

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

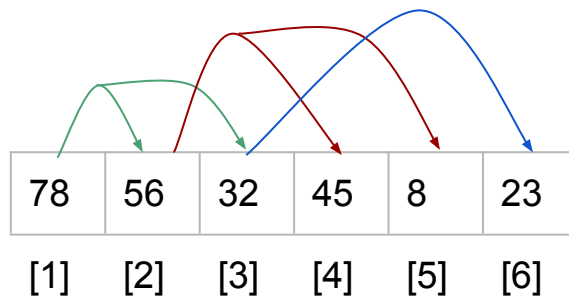
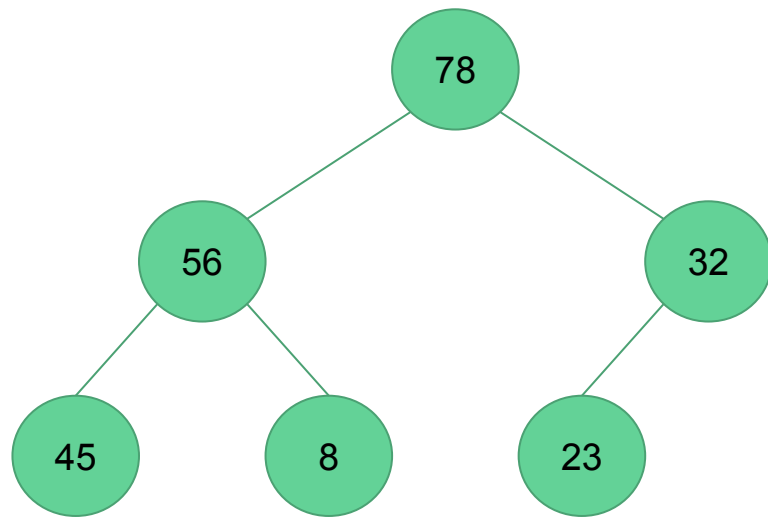
There are two kinds of binary heaps: **max-heaps** and **min-heaps**.

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)

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

BUILD-MAX-HEAP(A)

```
1   $A.\text{heap-size} = A.\text{length}$ 
2  for  $i = \lfloor A.\text{length}/2 \rfloor$  downto 1
3      MAX-HEAPIFY( $A, i$ )
```

HEAPSORT(A)

```
1  BUILD-MAX-HEAP( $A$ )
2  for  $i = A.\text{length}$  downto 2
3      exchange  $A[1]$  with  $A[i]$ 
4       $A.\text{heap-size} = A.\text{heap-size} - 1$ 
5      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)

```
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( $A, 1$ )
```

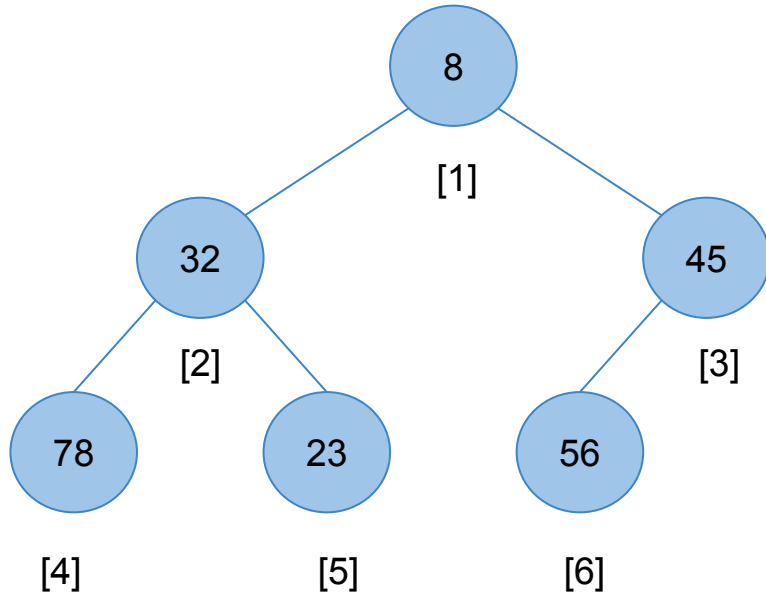
Heap sort

Example:

Sort the following data using heap sort:

8	32	45	78	23	56
---	----	----	----	----	----

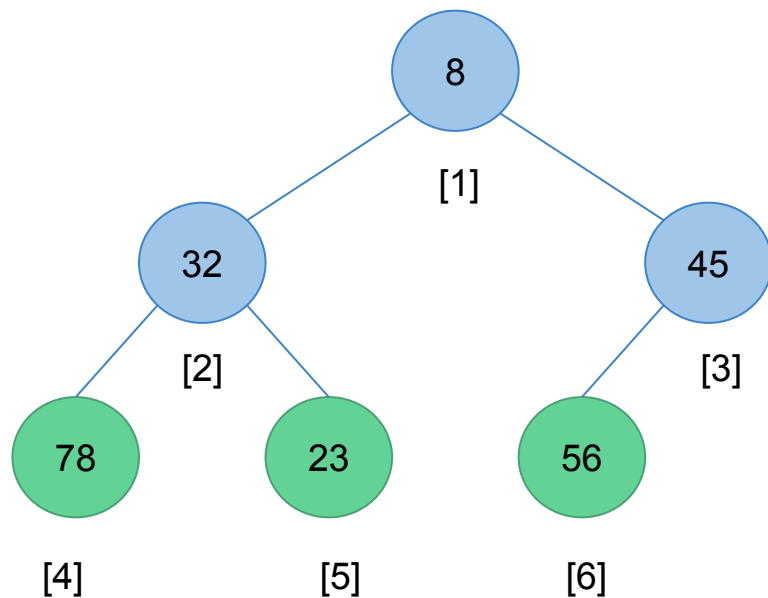
Heap sort



Convert the array into a max heap

8	32	45	78	23	56
---	----	----	----	----	----

Heap sort

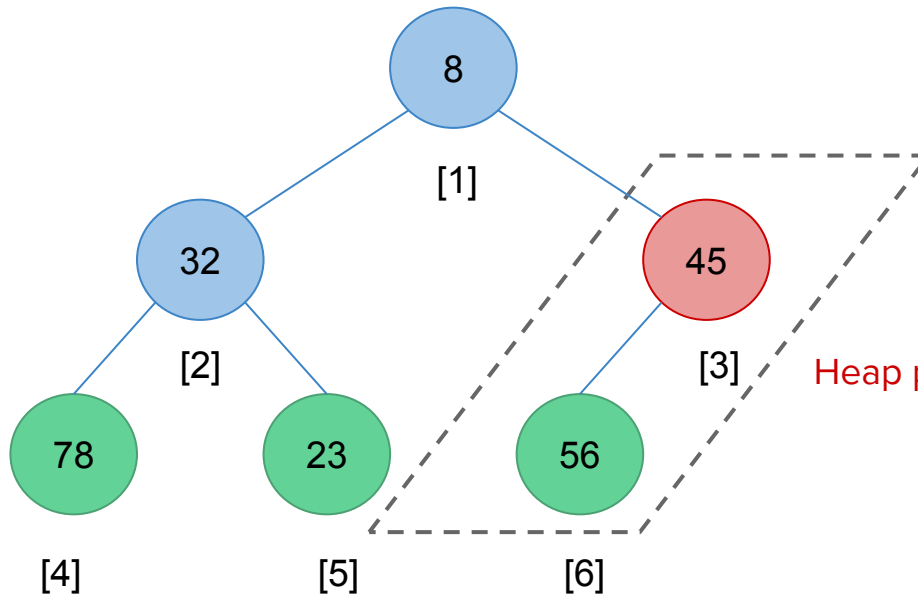


Convert the array into a max heap

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)

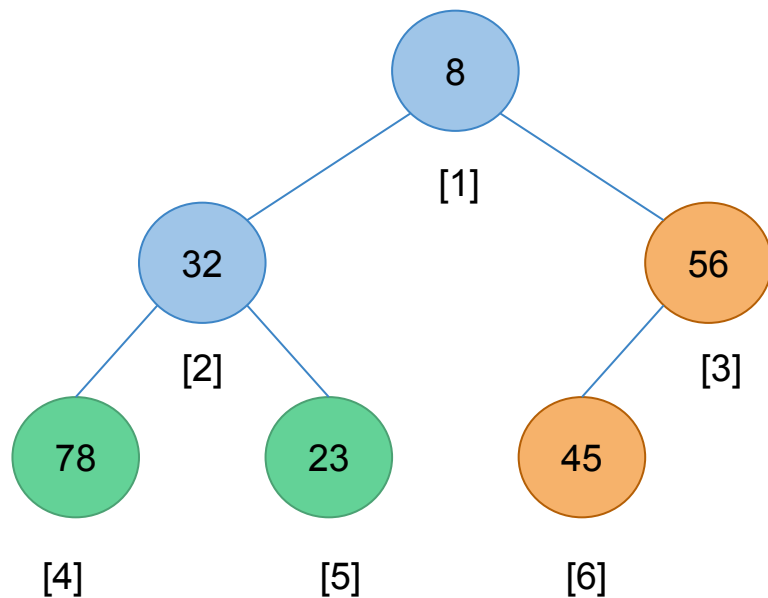
Heap sort



Convert the array into a max heap

8	32	45	78	23	56
---	----	----	----	----	----

Heap sort



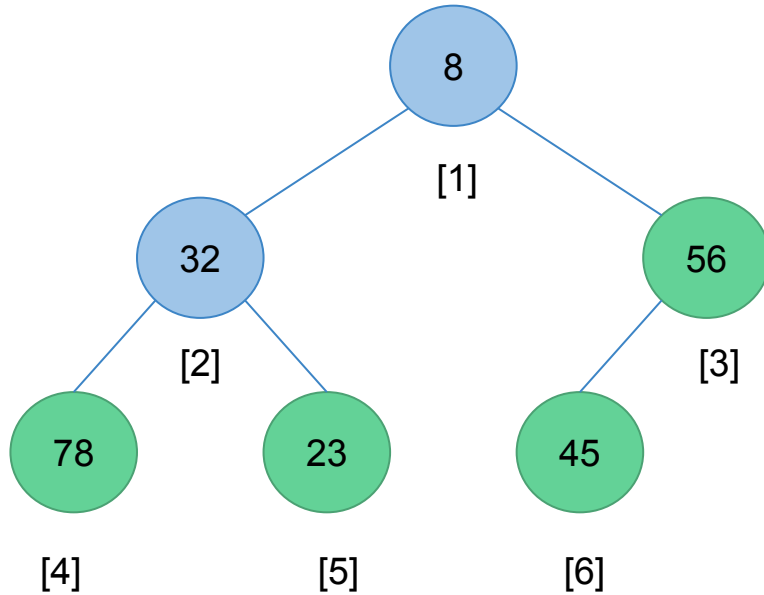
Convert the array into a max heap

8	32	56	78	23	45
---	----	----	----	----	----



Heap property is not satisfied. Therefore, swap them

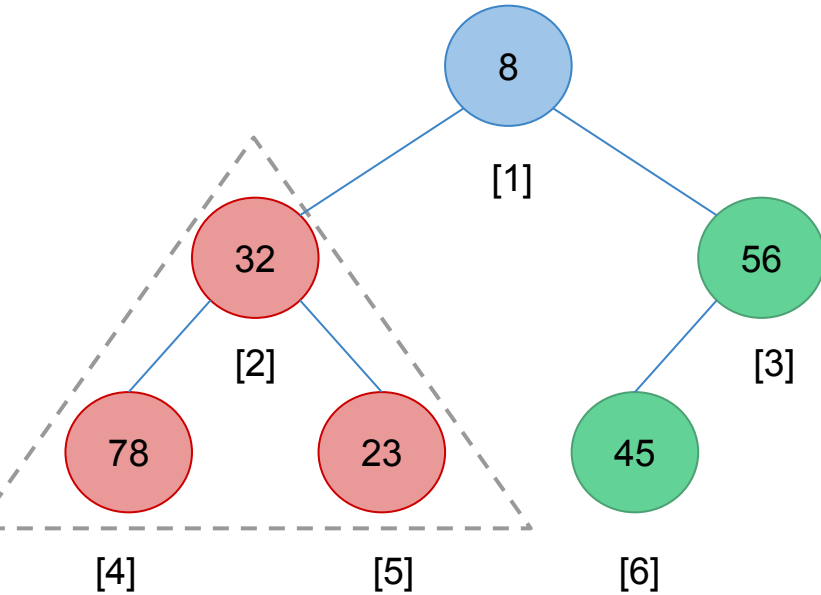
Heap sort



Convert the array into a max heap

8	32	56	78	23	45
---	----	----	----	----	----

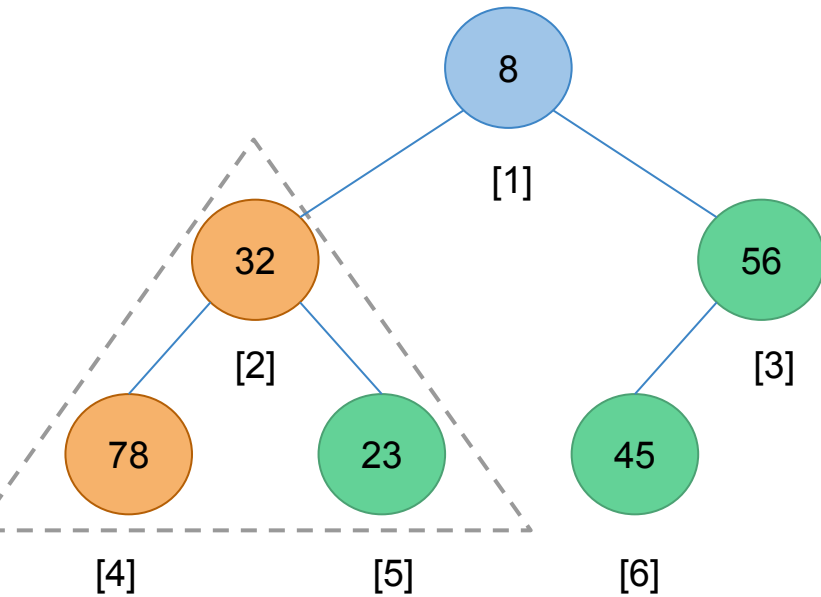
Heap sort



Convert the array into a max heap

8	32	56	78	23	45
---	----	----	----	----	----

Heap sort

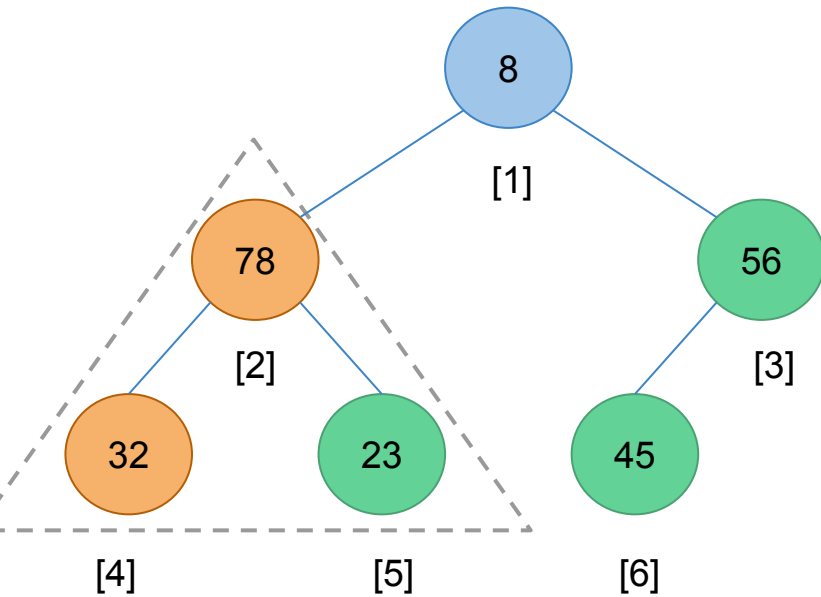


Convert the array into a max heap

8	32	56	78	23	45
---	----	----	----	----	----

Swap the root with the largest child

Heap sort

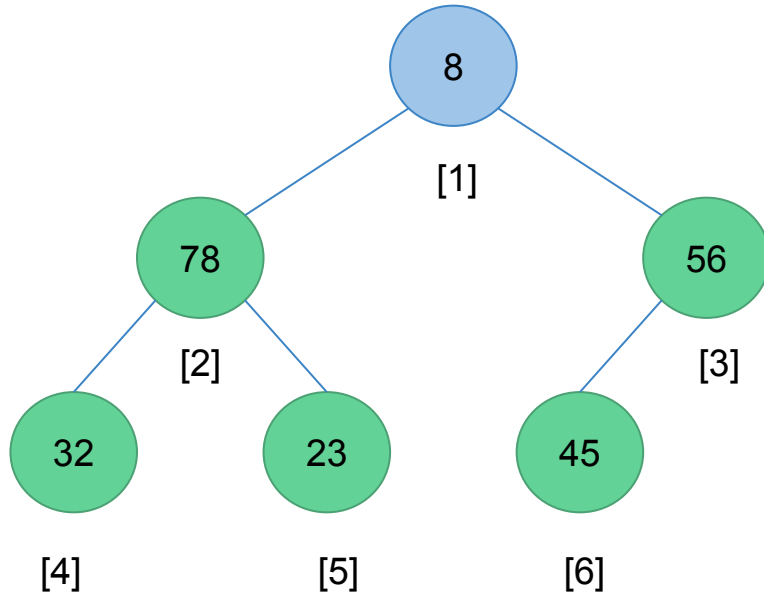


Convert the array into a max heap

8	78	56	32	23	45
---	----	----	----	----	----

Swap the root with the largest child

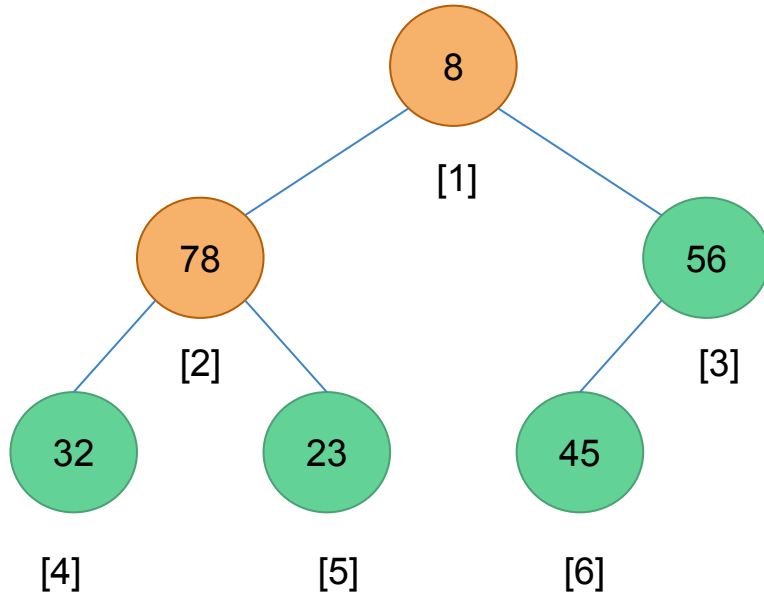
Heap sort



Convert the array into a max heap

8	78	56	32	23	45
---	----	----	----	----	----

Heap sort

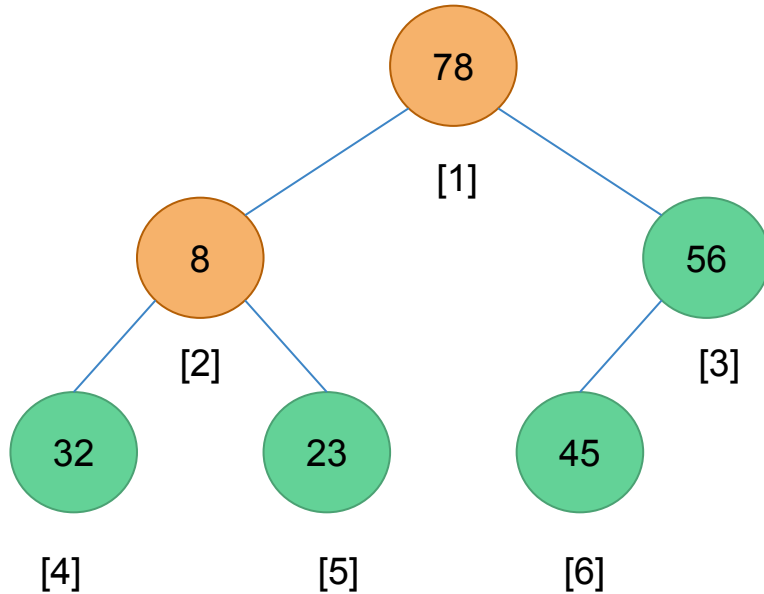


Convert the array into a max heap

8	78	56	32	23	45
---	----	----	----	----	----

Swap the root with the largest child

Heap sort

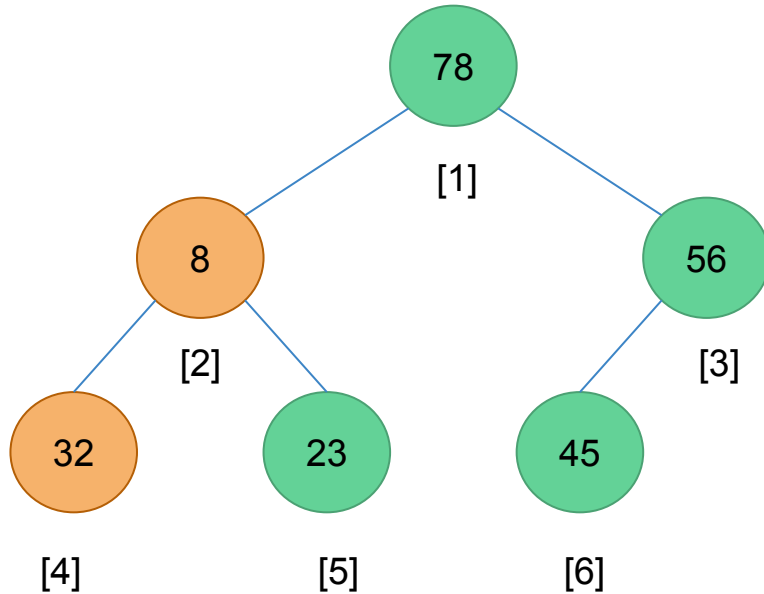


Convert the array into a max heap

78	8	56	32	23	45
----	---	----	----	----	----

Swap the root with the largest child

Heap sort

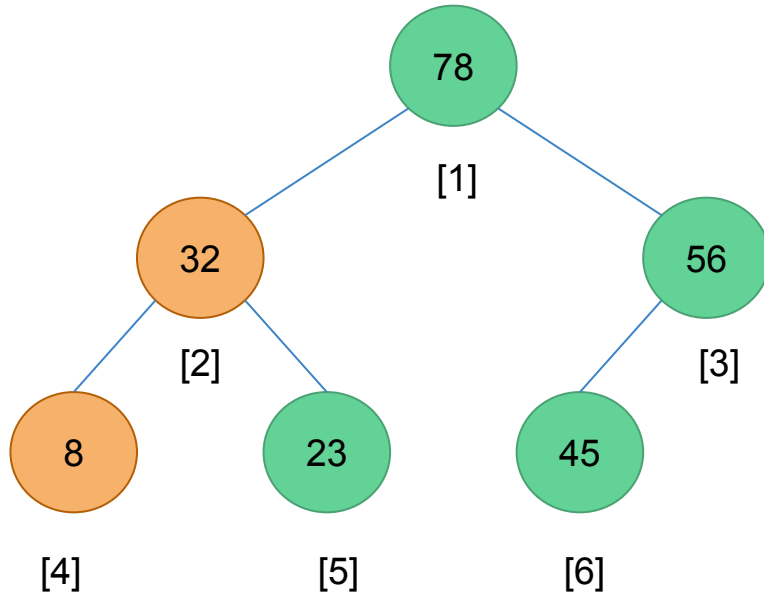


Convert the array into a max heap

78	8	56	32	23	45
----	---	----	----	----	----

Swap the root with the largest child

Heap sort

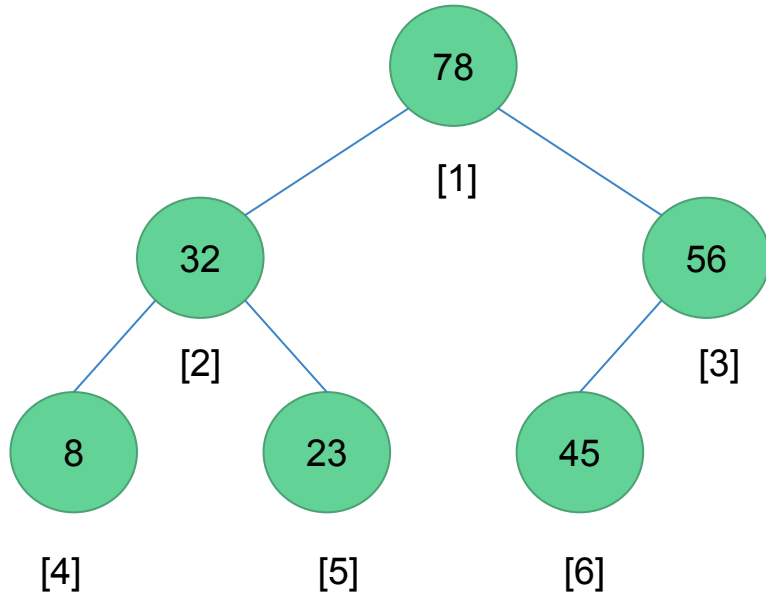


Convert the array into a max heap

78	32	56	8	23	45
----	----	----	---	----	----

Swap the root with the largest child

Heap sort

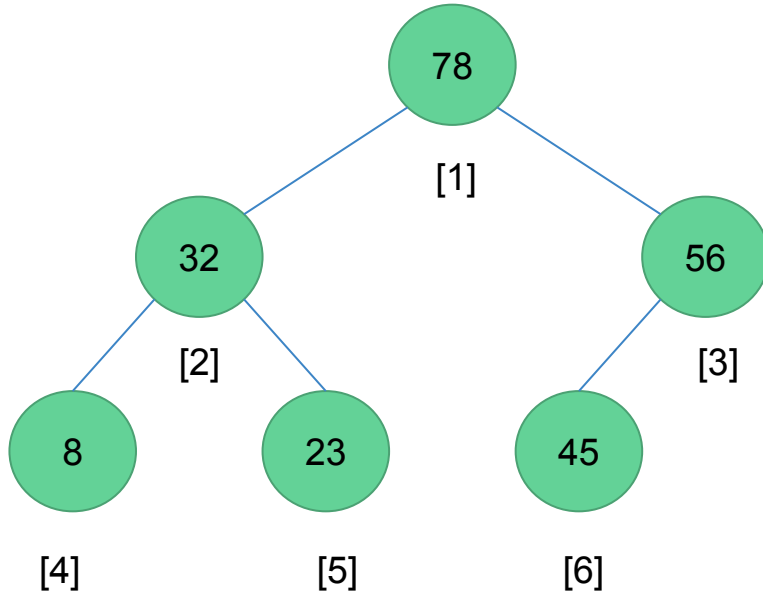


Convert the array into a max heap

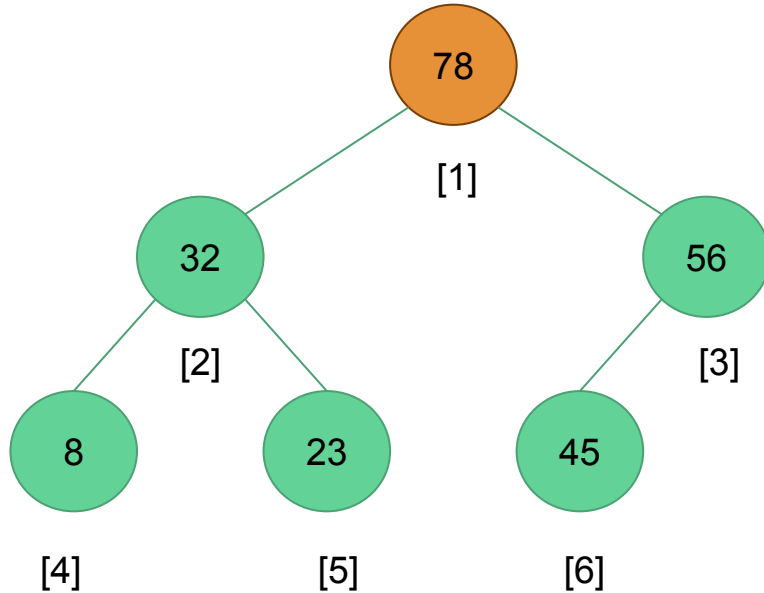
78	32	56	8	23	45
----	----	----	---	----	----

Heap sort

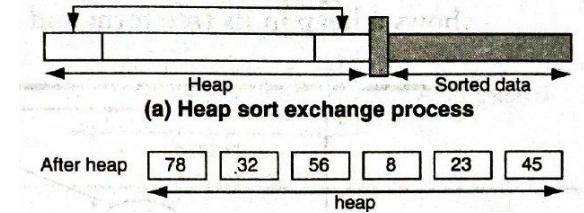
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



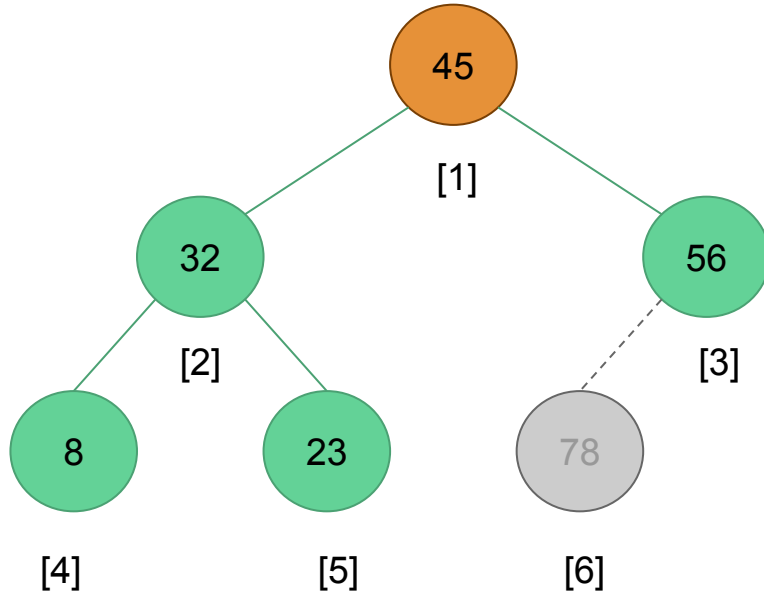
Heap sort



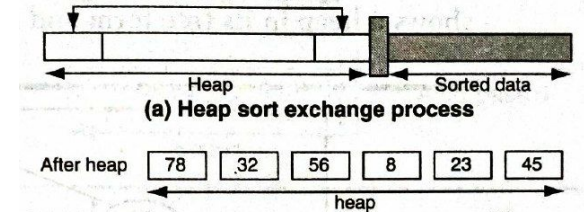
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



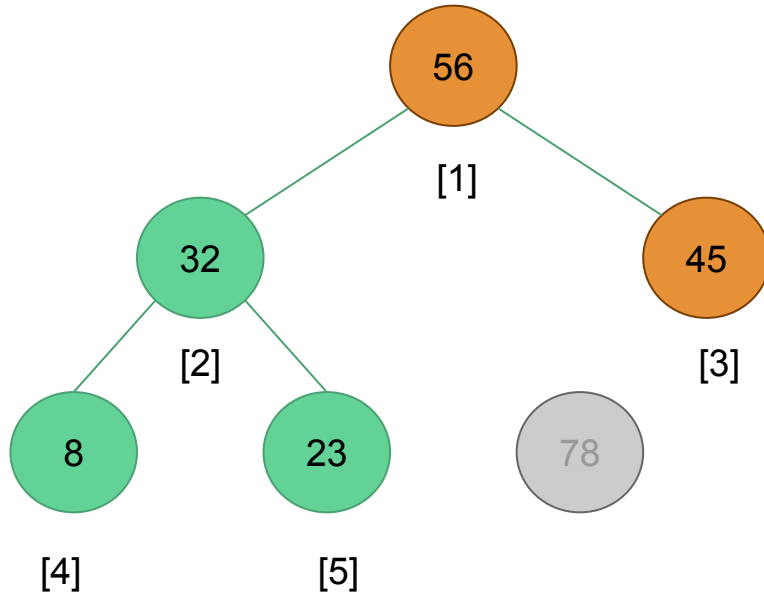
Heap sort



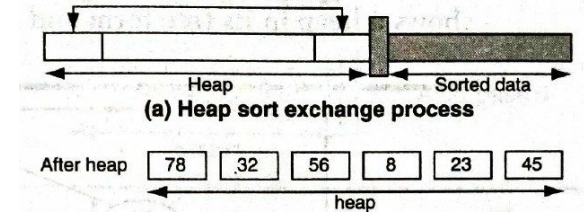
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



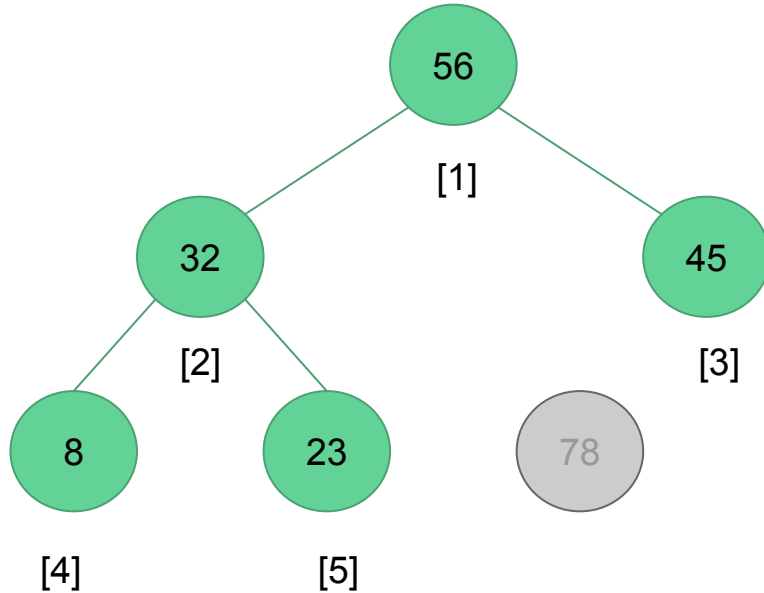
Heap sort



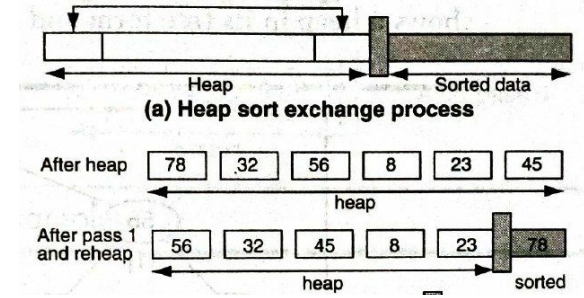
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



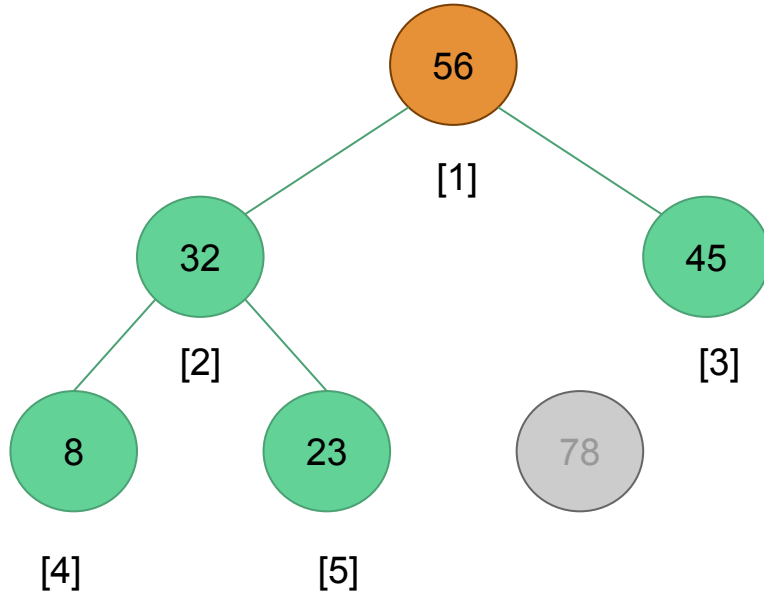
Heap sort



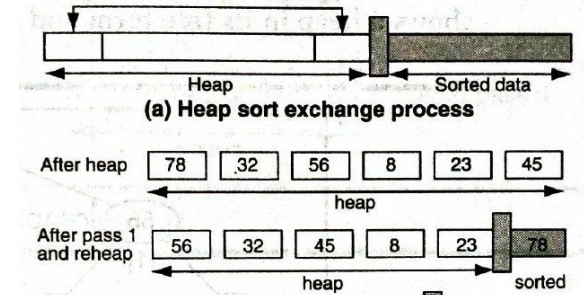
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



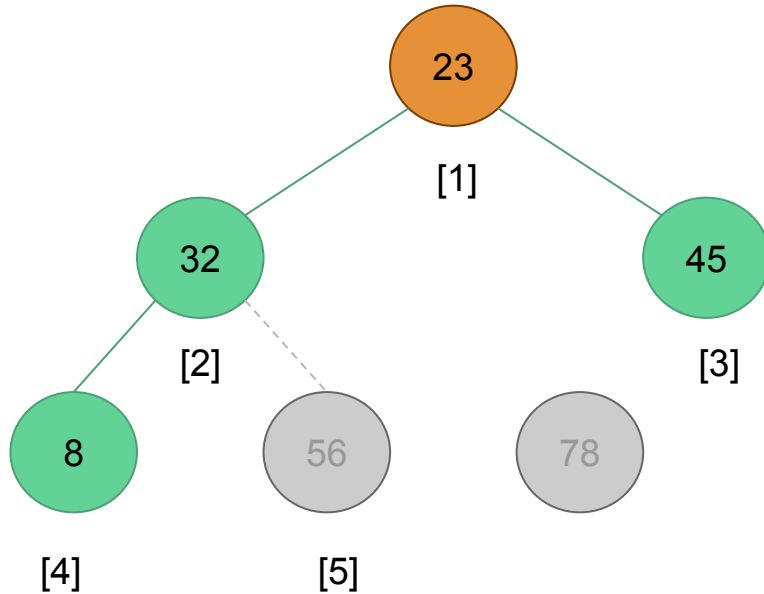
Heap sort



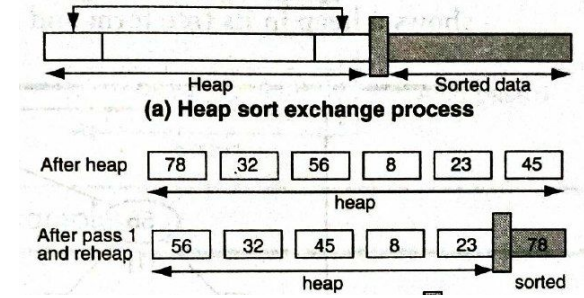
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



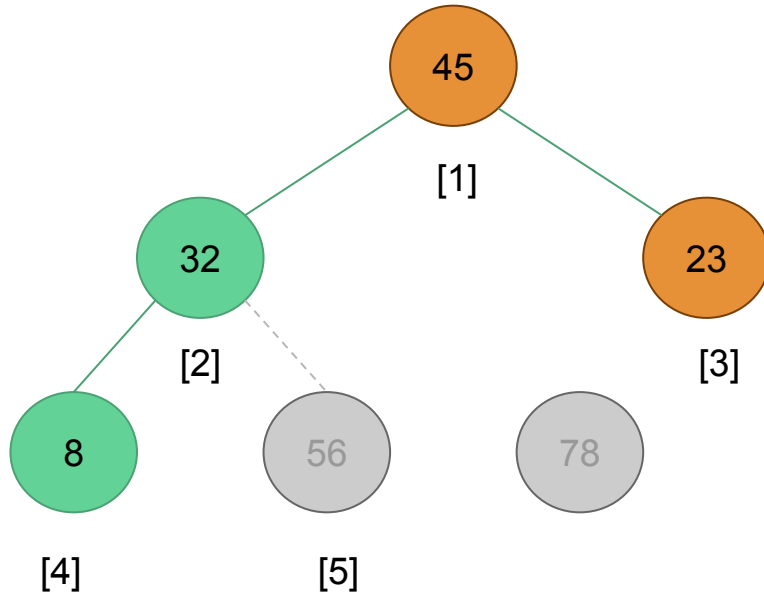
Heap sort



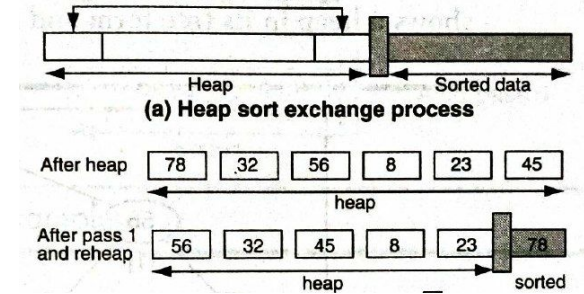
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



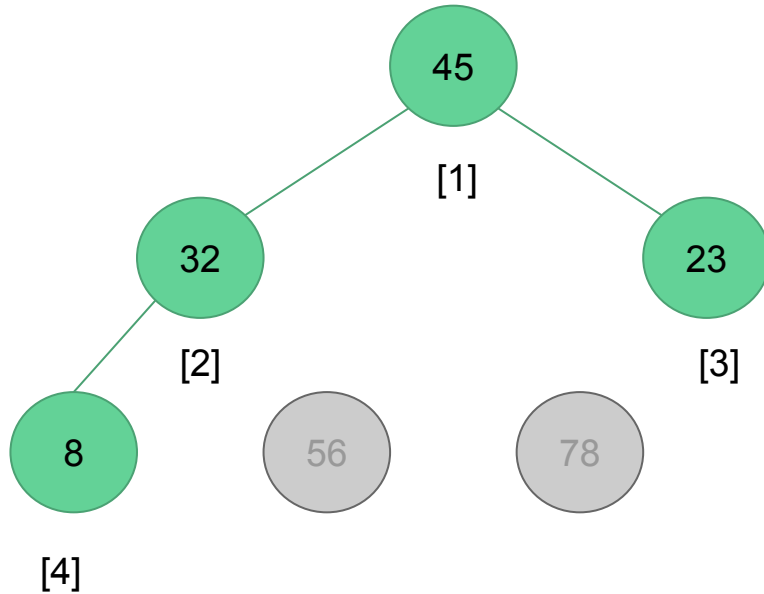
Heap sort



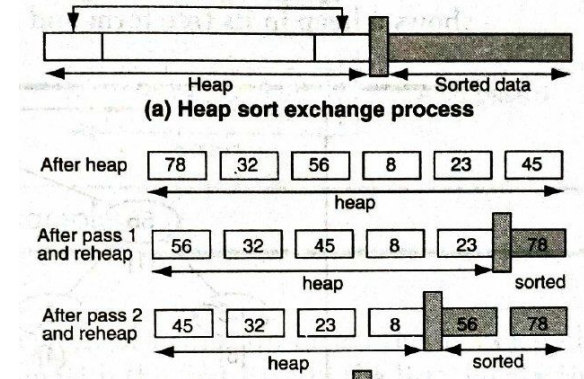
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



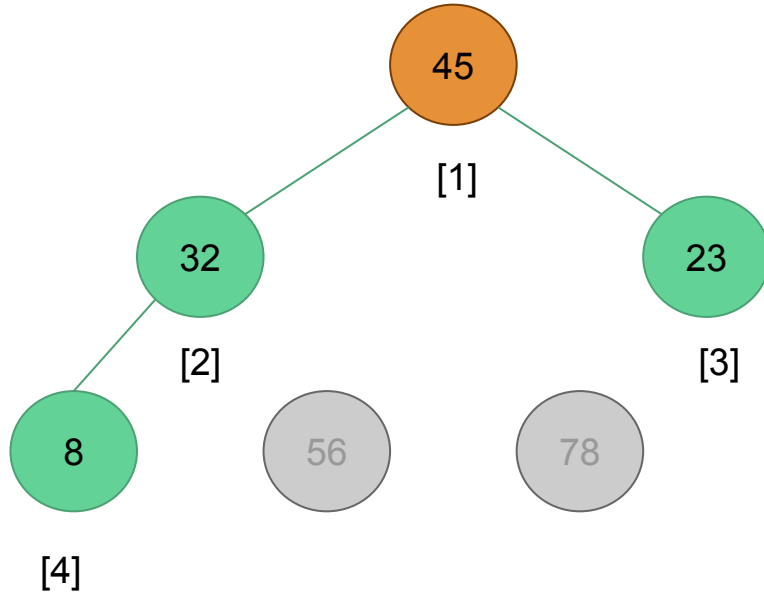
Heap sort



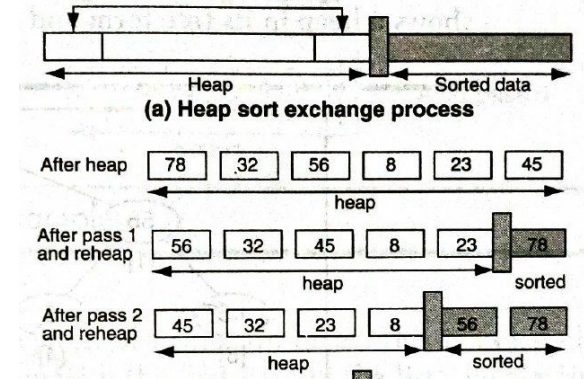
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



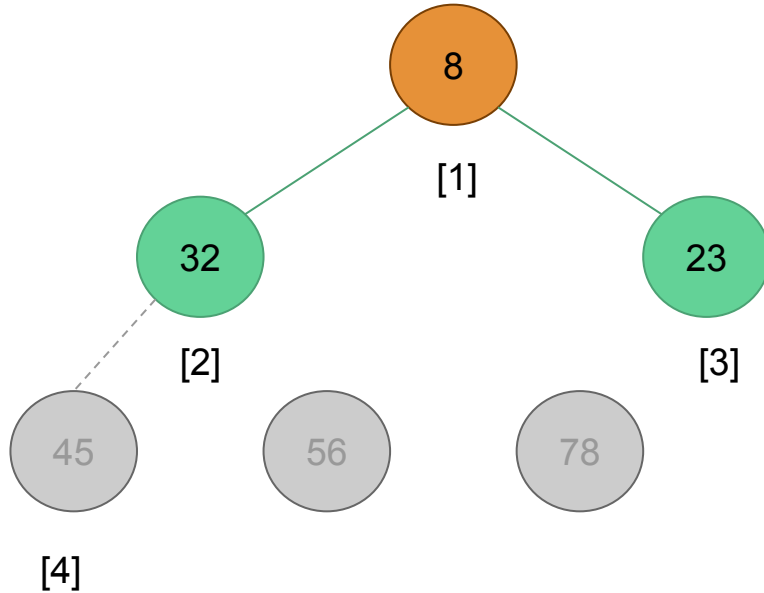
Heap sort



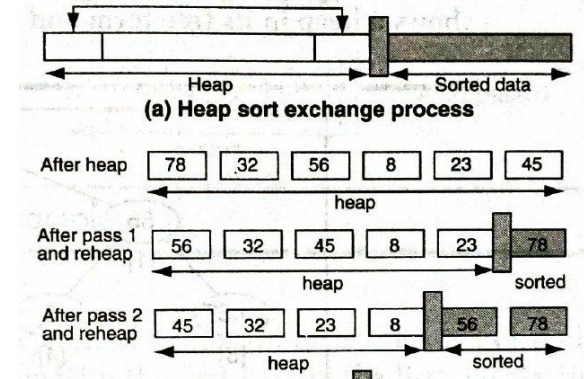
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



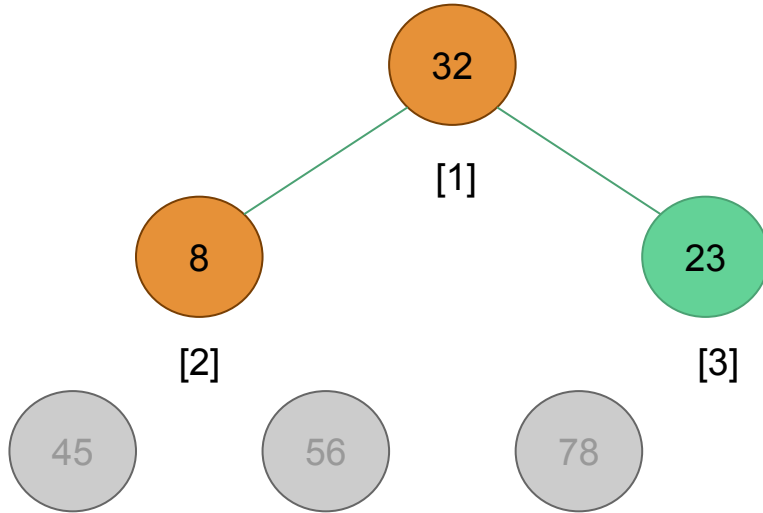
Heap sort



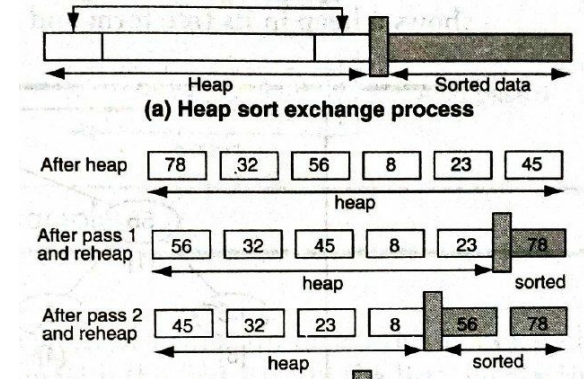
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



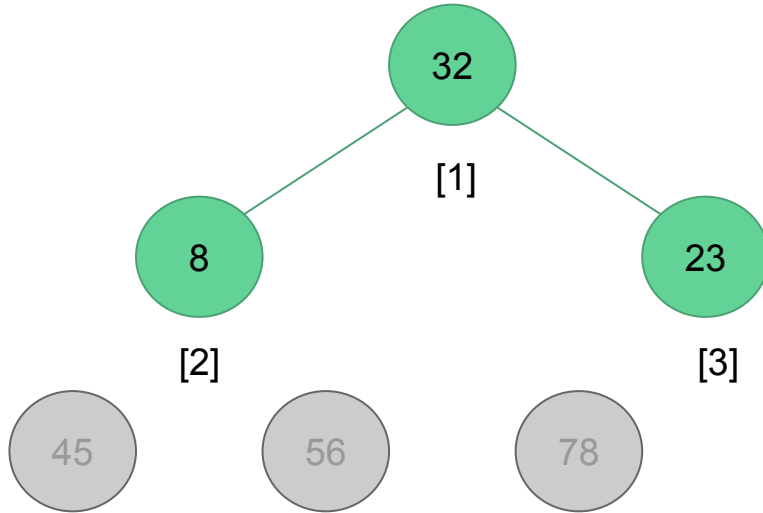
Heap sort



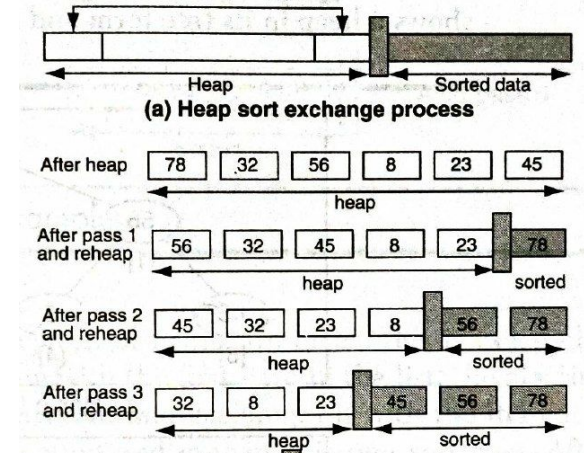
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



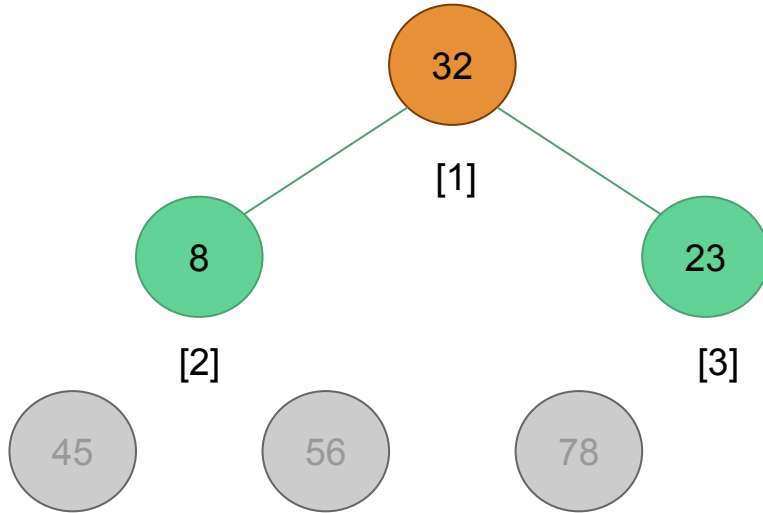
Heap sort



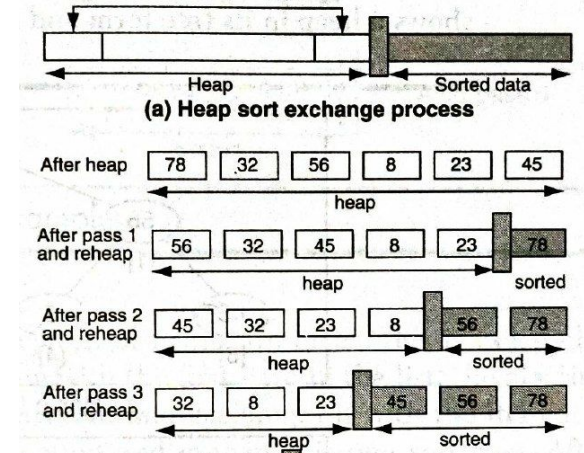
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



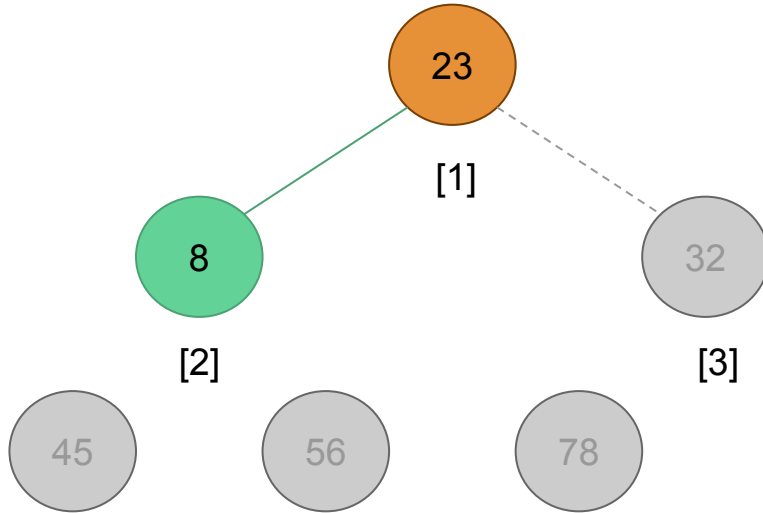
Heap sort



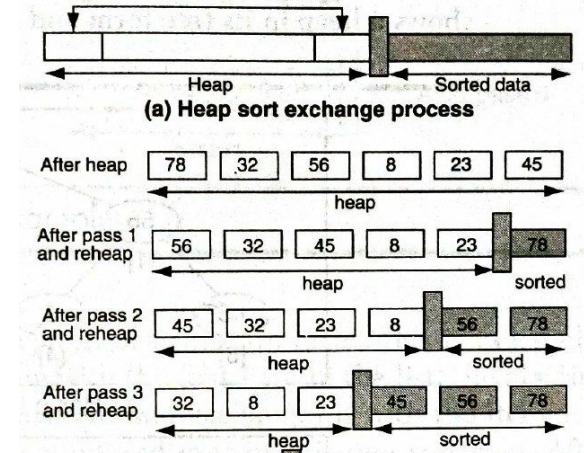
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



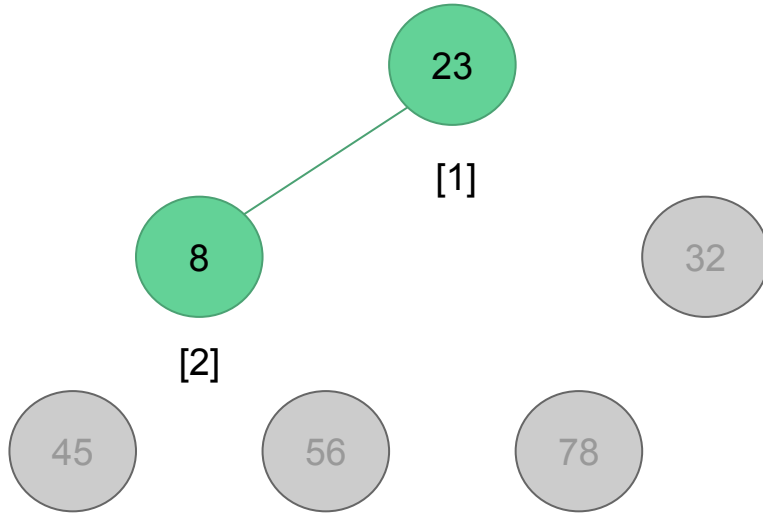
Heap sort



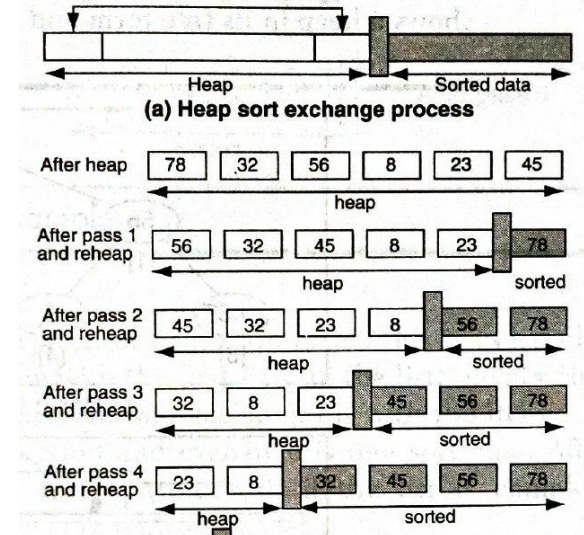
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



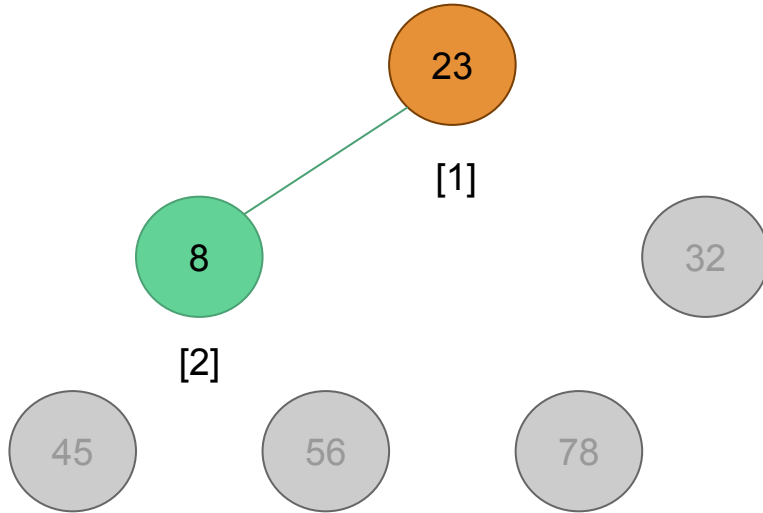
Heap sort



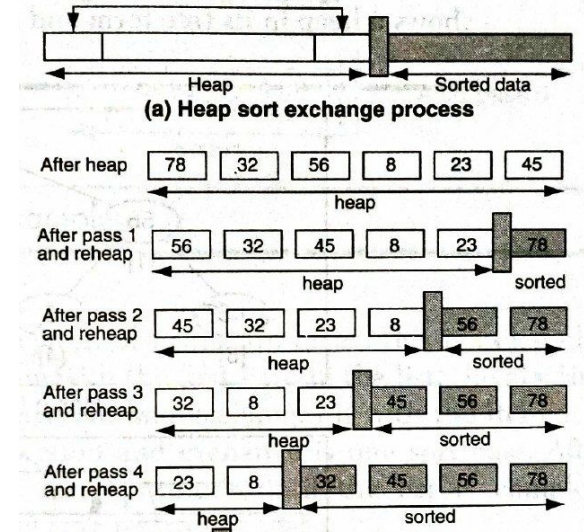
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



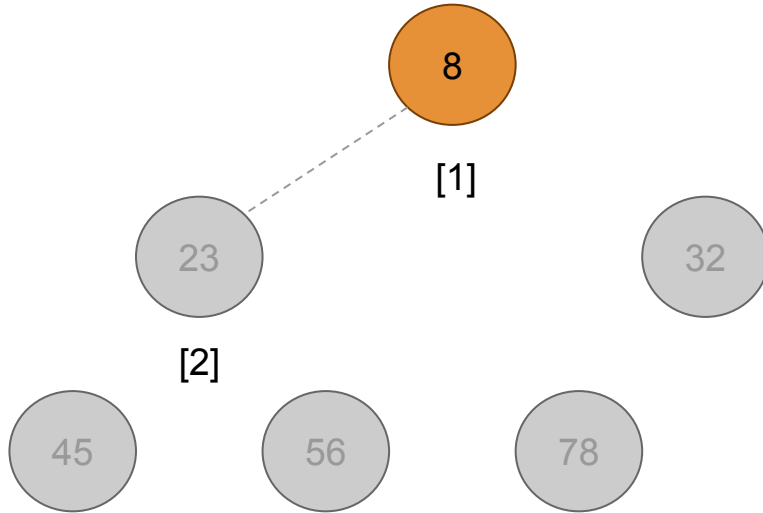
Heap sort



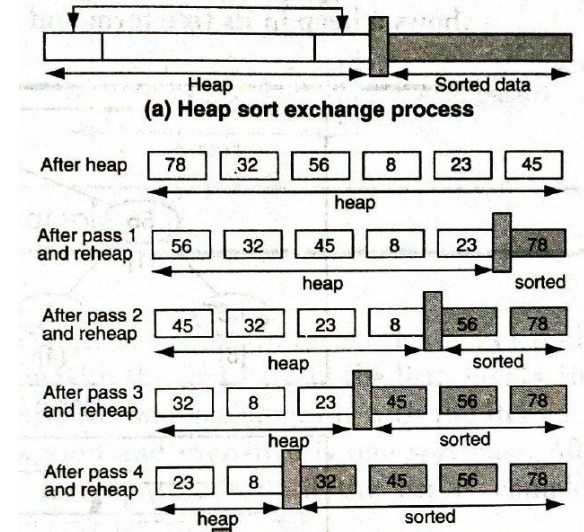
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



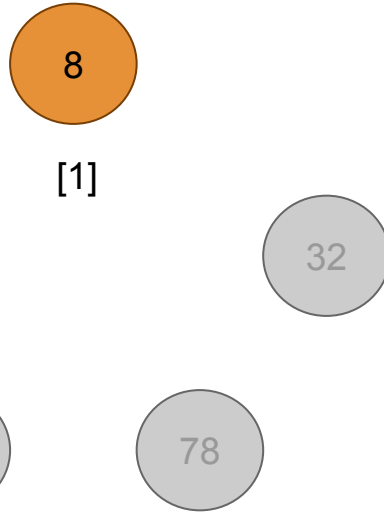
Heap sort



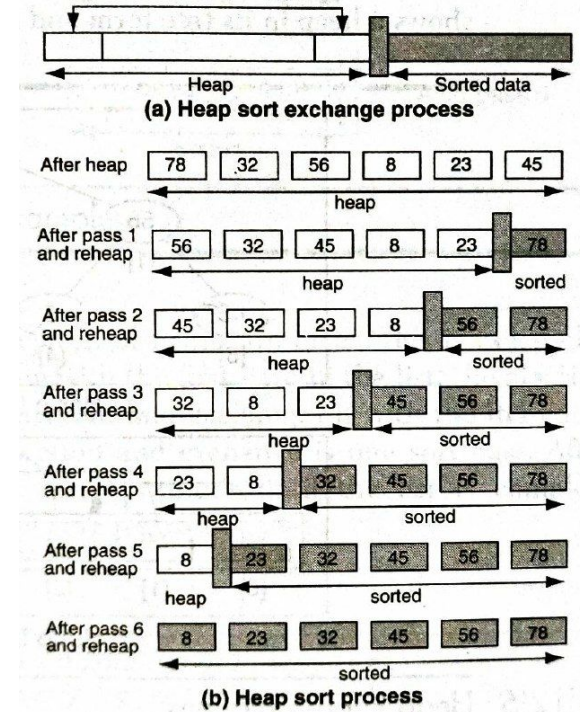
Swap the root of the heap with the element at the end of the list. Decrement the heap size by 1 and readjust the heap.



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.



Analysis of Max-Heapify

MAX-HEAPIFY(A, i)

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

Analysis of Max-Heapify

MAX-HEAPIFY(A, i)

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

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)

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

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)

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

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)

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

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)

```
1   $A.heap-size = A.length$   
2  for  $i = \lfloor A.length/2 \rfloor$  downto 1  
3      MAX-HEAPIFY( $A, i$ )
```

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

Analysis of Build-Max-Heap

BUILD-MAX-HEAP(A)

```
1   $A.heap-size = A.length$   
2  for  $i = \lfloor A.length/2 \rfloor$  downto 1  
3      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.

$$n = 2^{h+1} - 1$$

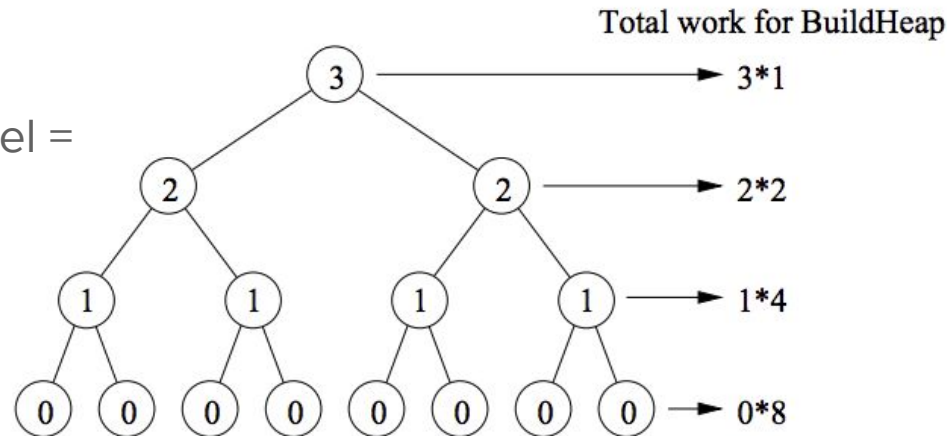
Analysis of build_heap

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.

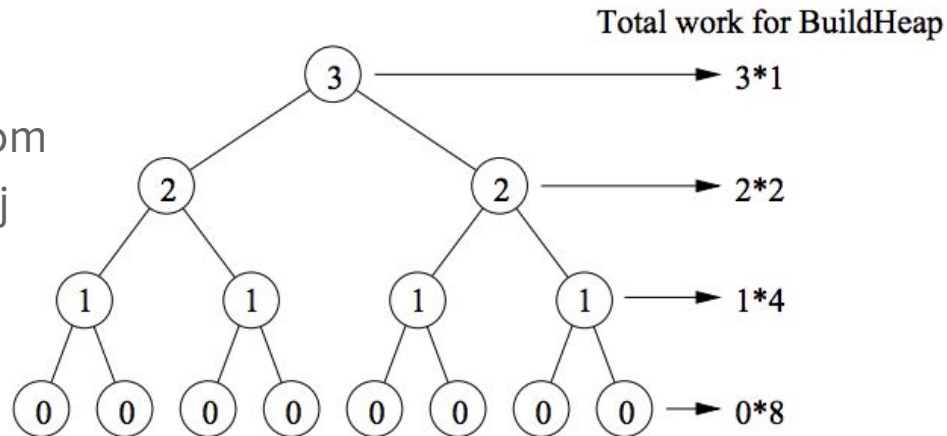


Analysis of build_heap

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



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

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 $x = 1/2$, 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 $\Omega(n)$ since every element of the array must be accessed at least once.

Therefore, $T(n)$ is $\Theta(n)$.

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($A, 1$)

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

$$T(n) = O(n) + O(n) + O(n \log n) = O(n \log n)$$

Stability

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