Chapter 1: Introduction to Algorithms

Part IV

Contents

- Algorithms (Part I)
- Mathematical preliminaries
  - Order of growth (Part I)
  - Summation (Part II)
  - Recurrence (Part II)
- Proof of correctness (Part III)
- Analysis of sorting algorithms
- Sorting in Linear time

Analysis of Sorting Algorithms

Sorting

One of the most common data-processing applications

The process of arranging a collection of items/records in a specific order

The items/records consist of one or more **fields** or **members**

One of these fields is designated as the "**sort key**" in which the records are ordered

Sort order: Data may be sorted in either ascending sequence or descending sequence

Sorting

Input

A sequence of numbers

$a_1, a_2, a_3, a_4, \dots, a_n$

Example: 2 5 6 1 12 10

Sorting

Output

A permutation of the sequence of numbers

$b_1, b_2, b_3, b_4, \dots, b_n$

Example: 1 2 5 6 10 12

Sorting

Types

1. Internal sort
   - All of the data are held in primary memory during the sorting process
   - Examples: Insertion, selection, heap, bubble, quick, shell sort
2. External sort
   - Uses primary memory for the data currently being sorted and secondary storage for any data that does not fit in primary memory
   - Examples: merge sort

Sorting

Sort stability

Indicates that data with equal keys maintain their relative input order in the output

365	blue
212	green
876	white
212	yellow
119	purple
212	blue

Unsorted data

119	purple
212	green
212	yellow
212	blue
365	blue
876	white

Stable sort

119	purple
212	blue
212	green
212	yellow
365	blue
876	white

Unstable sort

Sorting algorithms

- Selection Sort ✔
- Insertion Sort ✔
- Merge Sort ✔
- Quick Sort ✔
- Heap Sort

Sorting algorithms

Algorithm	**Best case**	**Worst case**	**Average case**
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Quick sort	$O(n \log n)$	$O(n^2)$	$O(n \log n)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heap sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Quick sort vs merge sort

In merge sort, the divide step does hardly anything, and all the real work happens in the combine step whereas in quick sort, the real work happens in the divide step.

Quicksort works in place.

In practice, quicksort outperforms merge sort, and it significantly outperforms selection sort and insertion sort.

Heap sort

- Is among the fastest sorting algorithms
- Is used with very large arrays
- Uses heap data structure for sorting

Heap

- A *nearly complete binary tree* in which the root contains the largest (or smallest) element in the tree.
- Is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.

Heap property

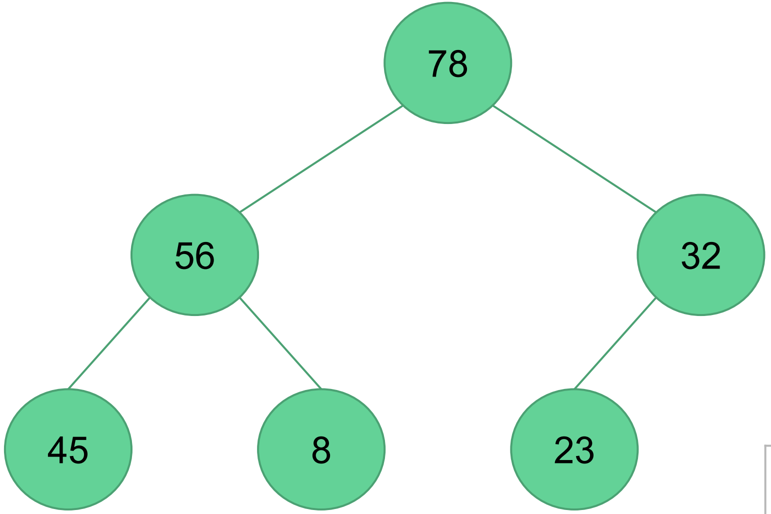
There are two kinds of binary heaps: **max-heaps** and **min-heaps**.

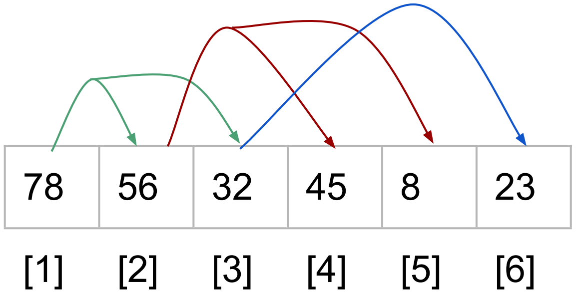
In both kinds, the values in the nodes satisfy a **heap property**.

In a max-heap, the **max-heap property** is that for every node other than the root, **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.

Similarly, in a min-heap, the smallest element is at the root.

Heap





Heap in its array form

PARENT(i)

1 **return** $\lfloor i/2 \rfloor$

LEFT(i)

1 **return** $2i$

RIGHT(i)

1 **return** $2i + 1$

Heap sort

To implement the heap sort using a max-heap, we need two basic algorithms:

1. **Max-heapify**
   Maintains the max-heap property by pushing the root down the tree until it is in its correct position in the heap.
2. **Build-max-heap**
   Produces a max-heap from an unordered input array.

Heap sort

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```

Heap sort

Steps:

1. Convert the array into a max heap
2. Find the largest element of the list (i.e., the root of the heap) and then place it at the end of the list. Decrement the heap size by 1 and readjust the heap
3. Repeat Step 2 until the unsorted list is empty

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```

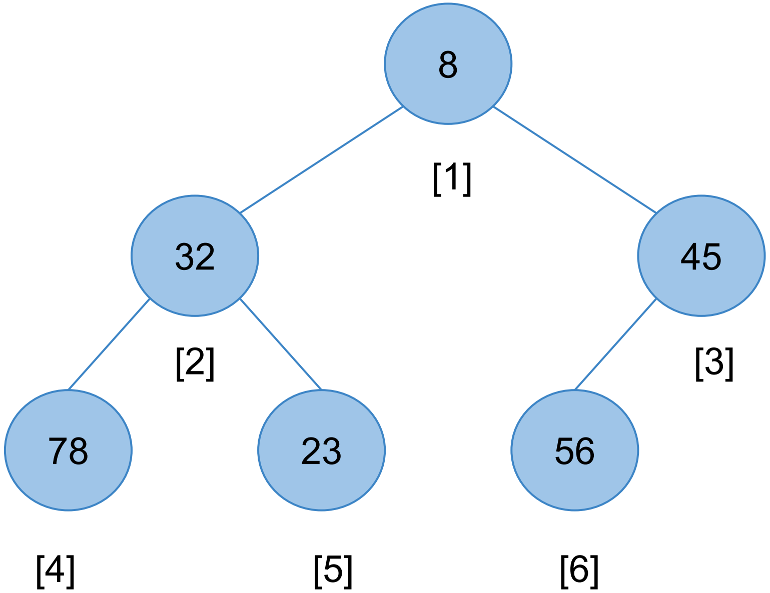
Heap sort

Example:

Sort the following data using heap sort:

8	32	45	78	23	56

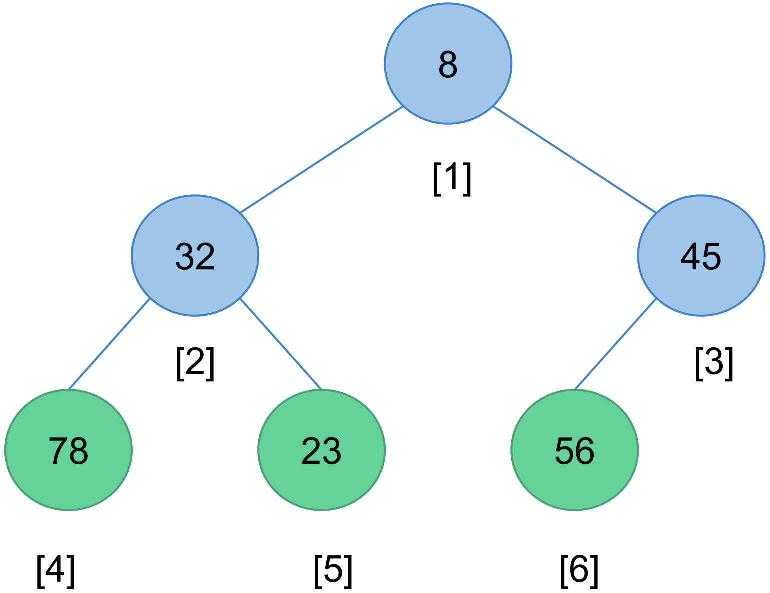
Heap sort



Convert the array into a max heap

8	32	45	78	23	56

Heap sort



Convert the array into a max heap

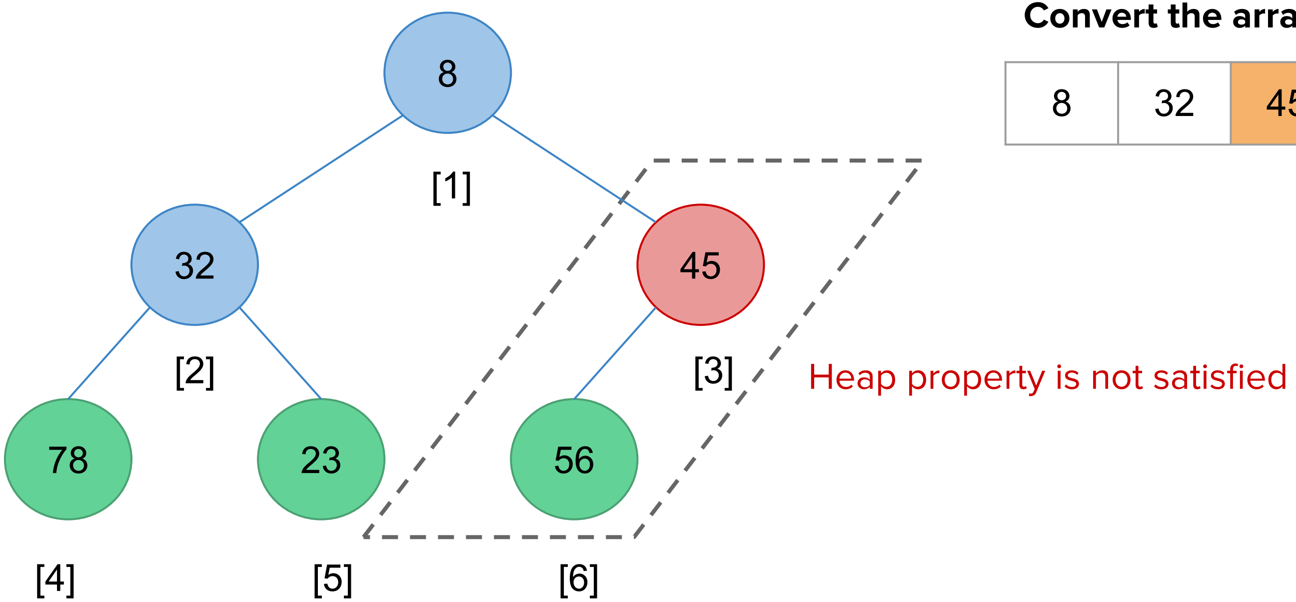
8	32	45	78	23	56

Starting from the index, i, of the node just above the leaf level, check if the tree starting at i is a max-heap. If not, fix it (by reheap down)

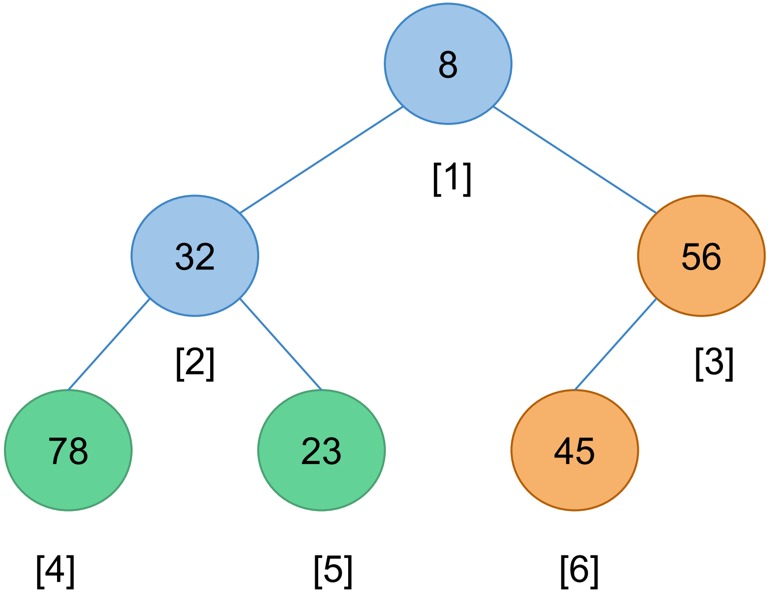
Heap sort

Convert the array into a max heap

8	32	45	78	23	56



Heap sort

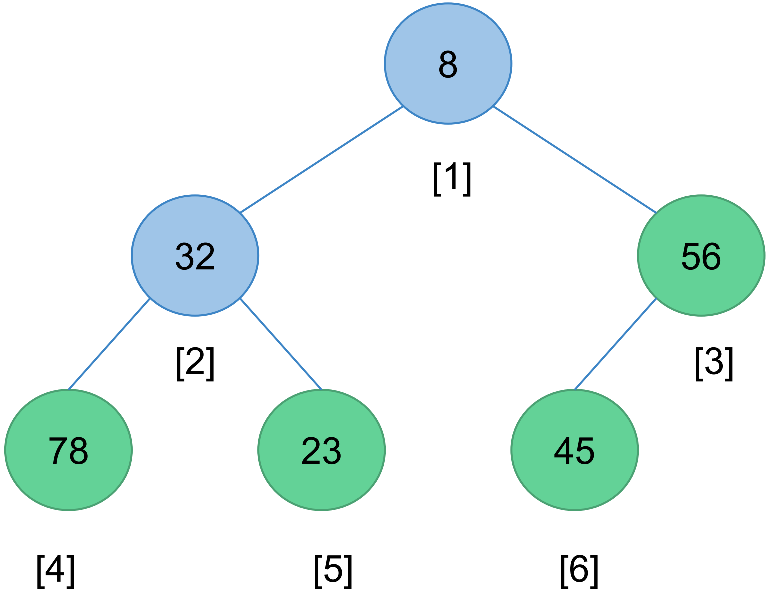


Convert the array into a max heap

8	32	56	78	23	45

Heap property is not satisfied. Therefore, swap them

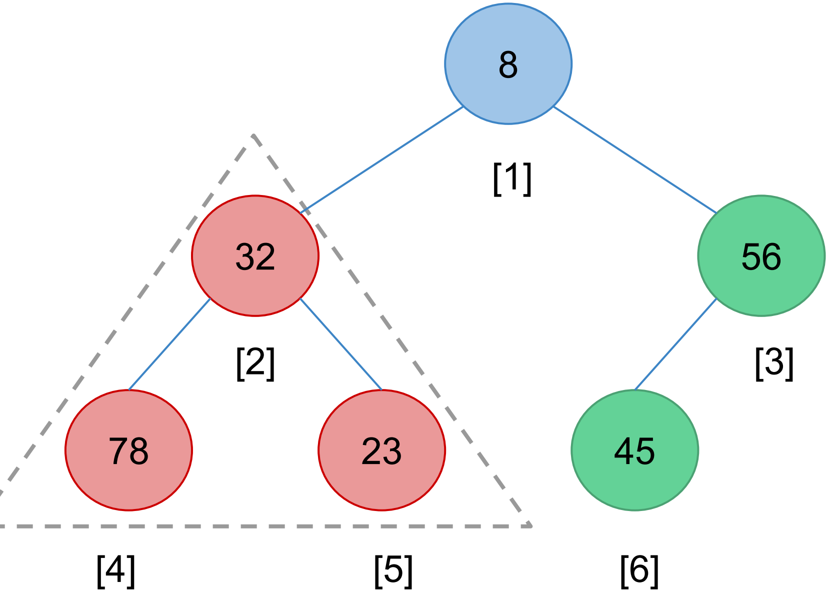
Heap sort



Convert the array into a max heap

8	32	56	78	23	45

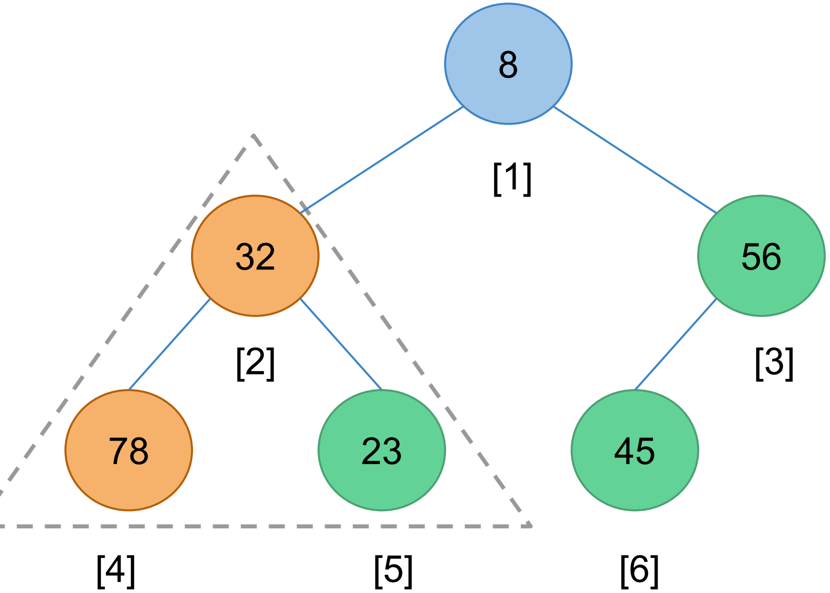
Heap sort



Convert the array into a max heap

8	32	56	78	23	45

Heap sort

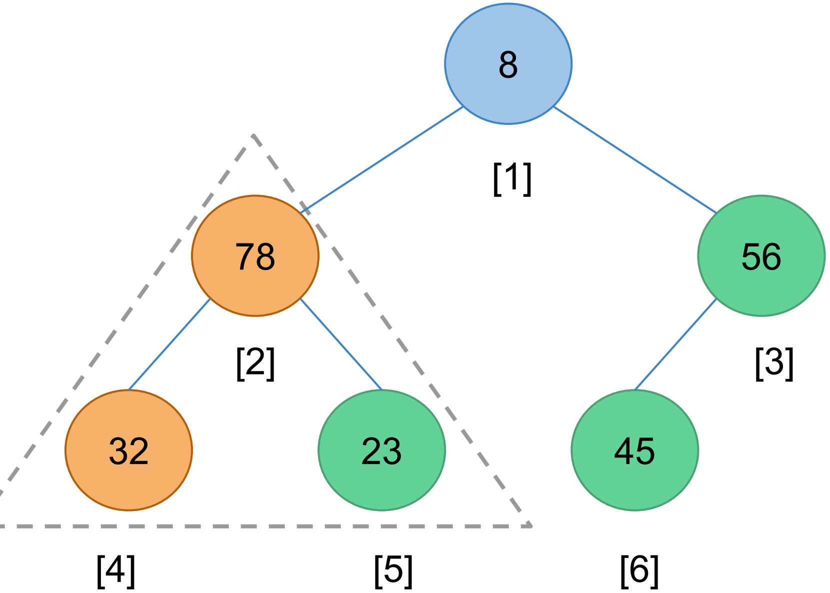


Convert the array into a max heap

8	32	56	78	23	45

Swap the root with the largest child

Heap sort

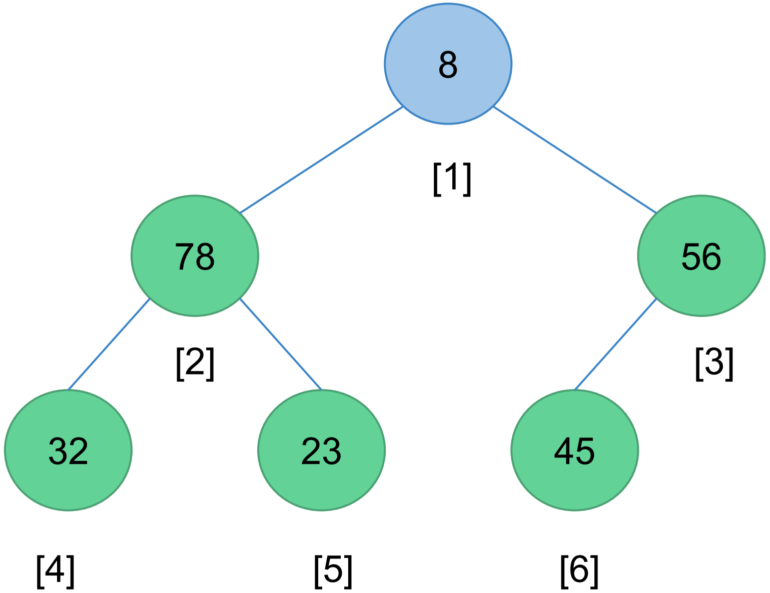


Convert the array into a max heap

8	78	56	32	23	45

Swap the root with the largest child

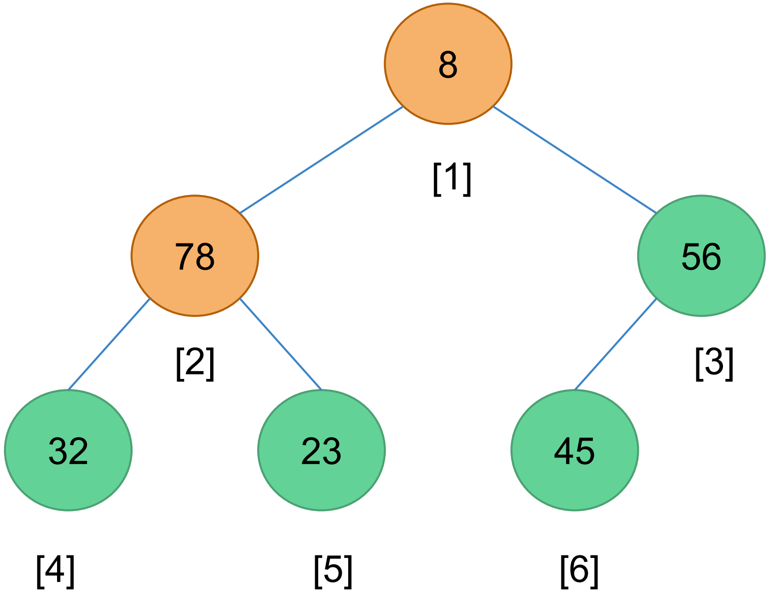
Heap sort



Convert the array into a max heap

8	78	56	32	23	45

Heap sort

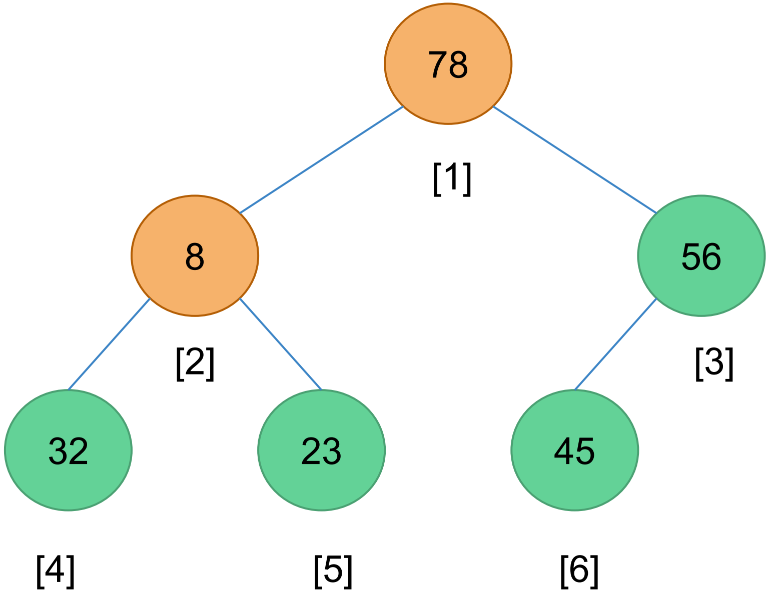


Convert the array into a max heap

8	78	56	32	23	45

Swap the root with the largest child

Heap sort

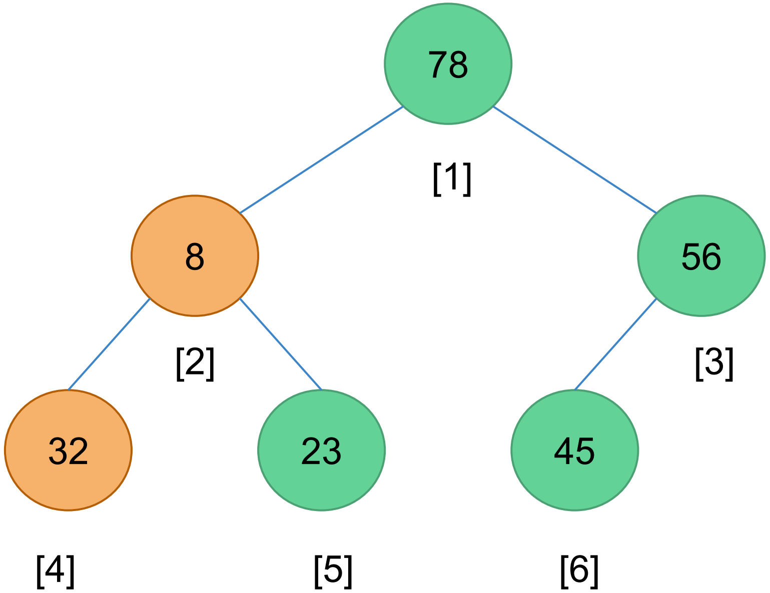


Convert the array into a max heap

78	8	56	32	23	45

Swap the root with the largest child

Heap sort

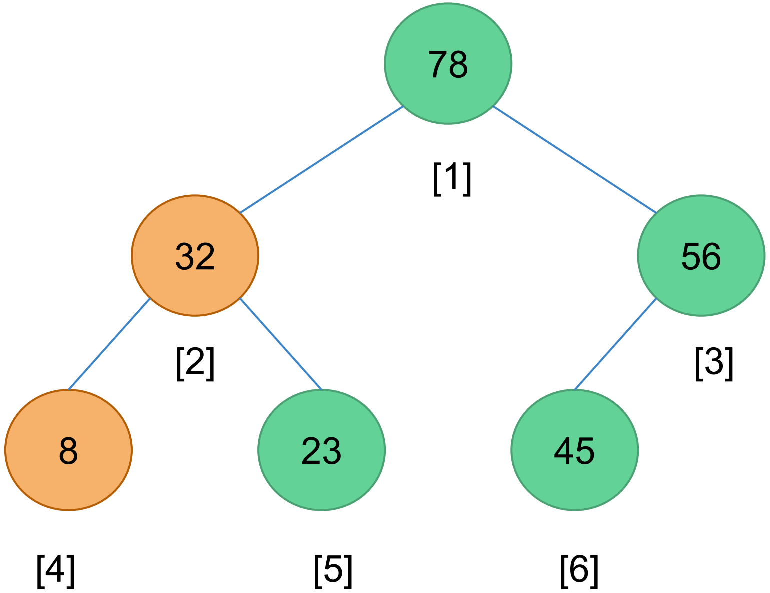


Convert the array into a max heap

78	8	56	32	23	45

Swap the root with the largest child

Heap sort

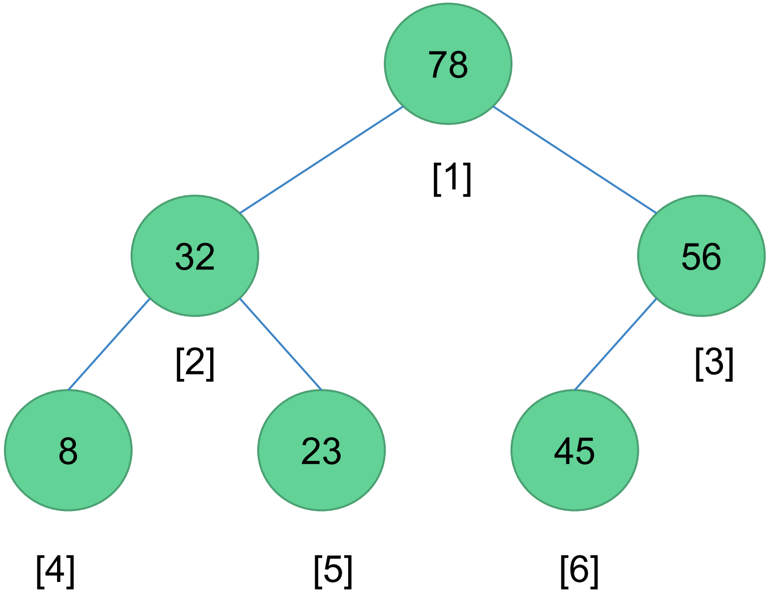


Convert the array into a max heap

78	32	56	8	23	45

Swap the root with the largest child

Heap sort

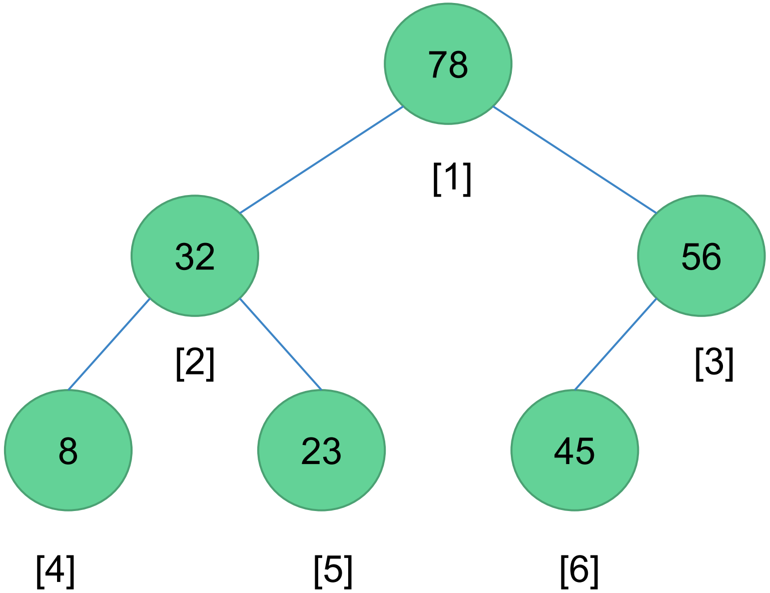


Convert the array into a max heap

78	32	56	8	23	45

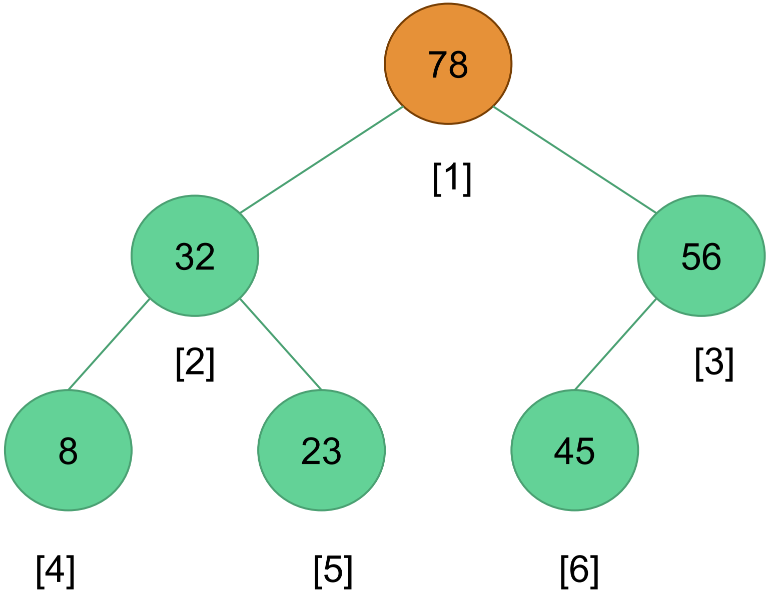
Heap sort

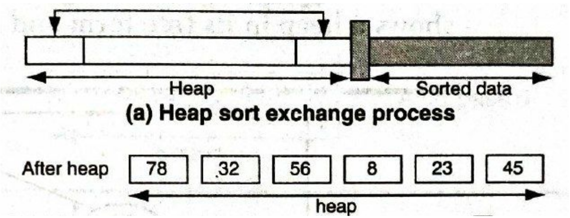
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



Heap sort

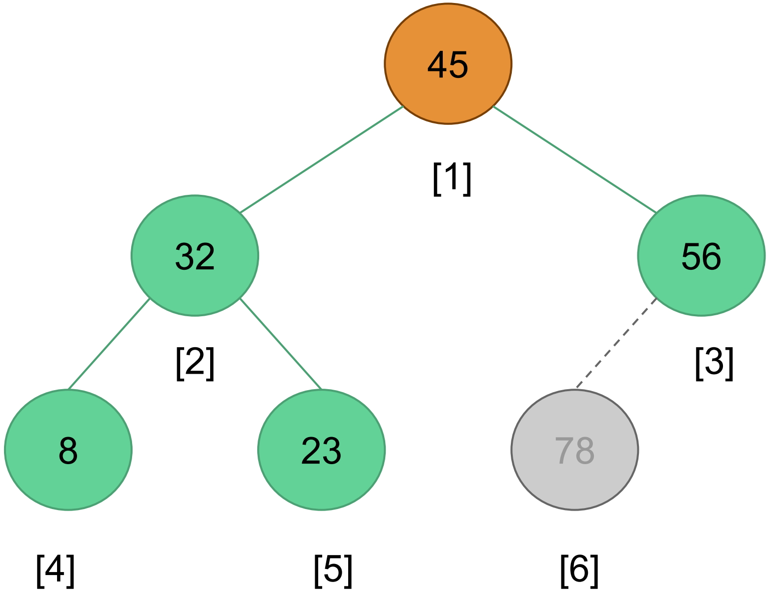
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.

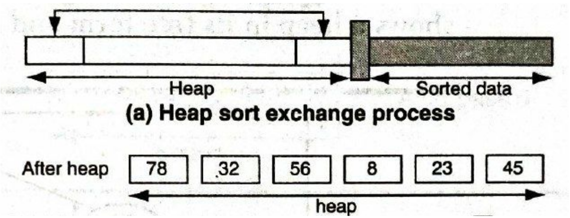




Heap sort

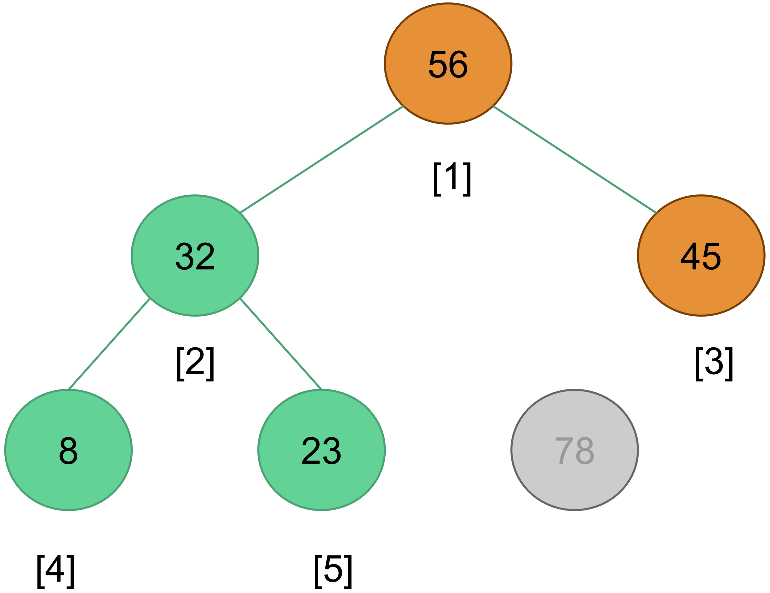
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.

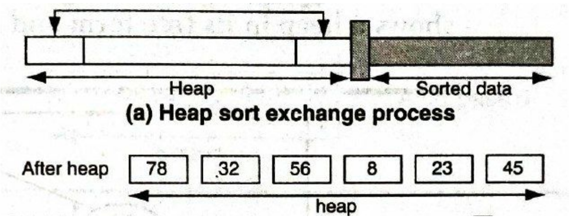




Heap sort

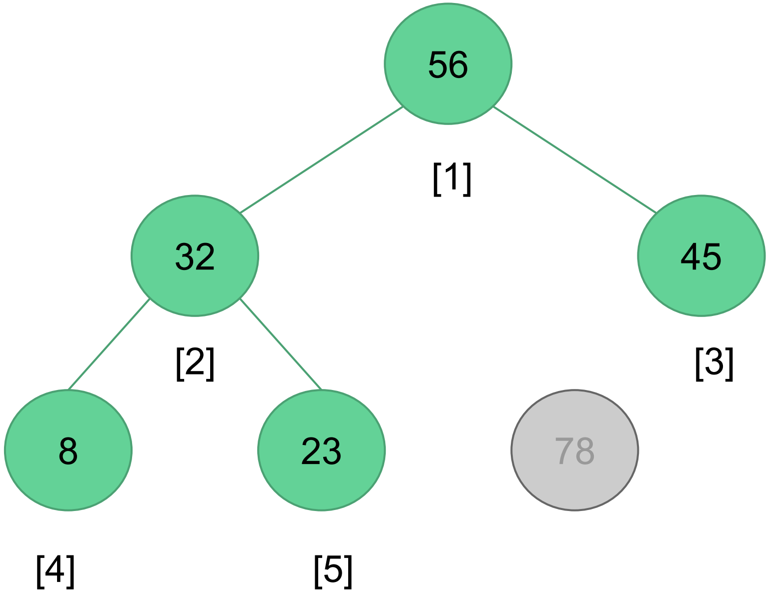
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.

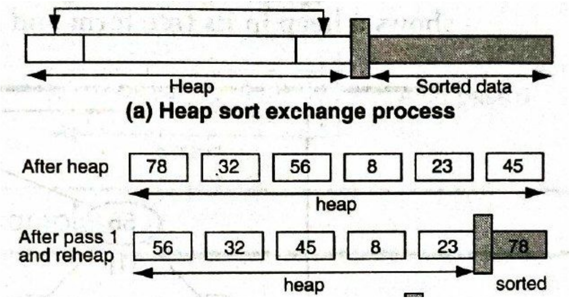




Heap sort

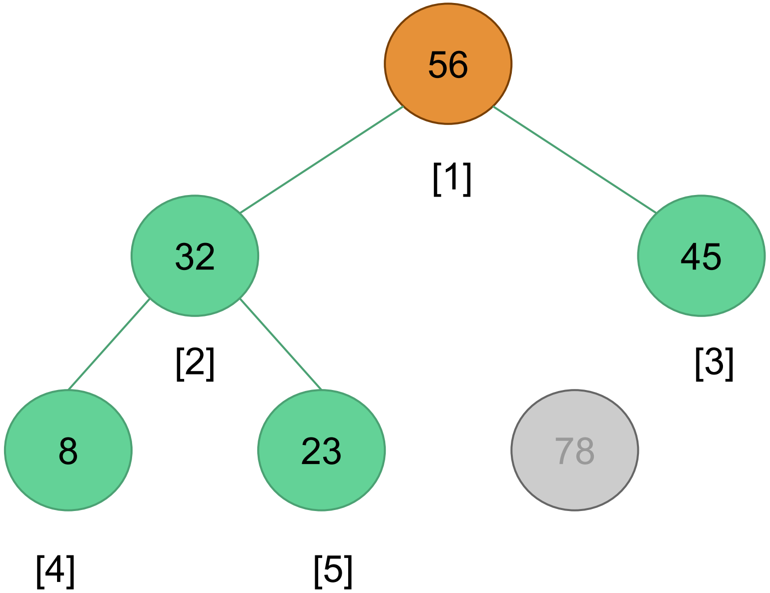
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.

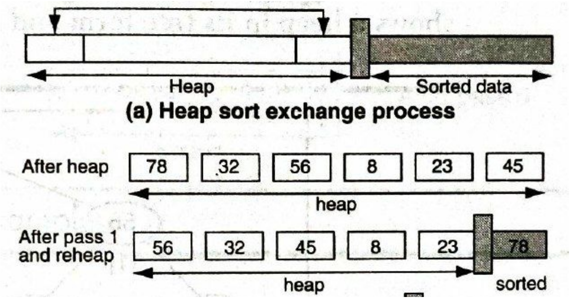




Heap sort

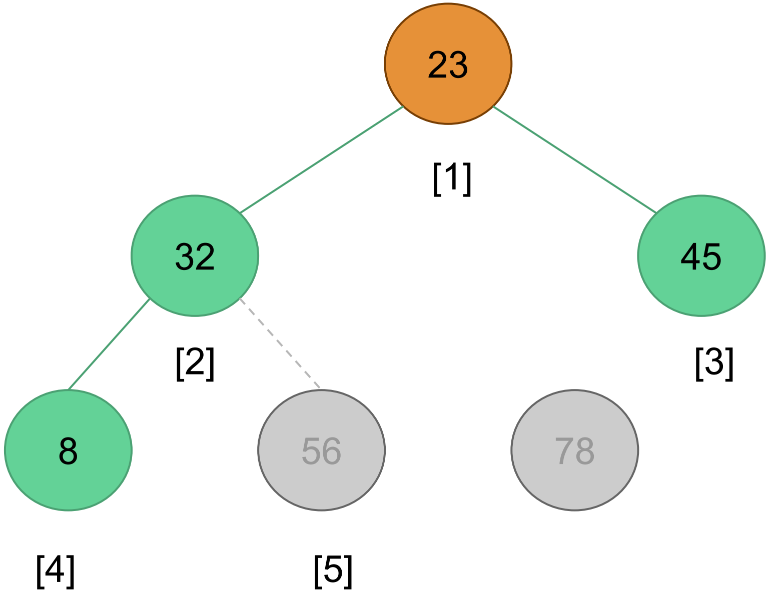
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.

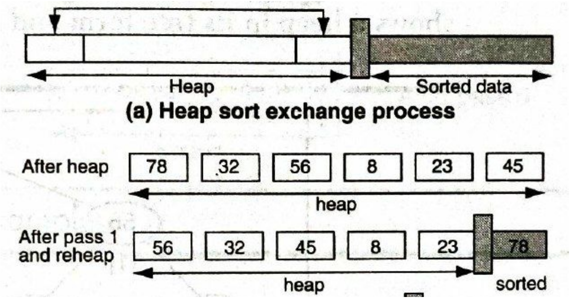




Heap sort

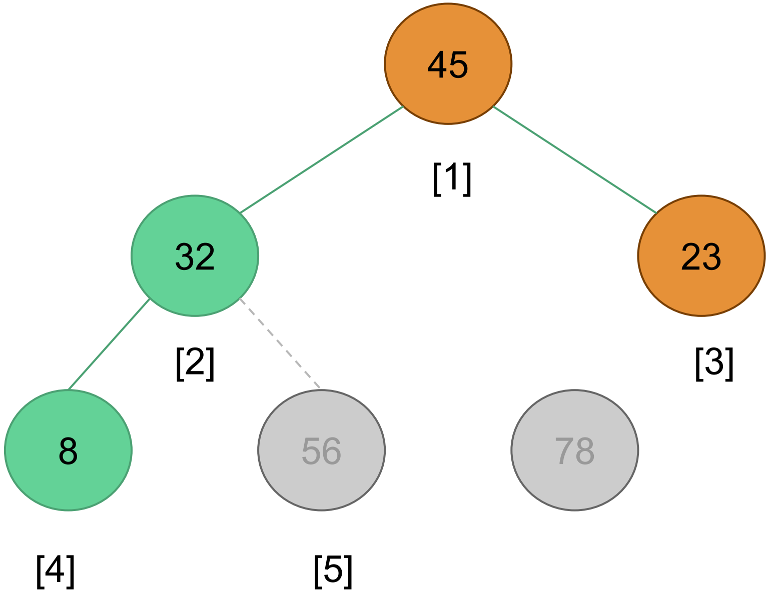
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.

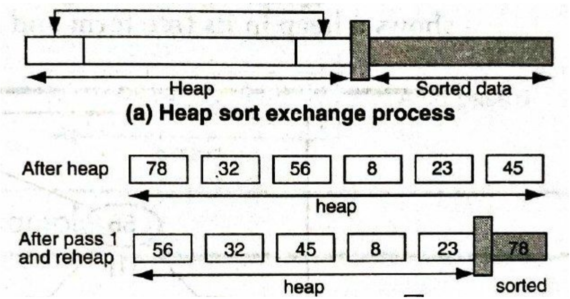




Heap sort

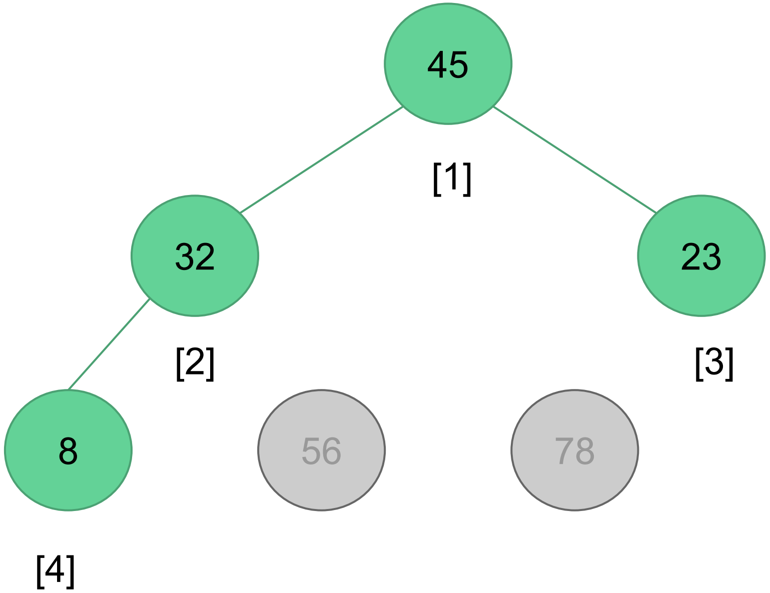
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.

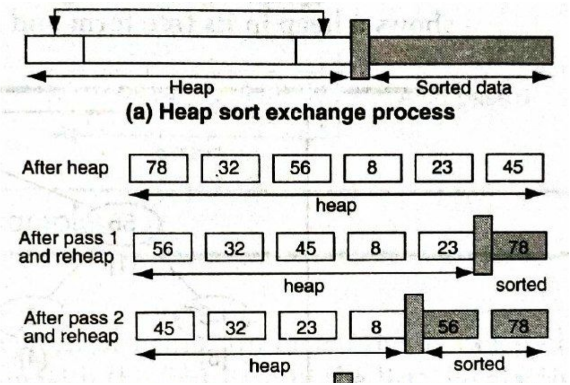




Heap sort

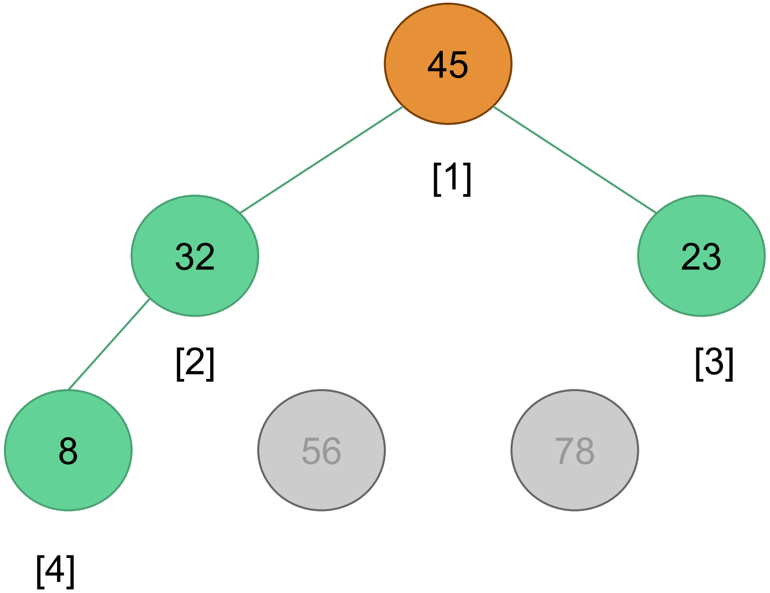
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.

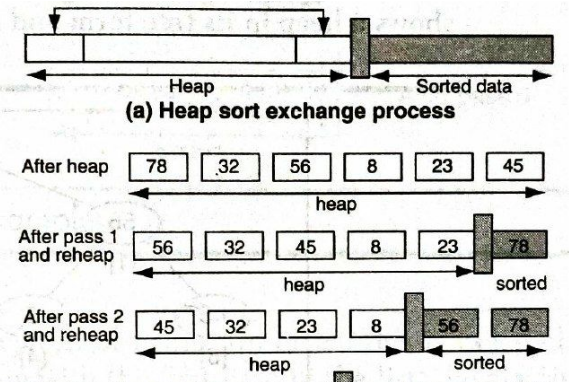




Heap sort

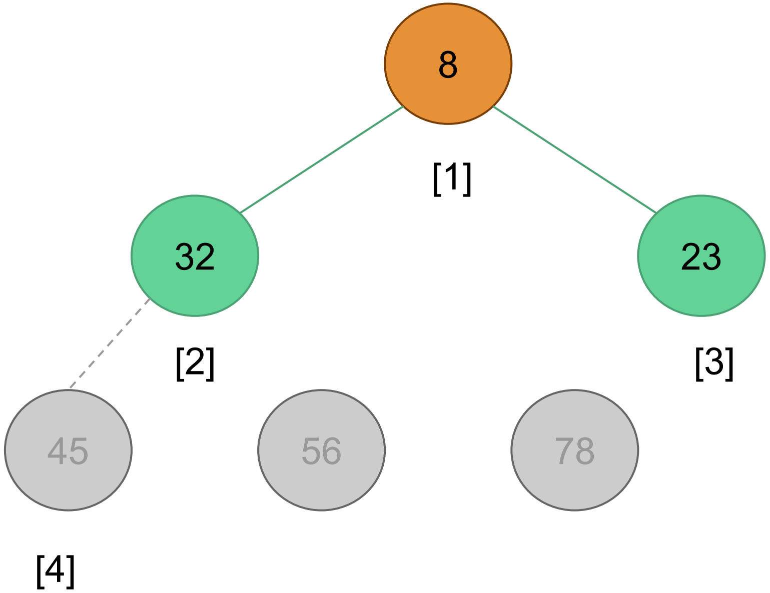
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.

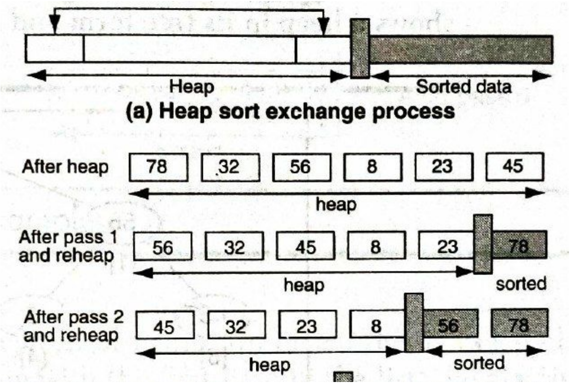




Heap sort

Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.

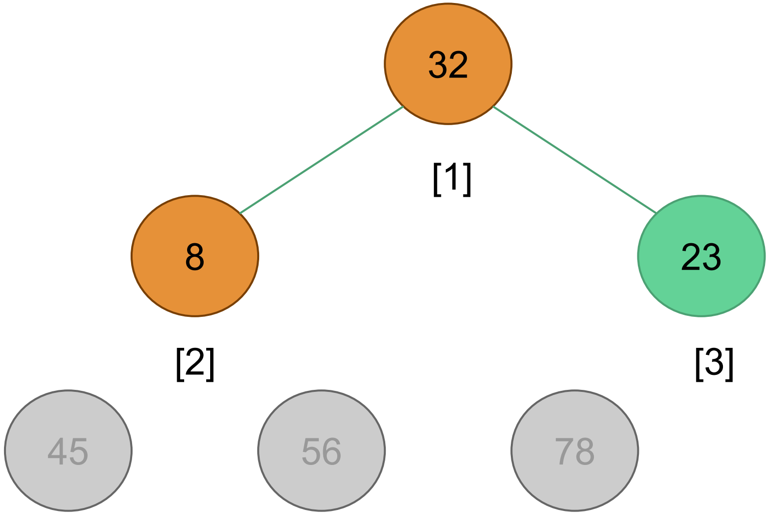


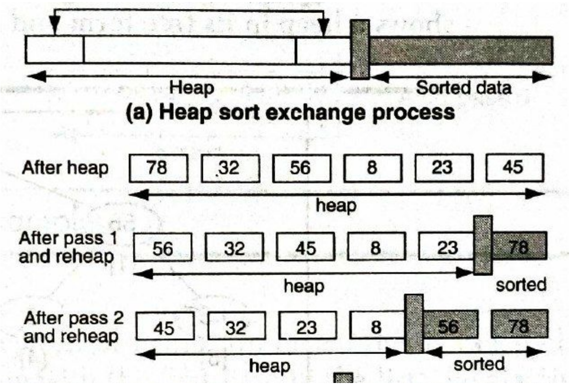


(a) Heap sort exchange process

Heap sort

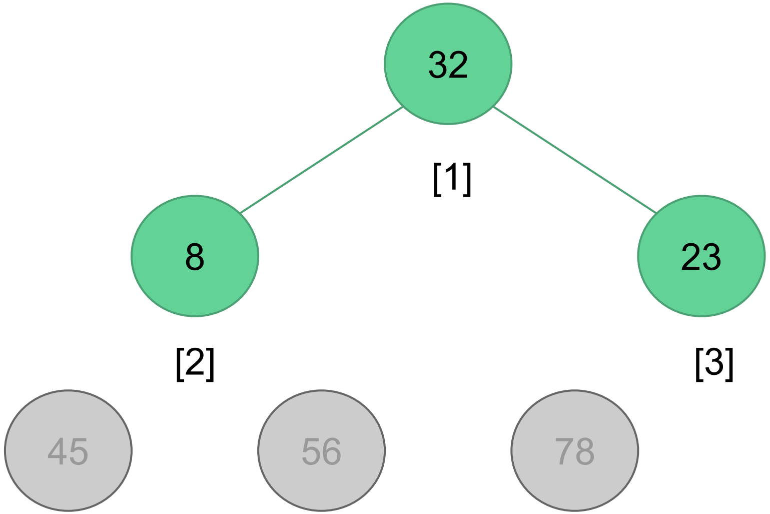
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.

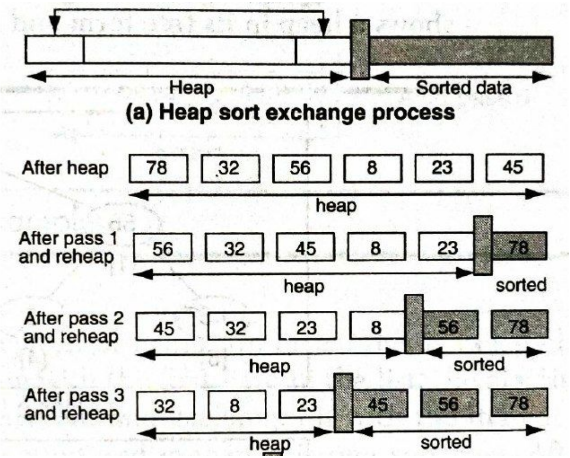




Heap sort

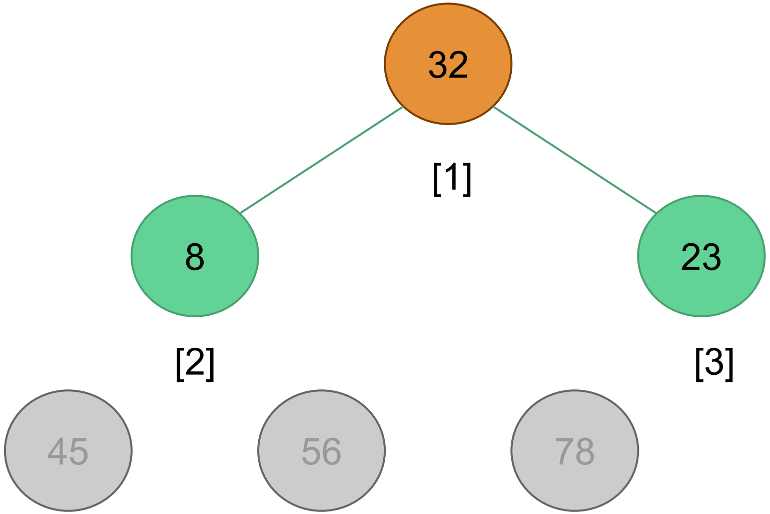
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.

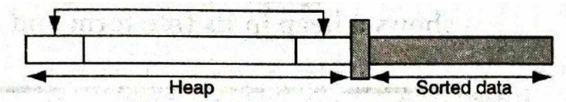




Heap sort

Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.

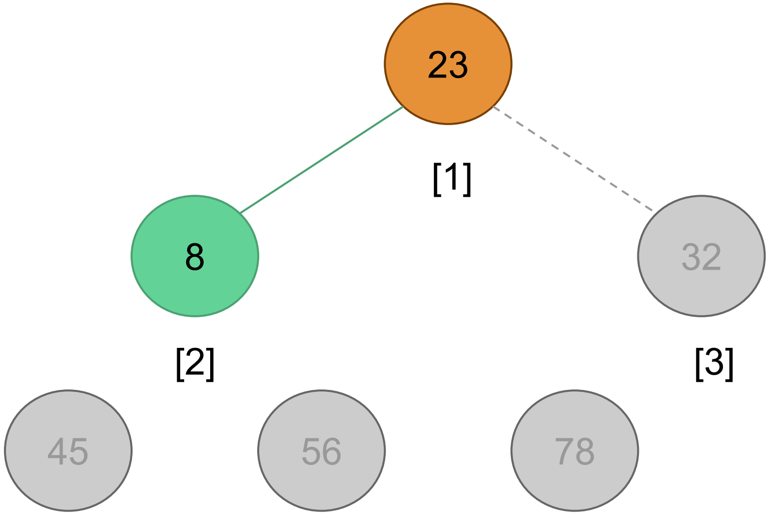


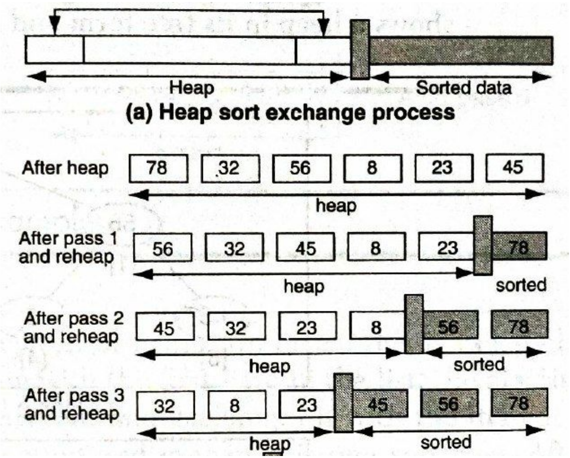


(a) Heap sort exchange process

Heap sort

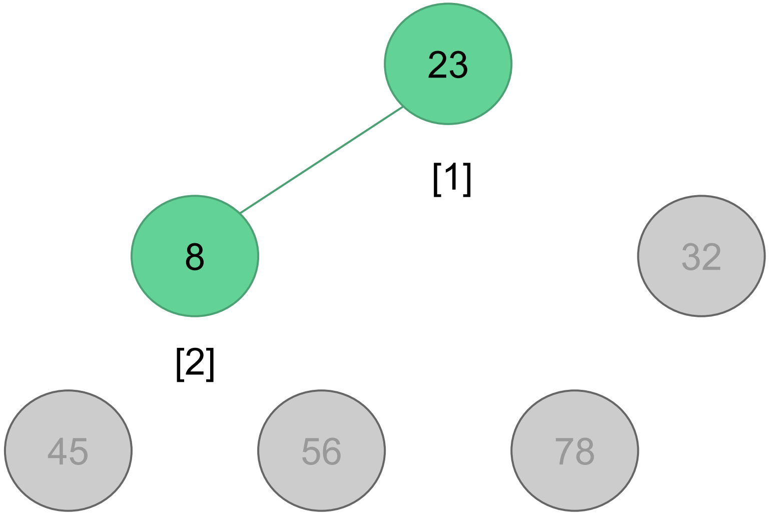
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.

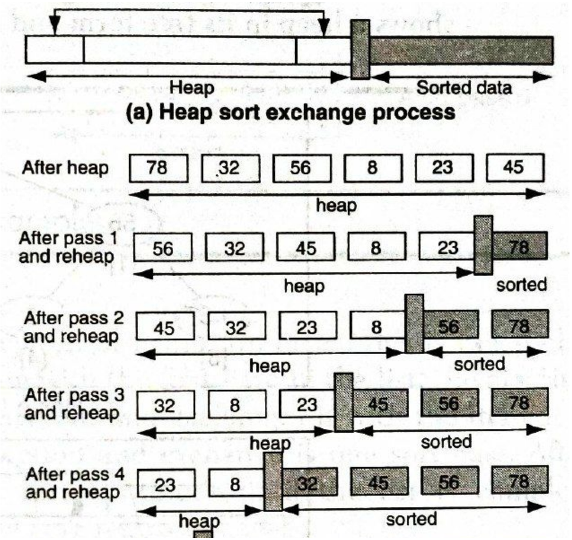




Heap sort

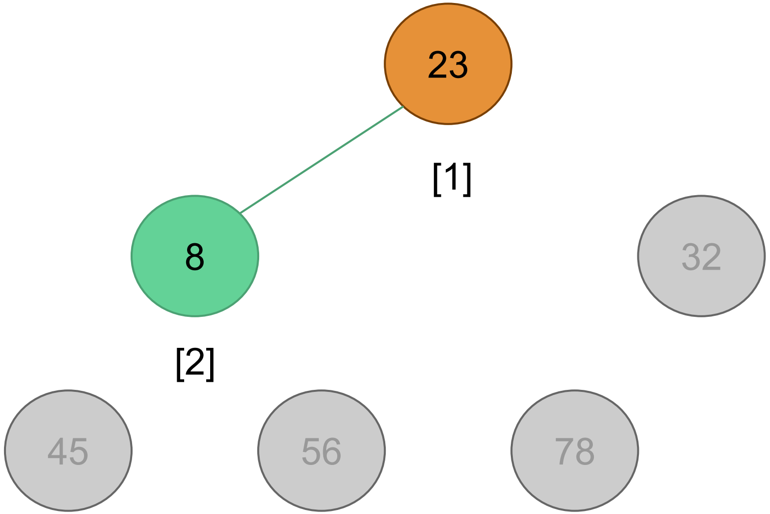
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.

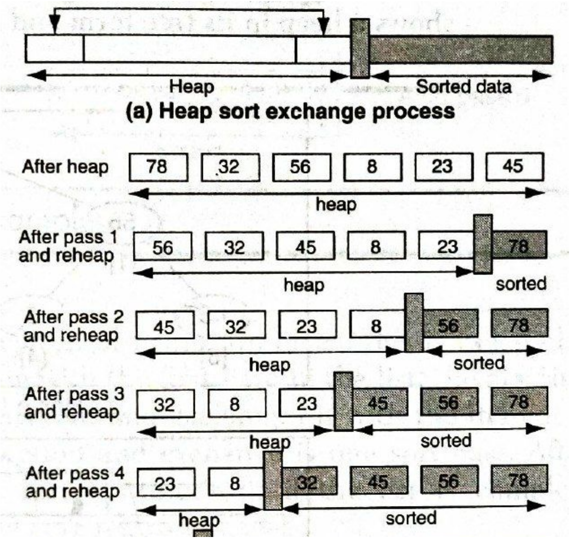




Heap sort

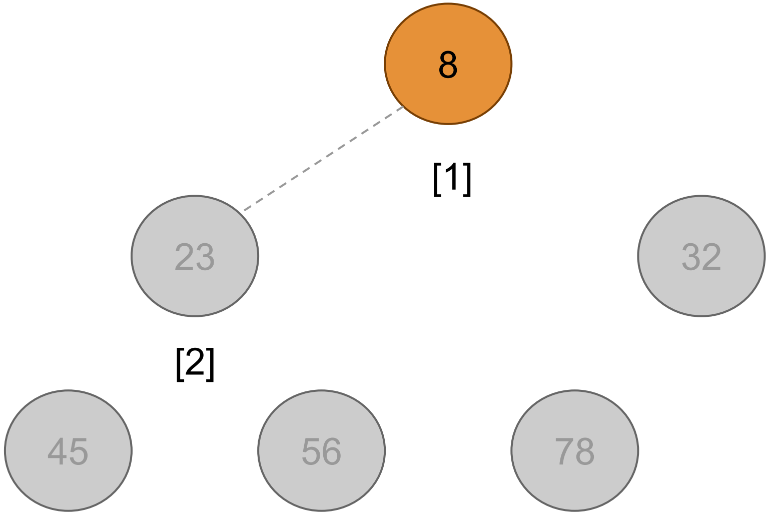
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.

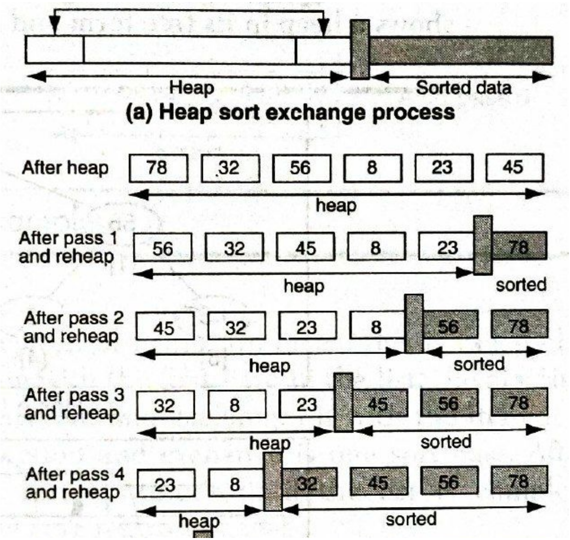




Heap sort

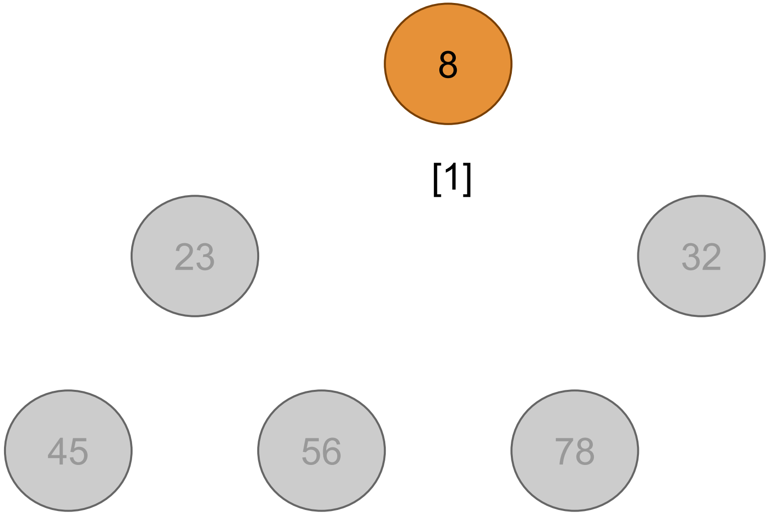
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.

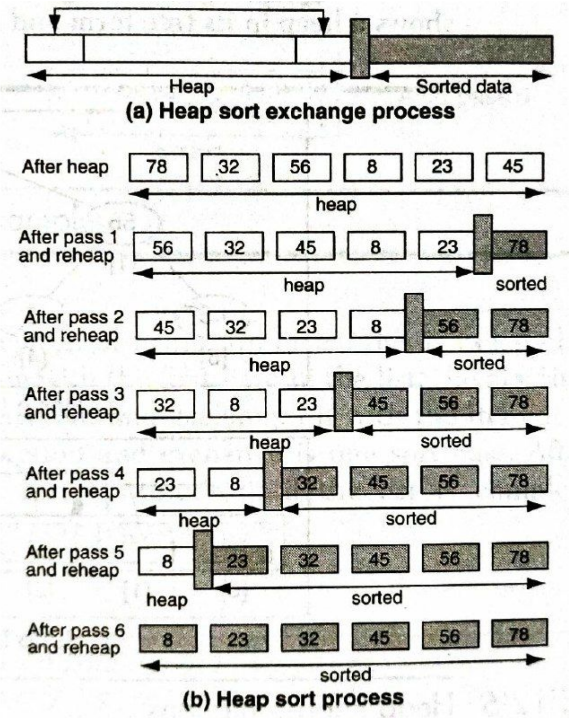




Heap sort

Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.





Analysis of Max-Heapify

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

Analysis of Max-Heapify

```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4      largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7      largest = r
8  if largest ≠ i
9      exchange A[i] with A[largest]
10     MAX-HEAPIFY(A, largest)
```

Running time of lines 1 - 9 = $\theta(1)$

Running time of line 10 = ?

Analysis of Max-Heapify

```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4          largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7          largest = r
8  if largest ≠ i
9          exchange A[i] with A[largest]
10         MAX-HEAPIFY(A, largest)
```

Running time of lines 1 - 9 = $\theta(1)$

Running time of line 10 = ?

The running time T(n) of Max-Heapify on a subtree of size n rooted at a given node i is

$\theta(1)$ + Time to run Max-Heapify on a subtree rooted at one of the children of node i

Analysis of Max-Heapify

```
MAX-HEAPIFY(A, i)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ A.heap-size and A[l] > A[i]
      largest = l
  else largest = i
  if r ≤ A.heap-size and A[r] > A[largest]
      largest = r
  if largest ≠ i
      exchange A[i] with A[largest]
      MAX-HEAPIFY(A, largest)
```

Max-Heapify may need to be called all the way down to the bottom level.

Recall

What is the maximum number of nodes on level i of a binary tree?

What is the maximum number of nodes in a binary tree of height h?

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}$$

Analysis of Max-Heapify

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
        largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
if largest ≠ i
        exchange A[i] with A[largest]
        MAX-HEAPIFY(A, largest)
```

Max-Heapify may need to be called all the way down to the bottom level.

In such case, children's subtrees each will have size of n/2 - 1 if the heap is a complete binary tree, and at most 2n/3 if the bottom level of the tree is exactly half full. (The latter is the worst case.)

$$T(n) \leq T(2n/3) + \theta(1)$$

From the master theorem, this recurrence solves to $T(n) = O(\log_2 n)$

Analysis of Max-Heapify

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

Alternatively, we can characterize the running time, T(n) of Max-Heapify on a node of height h as O(h).

The height of a heap with n nodes is

$$\log_2 n$$

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

Analysis of Build-Max-Heap

```
BUILD-MAX-HEAP(A)
    A.heap-size = A.length
    for i = ⌊A.length/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```

Trivial Analysis: Each call to `Max-Heapify` requires log n time, we make n/2 such calls ⇒ O(n log n).

Analysis of Build-Max-Heap

```
BUILD-MAX-HEAP(A)
    A.heap-size = A.length
    for i = ⌊A.length/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```

Tighter Bound: When `Max-Heapify` is called, the running time depends on how far an element might shift down before the process terminates.

In the worst case, the element might shift down all the way to the leaf level.

To simplify the analysis, let's assume that the heap is a complete binary tree, i.e.
$n = 2^{h+1} - 1$

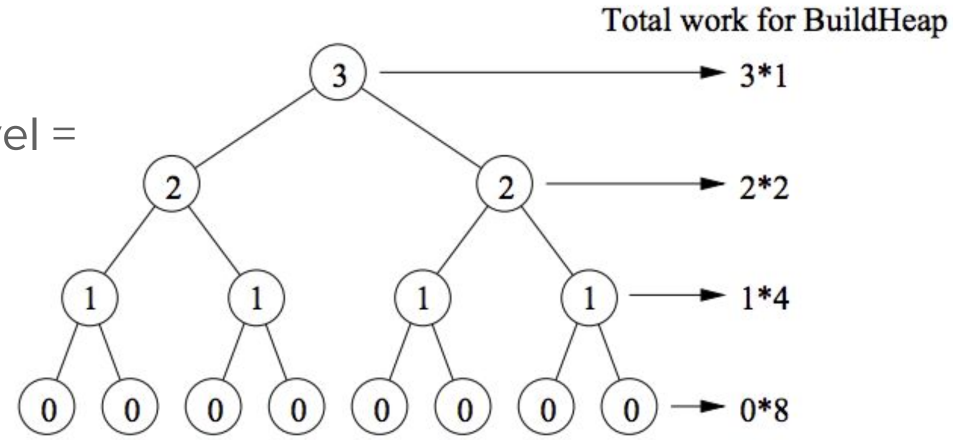
Analysis of build_heap

Tighter bound (contd.)

Number of nodes at the bottommost level = 2^h but we do not call heapify on any of these.

Number of nodes at the next to bottommost level = 2^{h-1}, each might shift down 1 level.

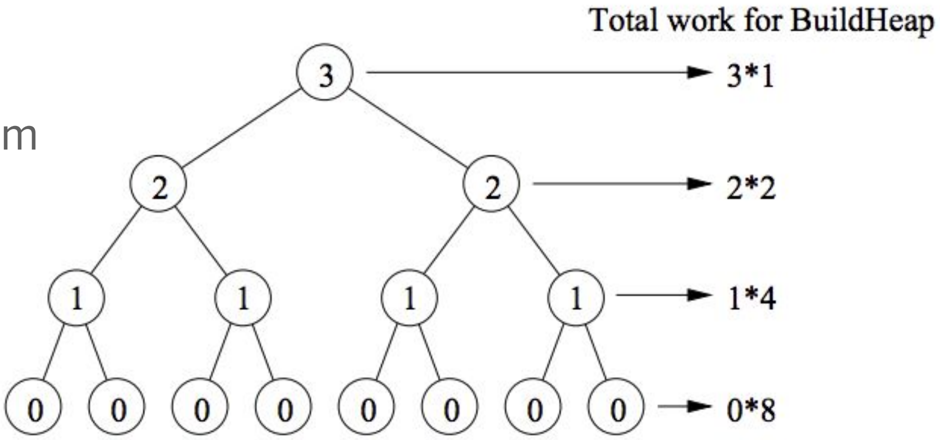
And so on.



Analysis of build_heap

Tighter bound (contd.)

In general, number of nodes at level j from the bottom = 2^{h-j}, each might shift down j level



So, the total time is proportional to

$$T(n) = \sum_{j=0}^{h} j2^{h-j} = \sum_{j=0}^{h} j\frac{2^h}{2^j} = 2^h \sum_{j=0}^{h} \frac{j}{2^j}.$$

Analysis of build_heap

Recall: The infinite geometric series, for any constant x < 1

$$\sum_{j=0}^{\infty} x^j = \frac{1}{1-x}.$$

Taking the derivative of both sides w.r.t. x gives

$$\sum_{j=0}^{\infty} jx^{j-1} = \frac{1}{(1-x)^2} \qquad \sum_{j=0}^{\infty} jx^j = \frac{x}{(1-x)^2},$$

Analysis of build_heap

When x = ½, we get

$$\sum_{j=0}^{\infty} \frac{j}{2^j} = \frac{1/2}{(1-(1/2))^2} = \frac{1/2}{1/4} = 2.$$

Using this as an approximation, we get

$$T(n) = 2^h \sum_{j=0}^{h} \frac{j}{2^j} \leq 2^h \sum_{j=0}^{\infty} \frac{j}{2^j} \leq 2^h \cdot 2 = 2^{h+1}.$$

Since $n = 2^{h+1} - 1$, so we have $T(n) \leq n + 1$, which implies T(n) is O(n).

Also, T(n) is Ω(n) since every element of the array must be accessed at least once.

Therefore, T(n) is Θ(n).

Analysis of heap sort

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size - 1
    MAX-HEAPIFY(A, 1)
```

Analysis of heap sort

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size - 1
5      MAX-HEAPIFY(A, 1)
```

Running time of line 1 = O(n)

Running time of line 3-4 = O(1)

Running time of line 5 = O(log n)

Lines 3-5 are executed n-1 times.

Therefore, the running time of Heap-Sort is

T(n) = O(n) + O(1) + (n - 1) O(log n) = O(n log n)

Stability

Algorithm	**Stable ?**
Selection sort	No
Insertion sort	Yes
Heap sort	No
Merge sort	Yes
Quick sort	No

Sorting in linear time

Sorting algorithms that run in linear time

- Counting sort
- Radix sort
- Bucket sort

Readings

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Chapter 8: Sorting in Linear Time. *In Introduction to algorithms*. MIT press.

Andersson, A., Hagerup, T., Nilsson, S., & Raman, R. (1998). Sorting in linear time?. *Journal of Computer and System Sciences*, 57(1), 74-93.

Comparison sorts

Sorting algorithms studied so far are **comparison sorts**, i.e. they are based on comparisons of input elements.

Sorting algorithm	**Time complexity**
Insertion sort	$O(n^2)$
Selection sort	$O(n^2)$
Merge sort	$O(n \lg n)$
Heap sort	$O(n \lg n)$
Quick sort	$O(n^2)$

Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

Comparison sorts

To get information about an input sequence $\langle a_1, a_2, \ldots, a_n \rangle$, comparison sort algorithms only use comparisons of two elements a_i and a_j

- $a_i < a_j$,
- $a_i \leq a_j$,
- $a_i = a_j$,
- $a_i \geq a_j$,
- $a_i > a_j$

If we assume all input elements are distinct, then comparisons of the form $a_i = a_j$ are useless.
Also, $a_i < a_j$, $a_i \leq a_j$, $a_i \geq a_j$, and $a_i > a$ are equivalent as they yield identical information about the relative order of a_i and a_j.

Comparison sorts can be viewed abstractly in terms of **decision trees**.

The decision tree model

- A **decision tree** is a full binary tree that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size.
- It does not consider control, data movement, and all other aspects of the algorithm but only comparisons.
- In a decision tree, for an input of size n,
    - Each internal node is annotated by $i : j$ for some i and j in the range $1 \le i, j \le n$, indicating a comparison between a_i and a_j
    - Each leaf is annotated by a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$, indicating the ordering $a_{\pi(1)} \le a_{\pi(2)} \le a_{\pi(3)} \ldots \le a_{\pi(n)}$

The decision tree model

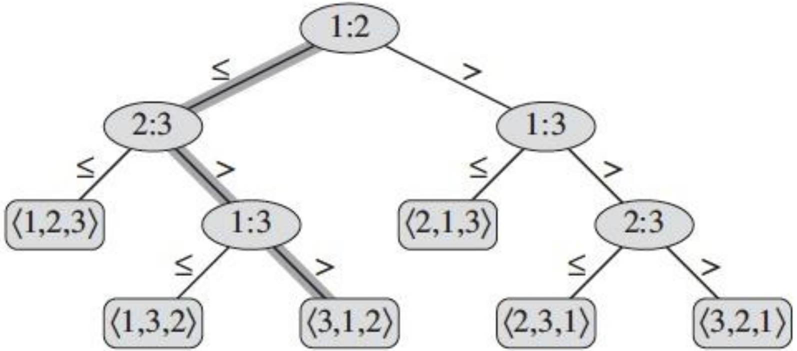
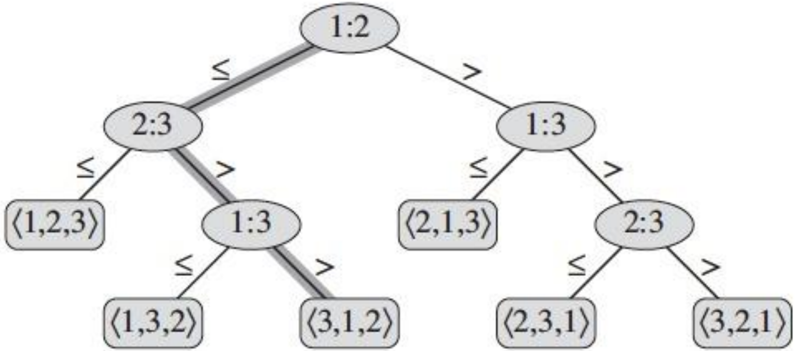


Figure 8.1 The decision tree for insertion sort operating on three elements. An internal node annotated by $i\!:\!j$ indicates a comparison between a_i and a_j. A leaf annotated by the permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ indicates the ordering $a_{\pi(1)} \le a_{\pi(2)} \le \cdots \le a_{\pi(n)}$. The shaded path indicates the decisions made when sorting the input sequence $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$; the permutation $\langle 3, 1, 2 \rangle$ at the leaf indicates that the sorted ordering is $a_3 = 5 \le a_1 = 6 \le a_2 = 8$. There are $3! = 6$ possible permutations of the input elements, and so the decision tree must have at least 6 leaves.

The decision tree model



Because any correct sorting algorithm must be able to produce each permutation of its input, **each of the n! permutations on n elements must appear as one of the leaves of the decision tree.**

Figure 8.1 The decision tree for insertion sort operating on three elements. An internal node annotated by $i:j$ indicates a comparison between a_i and a_j. A leaf annotated by the permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ indicates the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$. The shaded path indicates the decisions made when sorting the input sequence $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$; the permutation $\langle 3, 1, 2 \rangle$ at the leaf indicates that the sorted ordering is $a_3 = 5 \leq a_1 = 6 \leq a_2 = 8$. There are $3! = 6$ possible permutations of the input elements, and so the decision tree must have at least 6 leaves.

Lower bound of comparison sort algorithms

The length of the longest simple path from the root of a decision tree to any of its reachable leaves

= The worst-case number of comparisons that the corresponding sorting algorithm performs

= The height of its decision tree

Lower bound of comparison sort algorithms

Consider a decision tree of height h with l reachable leaves.

Since each of the n! permutations of the input appears as some leaf, we have

$n! \le l$

Since a binary tree of height h has no more than 2^h leaves, we have

$n! \le l \le 2^h$

By taking logarithms, we get $h \ge \lg(n!) = \Omega(n \lg n)$

$\therefore$ Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

Counting sort

- Assumes that each of the n input elements is an integer in the range from 0 to k, for some integer k
- When $k = O(n)$, the sort runs in $\Theta(n)$ time
- Determines, for each input element x, the number of elements less than x.
  - This is how it positions x in its place in the output array
  - If there are 17 elements smaller than x, then x will be assigned position 18
- To sort an array A[1..n], we need two additional arrays
  - B[1..n] holds the sorted output
  - C[1..k] stores the number of repetitions of number in A

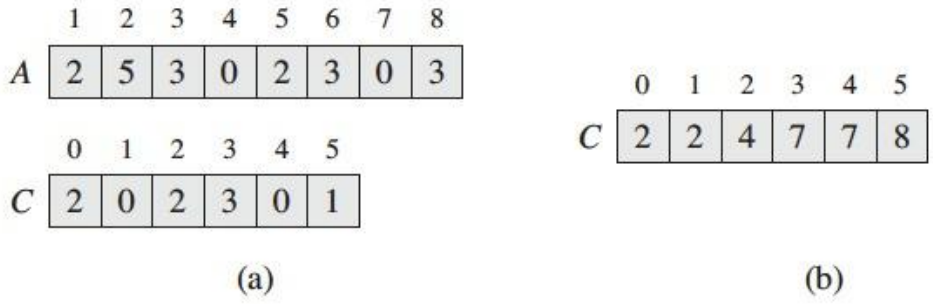
Counting sort

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

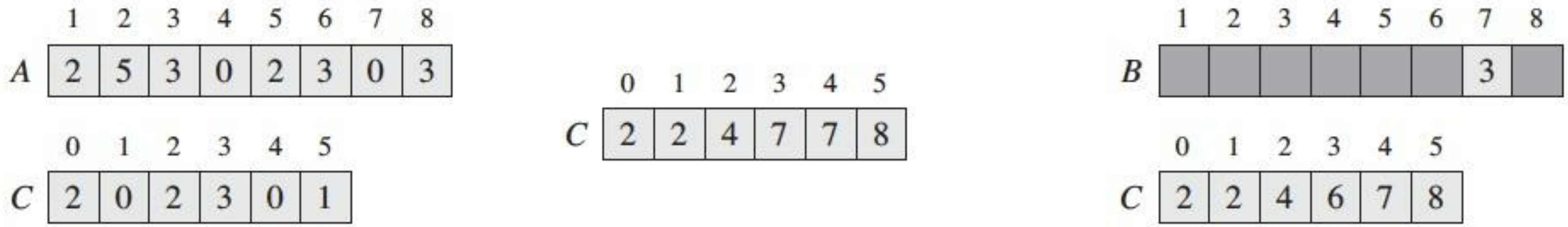
	0	1	2	3	4	5
C	2	0	2	3	0	1

(a)

Counting sort

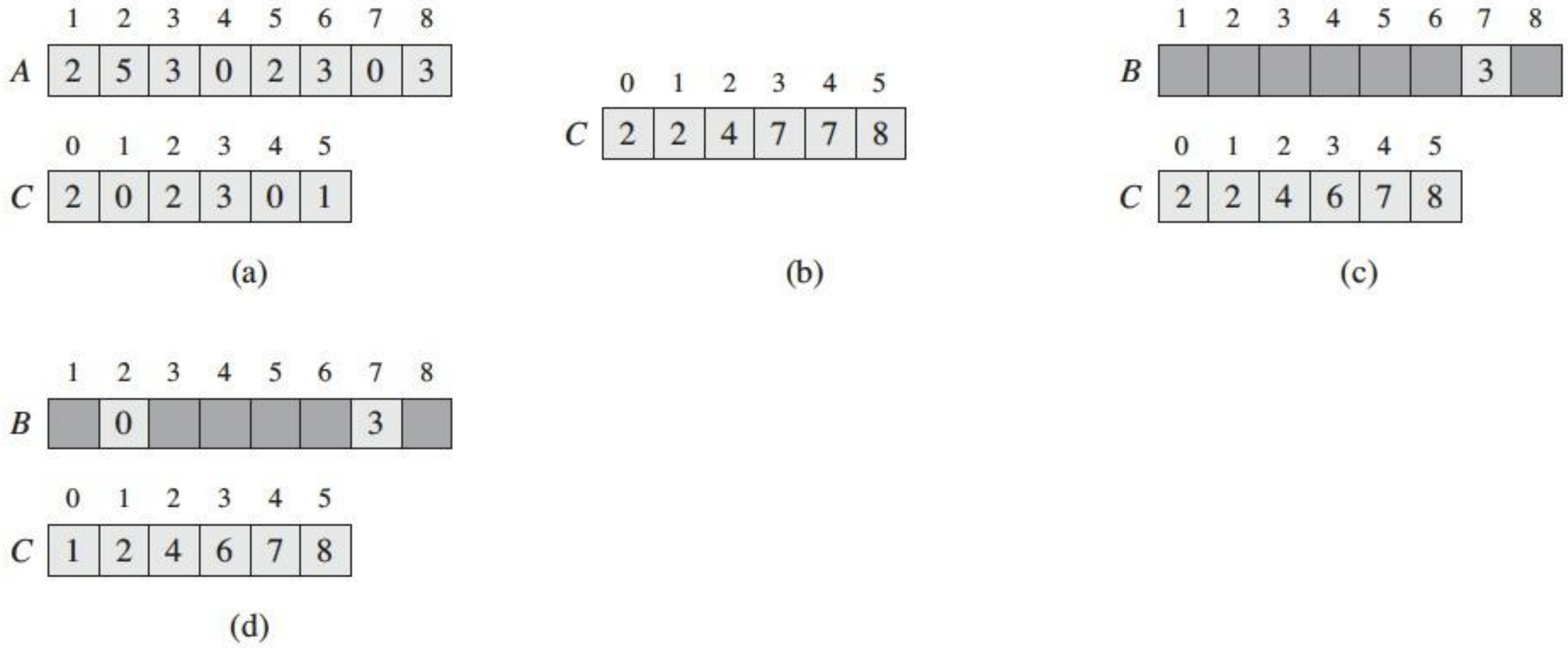


Counting sort

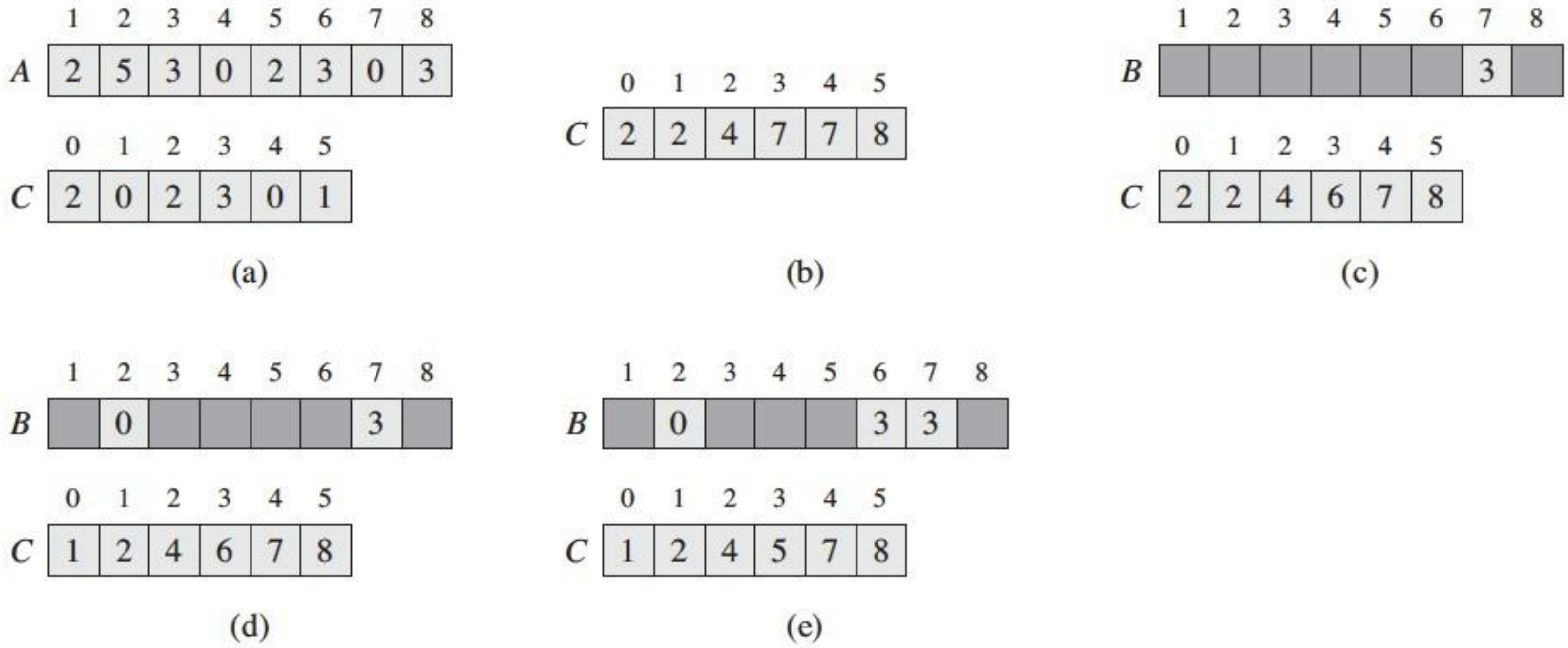


(a) (b) (c)

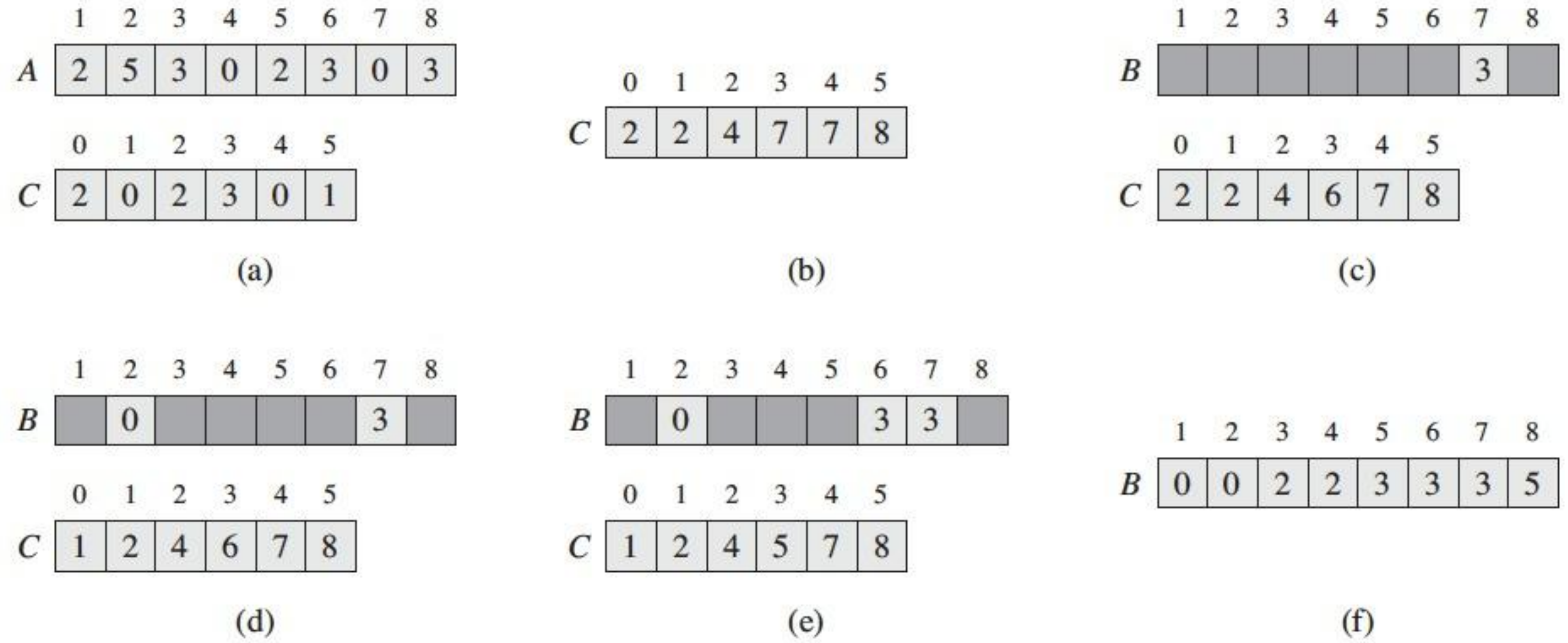
Counting sort



Counting sort



Counting sort



Counting sort

```
COUNTING-SORT(A, B, k)
1   let C[0..k] be a new array
2   for i = 0 to k
3       C[i] = 0
4   for j = 1 to A.length
5       C[A[j]] = C[A[j]] + 1
6   // C[i] now contains the number of elements equal to i.
7   for i = 1 to k
8       C[i] = C[i] + C[i - 1]
9   // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] - 1
```

Line 2 - 3 → $\Theta(k)$

Line 4 - 5 → $\Theta(n)$

Line 7 - 8 → $\Theta(k)$

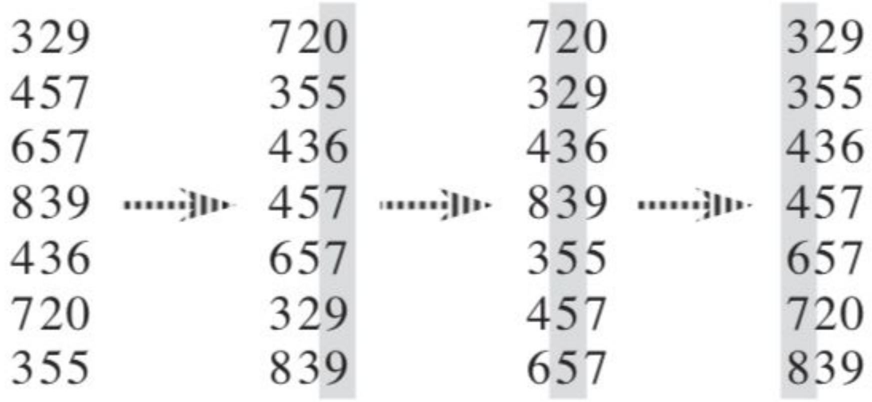
Line 10 - 12 → $\Theta(n)$

Counting sort

- Is **not** a comparison sort
- Beats the lower bound of $\Omega(n \lg n)$
- Is **stable**
- Is often used as a subroutine in radix sort

Radix sort

- The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit.
- In order for radix sort to work correctly, the digit sorts must be stable.
- Radix sort uses counting sort as a subroutine to sort.



Radix sort

```
RADIX-SORT(A, d)
for i = 1 to d
    use a stable sort to sort array A on digit i
```

Each element in the n-element array A has d digits, where digit 1 is the lowest-order digit and digit d is the highest-order digit.

Radix sort

```
RADIX-SORT(A, d)
for i = 1 to d
    use a stable sort to sort array A on digit i
```

When each digit is in the range 0 to k - 1 (so that it can take on k possible values), and k is not too large, counting sort is the obvious choice.

Each step for a digit takes $\Theta(n+k)$.

For d digits $\Theta(dn+dk) \Rightarrow$ The total time for radix sort is $\Theta(d(n+k))$

When d is constant and $k = O(n)$, we can make radix sort run in linear time.

Bucket sort

- Assumes that the input is drawn from a uniform distribution over the interval [0, 1)
- Divides the interval [0, 1) into n equal-sized subintervals, or **buckets**, and then distributes the n input numbers into the buckets
- To produce the output, we sort numbers in each bucket, then go through buckets in order

Bucket sort

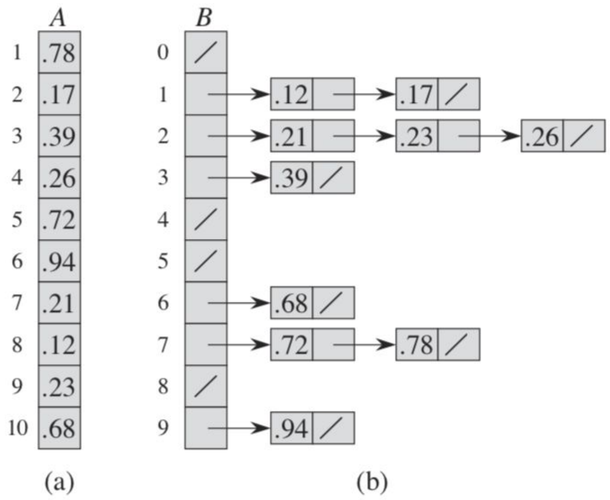


Figure 8.4 The operation of BUCKET-SORT for $n = 10$. **(a)** The input array $A[1..10]$. **(b)** The array $B[0..9]$ of sorted lists (buckets) after line 8 of the algorithm. Bucket i holds values in the half-open interval $[i/10, (i+1)/10)$. The sorted output consists of a concatenation in order of the lists $B[0], B[1], \ldots, B[9]$.

Bucket sort

BUCKET-SORT(A)

let $B[0\,..\,n-1]$ be a new array
$n = A.length$
for $i = 0$ **to** $n - 1$
    make $B[i]$ an empty list
for $i = 1$ **to** n
    insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
for $i = 0$ **to** $n - 1$
    sort list $B[i]$ with insertion sort
concatenate the lists $B[0], B[1], \ldots, B[n-1]$ together in order