Chapter 7
Sorting

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

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

Sort efficiency

- A measure of the relative efficiency of a sort
- Usually an estimate of the number of comparisons and moves required to order an unordered list

Passes

During the sort process, the data are traversed many times. Each traversal of the data is referred to as a **sort pass**

Sorting algorithms

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

Selection sort

Among the most intuitive of all sorting algorithms

Basic idea

- Given a list of data to be sorted,
  - Select the smallest item and place it in a sorted list
  - Repeat until all of the data are sorted

Straight selection sort

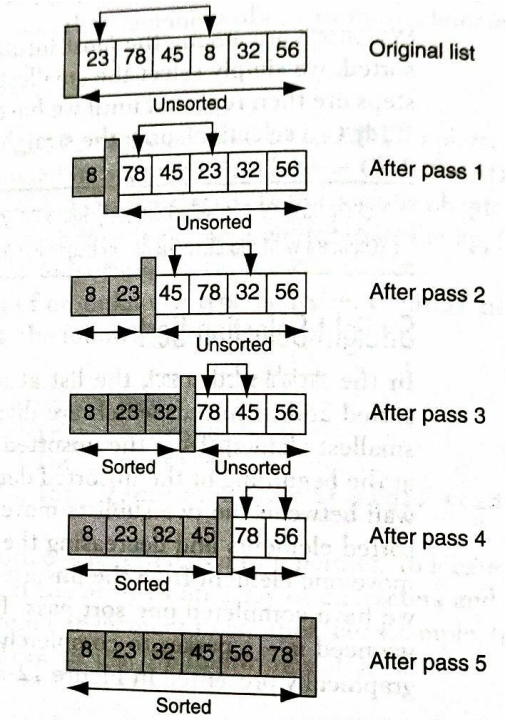
The list at any moment is divided into two sublists, sorted and unsorted, which are divided by an imaginary wall

Steps:

1. Select the smallest element from the unsorted sublist and exchange it with the element at the beginning of the unsorted data
2. Repeat Step 1 until there is no element in the unsorted sublist

Each time we move one element from the unsorted sublist to the sorted sublist, we say that we have completed one sort pass. Therefore, a list of n elements need n-1 passes to completely rearrange the data

Straight selection sort



Straight selection sort

Algorithm: selectionSort (a)
Input: An unordered array a

Steps:

```
1.  n = length of a
2.  for (i = 0; i < n - 1; i++)
3.      min_index = i
4.      for (j = i + 1; j < n; j++)        # Find minimum
5.          if ( a[ j ] < a[ min_index] )
6.              min_index = j
7.          endif
8.      end for
9.      Swap a[min_index] and a[ i ]
10. end for
```

Straight selection sort

Algorithm: selectionSort (a)
Input: An unordered array a

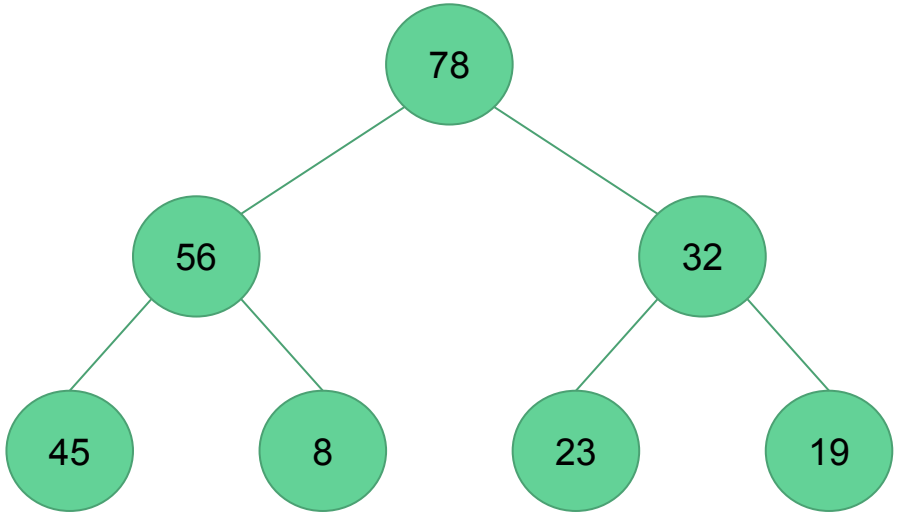
Steps:

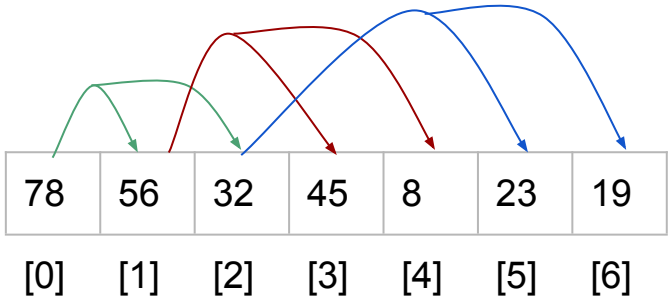
```
1.  n = length of a
2.  for (i = 0; i < n - 1; i++)
3.      min_index = i
4.      for (j = i + 1; j < n; j++)        # Find minimum
5.          if ( a[ j ] < a[ min_index] )
6.              min_index = j
7.          endif
8.      end for
9.      Swap a[min_index] and a[ i ]
10. end for
```

Complexity of selection sort = $O(n^2)$

Heap sort

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





Heap in its array form

(First element = Largest element)

Heap sort

Heap sort is among the fastest sorting algorithms.

Used with very large arrays

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

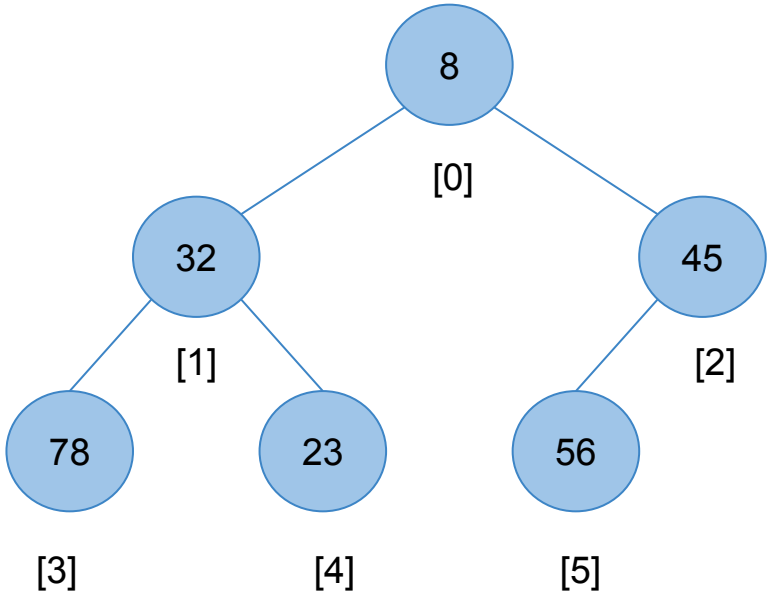
Heap sort

Example:

Sort the following data using heap sort:

8	32	45	78	23	56

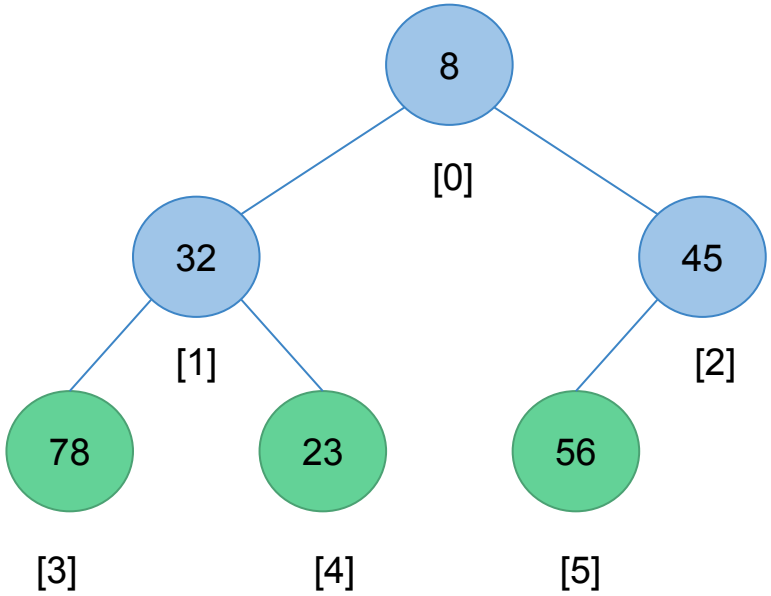
Heap sort



Convert the array into a max heap

8	32	45	78	23	56

Heap sort

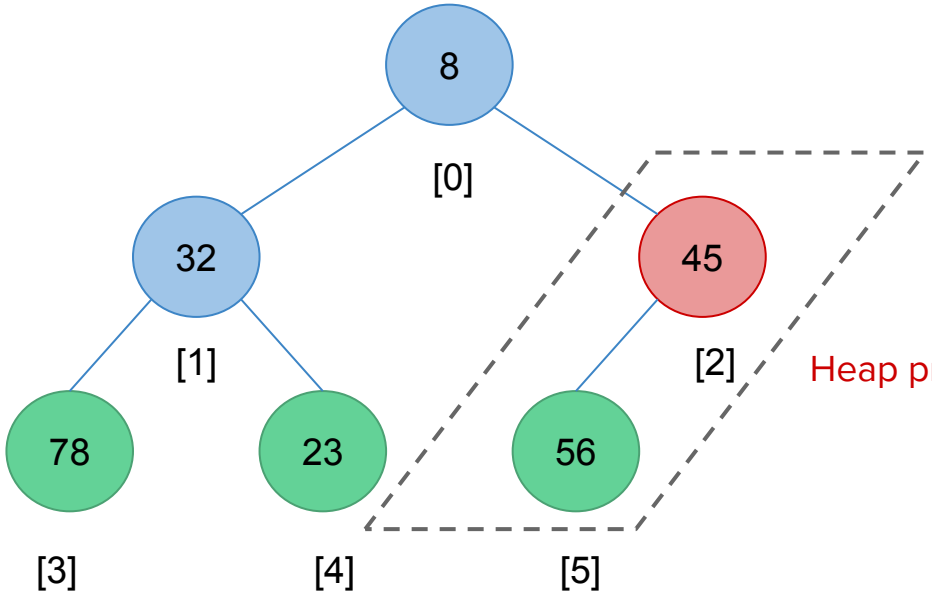


Convert the array into a max heap

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

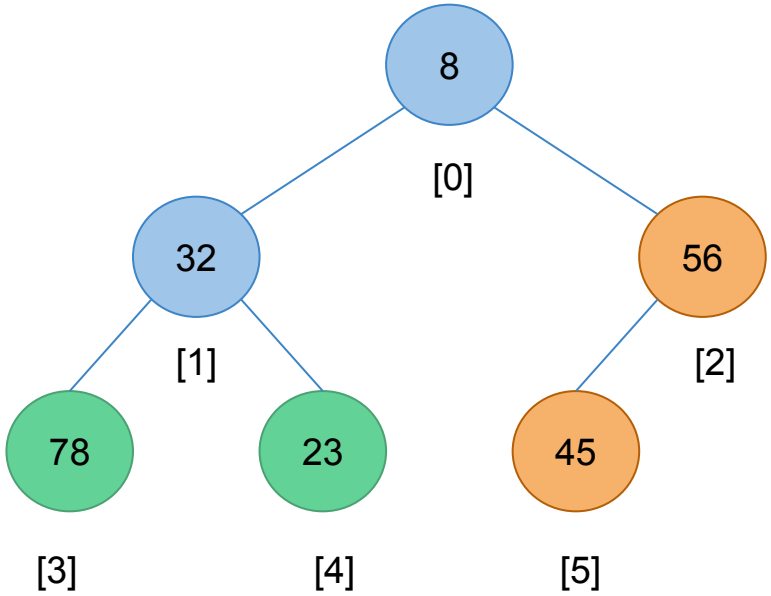


Convert the array into a max heap

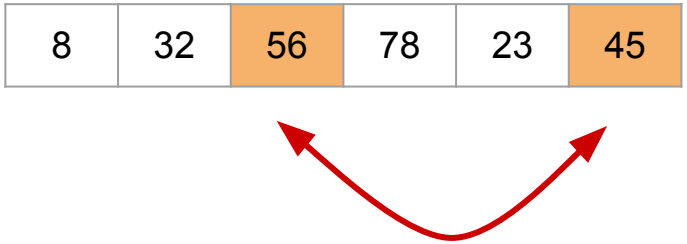
8	32	45	78	23	56

Heap property is not satisfied

Heap sort

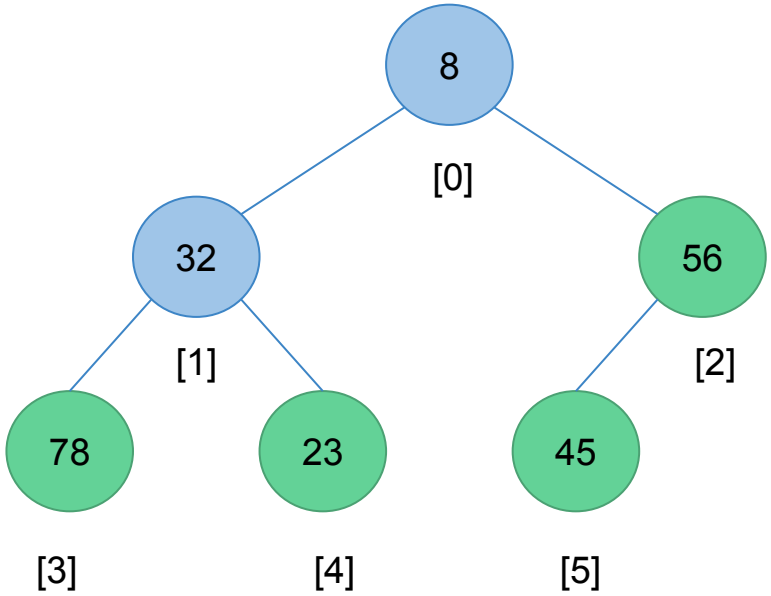


Convert the array into a max heap



Heap property is not satisfied. Therefore, swap them

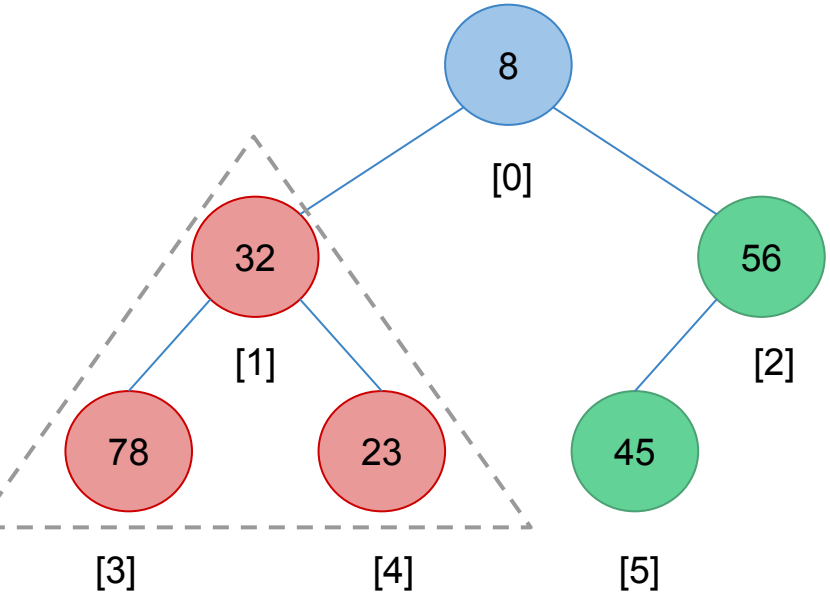
Heap sort



Convert the array into a max heap

8	32	56	78	23	45

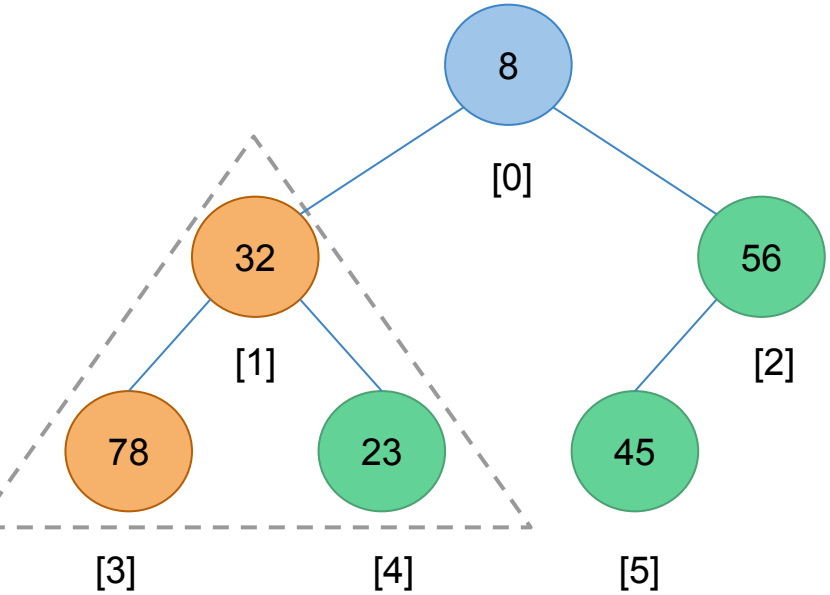
Heap sort



Convert the array into a max heap

8	32	56	78	23	45

Heap sort

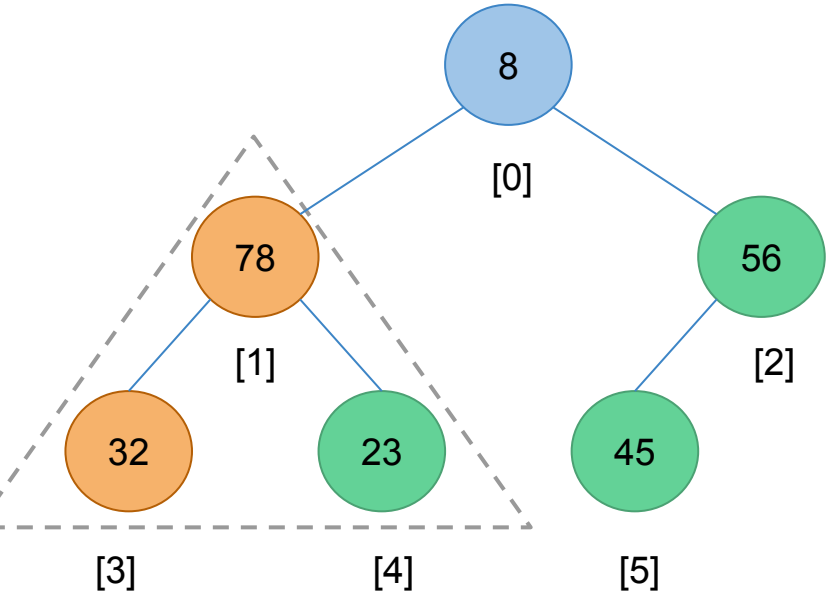


Convert the array into a max heap

8	32	56	78	23	45

Swap the root with the largest child

Heap sort

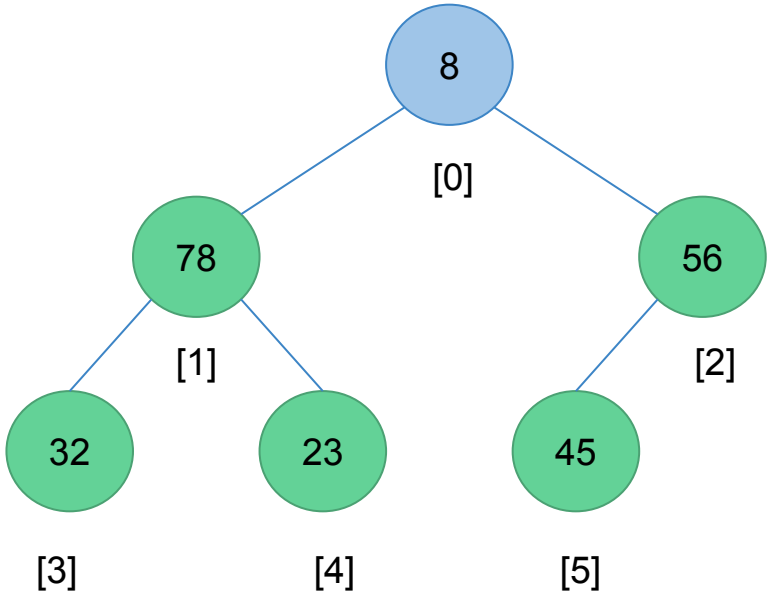


Convert the array into a max heap

8	78	56	32	23	45

Swap the root with the largest child

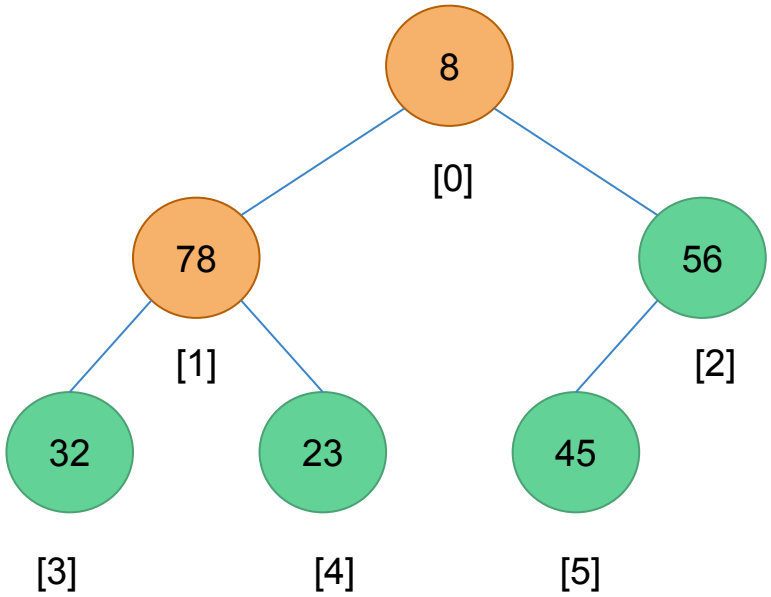
Heap sort



Convert the array into a max heap

8	78	56	32	23	45

Heap sort

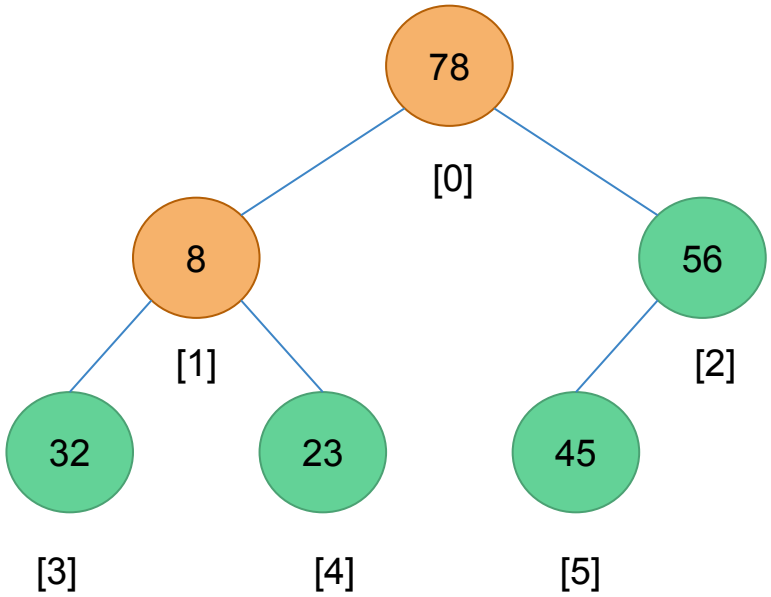


Convert the array into a max heap

8	78	56	32	23	45

Swap the root with the largest child

Heap sort

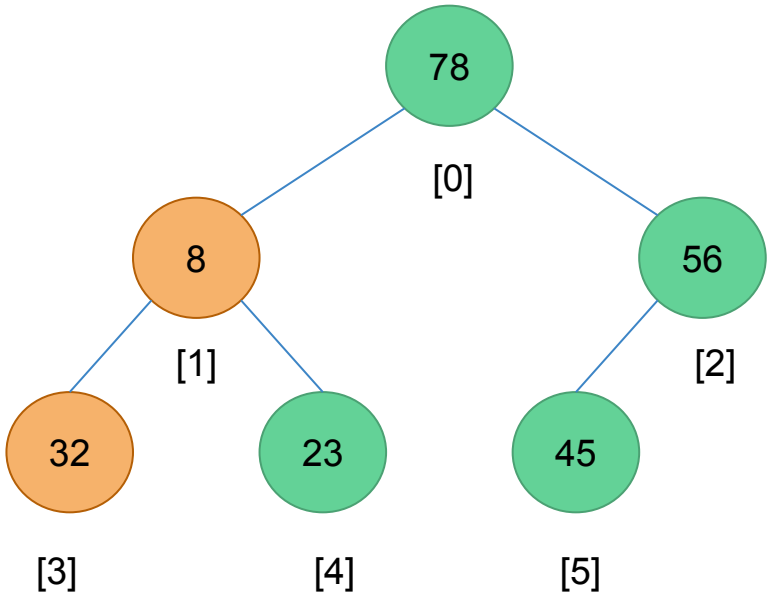


Convert the array into a max heap

78	8	56	32	23	45

Swap the root with the largest child

Heap sort

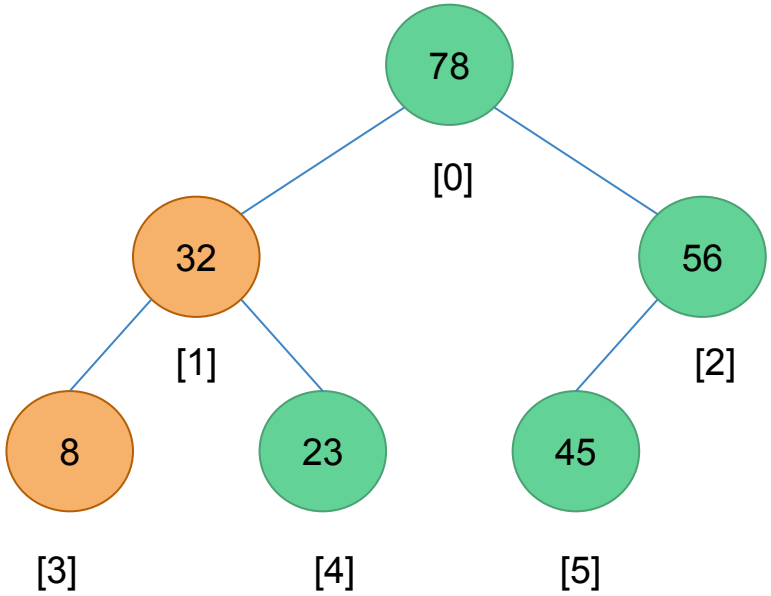


Convert the array into a max heap

78	8	56	32	23	45

Swap the root with the largest child

Heap sort

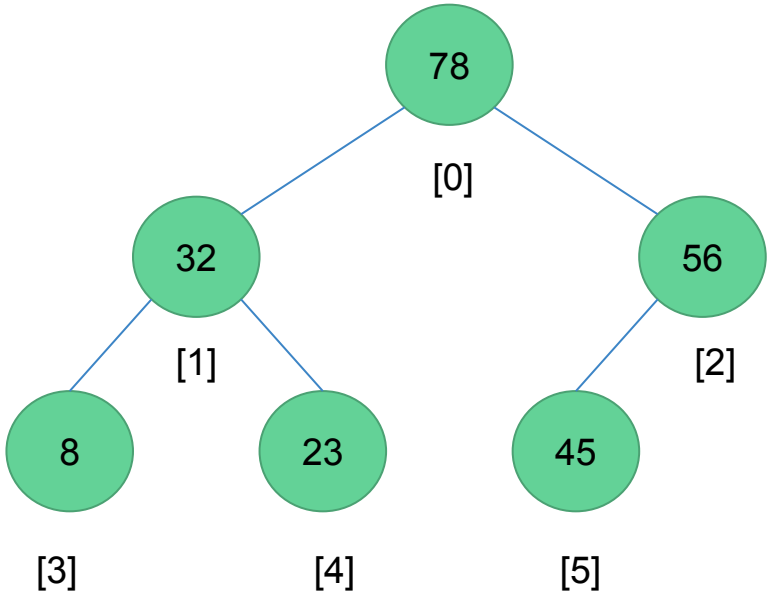


Convert the array into a max heap

78	32	56	8	23	45

Swap the root with the largest child

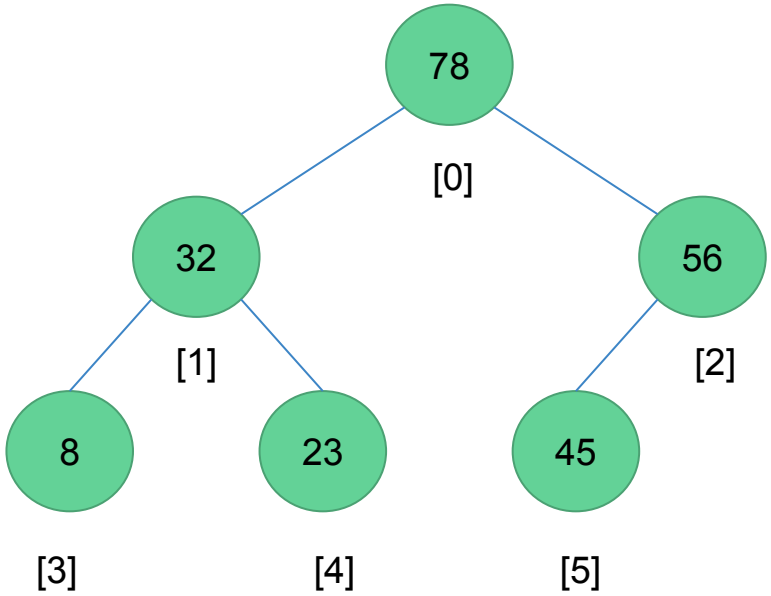
Heap sort



Convert the array into a max heap

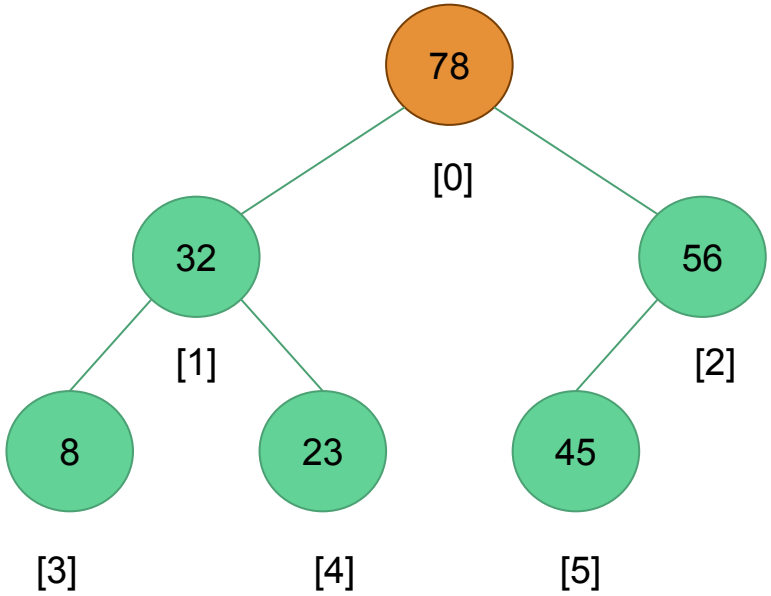
78	32	56	8	23	45

Heap sort

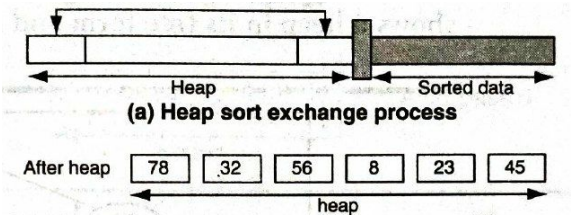


Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.

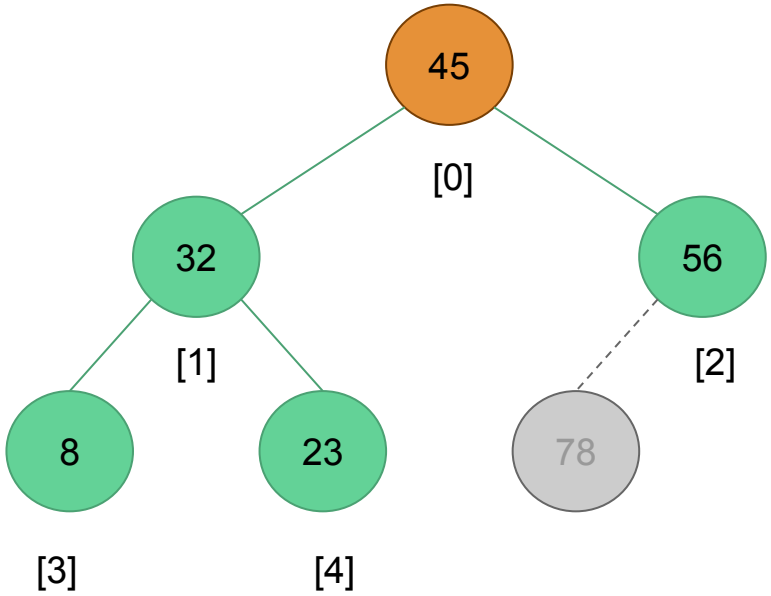
Heap sort



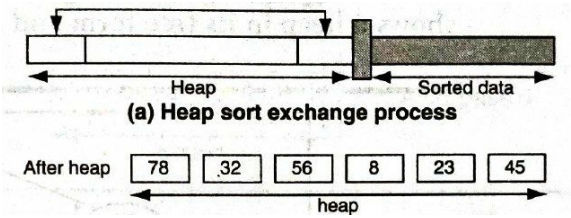
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



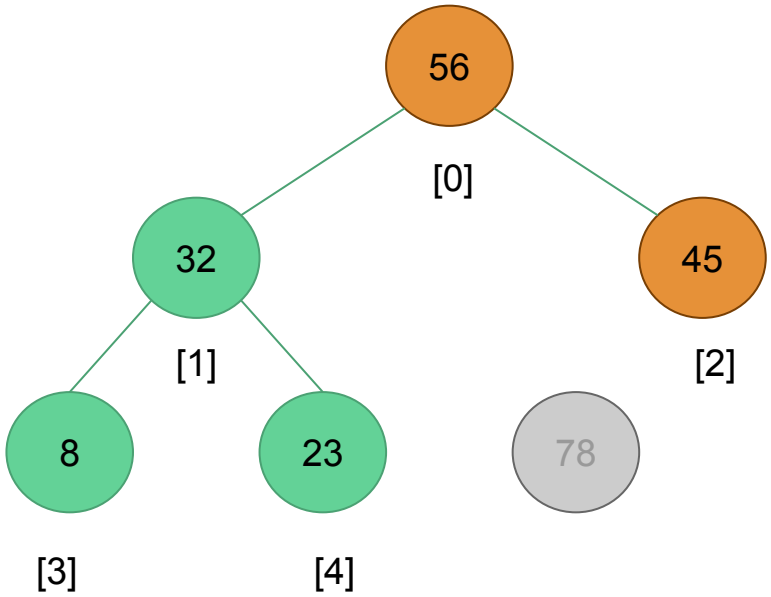
Heap sort



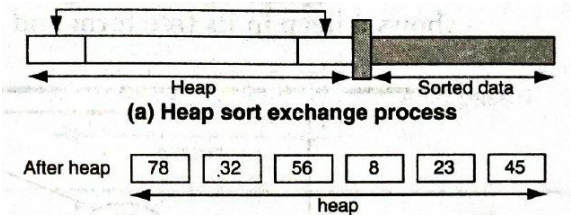
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



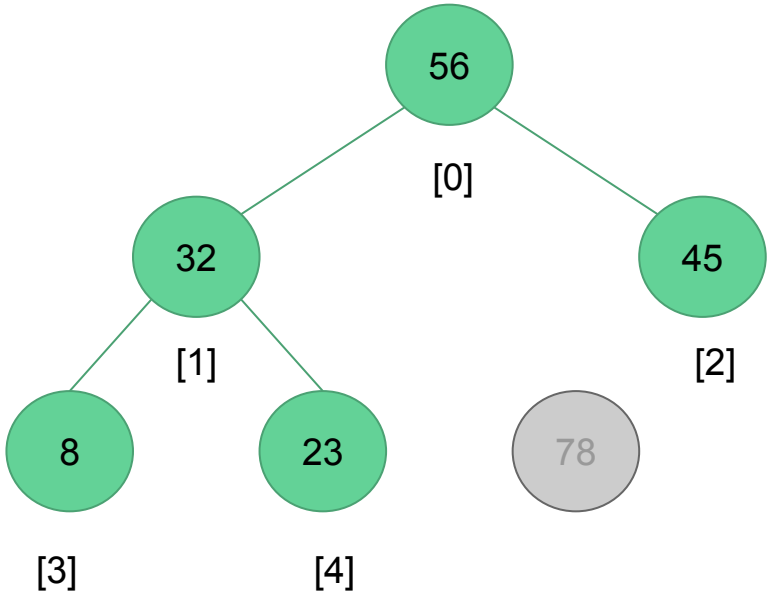
Heap sort



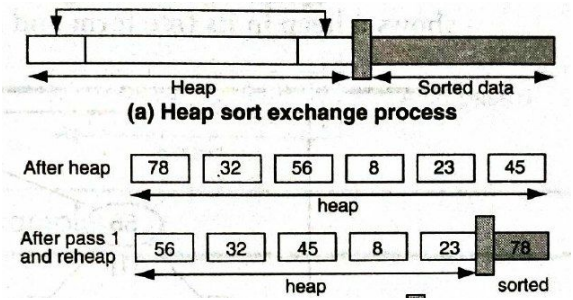
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



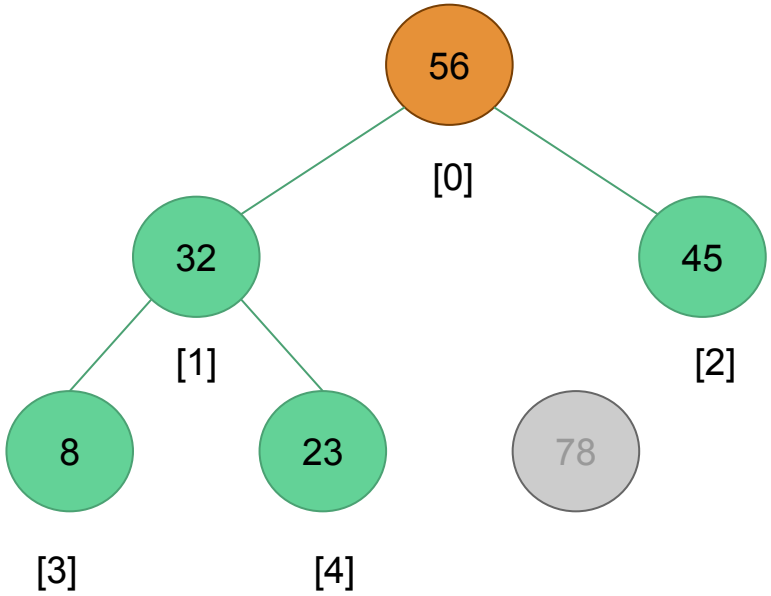
Heap sort



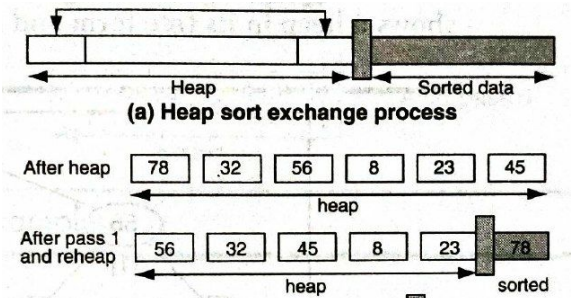
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



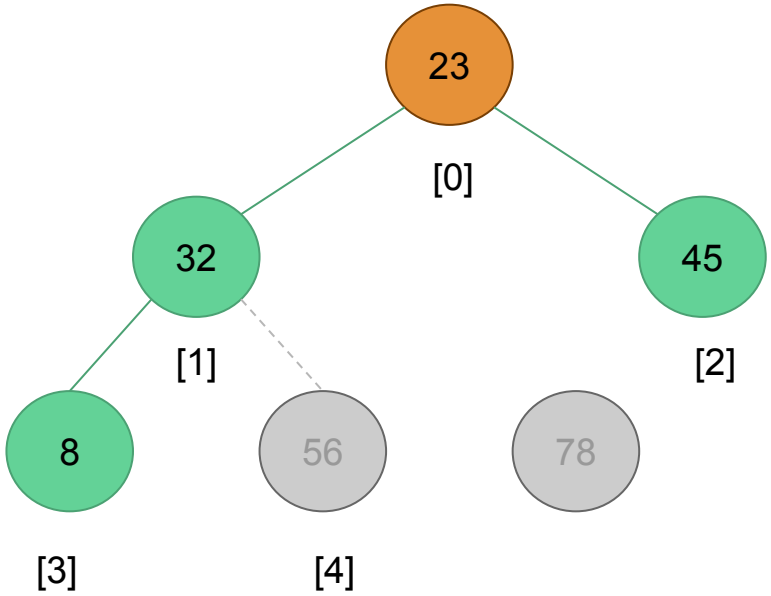
Heap sort



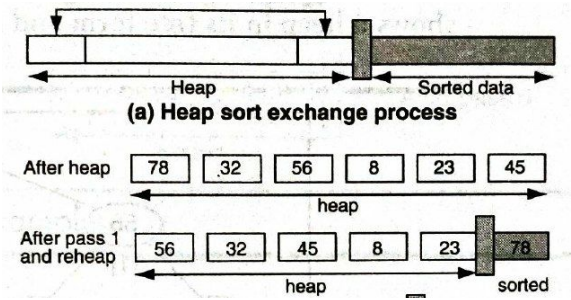
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



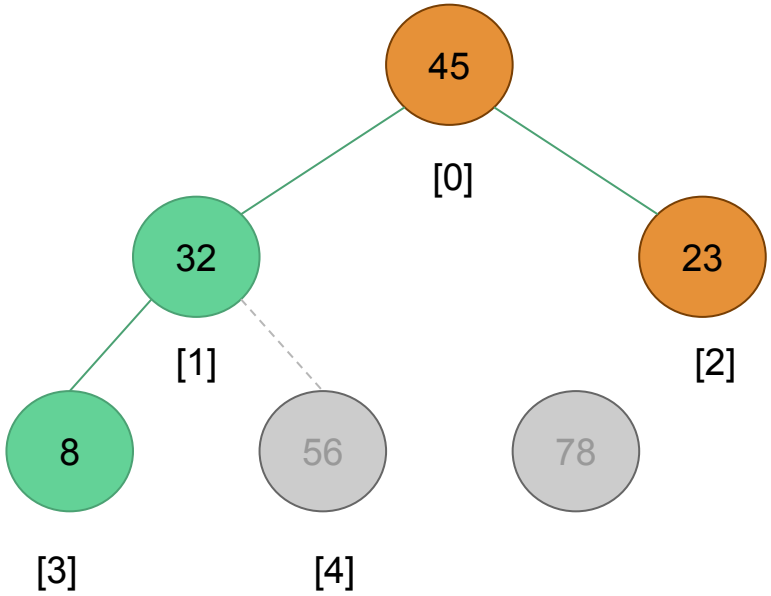
Heap sort



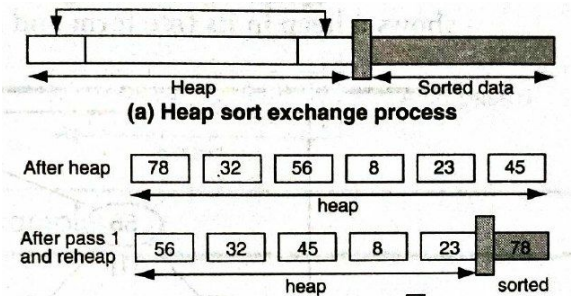
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



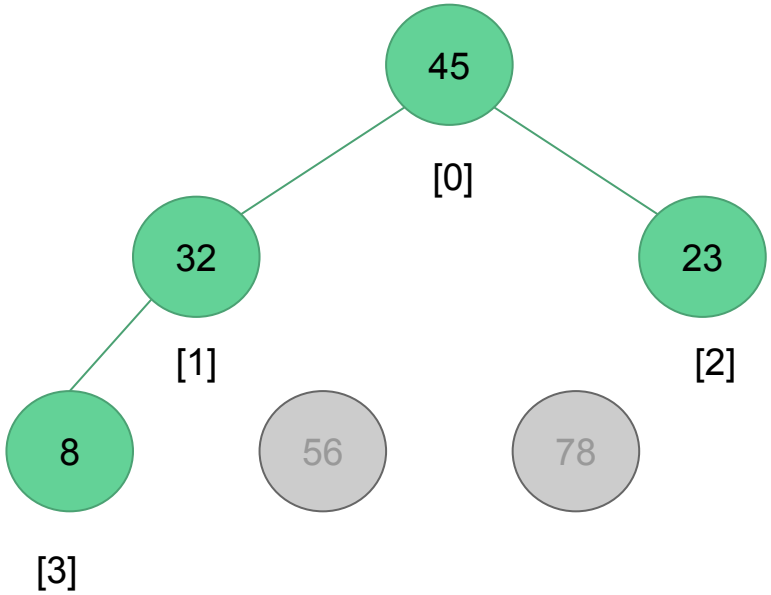
Heap sort



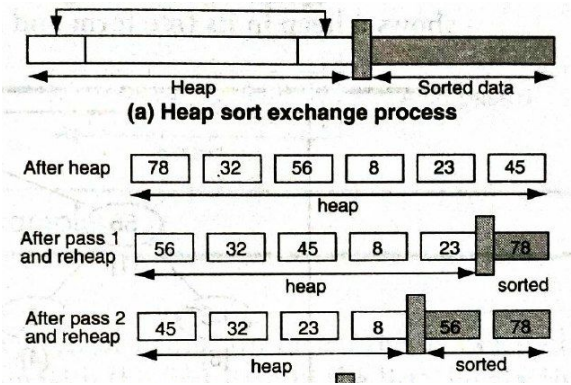
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



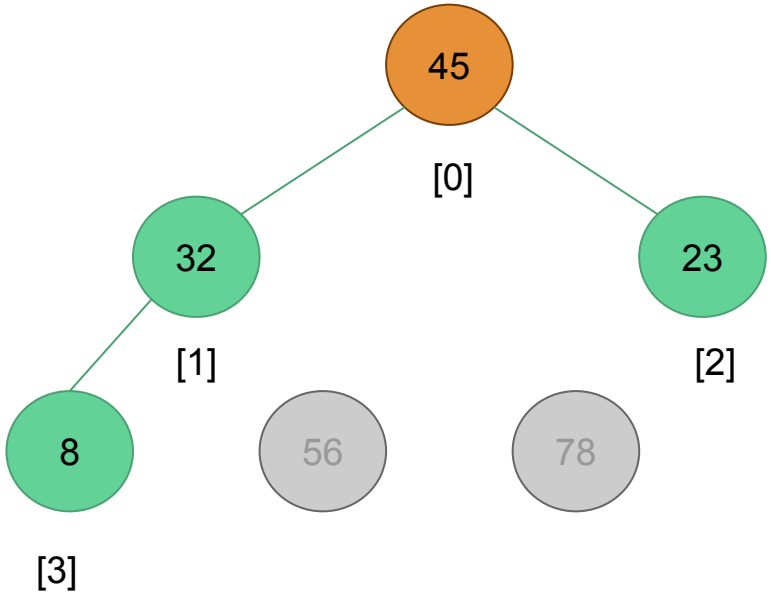
Heap sort



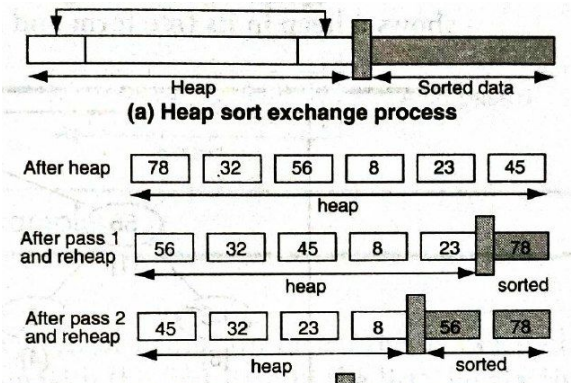
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



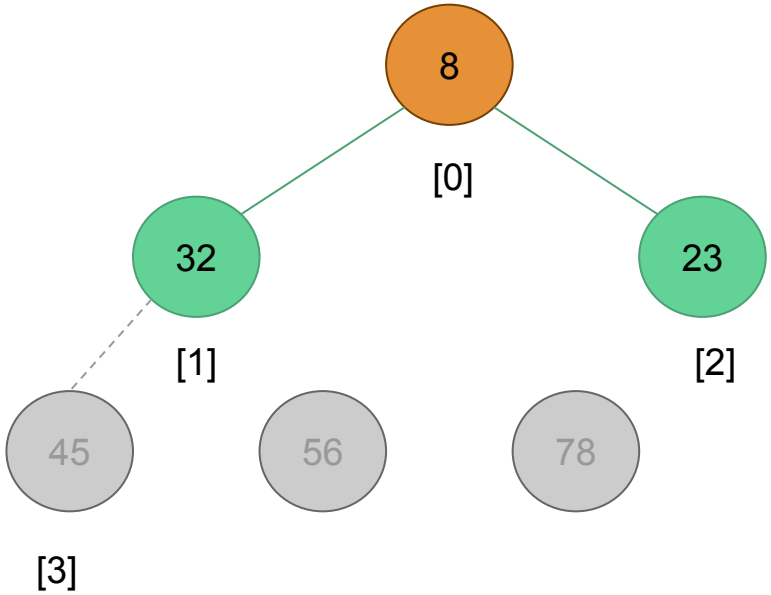
Heap sort



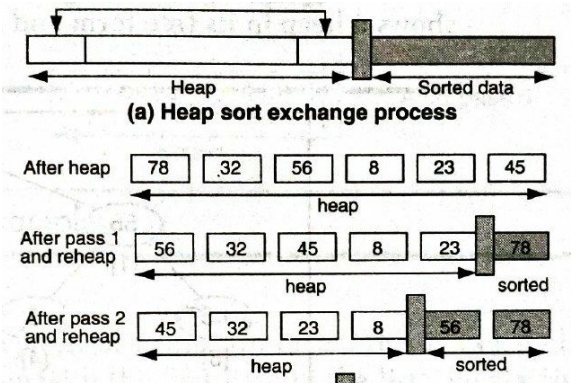
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



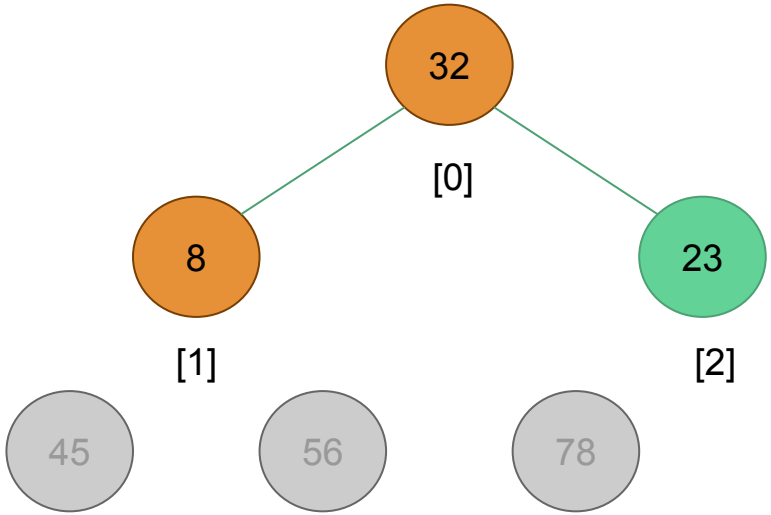
Heap sort



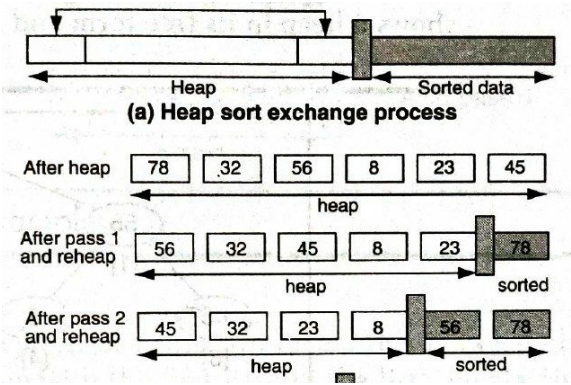
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



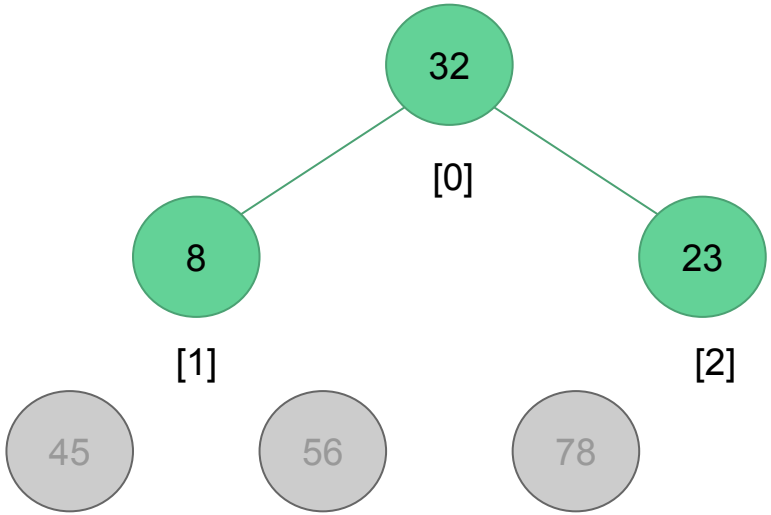
Heap sort



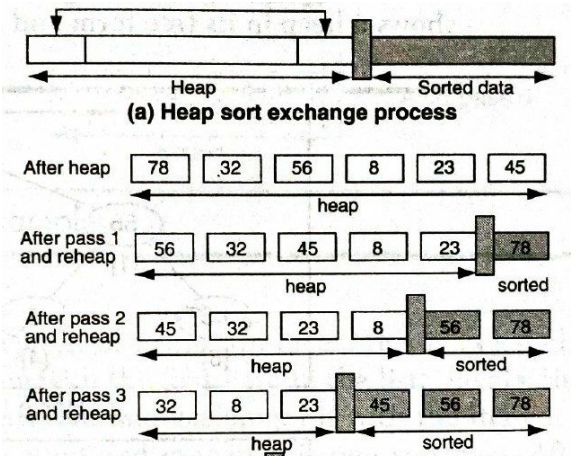
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



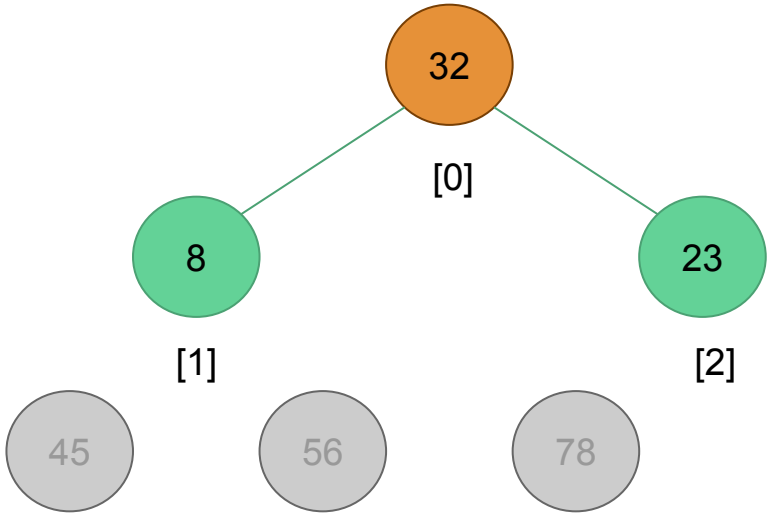
Heap sort



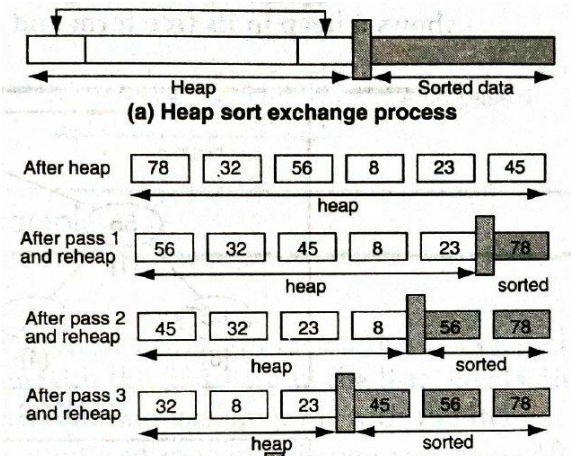
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



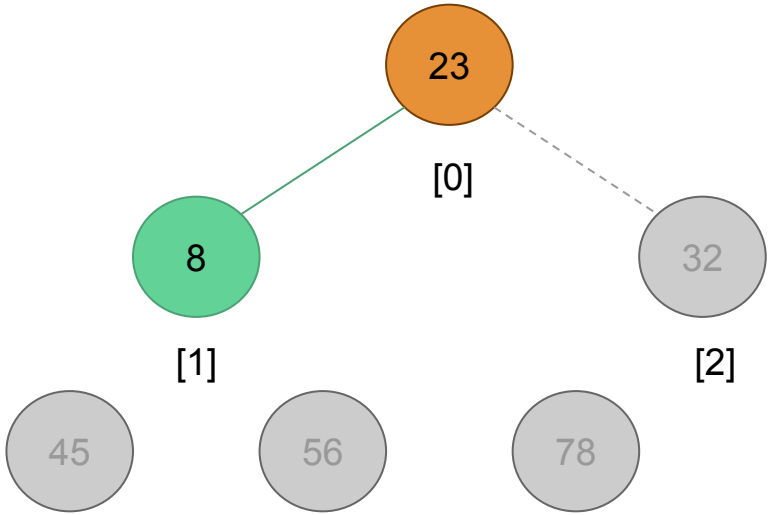
Heap sort



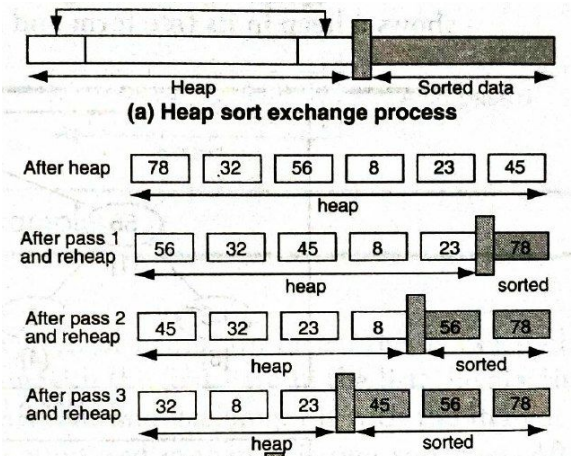
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



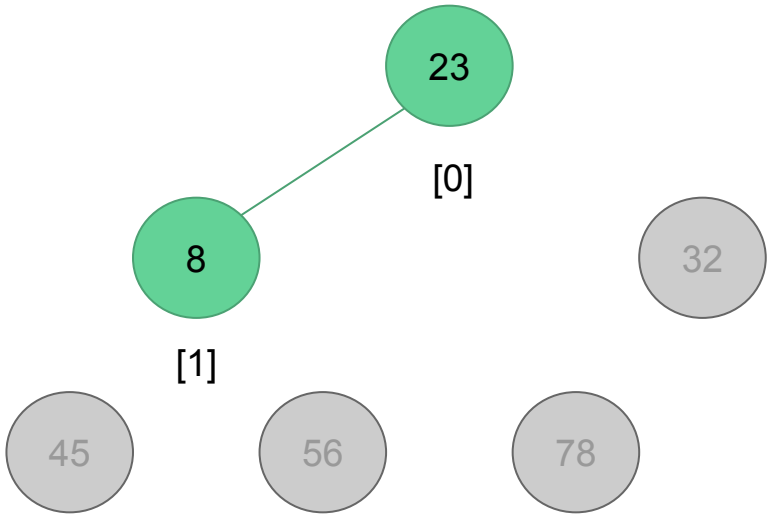
Heap sort



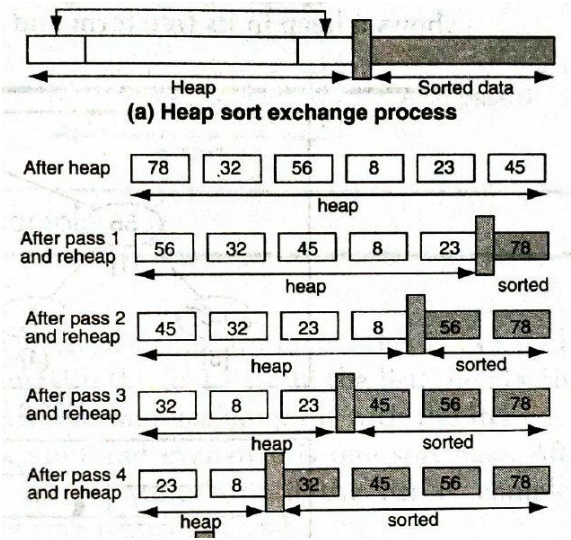
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



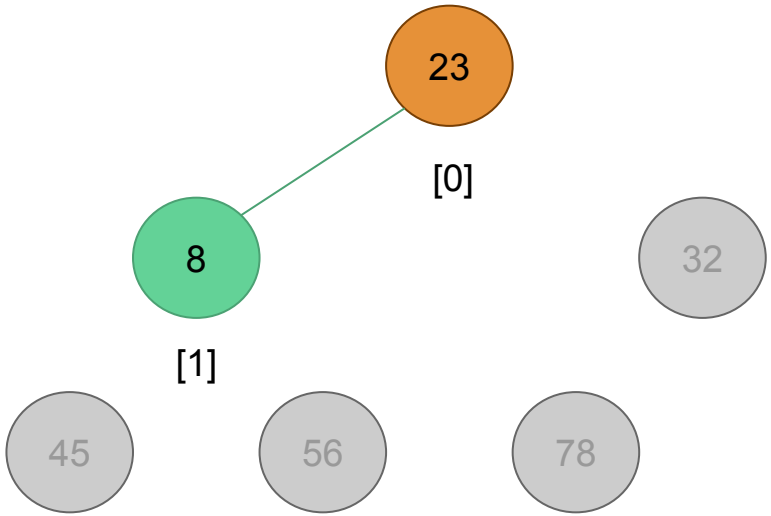
Heap sort



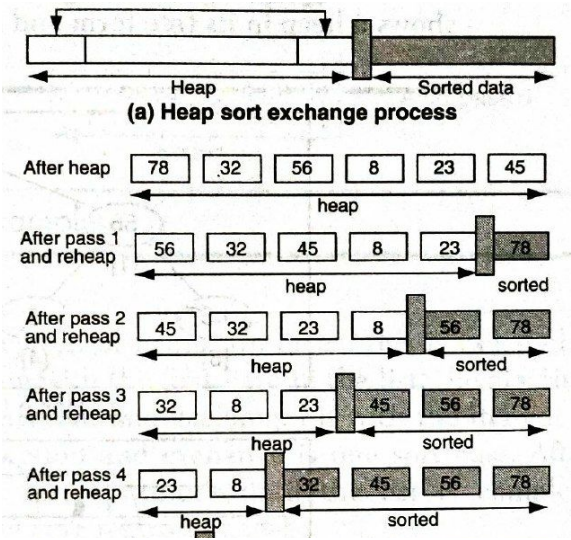
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



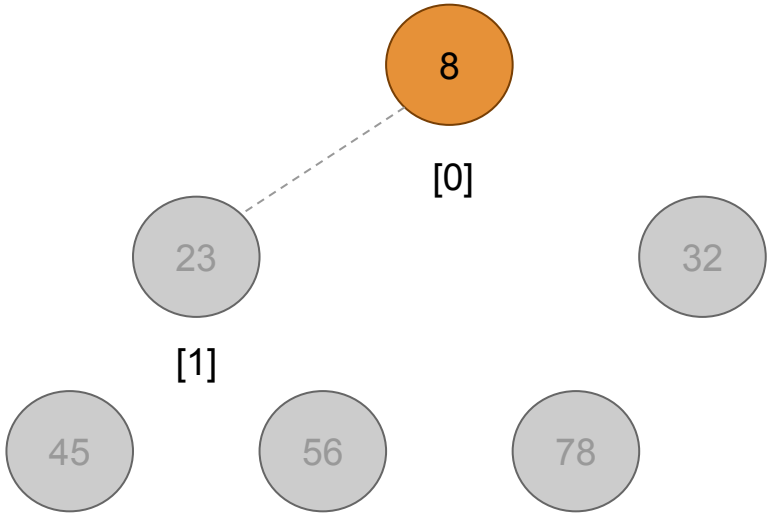
Heap sort



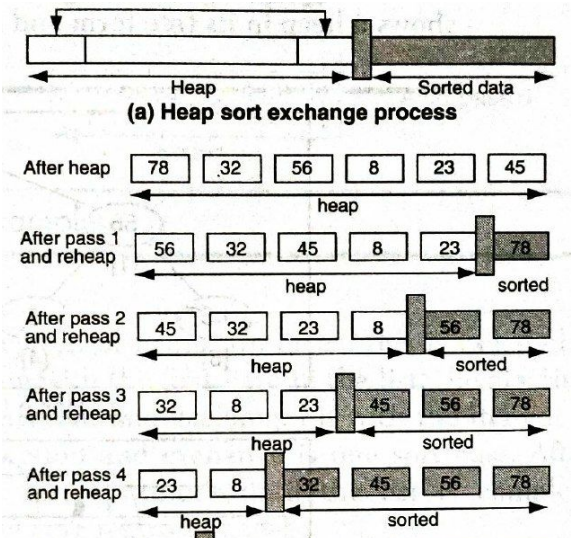
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



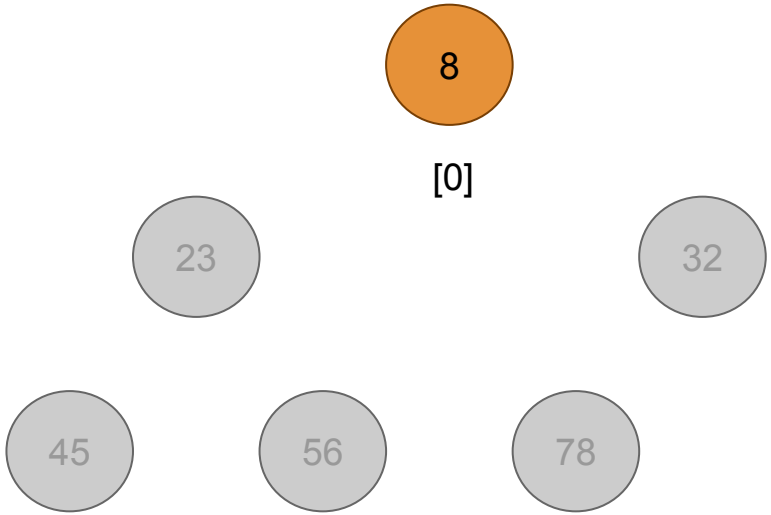
Heap sort



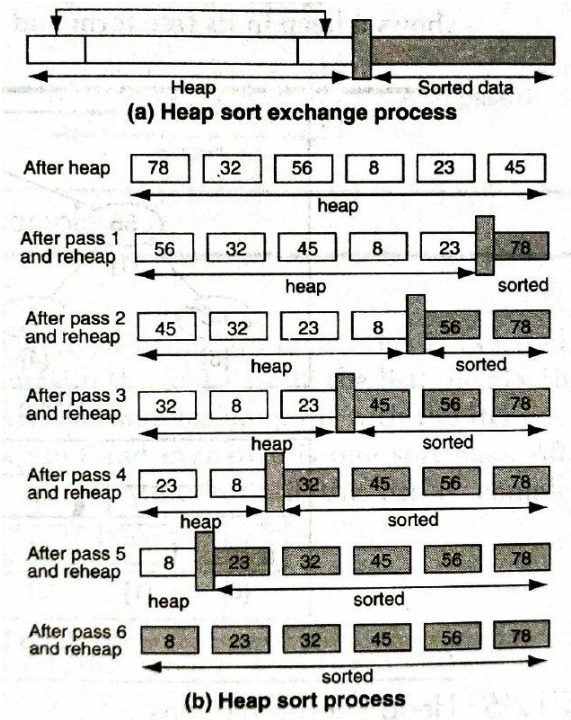
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



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

Algorithm: heapSort (a)
Input: An unordered array a

Steps:

1. n = length of a
2. build_heap(a) # Complexity: ? ; Nb execution: ?
3. **for** (i = n - 1; i > 0; i--)
4. Swap arr[i] and arr[0]
5. heapify(a, i, 0) # Complexity: ? ; Nb execution: ?
6. **end for**

Heap sort

Algorithm: Heapify(a, n, i)
Input: An array a of size n, and the root index i

Steps:

1. n = length of a
2. largest = i
3. l = 2 * i + 1
4. r = 2 * i + 2
5. if l < n and a[i] < a[l]
6. largest = l
7. endif
8. if r < n and a[largest] < a[r]:
9. largest = r
10. endif
11. if largest != i:
12. swap a[i] and a[largest]
13. heapify(a, n, largest)
14. endif

Heap sort

Algorithm: build_heap(a)
Input: An array a

Steps:

1. n = size of a
2. for i from n - 1 to 0
3. heapify(a, n, i)

Heap sort

Algorithm: heapSort (a)
Input: An unordered array a

Steps:

1. n = length of a
2. build_heap(a) # Complexity: O(n) ; Nb execution: 1
3. **for** (i = n - 1; i > 0; i--)
4. Swap arr[i] and arr[0]
5. heapify(a, i, 0) # Complexity: O(log n) ; Nb execution: n - 1
6. **end for**

Complexity of heap sort = O(n + n log n) = O(n log n)

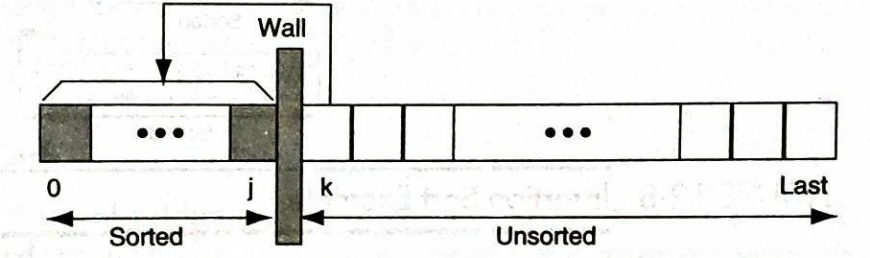
Insertion Sort

One of the most common sorting techniques used by card players. As they pick up each card, they insert it into the proper sequence in their hand.

Main idea:

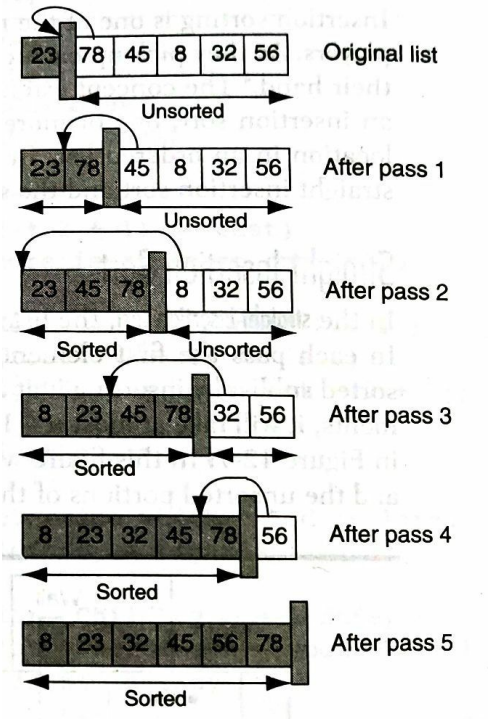
The list is divided into two parts: sorted and unsorted

In each pass of an insertion sort, the first element of the unsorted sublist is inserted into its correct location in the sorted sublist



Insertion sort

Example



Insertion sort

Algorithm: InsertionSort(a)
Input: An unsorted list, a
Output: The list a after sorting

```
1.  n ← length[a]
2.  for j ← 1 to n - 1
3.      key ← a[j]
4.      # Put a[j] into the sorted sequence a[0 . . j − 1]
5.      i ← j − 1
6.      while i ≥ 0 and a[i] > key
7.          a[i +1] ← a[i]
8.          i ← i − 1
9.      end while
10.     a[i + 1] ← key
11. end for
```

Insertion sort

Algorithm: InsertionSort(a)
Input: An unsorted list, a
Output: The list a after sorting

```
1.  n ← length[a]
2.  for j ← 1 to n - 1
3.      key ← a[j]
4.      # Put a[j] into the sorted sequence a[0 . . j − 1]
5.      i ← j − 1
6.      while i ≥ 0 and a[i] > key
7.          a[i +1] ← a[i]
8.          i ← i − 1
9.      end while
10.     a[i + 1] ← key
11. end for
```

Worst case time complexity: $O(n^2)$
Best case time complexity: $O(n)$

Merge sort

Merge sort uses **divide-and-conquer** strategy

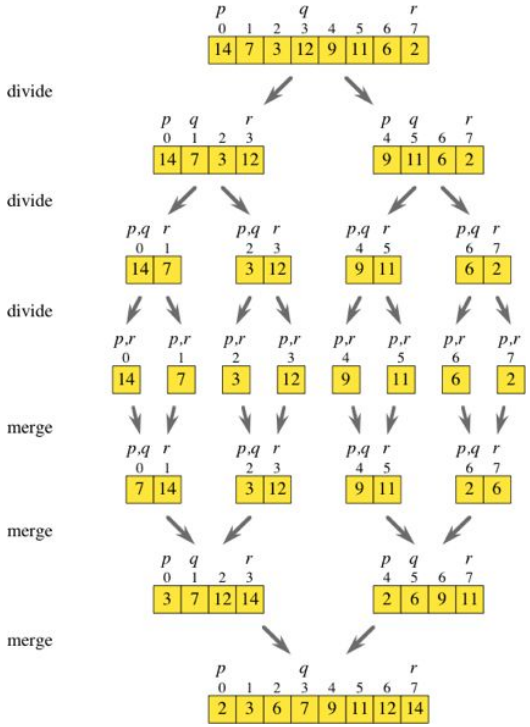
- The original problem is partitioned into simpler sub-problems, each sub problem is considered independently.
- Subdivision continues until sub problems obtained are simple.

Steps:

1. **Divide**: partition the list into two roughly equal parts, S1 and S2, called the left and the right sublists
2. **Conquer**: recursively sort S1 and S2
3. **Combine**: merge the sorted sublists.

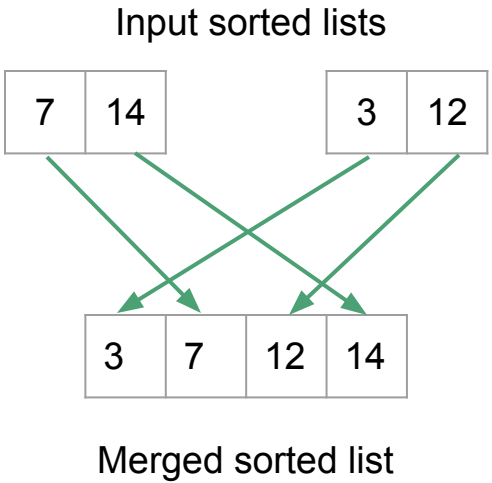
Merge Sort

Example



Merge sort

Merging two sorted lists into one



Algorithm: merge(L, R)

Input: Two sorted lists, L and R
Output: Merged sorted list, a

Steps:

1. i = j = k = 0
2. n1 = length of L
3. n2 = length of R
4. **while** (i < n1 and j < n2)
5. **if** (L[i] ≤ R[j])
6. a[k] = L[i]
7. i++
8. **else**
9. a[k] = R[j]
10. j++
11. **end if**
12. k++
13. **end while**
14. Copy the remaining elements of L to a, if there are any
15. Copy the remaining elements of R to a, if there are any

merge(L, R)

Example:

Input sorted lists

L = | 7 | 14 |

R = | 3 | 12 |

k	i	j
0	0	0

Comparison	
L[0] ≤ R[0]	False

a

3			

merge(L, R)

Example:

Input sorted lists

L = | 7 | 14 |

R = | 3 | 12 |

k	i	j
0	0	0
1	0	**1**

Comparison	
L[0] ≤ R[0]	False
L[0] ≤ R[1]	True

a			
3			
3	7		

merge(L, R)

Example:

Input sorted lists

L = | 7 | 14 |

R = | 3 | 12 |

k	i	j
0	0	0
1	0	1
2	**1**	1

Comparison	
L[0] ≤ R[0]	False
L[0] ≤ R[1]	True
L[1] ≤ R[1]	False

a

3			
3	7		
3	7	12	

merge(L, R)

Example:

Input sorted lists

L = | 7 | 14 |

R = | 3 | 12 |

k	i	j
0	0	0
1	0	1
2	1	1
3	**2**	1

Comparison	
L[0] ≤ R[0]	False
L[0] ≤ R[1]	True
L[1] ≤ R[1]	False
- (Copy the remaining data)	

a			
3			
3	7		
3	7	12	
3	7	12	14

Algorithm: mergeSort(a)

Complexity of merge sort = O(n log n)

Input: An unsorted list, a

Output: The sorted list, a

Steps:

1. n = length of a
2. if n == 1
3. return a
4. end if
5. # Divide Complexity = ?
6. mid = ⌊n/2⌋
7. Copy a[0] to a[mid] to L
8. Copy a[mid+1] to a[n-1] to R
9. # Recur Complexity = ?
10. L = mergeSort(L)
11. R = mergeSort(R)
12. #Conquer Complexity = ?
13. merge(L, R)

Algorithm: mergeSort(a)

Complexity of merge sort = O(n log n)

Input: An unsorted list, a

Output: The sorted list, a

Steps:

1. n = length of a
2. if n == 1
3. return a
4. end if
5. # Divide Complexity = O(1)
6. mid = ⌊n/2⌋
7. Copy a[0] to a[mid] to L
8. Copy a[mid+1] to a[n-1] to R
9. # Recur Complexity = O(n.logn)
10. L = mergeSort(L)
11. R = mergeSort(R)
12. #Conquer Complexity = O(n)
13. merge(L, R)

Quick Sort

Like merge sort, quick sort also uses divide-and-conquer paradigm

Steps:

1. **Divide**: Select any element from the list. Call it the **pivot**. Then partition the list into two non-empty sublists such that all the elements in the left sublist are less than or equal to the pivot and those of the right sublist are greater than or equal to the pivot
2. **Conquer**: Recursively sort the two sublists
3. **Combine**: Since the subarrays are sorted in place, no work is needed to combine them: the entire list is now sorted.

Quick sort

A pivot can be picked in different ways:

- Always pick first element as pivot
- Always pick last element as pivot
- Pick a random element as pivot
- Pick median as pivot

Quick sort

Algorithm: QuickSort(a, low, high)
Input: An unsorted list a[low ... high]
Output: The list a after sorting

Steps:

1. If low < high then # If there are more than 1 element
2. pivot ≔ Partition (a, low, high)
3. Quick Sort (a, low, pivot - 1)
4. Quick Sort (a, pivot + 1, high)
5. end if

Quick sort

Algorithm: Partition(a, low, high)
Input: An unsorted list, a, whose elements from index low to index high are to be partitioned
Output: The index of the pivot that partitions a into two sublists

Steps:

1. pi ← a[low] # Pivot = First element
2. i ← low
3. j ← high + 1
4. **while** i < j do
5. Repeat j ← j-1 until a[j] < pi
6. Repeat i ← i+1 until a[i] > pi
7. **if** i < j
8. exchange a[i] ↔ a[j]
9. **end if**
10. **end while**
11. exchange a[low] ↔ a[j]
12. return j # j = position of pivot

Quick sort example

Input:

26	5	37	1	61	11	59	15	48	19

Pivot = The first element in the list 26

Partition: Starting with i = 0 , j = 10, find the index i of the element larger than pivot and the index j of the element smaller than the pivot. Then exchange them if i < j

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
26	5	37	1	61	11	59	15	48	19

i	j
2	9

Exchange a[2] and a[9]

Quick sort example

Input:

26	5	37	1	61	11	59	15	48	19

Pivot = The first element in the list 26

Partition: Starting with i = 0 , j = 10, find the index i of the element larger than pivot and the index j of the element smaller than the pivot. Then exchange them if i < j

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
26	5	37	1	61	11	59	15	48	19
26	5	19	1	61	11	59	15	48	37

i	j
2	9
4	7

Exchange a[4] and a[7]

Quick sort example

Input:

26	5	37	1	61	11	59	15	48	19

Pivot = The first element in the list 26

Partition: Starting with i = 0 , j = 10, find the index i of the element larger than pivot and the index j of the element smaller than the pivot. Then exchange them if i < j

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
26	5	37	1	61	11	59	15	48	19
26	5	19	1	61	11	59	15	48	37
26	5	19	1	15	11	59	61	48	37

i	j	
2	9	
4	7	
6	5	i > j

Quick sort example

Input:

26	5	37	1	61	11	59	15	48	19

Pivot = The first element in the list 26

Partition: Starting with i = 0 , j = 10, find the index i of the element larger than pivot and the index j of the element smaller than the pivot. Then exchange them if i < j

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
26	5	37	1	61	11	59	15	48	19
26	5	19	1	61	11	59	15	48	37
26	5	19	1	15	11	59	61	48	37
11	5	19	1	15	26	59	61	48	37

i	j
2	9
4	7
6	5

i > j

Exchange a[0] and a[5]

Quick sort example

Similarly, partitioning the sublists recursively, we obtain the following result:

11	5	19	1	15	26	59	61	48	37

low	high
0	9

Quick sort example

Similarly, partitioning the sublists recursively, we obtain the following result:

11	5	19	1	15	26	59	61	48	37
1	5	11	19	15	26	59	61	48	37

low	high
0	9
0	4

Quick sort example

Similarly, partitioning the sublists recursively, we obtain the following result:

11	5	19	1	15	26	59	61	48	37
1	5	11	19	15	26	59	61	48	37
1	5	11	19	15	26	59	61	48	37

low	high
0	9
0	4
0	1

Quick sort example

Similarly, partitioning the sublists recursively, we obtain the following result:

11	5	19	1	15	26	59	61	48	37
1	5	11	19	15	26	59	61	48	37
1	5	11	19	15	26	59	61	48	37
1	5	11	15	19	26	59	61	48	37

low	high
0	9
0	4
0	1
3	4

Quick sort example

Similarly, partitioning the sublists recursively, we obtain the following result:

11	5	19	1	15	26	59	61	48	37
1	5	11	19	15	26	59	61	48	37
1	5	11	19	15	26	59	61	48	37
1	5	11	15	19	26	59	61	48	37
1	5	11	15	19	26	48	37	59	61
1	5	11	15	19	26	37	48	59	61
1	5	11	15	19	26	37	48	59	61
1	5	11	15	19	26	37	48	59	61

low	high
0	9
0	4
0	1
3	4
6	9
6	7
9	9

Quick sort performance

Best case, i.e. when the partition() always picks the middle element as pivot:

$O(n \log_2 n)$

Worst-case, i.e., when the list already is ordered but in reverse order:

$O(n^2)$

Average case:

$O(n \log_2 n)$

Quick sort

Differences from 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.

Stability

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