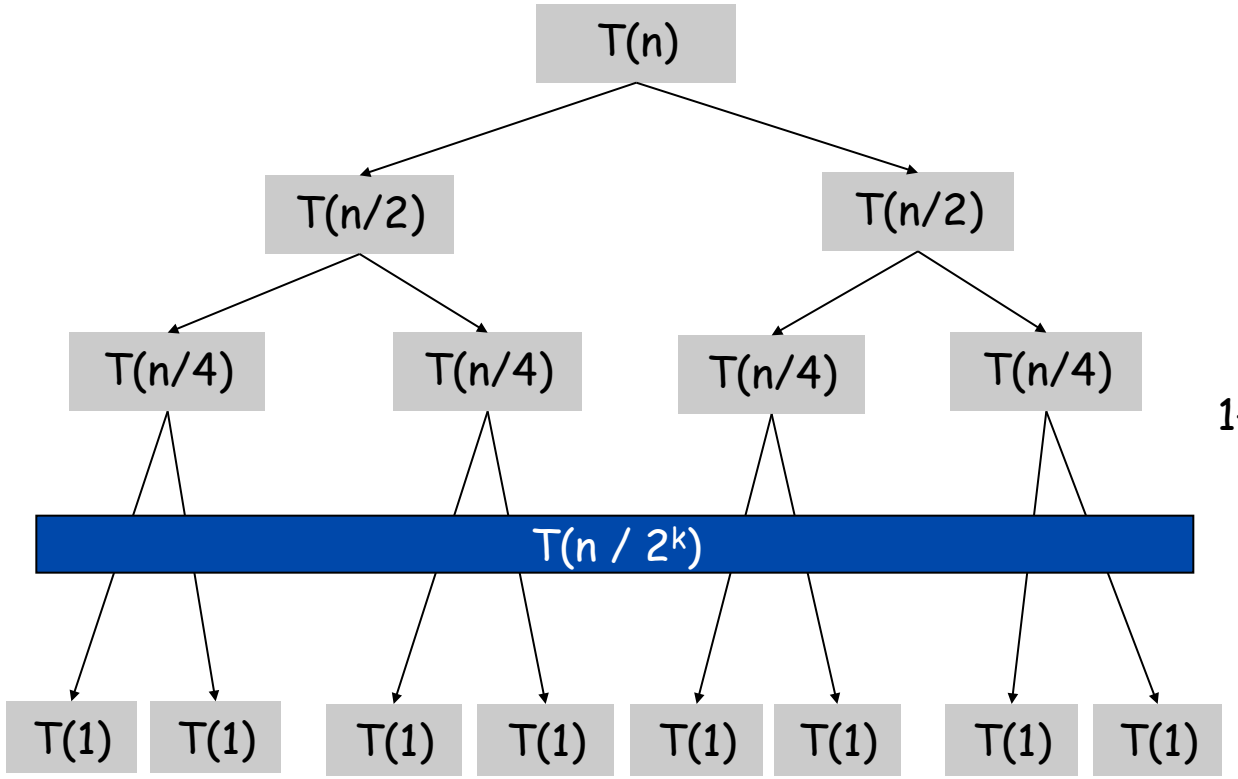
$$\mathrm{T}(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2\,T(n/2) + O(n) & \text{otherwise} \end{cases}$$

Solution. T(n) = $O(n \log_2 n)$.

Assorted proofs. We describe three ways to prove this recurrence: recursion tree, telescoping and induction.

Proof by Recursion Tree

$$T(n) \leq \begin{cases} d & \text{if } n = 1 \\ 2\,T(n/2) + dn & \text{otherwise} \end{cases}$$



d n

2 d (n/2)

4 d (n/4)

$1+\log_2 n$

. . .

2^k d (n / 2^k)

. . .

n d

(dn) ($\log_2 n+1$)

Another O(n log n) Sorting Algorithm

HeapSort: Underlying data structure is a heap

Definition: A min-heap is a balanced binary tree with nodes storing keys, with the property that for every parent w and child v, key(w) ≤ key(v)

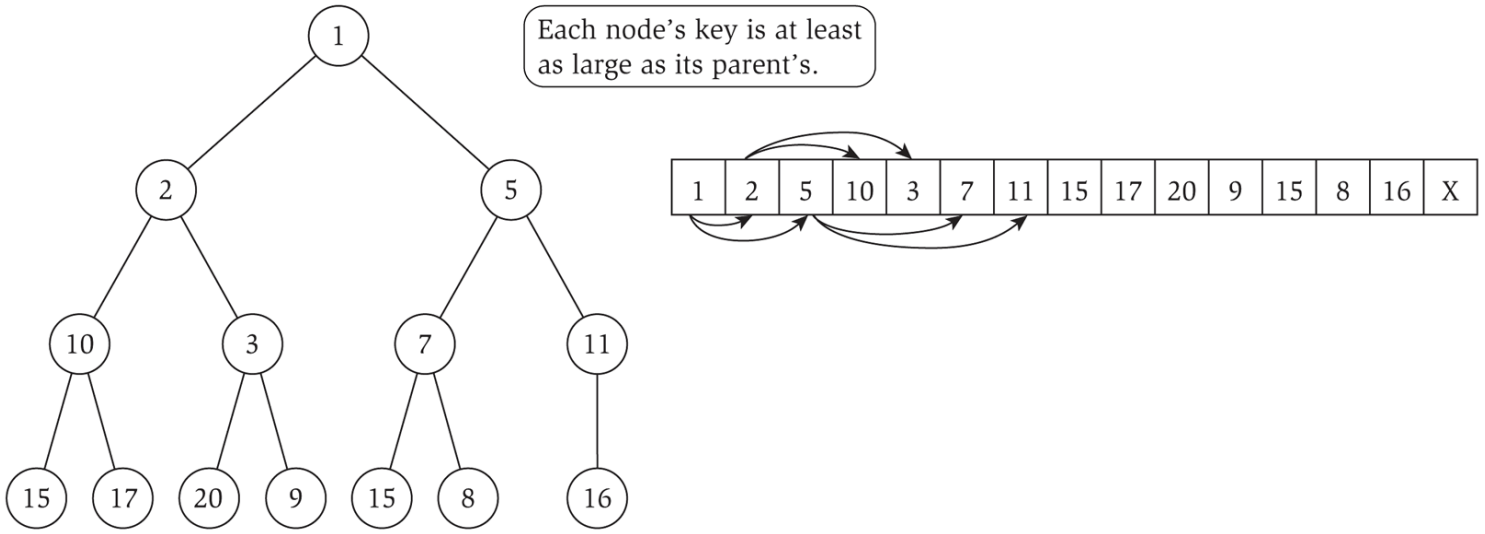


Figure 2.3 Values in a heap shown as a binary tree on the left, and represented as an array on the right. The arrows show the children for the top three nodes in the tree.

Heap Data Structure

Heap data structure: used to implement priority queues

Priority queues: supports operations such as:

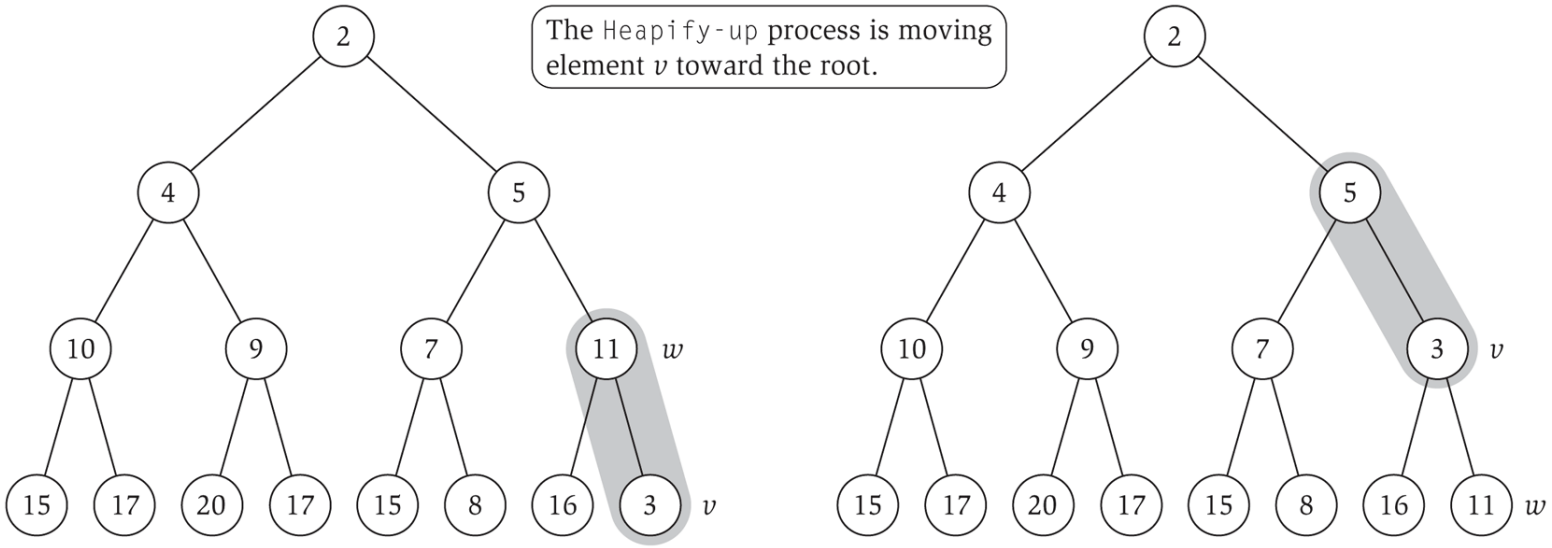
add_key
change_key
extract_min

Priority queue applications: used to efficiently identify the highest priority element of a dynamically changing list, without worrying about maintaining a completely sorted list. The highest priority element may have the minimum key (or maximum using an inverted heap called a max-heap).

E.g. scheduling processes on a computer.

Adding a Key to a Heap

Keys are added in the next available location in the tree. Then, we call the HEAPIFY-UP function to bubble the new key up the tree as necessary.




```
HEAPIFY-UP (H, i)
1.  while (i > 1) and (H[i] < H[Parent(i)]) do
2.     swap entries H[i] and H[Parent(i)]
3.     i = Parent(i)
```

```
ADD (H, key)
1.  H.length++
2.  H[H.length] = key
3.  HEAPIFY-UP (H, H.length)
```

What's the running time of ADD?

Changing a Key in a Heap

If the key changes to a smaller value, we can just call HEAPIFY-UP to move it higher in the tree as necessary.

Otherwise, we may need to move it down the tree.

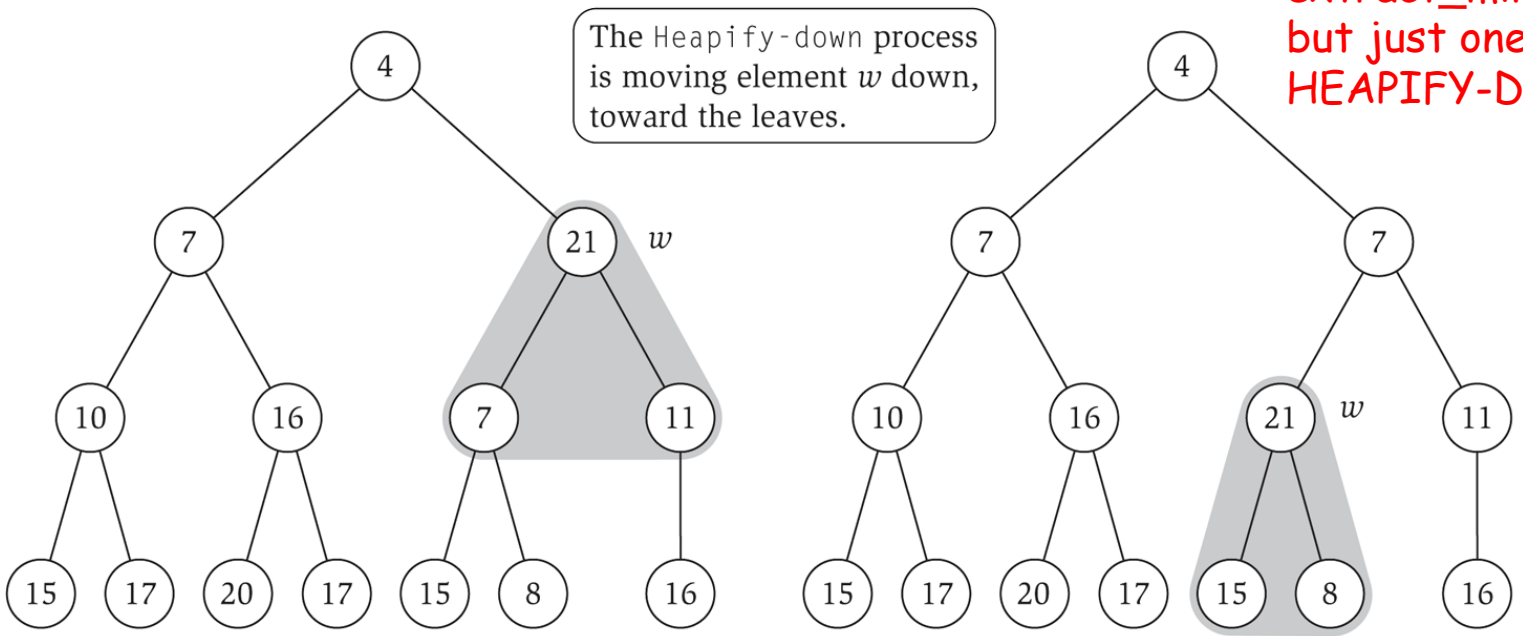
```
HEAPIFY-DOWN (H, i)
1.  n = H.length
2.  while (LeftChild(i) ≤ n) and (H[i] > H[LeftChild(i)])
        or (RightChild(i) ≤ n) and (H[i] > H[RightChild(i)])
3.      if (RightChild(i) > n) or
        (H[LeftChild(i)] < H[RightChild(i)]) then
4.          j = LeftChild(i)
5.      else
6.          j = RightChild(i)
7.      swap entries H[i] and H[j]
8.      i = j
```

Extracting the Minimum from a Heap

The minimum is easily identified and removed. But how to reform a proper heap from the remaining elements?

Take the last element of the heap and make it the new root.
Call HEAPIFY-DOWN as necessary to reorder.
What's the running time of extract_min?

Note this illustration doesn't show the whole extract_min process, but just one step of HEAPIFY-DOWN



Sorting using a Heap

What steps need to be taken to sort a sequence using a heap data structure?

First, the heap needs to be built. The elements of the sequence can be added to the heap (using the heap add_key operation) one at a time

Second, we iterate n times, each time extracting the minimum value from the heap (using the extract_min operation) and fixing the remaining structure so that it is valid heap again with that element removed.

Sorting using a Heap

What is the total running time, and running time of each step?

```
HEAPSORT (A)
1.  H = BUILD_HEAP (A)
2.  n = A.length
3.  for i=1 to n do
4.      A[i] = EXTRACT_MIN (H)
```

This pseudo-code suggests the usage of extra memory, but heapsort can be done in place.

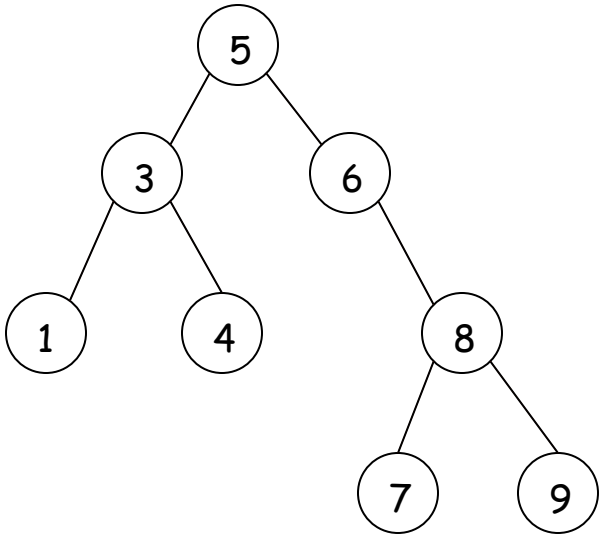
```
BUILD_HEAP (A)
1.  initially H is empty
2.  n = A.length
3.  for i=1 to n do
4.      ADD (H, A[i])
```

```
EXTRACT_MIN (H)
1.  min = H[1]
2.  H[1] = H[H.length]
3.  H.length--
4.  HEAPIFY_DOWN (H, 1)
5.  RETURN min
```

Related Data Structure

Binary Search Tree – has the property that for every node:

- The left subtree contains only nodes that have a key less than the node
- The right subtree contains only nodes that have a key greater than the node
- The left and right subtrees are also binary search trees



Sorting Faster than O(n log n)

Comparison-based sorts require at least n log n comparisons.

Is it possible to sort in linear time?

- Radix sort – a non-comparison sort. Belongs to a class of algorithms known as distribution-based sorts.

- Idea:

  sort by digit, starting from the least significant digit.

  Each next digit preserves the order already established for previous digits when breaking ties.

Radix Sort

Running time: $O(d(n + r))$ where d is the number of digits in the representation, and r is the radix (base) used to represent the numbers.

Linear time as long as d is a constant, and r is no worse than $O(n)$.

```
RADIXSORT (A)
1. d = length of the longest element in A
2. for j=1 to d do
3.     COUNTSORT(A,j) // a stable sort to sort A
                      // by the j-th last digit
```

(Below assumes we're using the normal base (radix) of 10)

```
COUNTSORT (A,j)
1. let B[1..10] be an array of (empty) linked-lists
2. n = A.length
3. for i=1 to n do
4.     let x be the j-th last digit of A[i]
5.     add A[i] at the end of the linked-list B[x]
6. copy B[1] followed by B[2], then B[3], etc. to A
```