Heapsort

ITX2010, CSX3003, IT2230

Data Structures and Algorithms, Information Structures

Learning Objectives

Students will be able to:

- Explain heap properties
- Distinguish between Min heap and Max heap
- Illustrate or implement Application Program Interfaces of heap
- Describe how to apply heap to the applications
- Heapsort and its performance

Chapter Outline

6

Heapsort

Heap

Binary Heap

- Heap can be divided into 2 types:
  1. Binary heap : Its degree is 2 which is focused and call as “Heap”.
  2. d-Heap : Its degree is d value.
- It is a complete binary tree designed to support sort.
- There are two main types:
  1. Maximum binary heap (Max heap)
  2. Minimum binary heap (Min heap)

Heap

Binary Heap

- There are two main types:
  1. Maximum binary heap (Max heap)
  2. Minimum binary heap (Min heap)
- Its construction inherits from complete binary tree whose node contains at most two children and extends the heap order property as follows:
  - Max heap: Root always keeps the maximum value.
  - Min heap: Root keeps the minimum value.

Heap

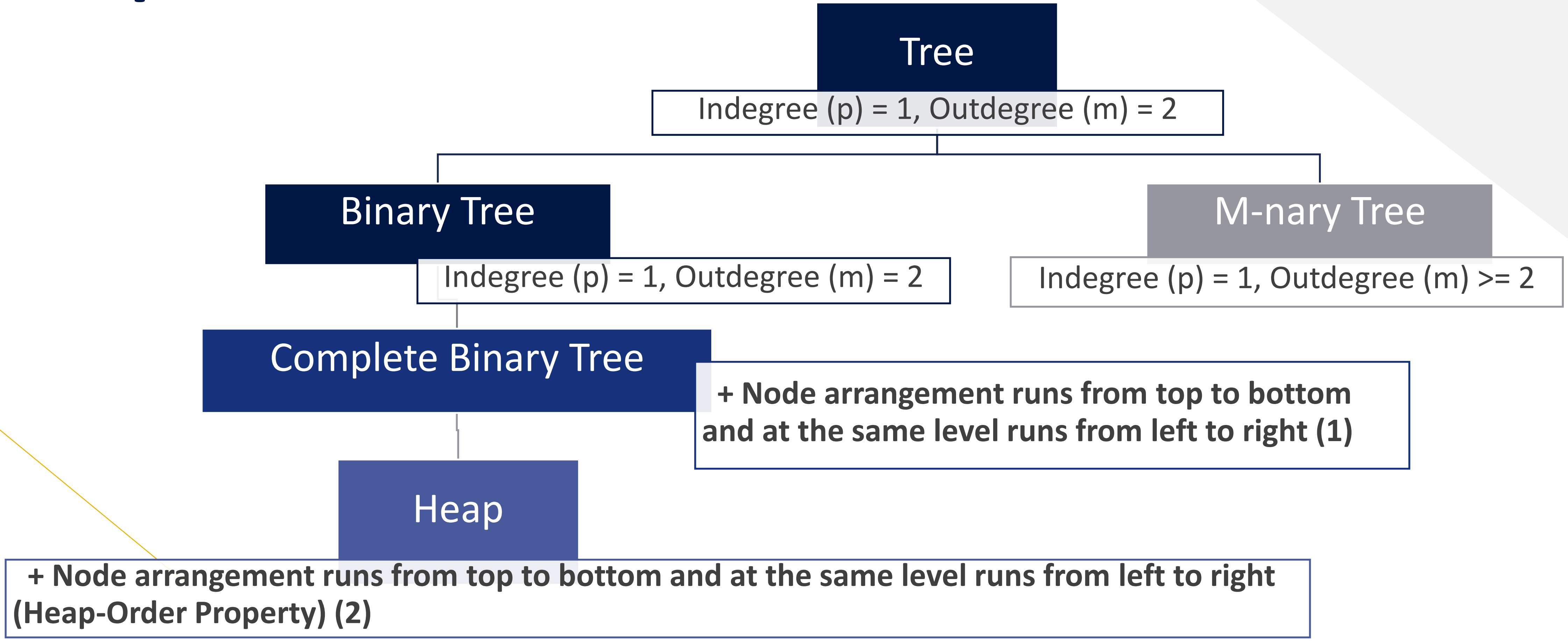

Tree
Indegree (p) = 1, Outdegree (m) = 2
Binary Tree
Indegree (p) = 1, Outdegree (m) = 2
M-nary Tree
Indegree (p) = 1, Outdegree (m) >= 2
Complete Binary Tree
+ Node arrangement runs from top to bottom and at the same level runs from left to right (1)
Heap
+ Node arrangement runs from top to bottom and at the same level runs from left to right (Heap-Order Property) (2)

Fig 6-1 Binary heap hierarchy

Heap

Heap property

- A heap is a binary tree that stores a collection of items at its positions and that satisfies two traditional properties: [1]
  1. **Complete binary tree shape**: Arrange all nodes from top to bottom and at the same level it runs from left to right
  2. **Heap-Order property**: Root keeps min (Min heap) or root keeps max (Max heap).

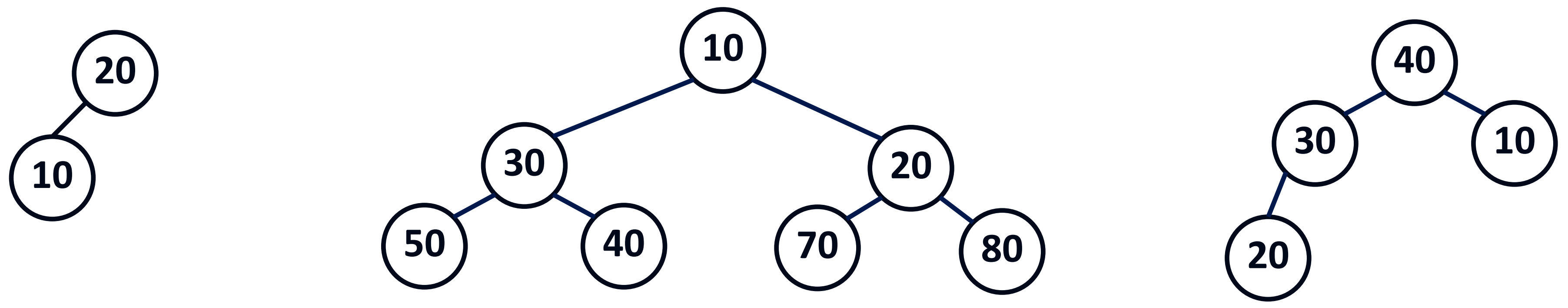
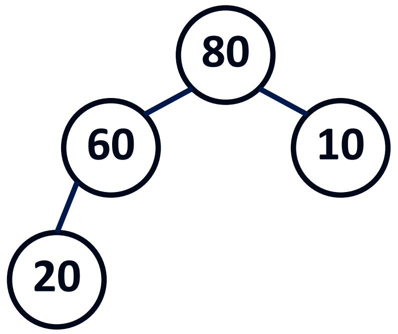


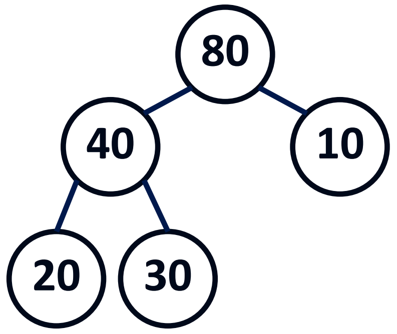
Fig 6-2 Binary heap examples

Heap

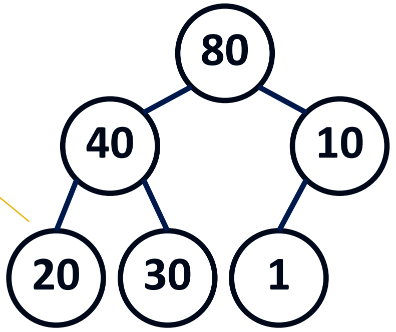
Complete binary tree shape



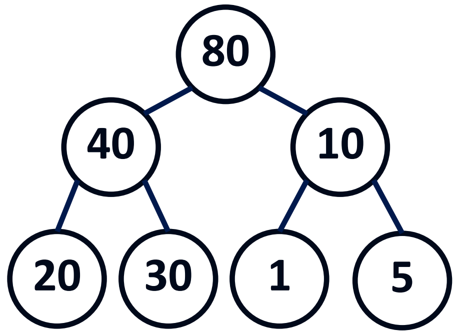
a) Complete binary tree 4 nodes



b) Complete binary tree and Full binary tree



c) Complete binary tree 6 nodes



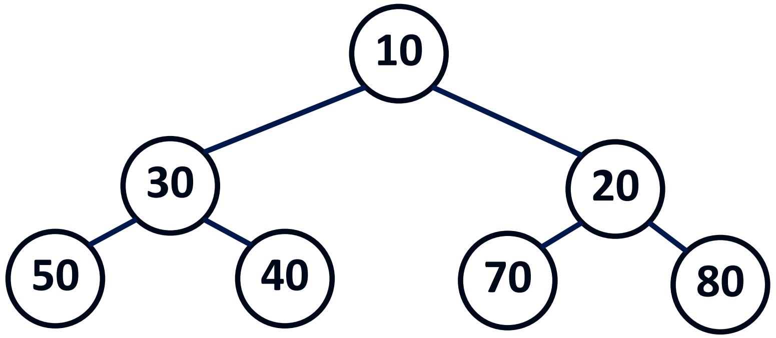
d) Complete binary tree 7 nodes, Full binary tree and Perfect binary tree

Fig 6-3 Complete binary trees whose height are 2

Heap

Heap-Order property

- Min heap $A[\mathrm{PARENT}(i)] \le A[i].$
- Max heap $A[\mathrm{PARENT}(i)] \ge A[i],$



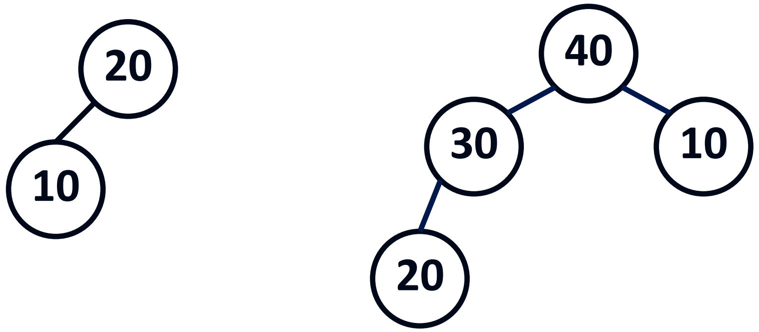


Fig 6-4 Heap-Order properties

Heap

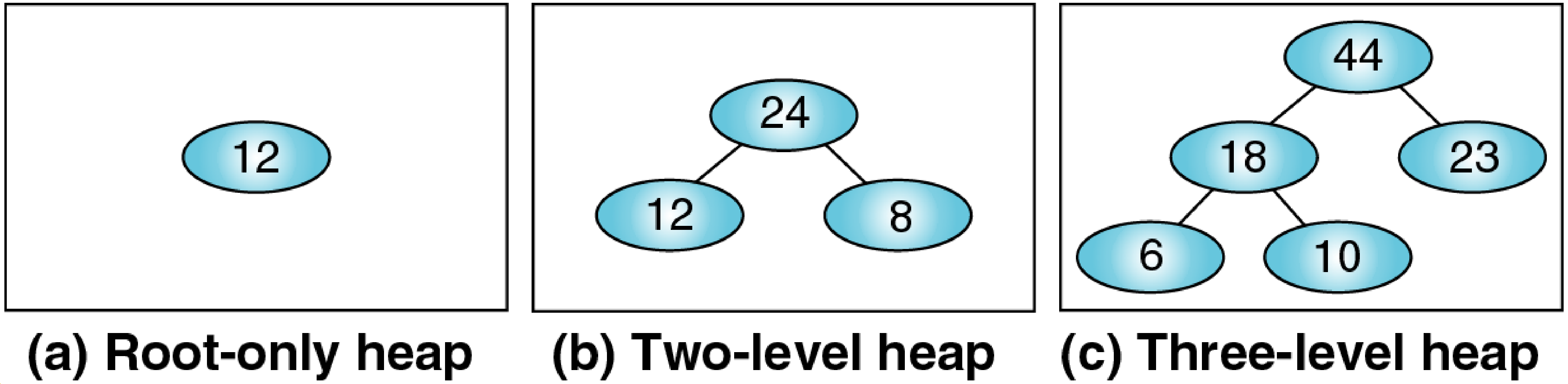
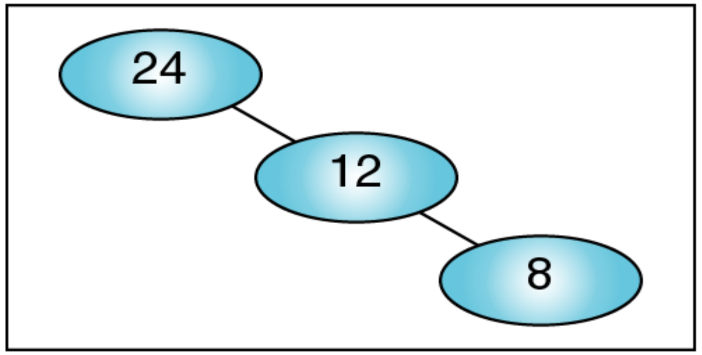
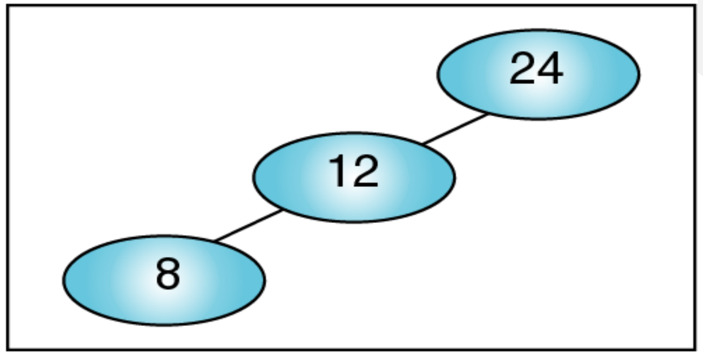


Fig 6-5 Max heap examples [3]

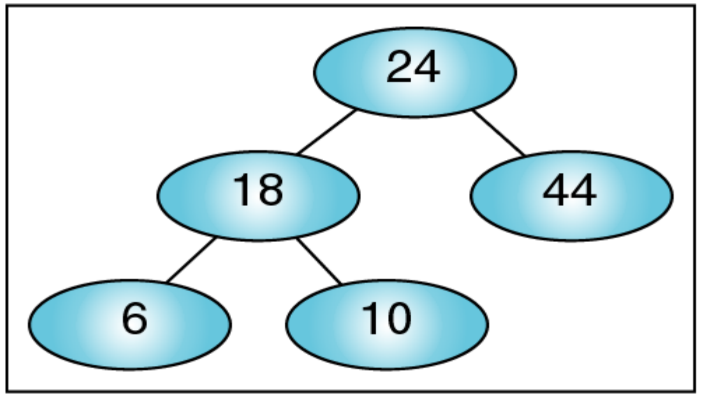
Heap



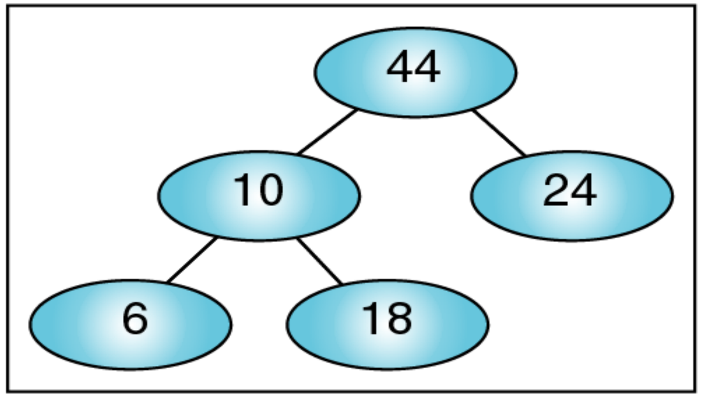
a)



b)



c)



d)

Fig 6-6 Binary tree examples [3]

Heap Operation

Methods / Operations	Description
Create a heap	Initial an empty heap
Search: MAX-HEAP-EXTRACT-MAX MAX-HEAP-MAXIMUM	Since heap is not design to support search, searching heap can be done sequentially. It runs in O(1) time.
Insertion, MAX-HEAP-INSERT	Heap insertion follows the complete binary tree rule but it extends Heap-Order property as well. It runs in O(lg n) time plus the time for mapping between objects being inserted into the priority queue and indices in the heap.
Deletion	Root node will be removed from structure automatically. It runs in O(1) time.

Table 6-1 Heap operations

Heap Operation

Methods / Operations	Description
MAX-HEAPIFY	Maintain the max-heap property which runs in O(lg n) time.
BUILD-MAX-HEAP	Since heap is complete binary tree, traverse every AVL node will be level order traversal.
HEAPSORT	Since heap is not design to support search, searching heap can be done sequentially.

Table 6-2 Heap operations

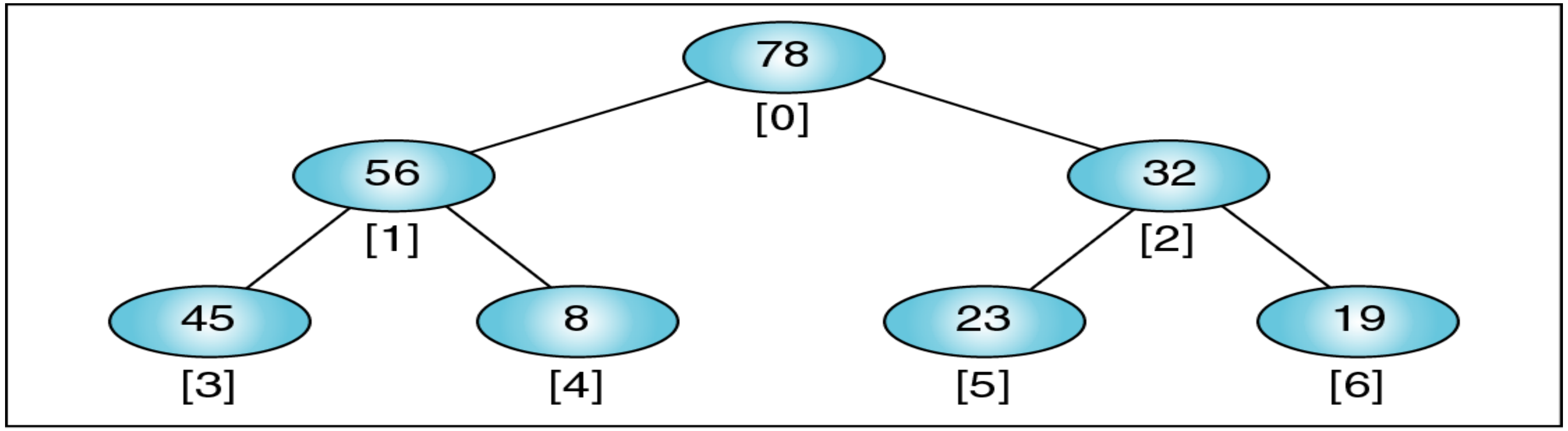
Heap Operation

Create heap and Node access

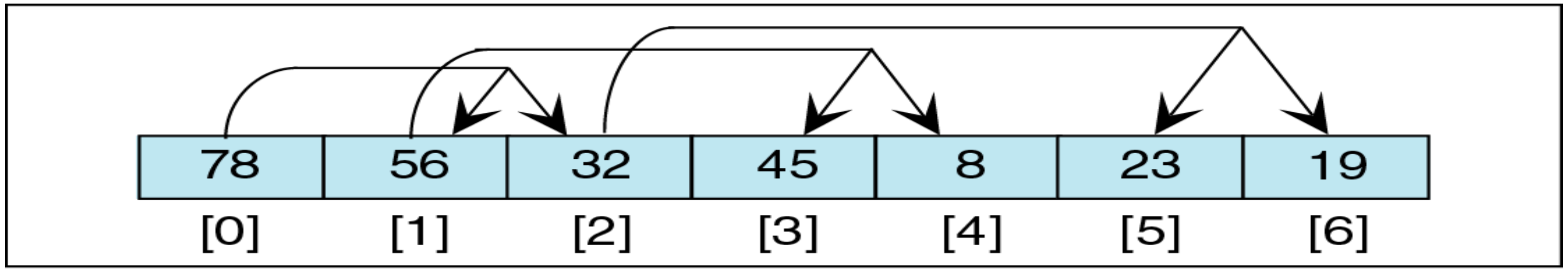
- Although a heap can be built in a dynamic tree structure, it is most often implemented in an array. [2]
- Given an array index i:
  - Parent node is at: Lover bound (i / 2)
  - Left child is at: 2 * i
  - Right child is at: 2 * i + 1

Heap Operation

Create heap and Node access



(a) A heap in its logical form



(b) A heap in an array

Fig 6-7 Heap representation [3]

Heap Operation

Create heap and Node access

```
class MaxHeap :
  def __init__( self, maxSize ):
    self._elements = Array( maxSize )
    self._count = 0

  def __len__( self ):
    return self._count

  def capacity( self ):
    return len( self._elements )

  def _swap( self, ind1, ind2):
    tmp = self._elements[ind1]
    self._elements[ind1] = self._elements[ind2]
    self._elements[ind1] = tmp

 # ...
```

Fig 6-8 Heap constructor [2]

Heap Operation

Reheap/Percolate/Shift up

- We have a nearly complete binary tree with N elements whose first N-1 elements satisfy the order property of heaps, but the last element does not. [3]
- The operation repairs the structure so that it is a heap by floating the last element up the tree until that element is in its correct location in the tree. [3]
- It is an operation to move a new key to the right position which is not conflict to Heap-Oder property.

Heap Operation

Reheap/Percolate/Shift up

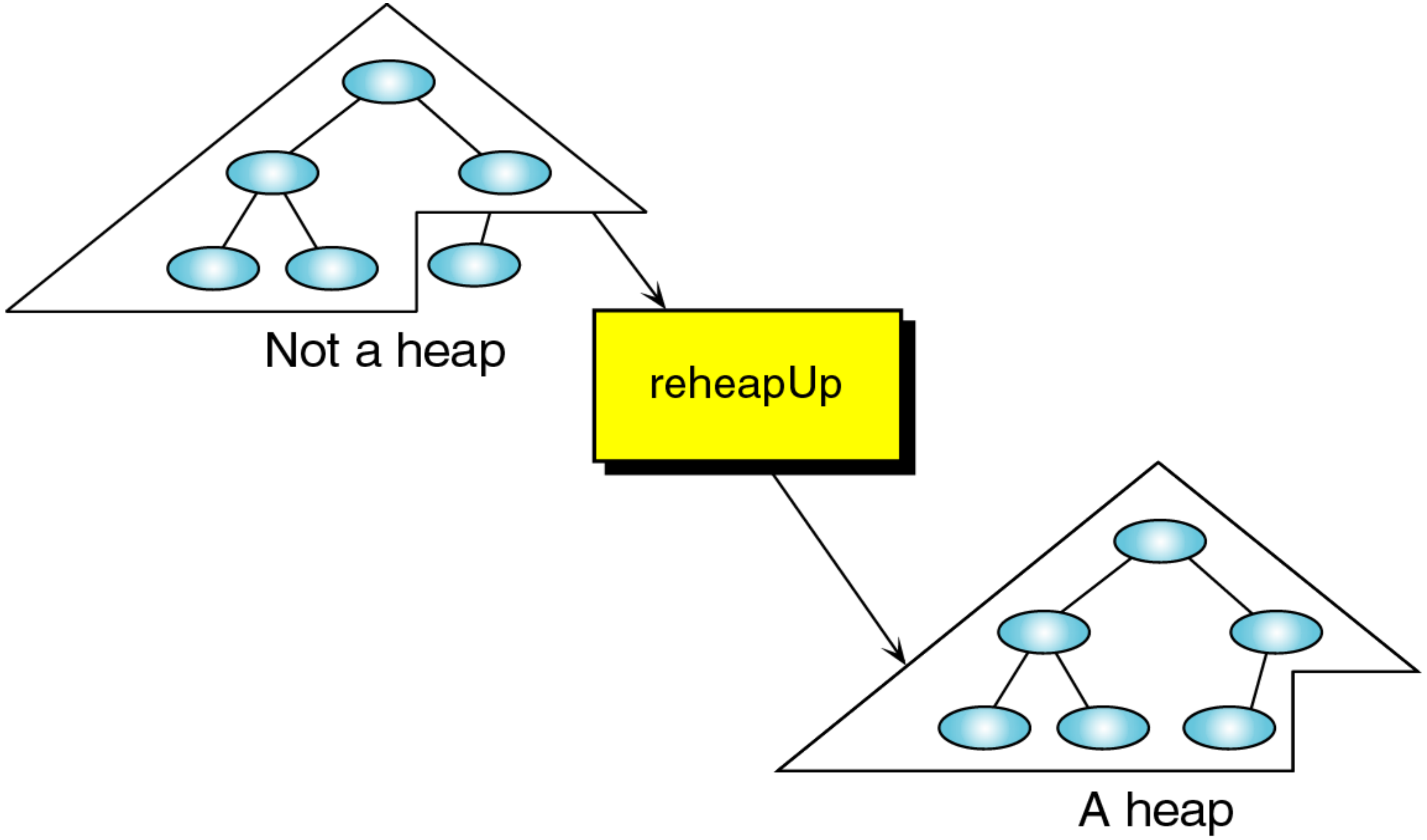


Fig 6-9 Logical view of reheap up [3]

Heap Operation

Reheap/Percolate/Shift up

- If the new node key is larger than its parent, it is floated up the tree by exchanging it with its parent. [3]
- The node eventually rises to its correct position in the heap by repeatedly exchanging it with its parent.

Heap Operation

Reheap/Percolate/Shift up

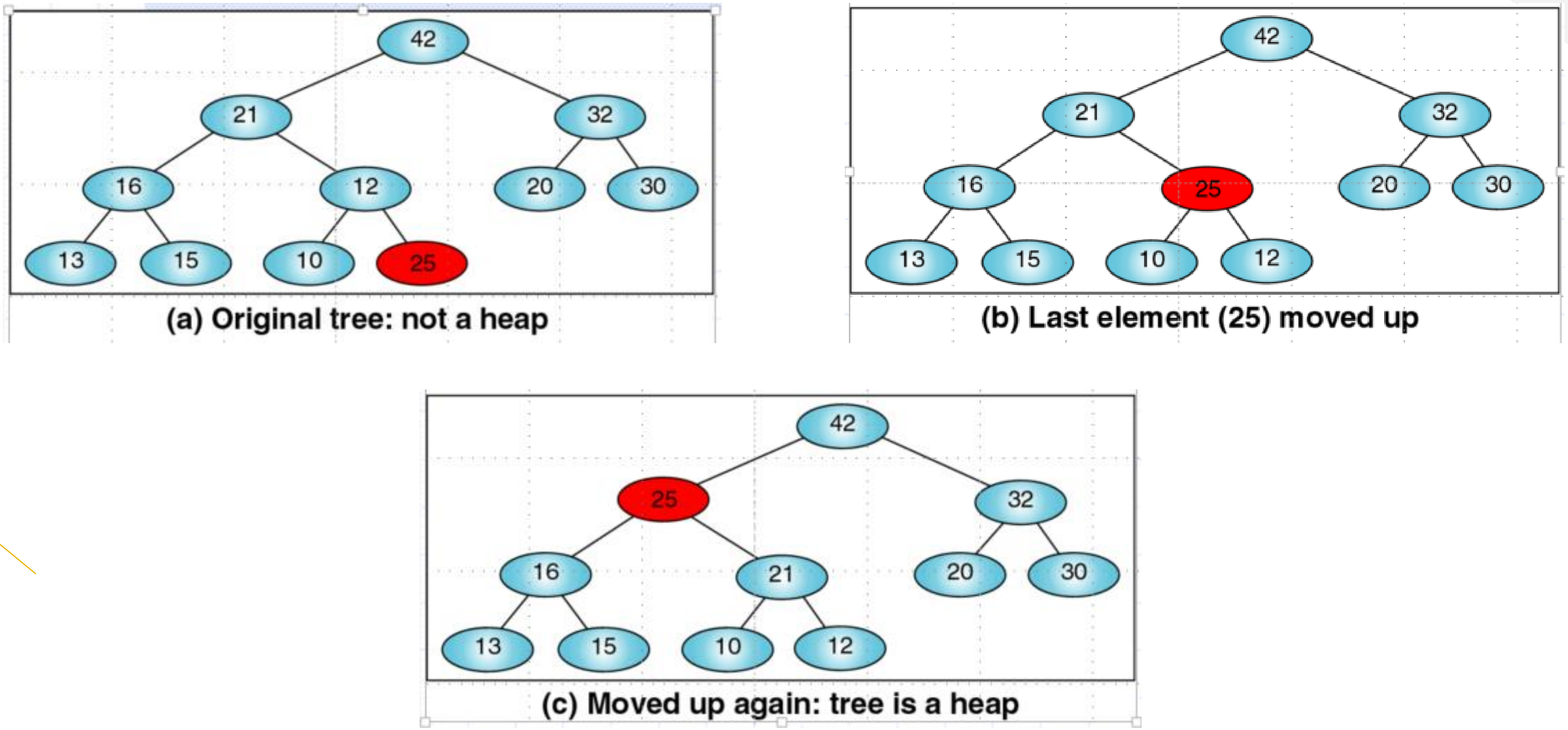

(a) Original tree: not a heap
(b) Last element (25) moved up
(c) Moved up again: tree is a heap

Fig 6-10 Reheap up operation [3]

Heap Operation

Reheap/Percolate/Shift down

- This situation occurs when the root is deleted from the tree, leaving two disjoint heaps. [2]
- Reheap down repairs a broken heap by pushing the root down the tree until it is in its correct position in the heap. [2]
- Move the removed key by swapping with an appropriated child until it is leaf node.

Heap Operation

Reheap/Percolate/Shift down

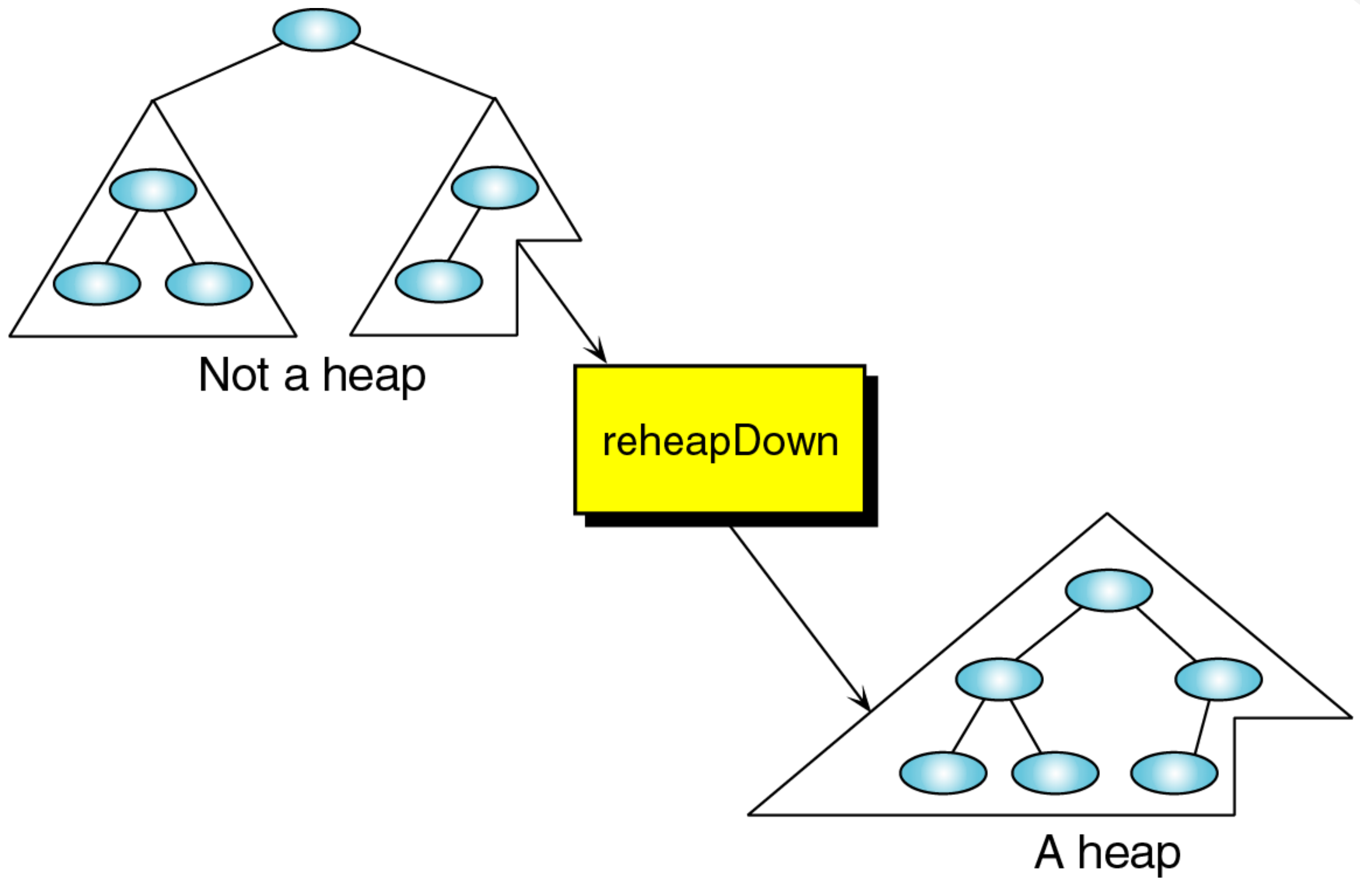


Fig 6-11 Reheap down operation [3]

Heap Operation

Reheap/Percolate/Shift down

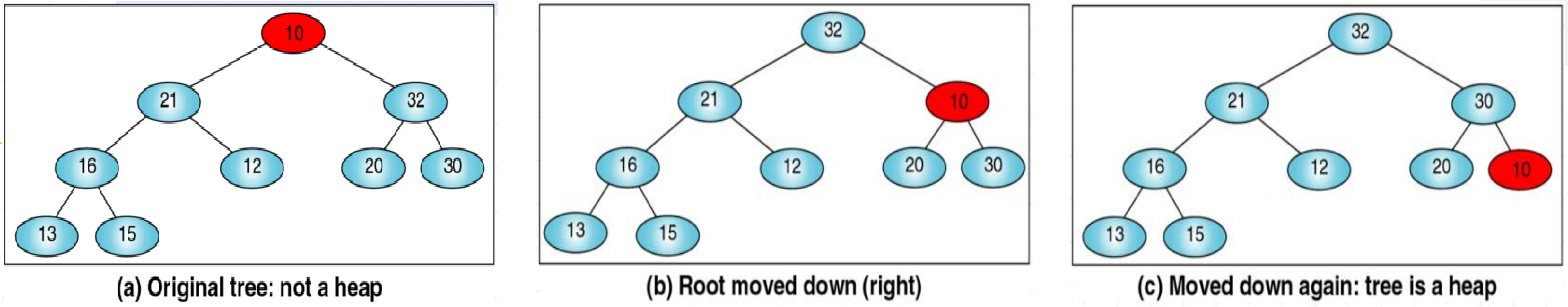

10
21
32
16
12
20
30
13
15
(a) Original tree: not a heap
32
21
10
16
12
20
30
13
15
(b) Root moved down (right)
32
21
30
16
12
20
10
13
15
(c) Moved down again: tree is a heap

Fig 6-12 Reheap down operation [3]

Heap Operation

Insertion

- Inserts into heaps take place at a leaf. [2]
- Steps of Work:
  1. Because the heap is a complete binary tree, the node must be placed in the last leaf level at the first (leftmost) empty position and
  2. Check Heap-Order property of insert node whether it is placed to the correct position or not.
- If the new node key is larger than its parent, it is floated up the tree by exchanging it with its parent (Reheap up).

Heap Operation

Insertion

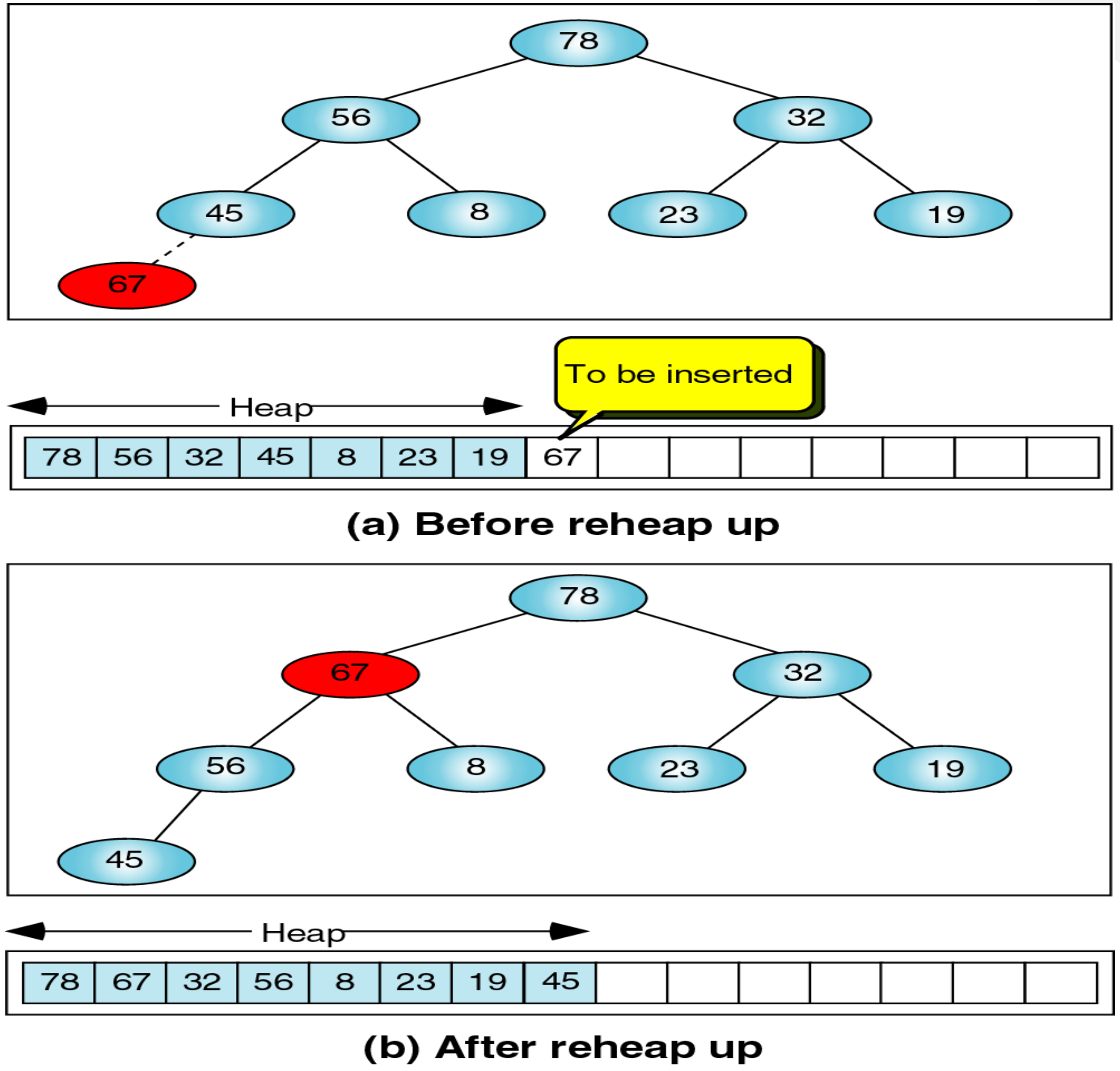
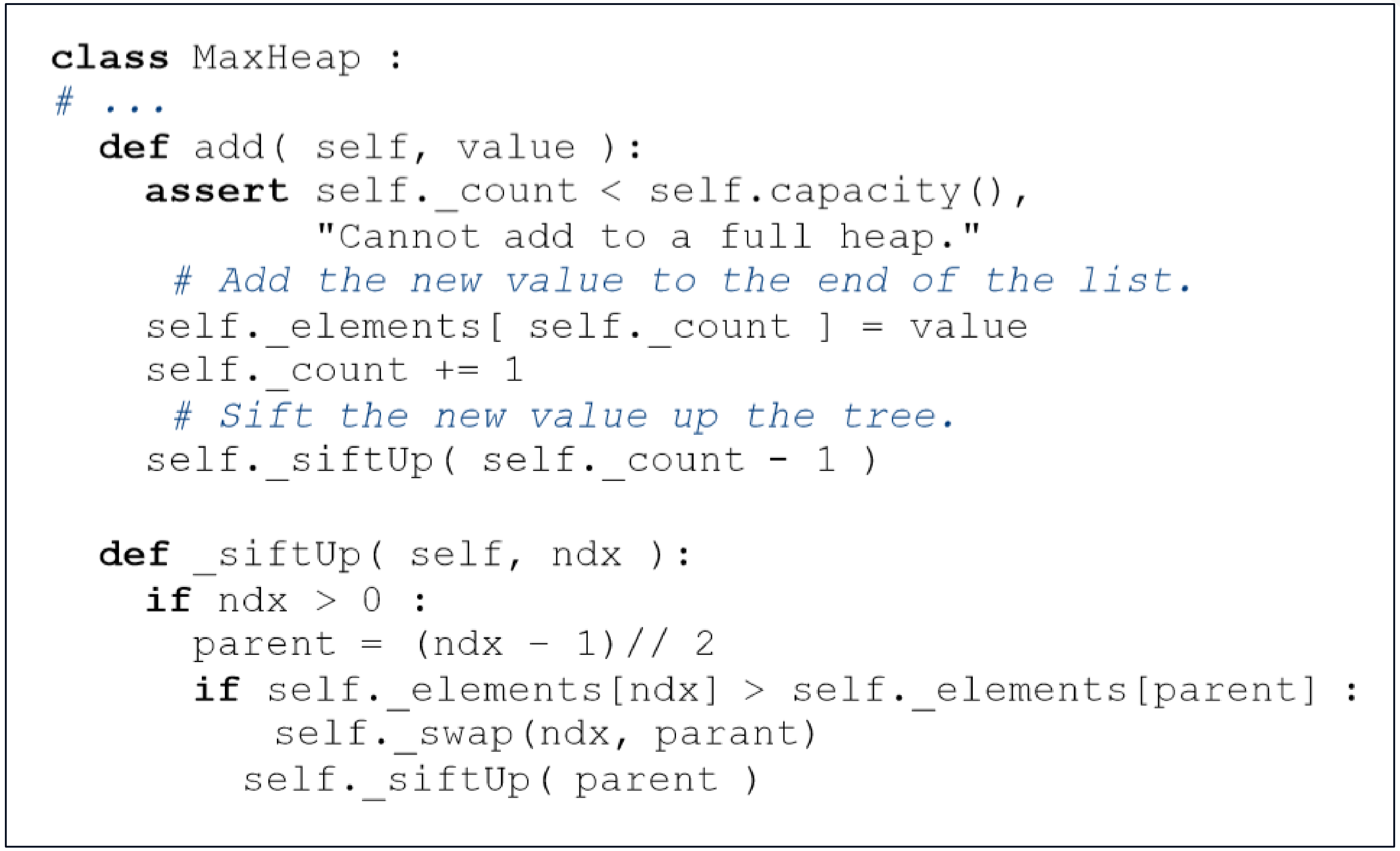


Fig 6-13 Heap insertion [3]

Heap Operation

Insertion



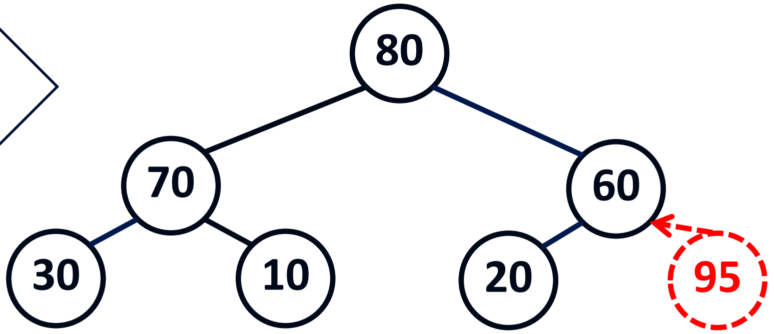
```
class MaxHeap :
# ...
  def add( self, value ):
    assert self._count < self.capacity(),
           "Cannot add to a full heap."
     # Add the new value to the end of the list.
    self._elements[ self._count ] = value
    self._count += 1
     # Sift the new value up the tree.
    self._siftUp( self._count - 1 )

  def _siftUp( self, ndx ):
    if ndx > 0 :
      parent = (ndx - 1)// 2
      if self._elements[ndx] > self._elements[parent] :
          self._swap(ndx, parant)
        self._siftUp( parent )
```

Fig 6-14 Insertion function [2]

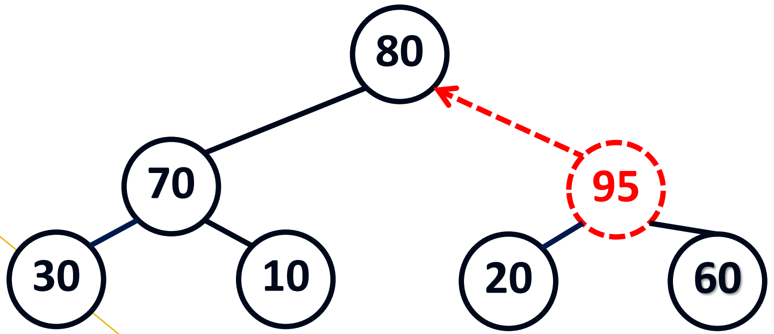
Tutorial 1: Heap Insertion Illustration

Insert (95)



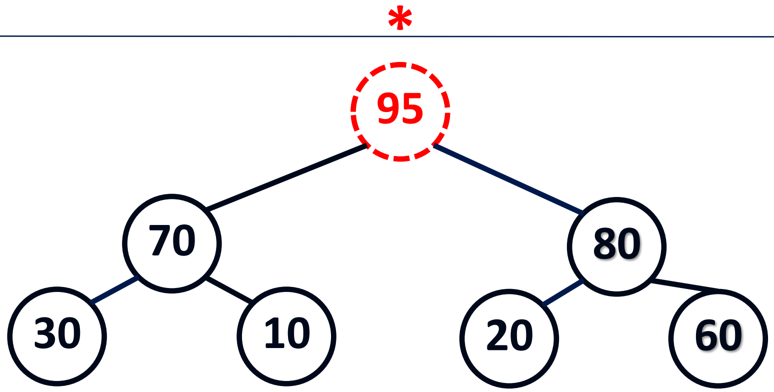
a)

1	2	3	4	5	6	7	8
80	70	60	30	10	20	95	



b)

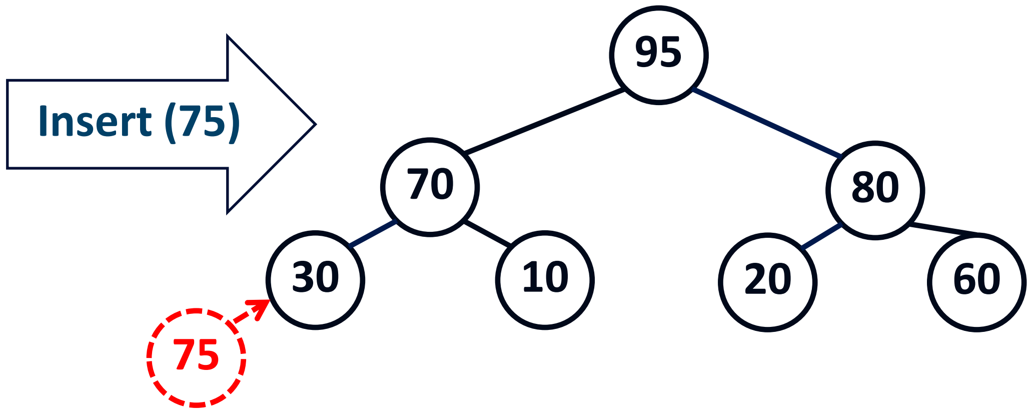
1	2	3	4	5	6	7	8
80	70	95	30	10	20	60	



c)

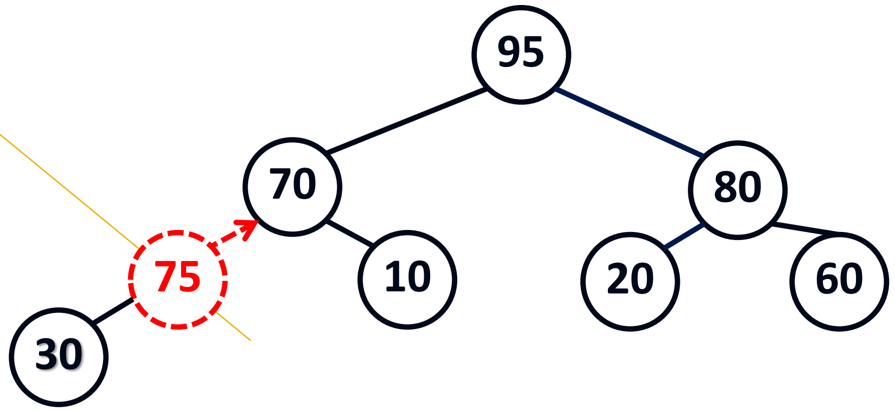
1	2	3	4	5	6	7	8
95	70	80	30	10	20	60	

Tutorial 1: Heap insertion Illustration (cont.)



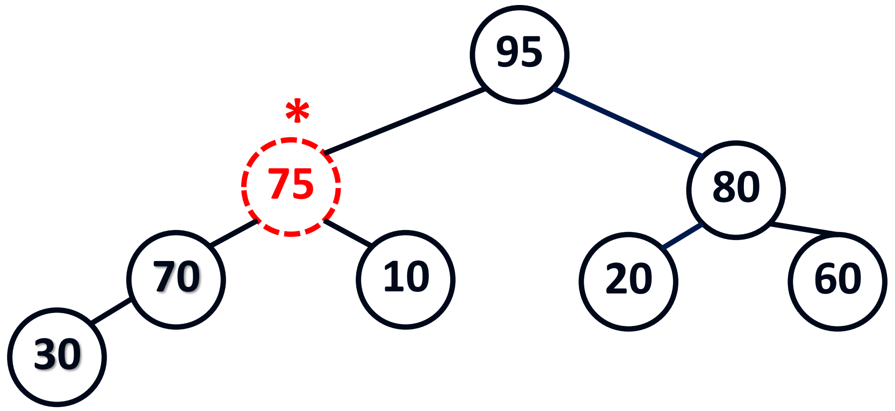
a)

1	2	3	4	5	6	7	8
95	70	80	30	10	20	60	75



b)

1	2	3	4	5	6	7	8
95	70	80	75	10	20	60	30



c)

1	2	3	4	5	6	7	8
95	70	80	75	10	20	60	30

[5]

Heap Operation

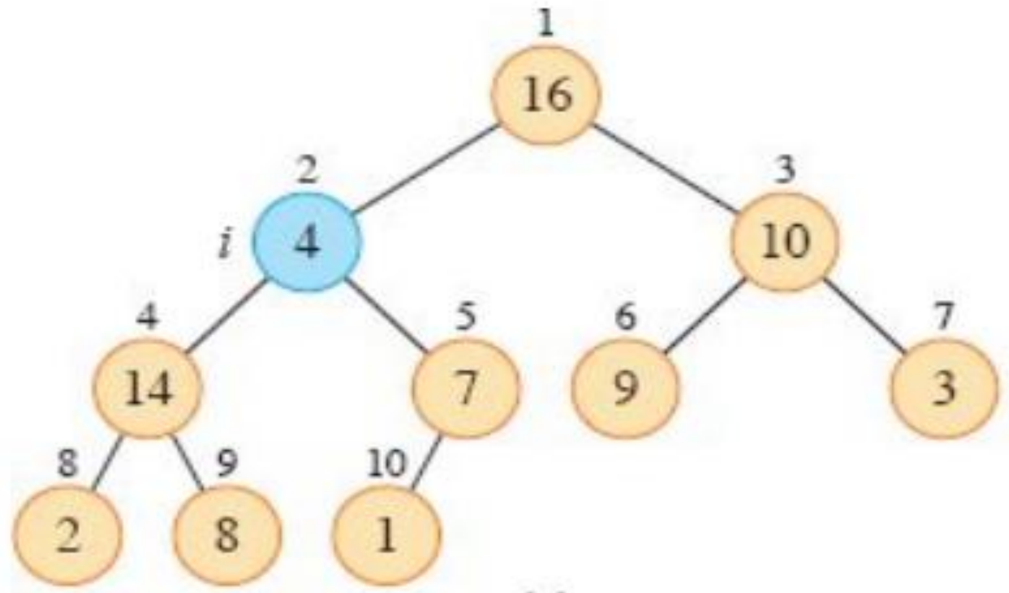
MAX-HEAPIFY

- Maintain the max heap property
- Each step determines the largest of elements between

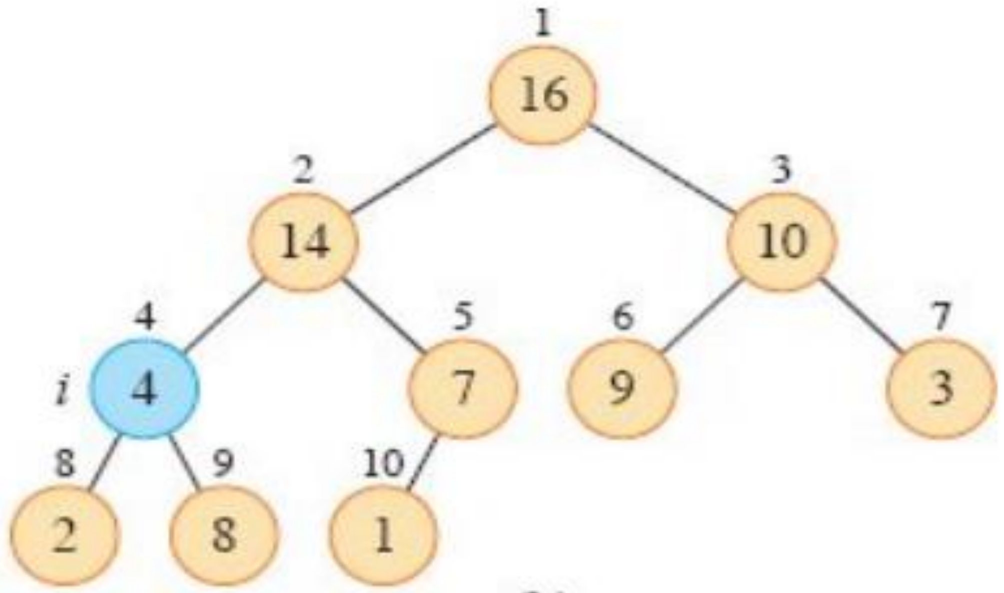
$A[i]$, $A[\mathrm{LEFT}(i)]$, and $A[\mathrm{RIGHT}(i)]$

6.2

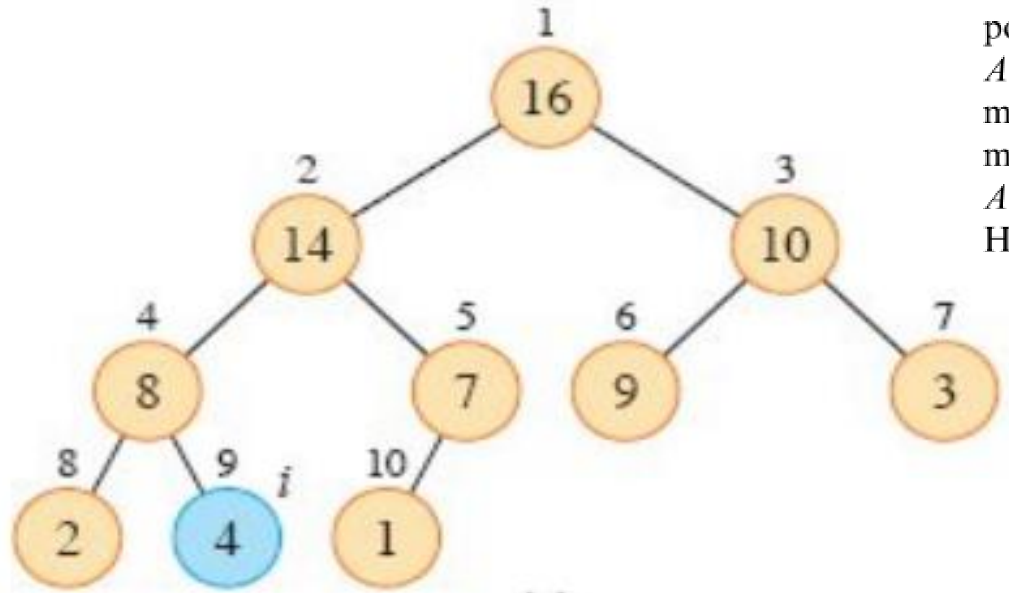
[5]



(a)



(b)



(c)

Figure 6.2 The action of MAX-HEAPIFY(*A*, 2), where *A.heap-size* = 10. The node that potentially violates the max-heap property is shown in blue. **(a)** The initial configuration, with *A*[2] at node *i* = 2 violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging *A*[2] with *A*[4], which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY(*A*, 4) now has *i* = 4. After *A*[4] and *A*[9] are swapped, as shown in **(c)**, node 4 is fixed up, and the recursive call MAX-HEAPIFY(*A*, 9) yields no further change to the data structure.

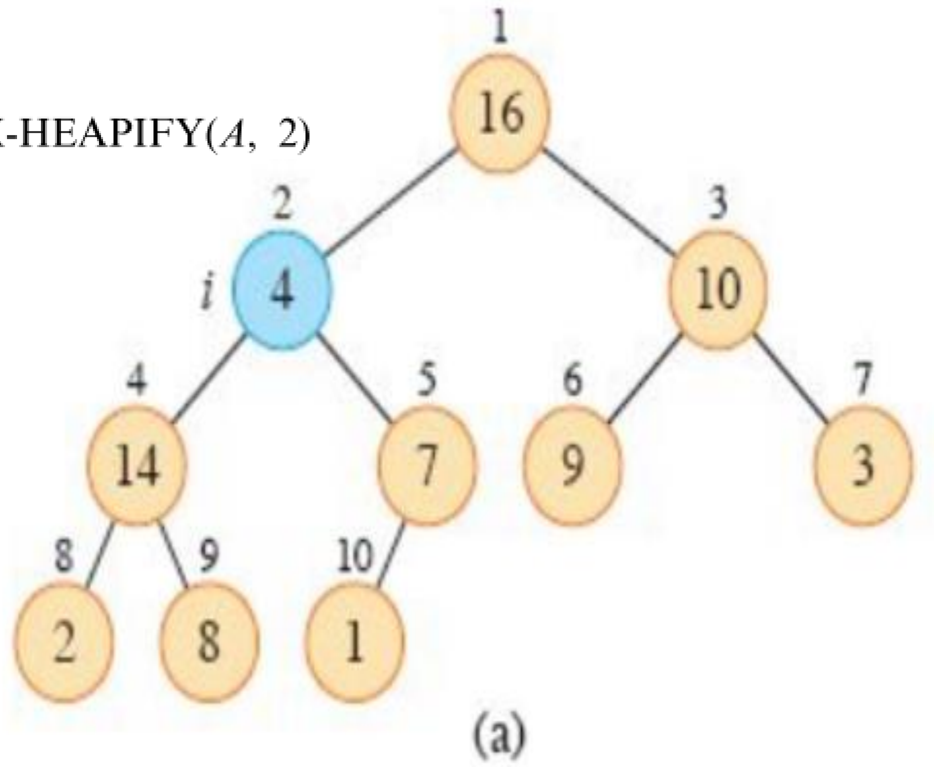
[5]

Heap Operation

MAX-HEAPIFY

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

MAX-HEAPIFY(A, 2)



[5]

Heap Operation

MAX-HEAPIFY

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

Let T(n) be the worst-case running time

$\rightarrow$ 1. $\Theta(1)$

$\rightarrow$ 2. $\Theta(1)$

$\rightarrow$ 3-5. Max [Then 3. $\Theta(1)$, Else 5. $\Theta(1)$] = $\Theta(1)$

$\rightarrow$ 6-7. Max [Then 3. $\Theta(1)$]

$\rightarrow$ = $\Theta(1)$

$\rightarrow$ 8. Max [9. $\Theta(1)$, 10. $T(2n/3) = O(\log_2 n)$]

$\rightarrow$ = $O(\log_2 n) \cong O(\text{height})$ *

The running time of MAX-HEAPIFY on a node of height h as O(height) #

Heap Operation

Build heap

- We can start with an empty array and insert elements into the array one at a time or rearrange the elements in the array to form a heap.
- Given a filled array of elements in random order, to build the heap we need to rearrange the data so that each node in the heap is greater (smaller) than its children. [3]

Heap Operation

Build heap

- This is done by dividing the array into two parts: [3]
  1. the left being a heap and
  2. the right being data to be inserted into the heap.
- Each iteration of the insertion operation uses the reheap up to insert the next element into the heap and moves the wall separating the elements one position to the right. [3]

Heap Operation

Build heap

- Steps of work: [2]
  1. we follow the parent path up the heap, swapping nodes that are out of order (If the nodes are in order, then the insertion terminates) and
  2. the next node is selected into the heap.
- This process is sometimes referred to as “heapify”. [2]

Heap Operation

Build heap

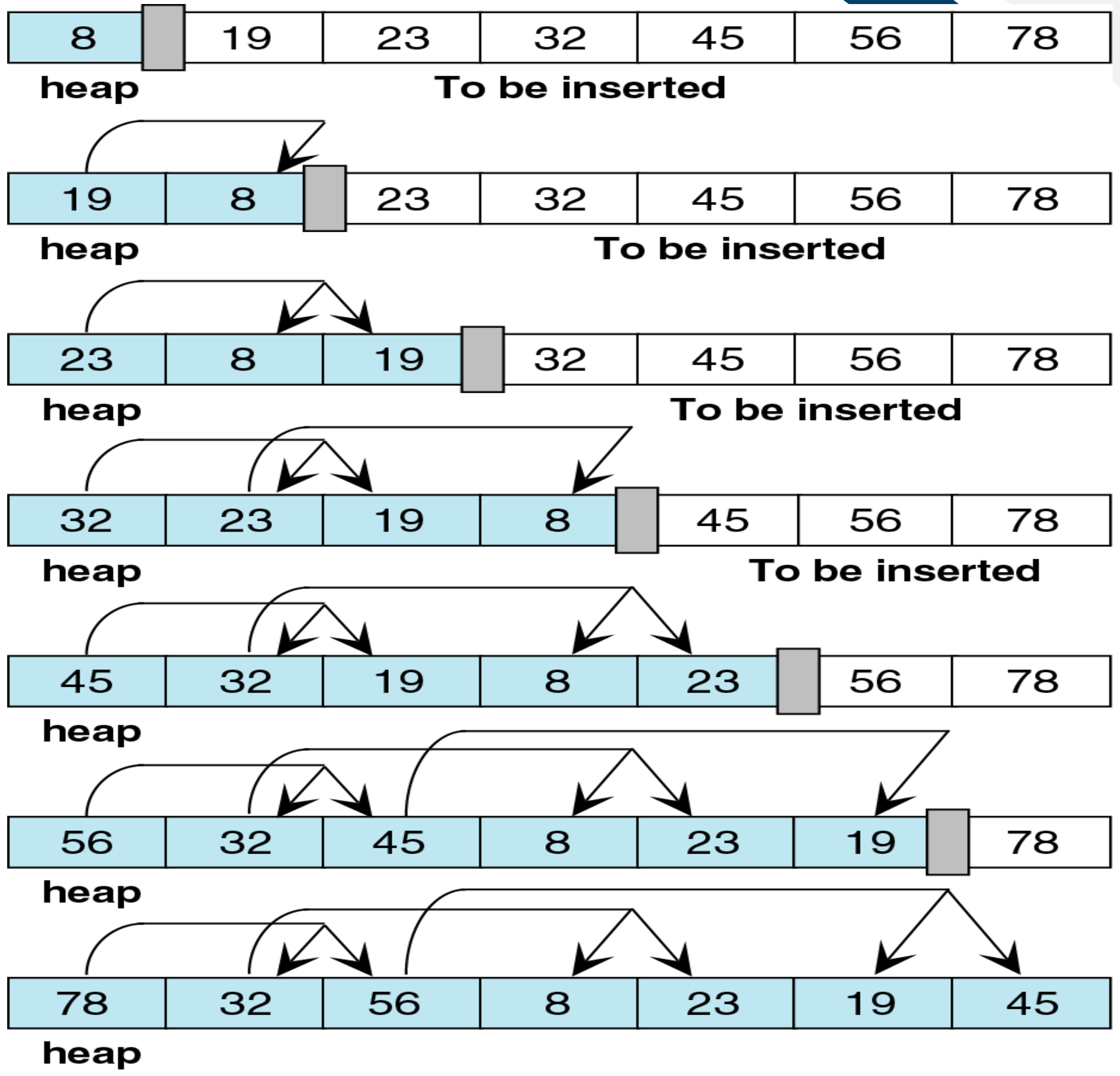
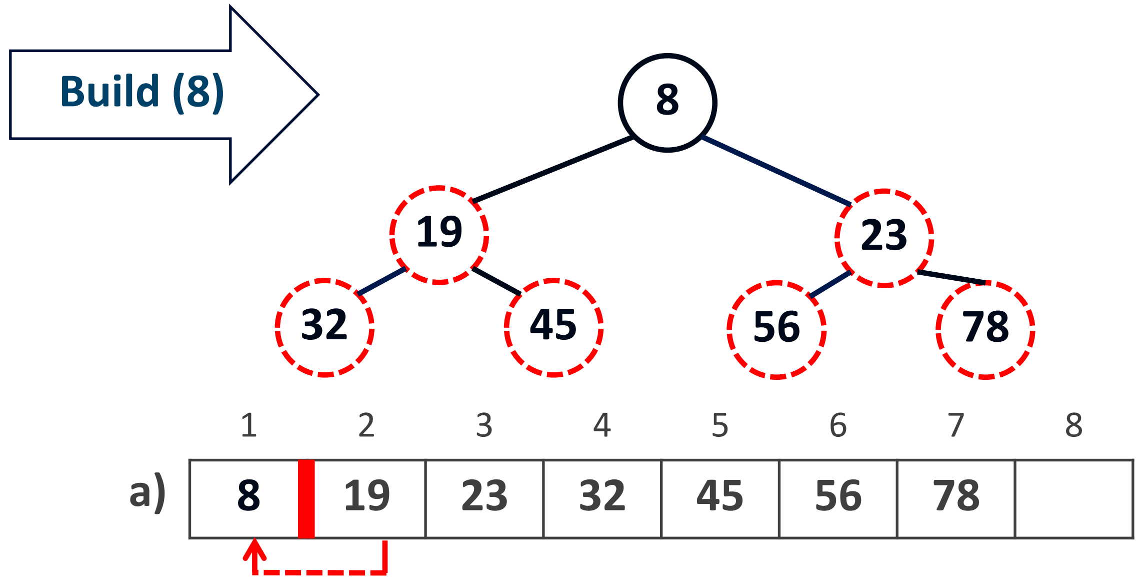


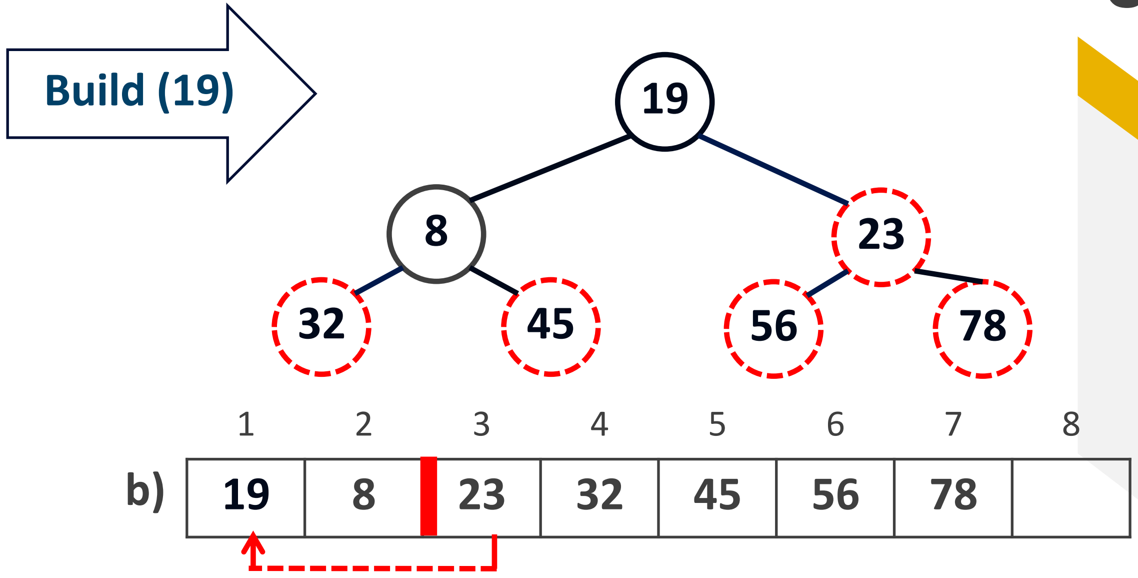
Fig 9-15 Heap insertion [3]

Tutorial 2: Build Max Heap Illustration

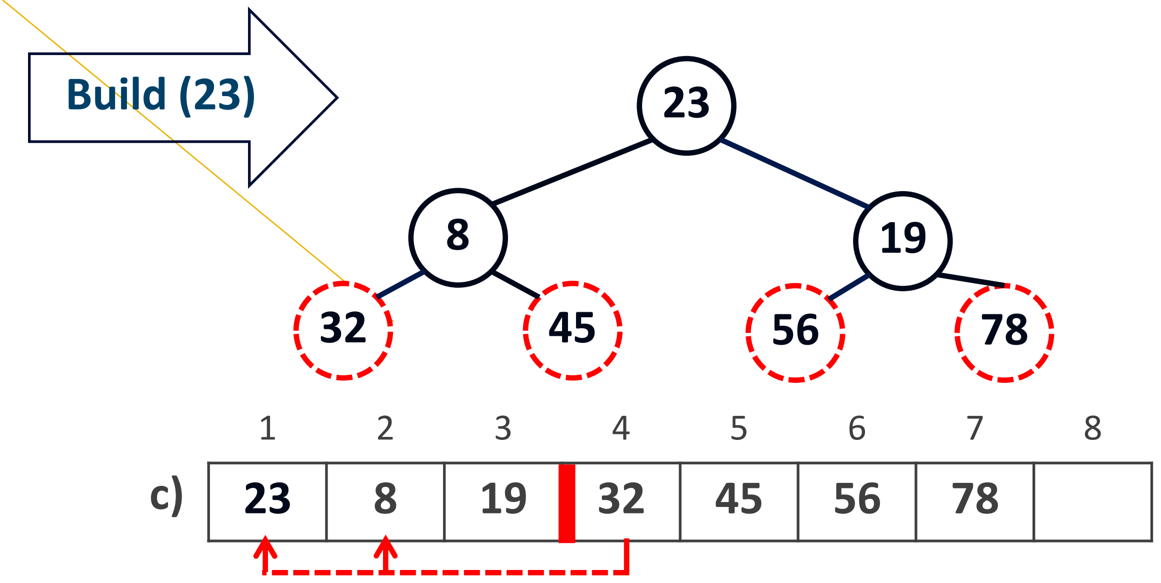
6.2



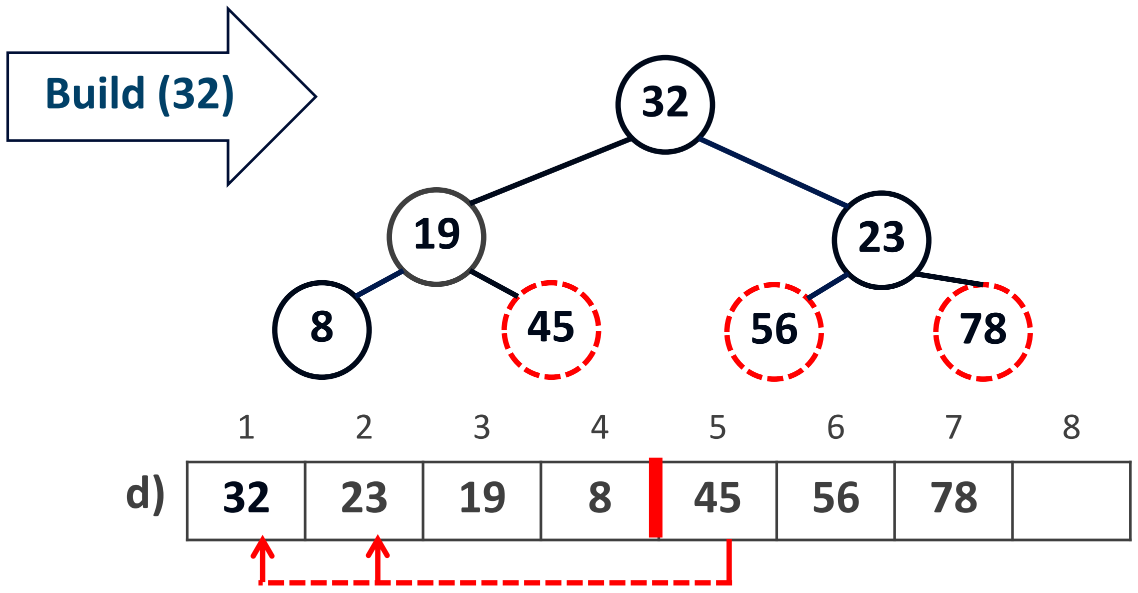
	1	2	3	4	5	6	7	8
a)	8	19	23	32	45	56	78	



	1	2	3	4	5	6	7	8
b)	19	8	23	32	45	56	78	

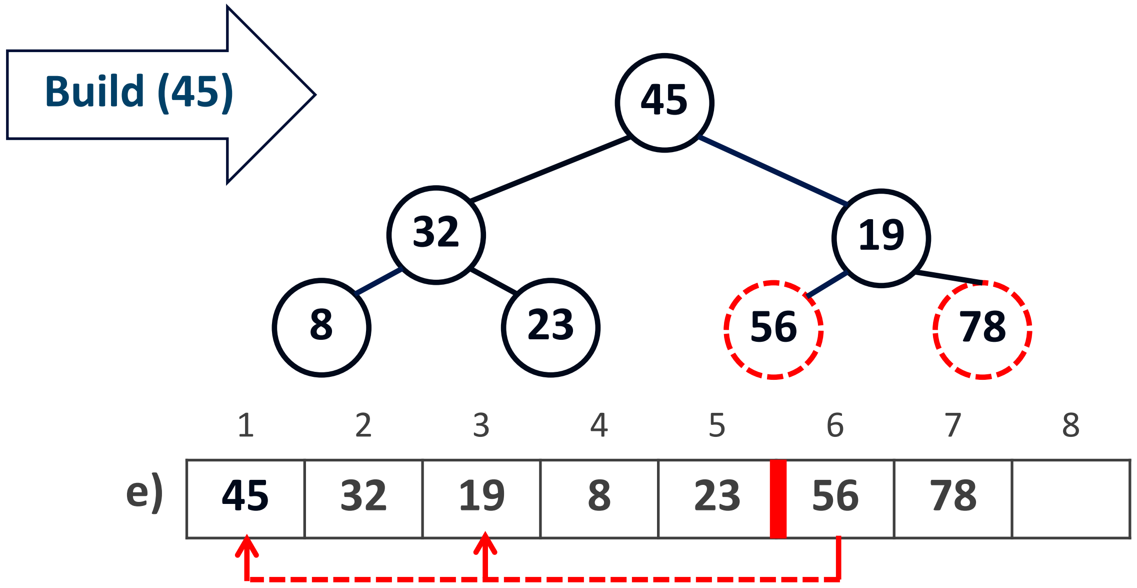


	1	2	3	4	5	6	7	8
c)	23	8	19	32	45	56	78	

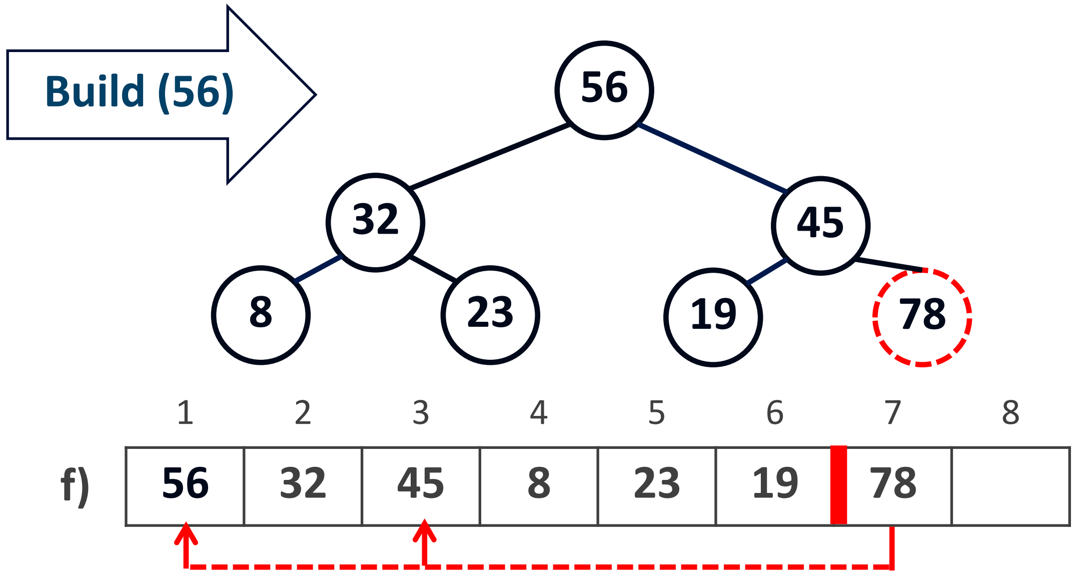


	1	2	3	4	5	6	7	8
d)	32	23	19	8	45	56	78	

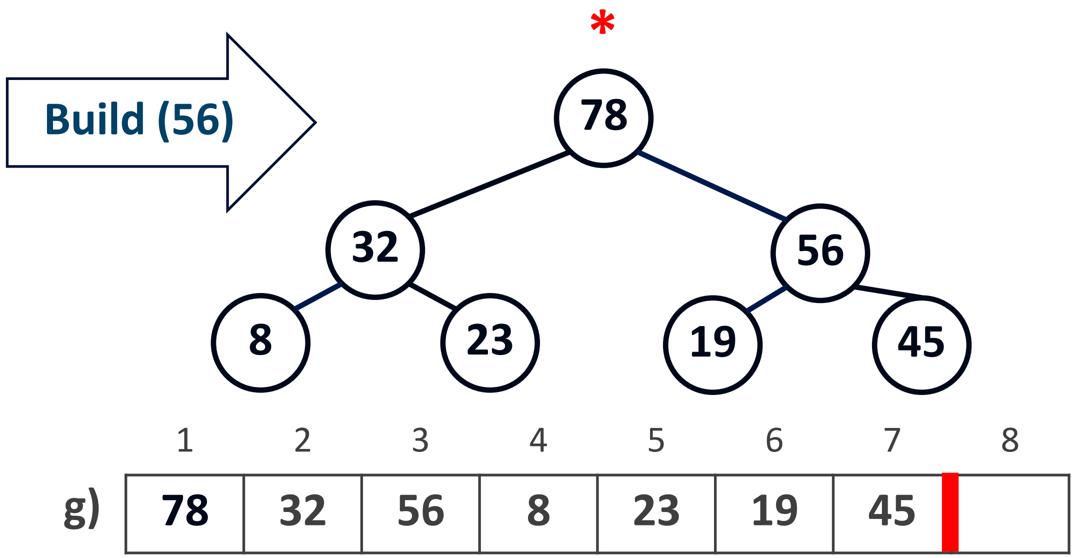
Tutorial 2: Build Max Heap Illustration (Cont.)



	1	2	3	4	5	6	7	8
e)	45	32	19	8	23	56	78	

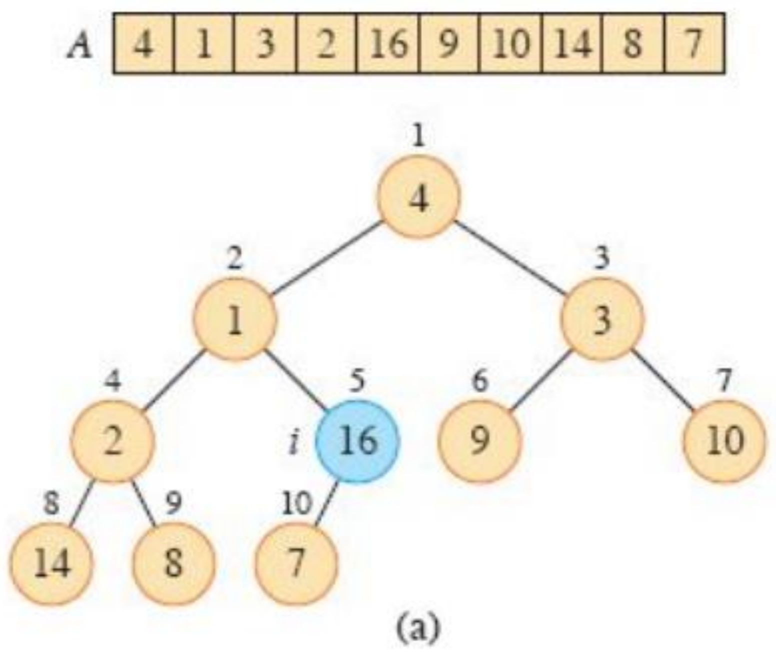


	1	2	3	4	5	6	7	8
f)	56	32	45	8	23	19	78	



	1	2	3	4	5	6	7	8
g)	78	32	56	8	23	19	45	

BUILD-MAX-HEAP [5]



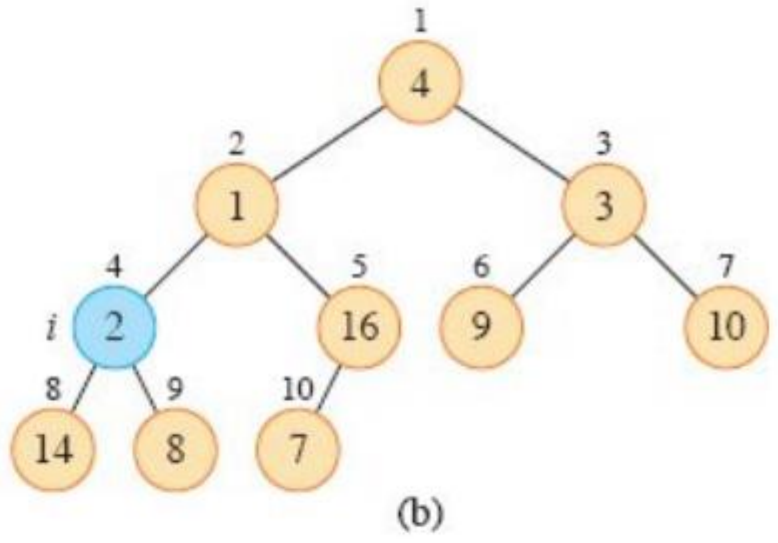
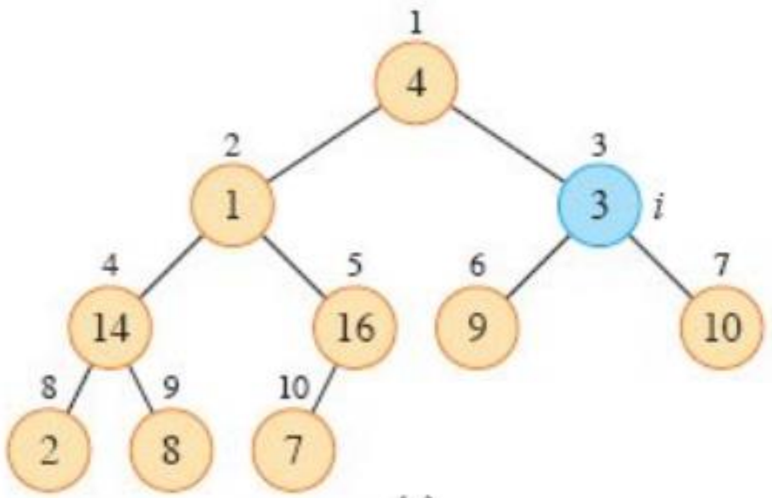


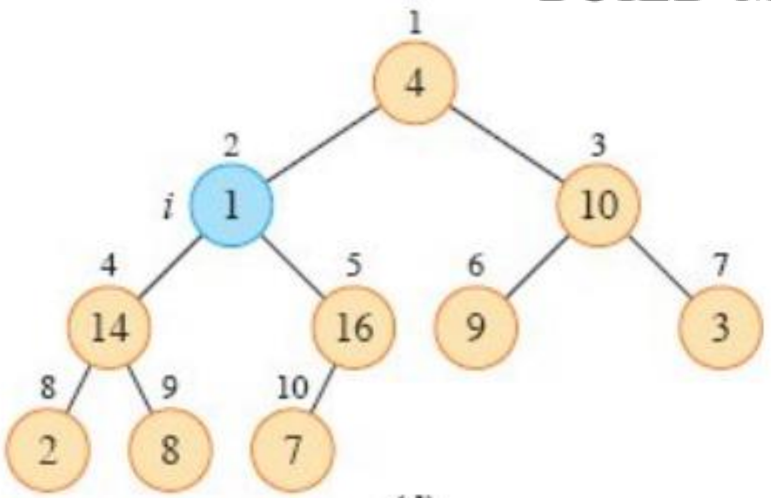
Figure 6.3 The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. The node indexed by *i* in each iteration is shown in blue. **(a)** A 10-element input array *A* and the binary tree it represents. The loop index *i* refers to node 5 before the call MAX-HEAPIFY(*A*, *i*). **(b)** The data structure that results. The loop index *i* for the next iteration refers to node 4. **(c)–(e)** Subsequent iterations of the **for** loop in BUILD-MAX-HEAP. Observe that whenever MAX-HEAPIFY is called on a node, the two subtrees of that node are both max-heaps. **(f)** The max-heap after BUILD-MAX-HEAP finishes.

BUILD-MAX-HEAP

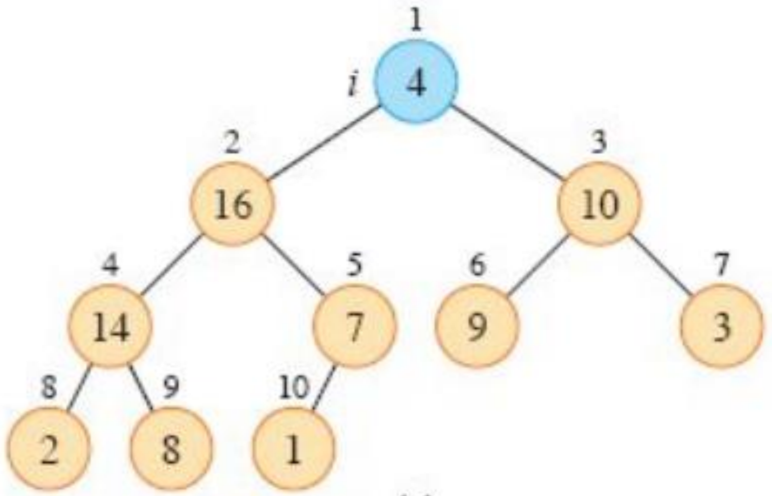
[5]



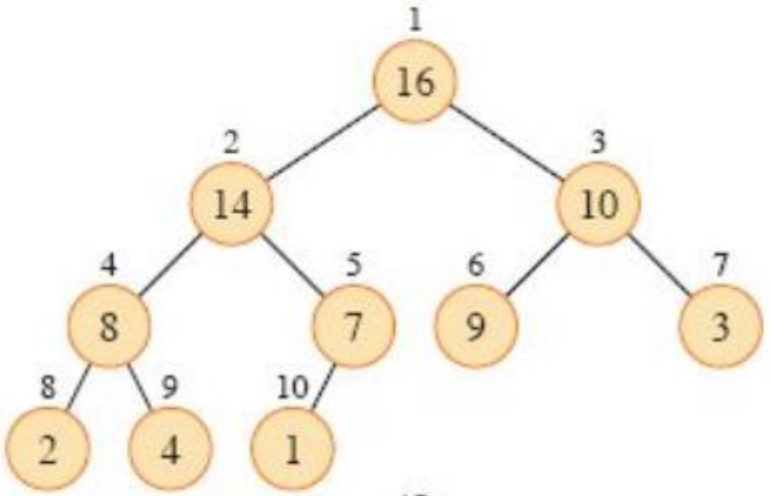
(c)



(d)



(e)



(f)

[5]

Heap Operation

BUILD-MAX-HEAP

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

Let T(n) be the worst-case running time

→1. $\Theta(1)$

→2. The loop, i=⌊n/2⌋ down to 1 → n times

→3. T(2n/3) = O(height) * $\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil n/2^{h+1} \rceil$.

The running time of BUILD-MAX-HEAP is O(n) #

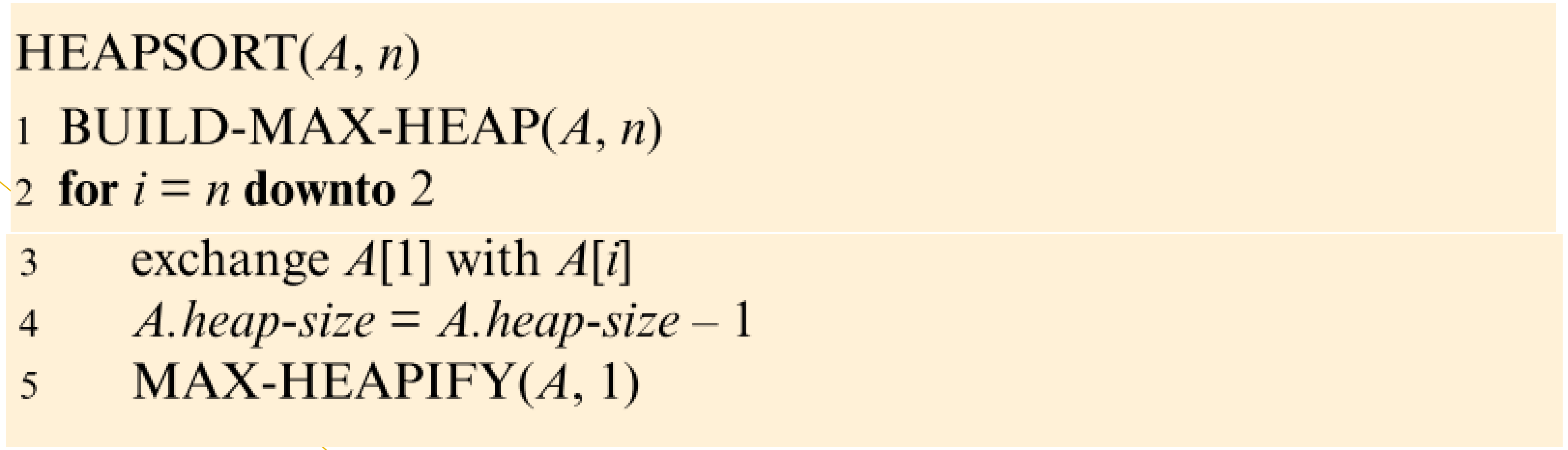
$$\begin{aligned}
\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil ch
&\le \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{n}{2^h} ch \\
&= cn \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \\
&\le cn \sum_{h=0}^{\infty} \frac{h}{2^h} \\
&\le cn \cdot \frac{1/2}{(1-1/2)^2} \\
&= O(n) .
\end{aligned}$$

[5]

Heap Operation

HEAPSORT

- Start by BUILD-MAX-HEAP and maintain max heap with MAX-HEAPIFY



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

Let T(n) be the worst-case running time

$\rightarrow$ 1. $T_1(n)=O(n)$

$\rightarrow$ 2. The loop, i=n down to 2 $\rightarrow$ n-1 times

$\rightarrow$ 3. $T_2(2n/3) = O(\text{height}) = O(\log_2 n)$

$\rightarrow \cong O(n \log_2 n)$ *

The running time of HEAPSORT is $O(n \log_2 n)$ #

6.2

HEAPSORT [5]

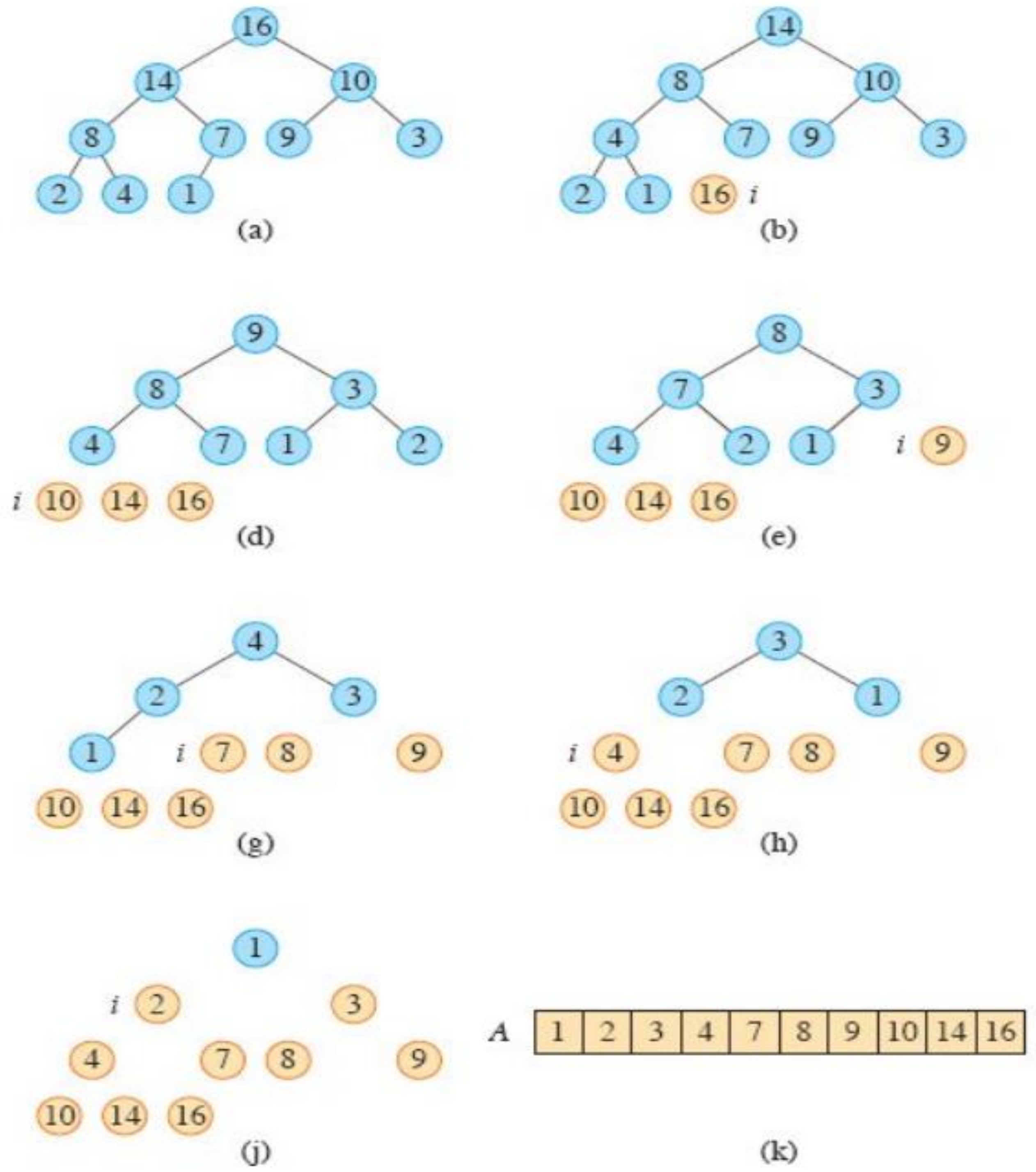


Figure 6.4 The operation of HEAPSORT. **(a)** The max-heap data structure just after BUILD-MAX-HEAP has built it in line 1. **(b)–(j)** The max-heap just after each call of MAX-HEAPIFY in line 5, showing the value of *i* at that time. Only blue nodes remain in the heap. Tan nodes contain the largest values in the array, in sorted order. **(k)** The resulting sorted array *A*.

Heap Operation

Deletion

- To reestablish the heap after deleting, we move the data in the last heap node to the root and call operation reheap down. [3]
- Node which can be deleted in heap is root node.
- Every deletion will deliver a maximum value from the structure (Max heap).

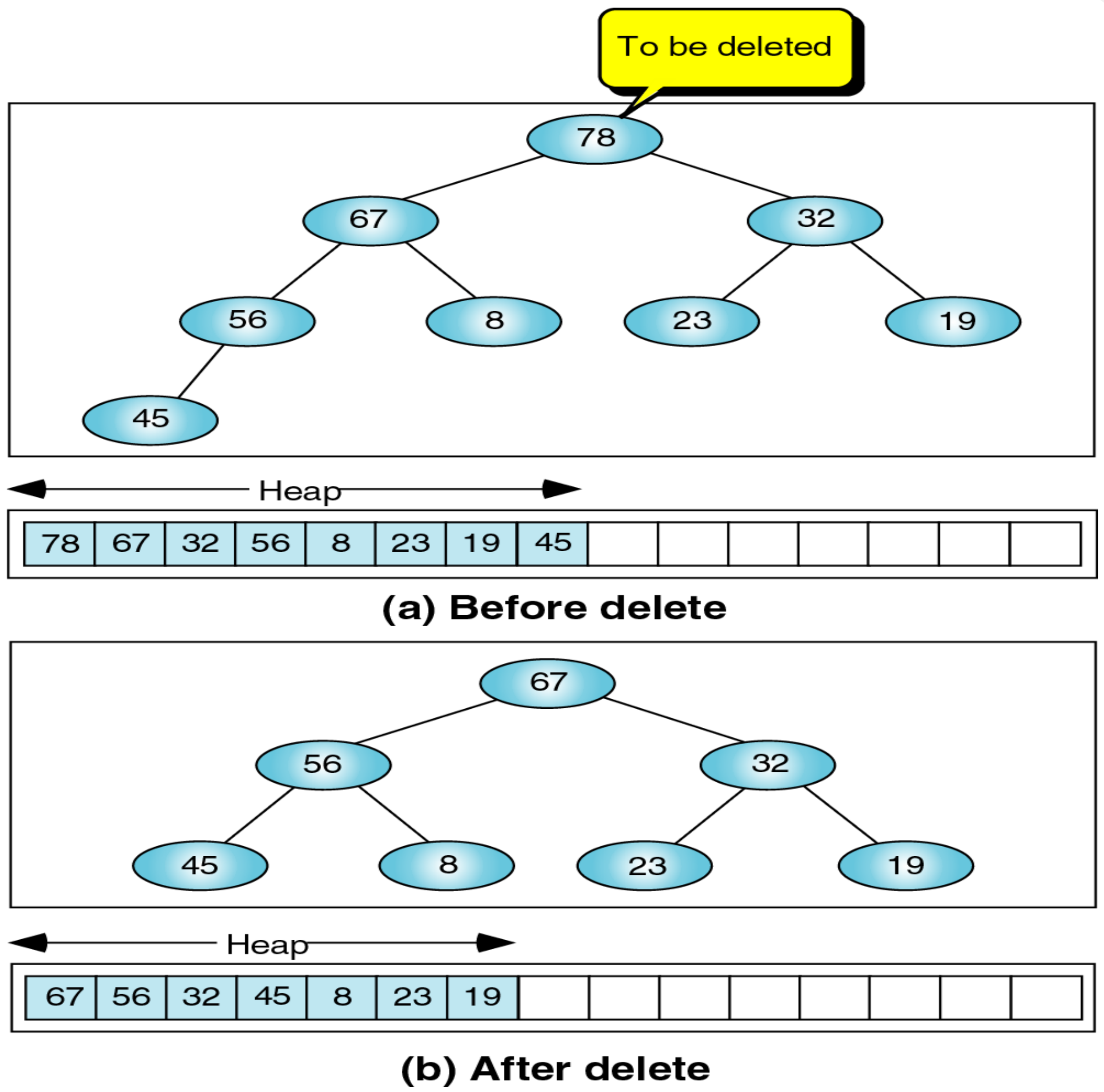
Heap Operation

Deletion

- Steps of work:
  1. Root is removed.
  2. Replace root by selecting a maximum value from its two children.
  3. Repeat until reach to leaf node and check whether it is the last element or not (if it is not, replace this position with the last node because complete binary tree shape must be maintained).

Heap Operation

Deletion



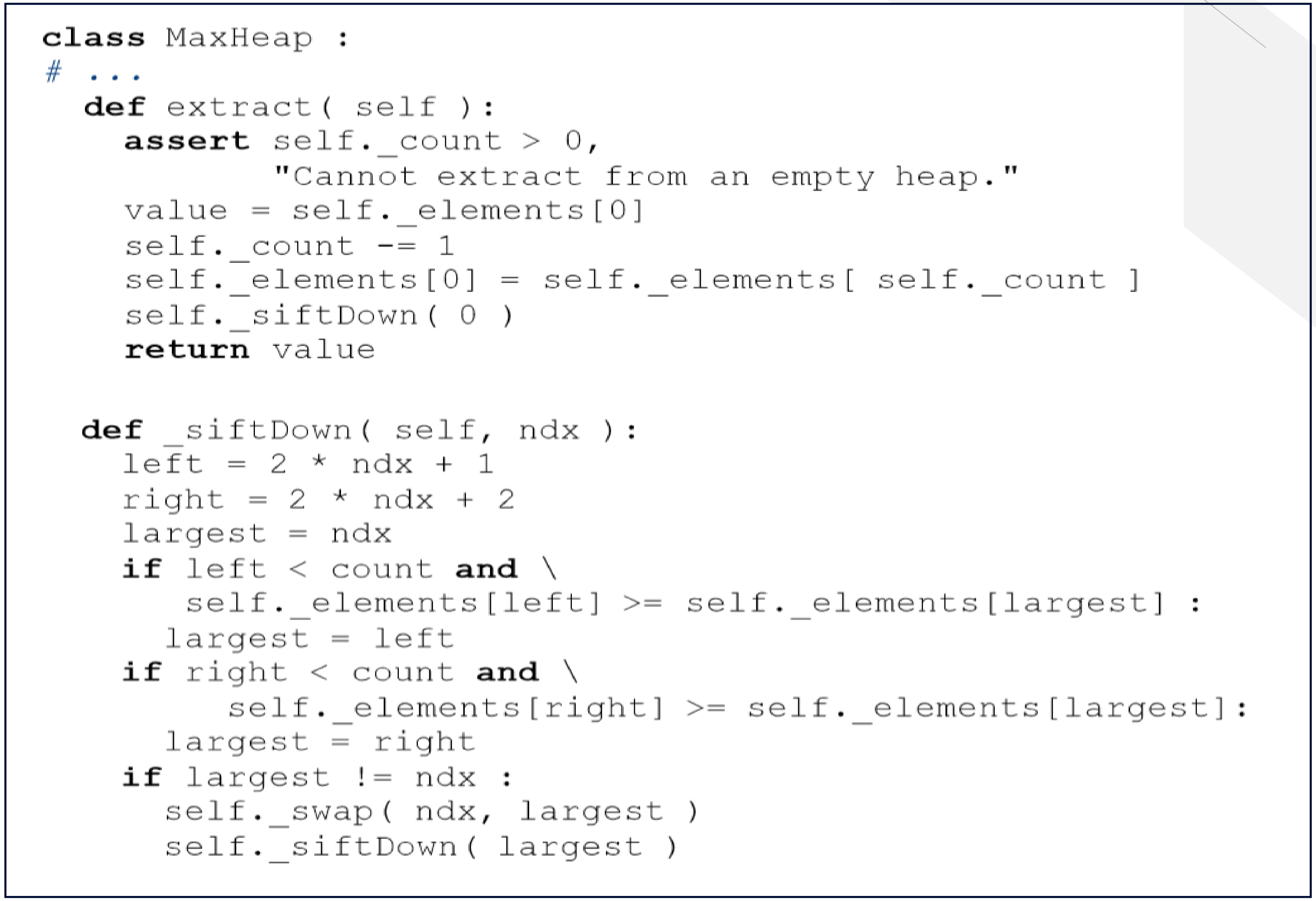
78	67	32	56	8	23	19	45							

67	56	32	45	8	23	19								

Fig 9-16 Heap deletion [3]

Heap Operation

Deletion



```
class MaxHeap :
# ...
  def extract( self ):
    assert self._count > 0,
            "Cannot extract from an empty heap."
    value = self._elements[0]
    self._count -= 1
    self._elements[0] = self._elements[ self._count ]
    self._siftDown( 0 )
    return value

  def _siftDown( self, ndx ):
    left = 2 * ndx + 1
    right = 2 * ndx + 2
    largest = ndx
    if left < count and \
       self._elements[left] >= self._elements[largest] :
      largest = left
    if right < count and \
         self._elements[right] >= self._elements[largest]:
      largest = right
    if largest != ndx :
      self._swap( ndx, largest )
      self._siftDown( largest )
```

Fig 9-17 Deletion function [2]

Tutorial 3: Heap Implementation

6.3

```
# An array-based implementation of the max-heap.
class MaxHeap :
   # Create a max-heap with maximum capacity of maxSize.
  def __init__( self, maxSize ):
    self._elements = Array( maxSize )
    self._count = 0

   # Return the number of items in the heap.
  def __len__( self ):
    return self._count

   # Return the maximum capacity of the heap.
  def capacity( self ):
    return len( self._elements )

   # Add a new value to the heap.
  def add( self, value ):
    assert self._count < self.capacity(), "Cannot add to a full heap."
     # Add the new value to the end of the list.
    self._elements[ self._count ] = value
    self._count += 1
     # Sift the new value up the tree.
    self._siftUp( self._count - 1 )

   # Extract the maximum value from the heap.
  def extract( self ):
    assert self._count > 0, "Cannot extract from an empty heap."
```

Tutorial 3: Heap Implementation (Cont.)

```
    # Save the root value and copy the last heap value to the root.
    value = self._elements[0]
    self._count -= 1
    self._elements[0] = self._elements[ self._count ]
    # Sift the root value down the tree.
    self._siftDown( 0 )

  # Sift the value at the ndx element up the tree.
  def _siftUp( self, ndx ):
    if ndx > 0 :
      parent = ndx // 2
      if self._elements[ndx] > self._elements[parent] :  # swap elements
        tmp = self._elements[ndx]
        self._elements[ndx] = self._elements[parent]
        self._elements[parent] = tmp
        self._siftUp( parent )

  # Sift the value at the ndx element down the tree.
  def _siftDown( self, ndx ):
    left = 2 * ndx + 1
    right = 2 * ndx + 2
    # Determine which node contains the larger value.
    largest = ndx
    if left < count and self._elements[left] >= self._elements[largest] :
      largest = left
    elif right < count and self._elements[right] >= self._elements[largest]:
      largest = right
    # If the largest value is not in the current node (ndx), swap it with
    # the largest value and repeat the process.
    if largest != ndx :
      swap( self._elements[ndx], self._elements[largest] )
```

Heap Applications

- Implement the priority queue
- Support heap sort

Heap Applications

Priority Queue

- A priority queue is a queue whose data item contain:
  1. Value and
  2. Priority
- Enqueue and Dequeue will consider items' priority as a core whose tasks are more complicated in array structure.
- If we implement it in a minimum binary heap and considering the items' priority instead of data, it can be done easier.

Heap Applications

Priority Queue

- Store both the priority and queue elements of queue array to the heap node.
- The heap ordering is based on the priority when the highest priority is the minimum value.

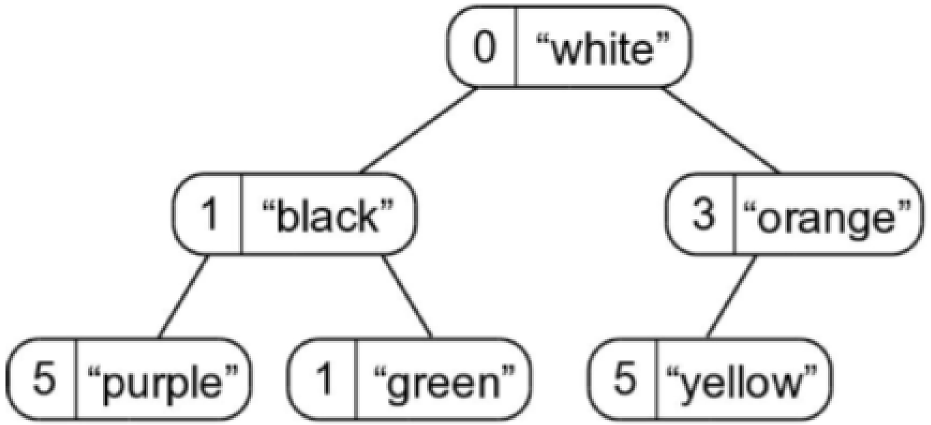
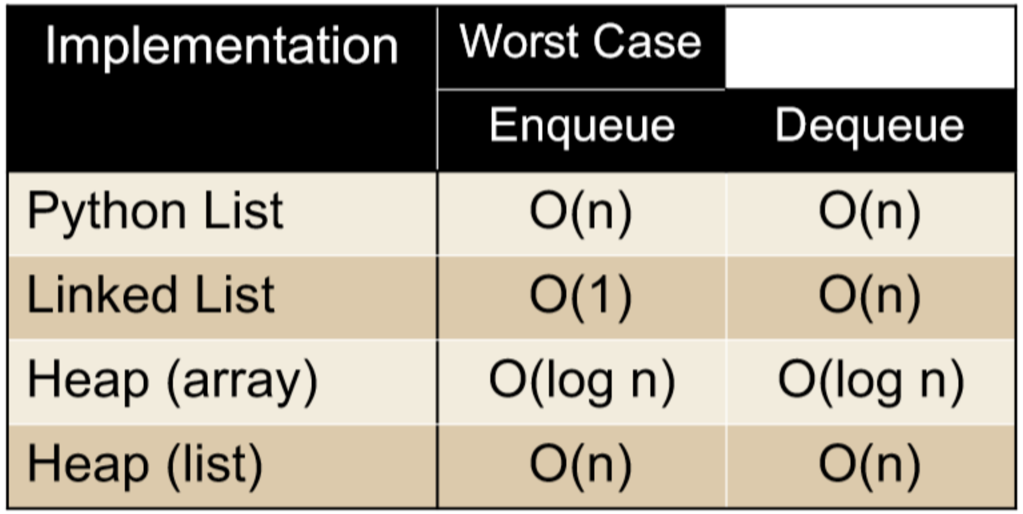


Fig 6-18 Priority queue implemented by Min heap [2]

Heap Applications

Priority Queue



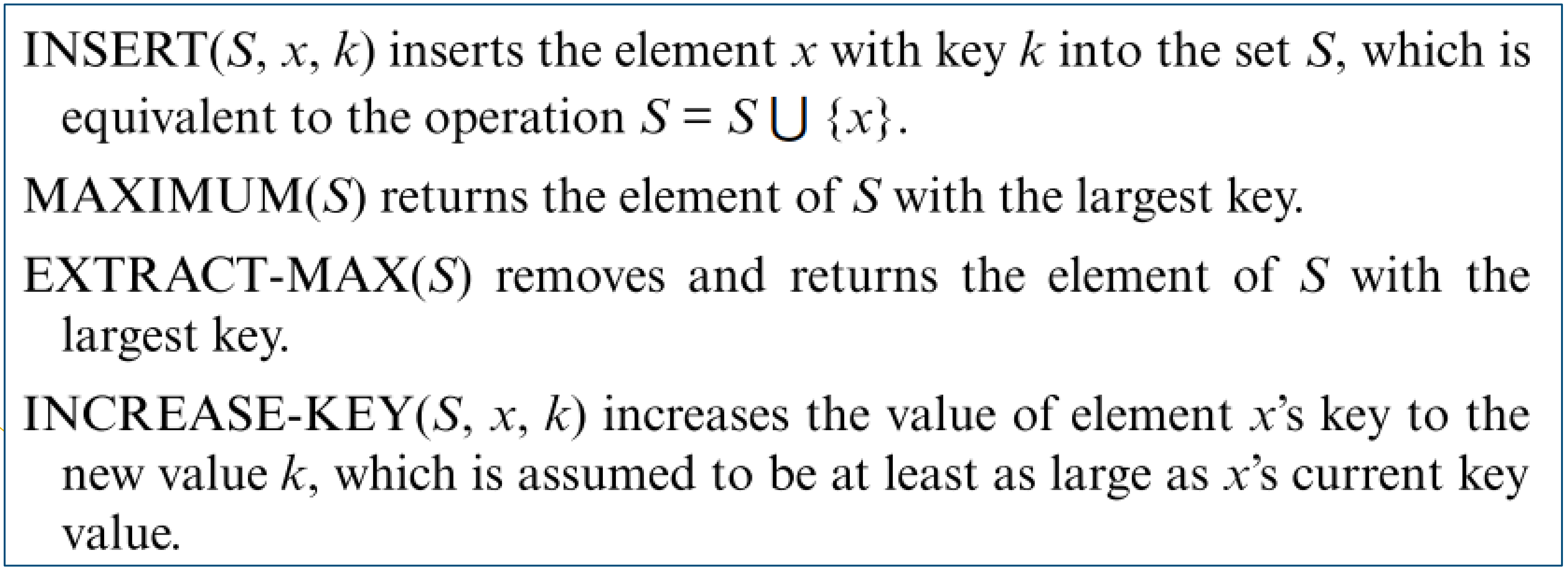
Implementation	Worst Case	
	Enqueue	Dequeue
Python List	O(n)	O(n)
Linked List	O(1)	O(n)
Heap (array)	O(log n)	O(log n)
Heap (list)	O(n)	O(n)

Fig 6-19 Comparative performance of implementing priority queue [2]

Heap Applications

Priority Queue

- A max-priority queue supports the following operations:



INSERT(S, x, k) inserts the element x with key k into the set S, which is equivalent to the operation $S = S \cup \{x\}$.

MAXIMUM(S) returns the element of S with the largest key.

EXTRACT-MAX(S) removes and returns the element of S with the largest key.

INCREASE-KEY(S, x, k) increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value.

Heap Applications

Priority Queue

- It helps schedule jobs on a computer shared among multiple users.
  - It keep tracks of jobs to be performed and their relative priorities.
    - When a job is finished or interrupted, the scheduler selects the highest-priority job from among those pending by calling EXTRACT-MAX.
    - The scheduler can add a new job to the queue at any time by calling INSERT.

Heap Applications

Priority Queue

```
MAX-HEAP-MAXIMUM($A$)
if $A.heap\text{-}size < 1$
    error "heap underflow"
return $A[1]$

MAX-HEAP-EXTRACT-MAX($A$)
$max$ = MAX-HEAP-MAXIMUM($A$)
$A[1] = A[A.heap\text{-}size]$
$A.heap\text{-}size = A.heap\text{-}size - 1$
MAX-HEAP-IFY($A$, 1)
return $max$
```

The running time of MAX-HEAP-MAXIMUMT is $\Theta(1)$ #

The running time of MAX-HEAP-MAXIMUMT is $O(\log_2 n)$ #

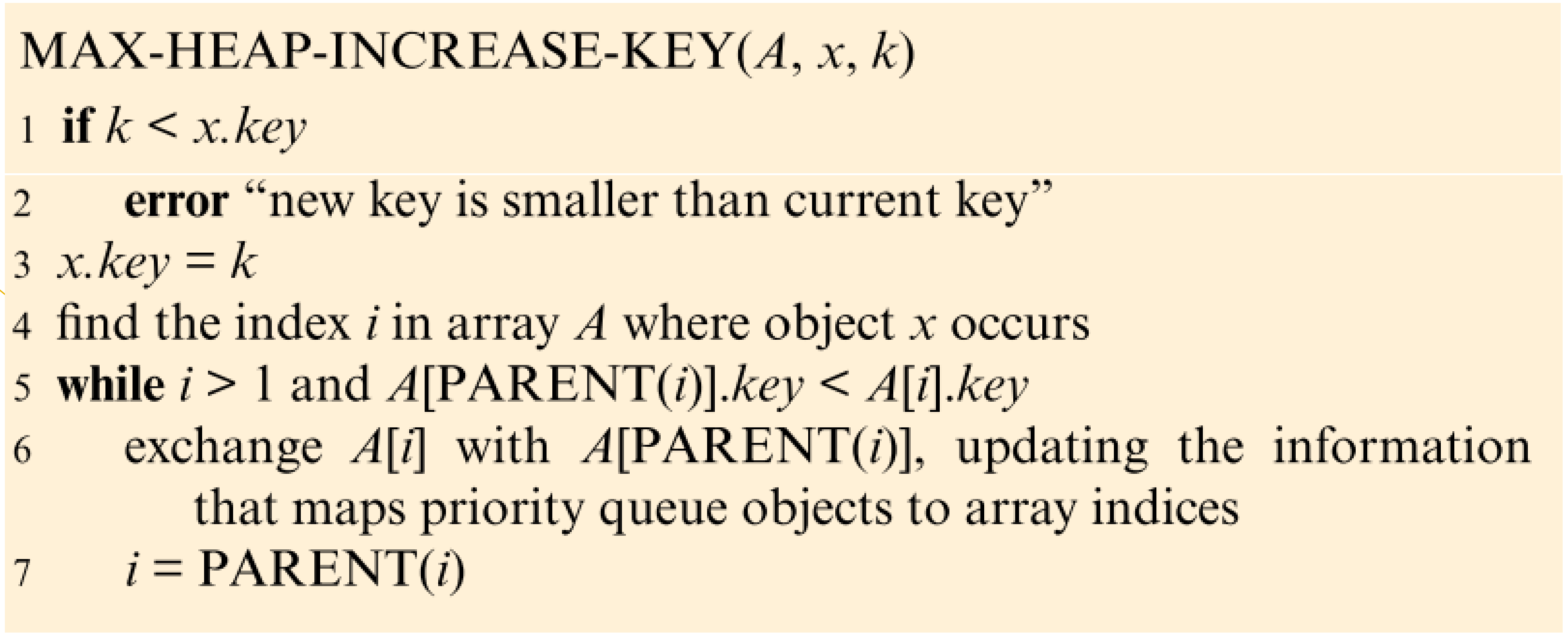
Heap Applications

Priority Queue

- As an alternative to incorporating handles in application objects, you can store within the priority queue a mapping from application objects to array indices in the heap

Heap Applications

Priority Queue

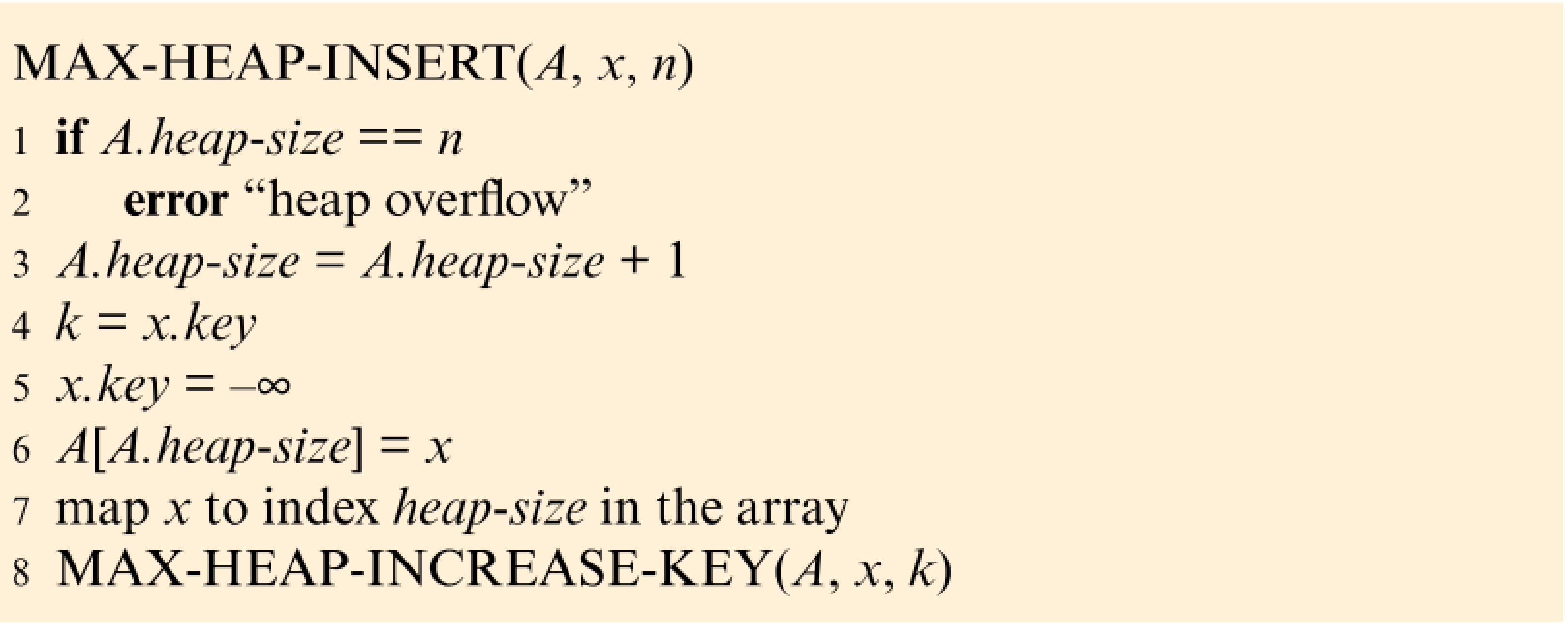


```
MAX-HEAP-INCREASE-KEY($A$, $x$, $k$)
if $k < x.key$
    error "new key is smaller than current key"
$x.key = k$
find the index $i$ in array $A$ where object $x$ occurs
while $i > 1$ and $A[\text{PARENT}(i)].key < A[i].key$
    exchange $A[i]$ with $A[\text{PARENT}(i)]$, updating the information
        that maps priority queue objects to array indices
    $i = \text{PARENT}(i)$
```

The running time of MAX-HEAP-INCREASE-KEY is $O(\log_2 n)$ #

Heap Applications

Priority Queue



```
MAX-HEAP-INSERT($A, x, n$)
if $A.heap\text{-}size == n$
    error "heap overflow"
$A.heap\text{-}size = A.heap\text{-}size + 1$
$k = x.key$
$x.key = -\infty$
$A[A.heap\text{-}size] = x$
map $x$ to index $heap\text{-}size$ in the array
MAX-HEAP-INCREASE-KEY($A, x, k$)
```

The running time of MAX-HEAP-INSERT is $O(\log_2 n)$ #

References

Texts | Integrated Development Environment (IDE)

[1] Data Structures and Algorithms in Python, Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser, Willy & Sons Inc., 2013.

[2] Data Structures: A Pseudocode Approach with C++, Richard F. Gilberg and Behrouz A. Forouzan, Brooks/Cole, 2001.

[3] Data Structures and Algorithms Using Python, Rance D. Necaise, John Winley & Sons, Inc., 2011.

[4] https://colab.research.google.com

[5] Introduction to Algorithms, H., author., Leiserson, Charles Eric, Rivest, Ronald L., Stein, Clifford, The MIT Press, Forth Edition 2022.