# Data Structure and Algorithms [CO2003]

## Chapter 8 - Heap

Lecturer: Duc Dung Nguyen, PhD.
Contact: nddung@hcmut.edu.vn

Faculty of Computer Science and Engineering
Hochiminh city University of Technology

# Contents

# Outcomes

- **L.O.4.1** - List some applications of Heap.
- **L.O.4.2** - Depict heap structure and relate it to array.
- **L.O.4.3** - List necessary methods supplied for heap structure, and describe them using pseudocode.
- **L.O.4.4** - Depict the working steps of methods that maintain the characteristics of heap structure for the cases of adding/removing elements to/from heap.

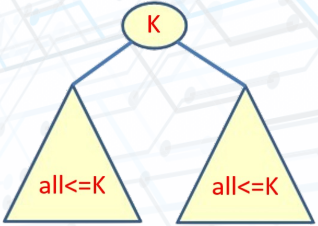
# Outcomes

- **L.O.4.5** - Implement heap using C/C++.
- **L.O.4.6** - Analyze the complexity and develop experiment (program) to evaluate methods supplied for heap structures.
- **L.O.8.4** - Develop recursive implementations for methods supplied for the following structures: list, tree, heap, searching, and graphs.
- **L.O.1.2** - Analyze algorithms and use Big-O notation to characterize the computational complexity of algorithms composed by using the following control structures: sequence, branching, and iteration (not recursion).

# Heap Definition

**Definition**
A heap (max-heap) is a binary tree structure with the following properties:

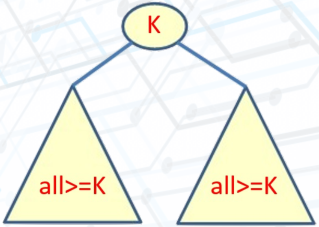
1. The tree is complete or nearly complete.
2. The key value of each node is greater than or equal to the key value in each of its descendents.




(Source: Data Structures - A Pseudocode Approach with C++)

**Definition**
A min-heap is a binary tree structure with the following properties:

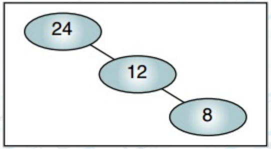
1. The tree is complete or nearly complete.
2. The key value of each node is less than or equal to the key value in each of its descendents.



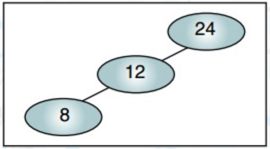

(Source: Data Structures - A Pseudocode Approach with C++)

# Heap Structure

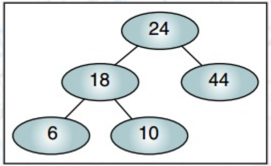
# Heap trees



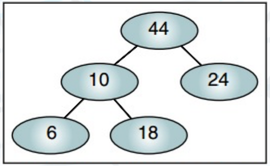

**(a) Not nearly complete (rule 1)**




**(b) Not nearly complete (rule 1)**




**(c) Root not largest (rule 2)**



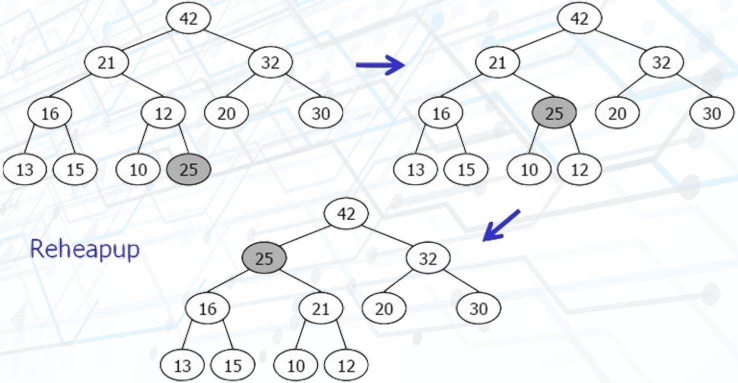

**(d) Subtree 10 not a heap (rule 2)**

(Source: Data Structures - A Pseudocode Approach with C++)

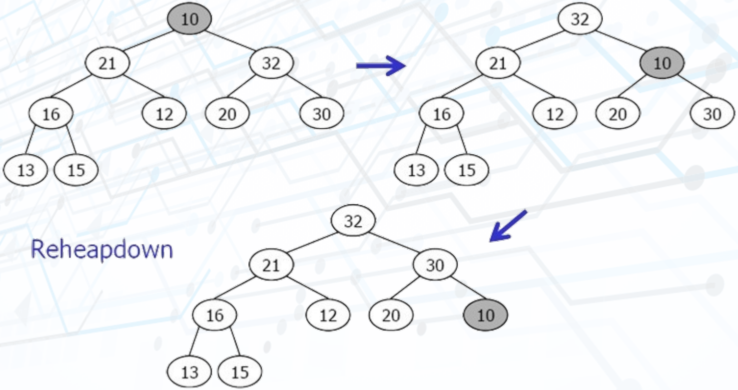
# Basic Heap Algorithms

# ReheapUp

The reheapUp operation repairs a "broken" heap by floating the last element up the tree until it is in its correct location in the heap.



The reheapDown operation repairs a "broken" heap by pushing the root down the tree until it is in its correct location in the heap.

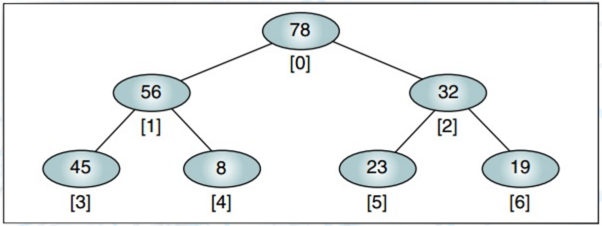


# Heap Data Structure

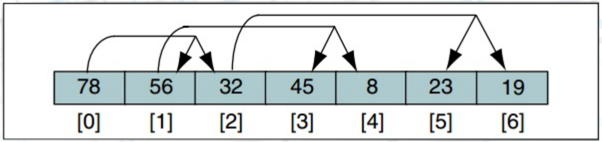
# Properties of Heaps

- A complete or nearly complete binary tree.
- If the height is $h$, the number of nodes $N$ is between $2^{h-1}$ and $2^h - 1$.
- Complete tree: $N = 2^h - 1$ when last level is full.
- Nearly complete: All nodes in the last level are on the left.

$\longrightarrow$ Heap can be represented in an array.




(a) Heap in its logical form




(b) Heap in an array

(Source: Data Structures - A Pseudocode Approach with C++)

The relationship between a node and its children is fixed and can be calculated:

1. For a node located at index $i$, its children are found at
   - Left child: $2i + 1$
   - Right child: $2i + 2$
2. The parent of a node located at index $i$ is located at $\lfloor (i-1)/2 \rfloor$.
3. Given the index for a left child, $j$, its right sibling, if any, is found at $j+1$. Conversely, given the index for a right child, $k$, its left sibling, which must exist, is found at $k-1$.
4. Given the size, $N$, of a complete heap, the location of the first leaf is $\lfloor N/2 \rfloor$.
5. Given the location of the first leaf element, the location of the last nonleaf element is 1 less.

# Heap Algorithms

**Algorithm** reheapUp(ref heap <array>, val position <integer>)
Reestablishes heap by moving data in `position` up to its correct location.

**Pre:** All data in the heap above this `position` satisfy key value order of a heap, except the data in `position`

**Post:** Data in `position` has been moved up to its correct location.

# ReheapUp Algorithm

```
if position > 0 then
    parent = (position-1)/2
    if heap[position].key > heap[parent].key then
        swap(position, parent)
        reheapUp(heap, parent)
    end
end
return
End reheapUp
```

# ReheapDown Algorithm

**Algorithm** reheapDown(ref heap <array>, val position <integer>, val lastPosition <integer>)

Reestablishes heap by moving data in `position` down to its correct location.

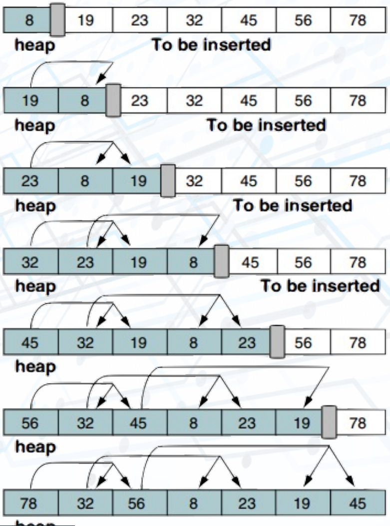
**Pre:** All data in the subtree of `position` satisfy key value order of a heap, except the data in `position`

`lastPosition` is an index to the last element in heap

**Post:** Data in `position` has been moved down to its correct location.

```
leftChild = position * 2 + 1
rightChild = position * 2 + 2
if leftChild <= lastPosition then
    if (rightChild <= lastPosition) AND (heap[rightChild].key > heap[leftChild].key then
        largeChild = rightChild
    else
        largeChild = leftChild
    end
    if heap[largeChild].key > heap[position].key then
        swap(largeChild, position)
        reheapDown(heap, largeChild, lastPosition)
    end
end
return
End reheapDown
```

- 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 than its children.
- We begin by dividing the array into two parts, the left being a heap and the right being data to be inserted into the heap. Note the "wall" between the first and second parts.
- At the beginning the root (the first node) is the only node in the heap and the rest of the array are data to be inserted.
- Each iteration of the insertion algorithm uses reheap up to insert the next element into the heap and moves the wall separating the elements one position to the right.


8
19
23
32
45
56
78
heap
To be inserted
19
8
23
32
45
56
78
heap
To be inserted
23
8
19
32
45
56
78
heap
To be inserted
32
23
19
8
45
56
78
heap
To be inserted
45
32
19
8
23
56
78
heap
56
32
45
8
23
19
78
heap
78
32
56
8
23
19
45

**Algorithm** buildHeap(ref heap <array>, val size <integer>)
Given an array, rearrange data so that they form a heap.

**Pre:** `heap` is array containing data in nonheap order
`size` is number of elements in array

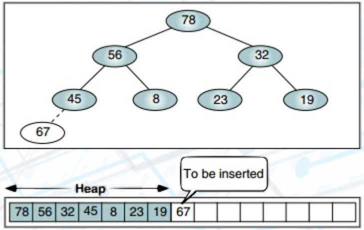
**Post:** array is now a heap.

walker = 1
**while** *walker* < *size* **do**
    reheapUp(heap, walker)
    walker = walker + 1
**end**
**End** buildHeap

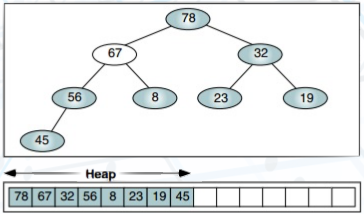
# Insert a Node into a Heap

- To insert a node, we need to locate the first empty leaf in the array.
- We find it immediately after the last node in the tree, which is given as a parameter.
- To insert a node, we move the new data to the first empty leaf and reheap up.

# Insert a Node into a Heap




(a) Before reheap up




(b) After reheap up

(Source: Data Structures - A Pseudocode Approach with C++)

**Algorithm** insertHeap(ref heap <array>, ref last <integer>, val data <dataType>)
Inserts `data` into heap.

**Pre:** `heap` is a valid heap structure
`last` is reference parameter to last node in heap
`data` contains data to be inserted

**Post:** `data` have been inserted into heap.

**Return** `true` if successful; `false` if array full

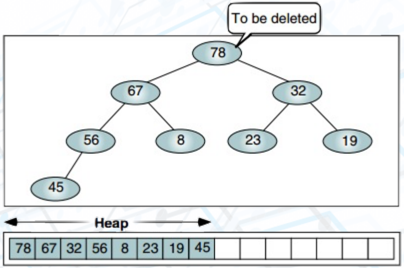
# Insert a Node into a Heap

```
if heap full then
  | return false
end
last = last + 1
heap[last] = data
reheapUp(heap, last)
return true
End insertHeap
```

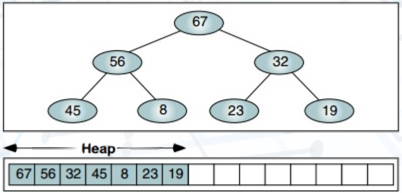
# Delete a Node from a Heap

- When deleting a node from a heap, the most common and meaningful logic is to delete the root.
- After it has been deleted, the heap is thus left without a root.
- To reestablish the heap, we move the data in the last heap node to the root and reheap down.

# Delete a Node from a Heap




(a) Before delete




(b) After delete

# Delete a Node from a Heap

**Algorithm** deleteHeap(ref heap <array>, ref last <integer>, ref dataOut <dataType>)
Deletes root of heap and passes data back to caller.

**Pre:** `heap` is a valid heap structure
`last` is reference parameter to last node
`dataOut` is reference parameter for output data

**Post:** root deleted and heap rebuilt
root data placed in `dataOut`

**Return** `true` if successful; `false` if array empty

```
if heap empty then
    return false
end
dataOut = heap[0]
heap[0] = heap[last]
last = last - 1
reheapDown(heap, 0, last)
return true
End deleteHeap
```

# Complexity of Binary Heap Operations

- ReheapUp: $O(\log_2 n)$
- ReheapDown: $O(\log_2 n)$
- Build a Heap: $O(n \log_2 n)$
- Insert a Node into a Heap: $O(\log_2 n)$
- Delete a Node from a Heap: $O(\log_2 n)$

# Heap Applications

# Heap Applications

Three common applications of heaps are:

1. selection algorithms,
2. priority queues,
3. and sorting.

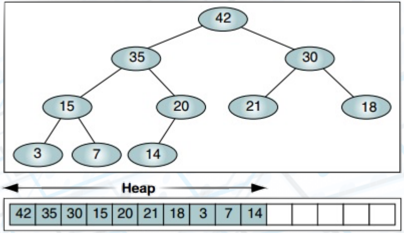
We discuss heap sorting in Chapter 10 and selection algorithms and priority queues here.

**Problem**
**Determining the $k^{th}$ element in an unsorted list.**

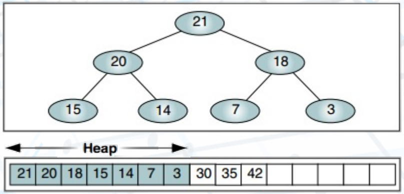
Two solutions:

1. Sort the list and select the element at location $k$. The complexity of a simple sorting algorithm is $O(n^2)$.
2. Create a heap and delete $k-1$ elements from the heap, leaving the desired element at the top. The complexity is $O(n \log_2 n)$.

Rather than simply discarding the elements at the top of the heap, a better solution would be to place the deleted element at the end of the heap and reduce the heap size by 1.
After the $k^{th}$ element has been processed, the temporarily removed elements can then be inserted into the heap.




(a) Original heap




(b) After three deletions

(Source: Data Structures - A Pseudocode Approach with C++)

**Algorithm** selectK(ref heap <array>, ref k <integer>, ref last <integer>)
Select the k-th largest element from a list.

**Pre:** `heap` is an array implementation of a heap
`k` is the ordinal of the element desired
`last` is reference parameter to last element

**Post:** k-th largest value returned

```
if k > last + 1 then
    return 0
end
i = 1
originalSize = last + 1
while i < k do
    temp = heap[0]
    deleteHeap(heap, last, dataOut)
    heap[last + 1] = temp
    i = i + 1
end
```

```
// Desired element is now at top of heap
holdOut = heap[0]

// Reconstruct heap
while last < originalSize do
    last = last + 1
    reheapUp(heap, last)
end
return holdOut
End selectK
```

The heap is an excellent structure to use for a priority queue.

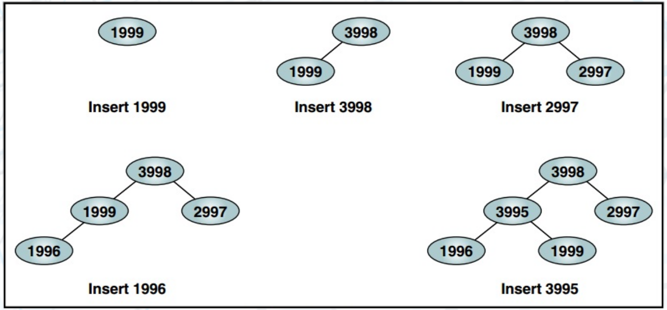
**Example**

Assume that we have a priority queue with three priorities: high (3), medium (2), and low (1). Of the first five customers who arrive, the second and the fifth are high-priority customers, the third is medium priority, and the first and the fourth are low priority.

| Arrival | Priority | Priority |
|---|---|---|
| 1 | low | 1999 (1 & (1000 - 1) |
| 2 | high | 3998 (3 & (1000 - 2) |
| 3 | medium | 2997 (2 & (1000 - 3) |
| 4 | low | 1996 (1 & (1000 - 4) |
| 5 | high | 3995 (3 & (1000 - 5) |

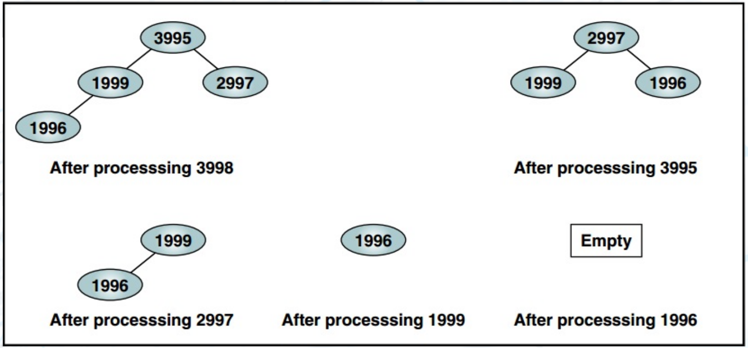
(Source: Data Structures - A Pseudocode Approach with C++)

The customers are served according to their priority and within equal priorities, according to their arrival. Thus we see that customer 2 (3998) is served first, followed by customer 5 (3995), customer 3 (2997), customer 1 (1999), and customer 4 (1996).



(a) Insert customers

(Source: Data Structures - A Pseudocode Approach with C++)




(b) Process customers

(Source: Data Structures - A Pseudocode Approach with C++)