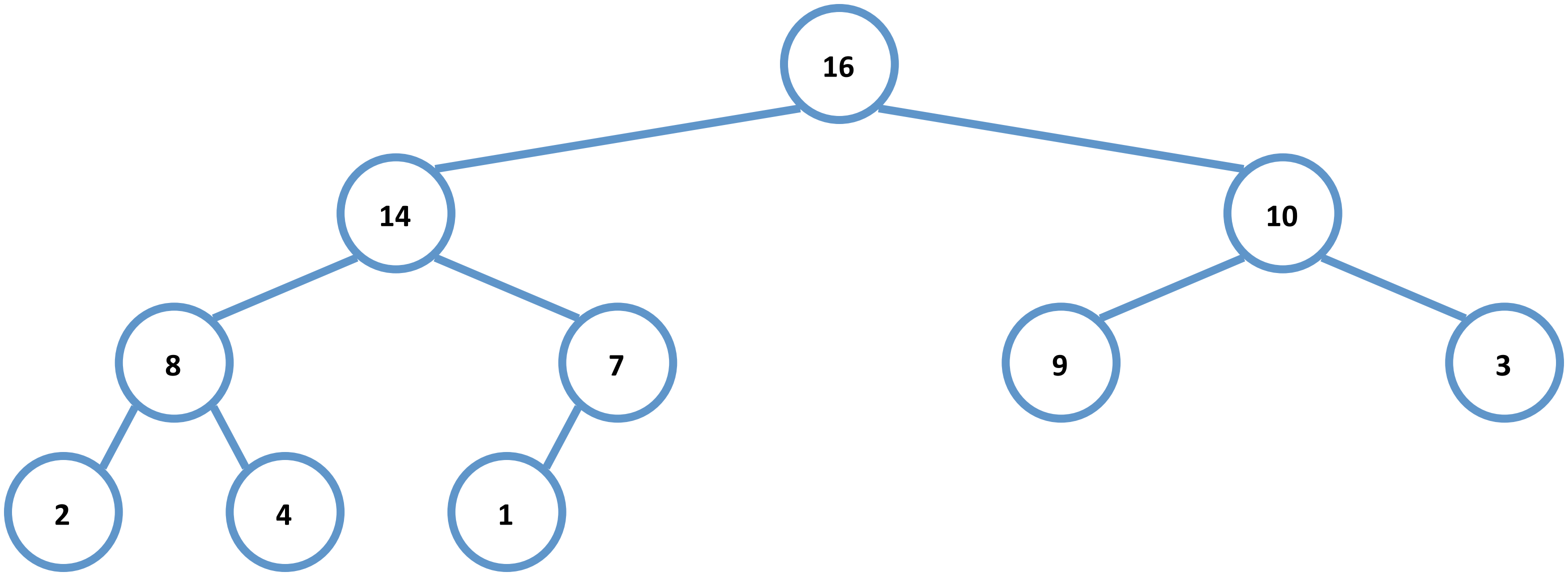
Heap Sort

Heaps

- A *heap* can be seen as a complete binary tree:



  - *What makes a binary tree complete?*
  - *Is the example above complete?*

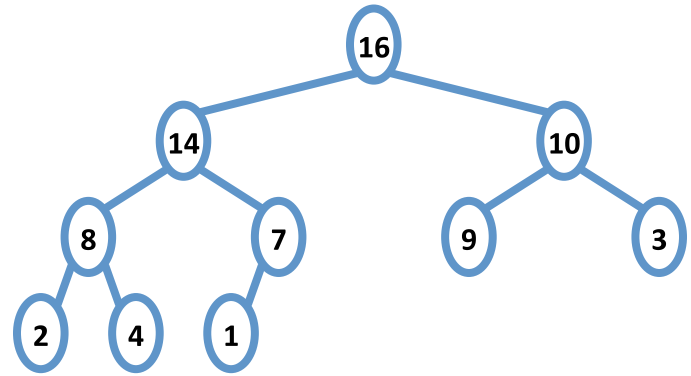
Heaps

- In practice, heaps are usually implemented as arrays:

A =

16	14	10	8	7	9	3	2	4	1

=



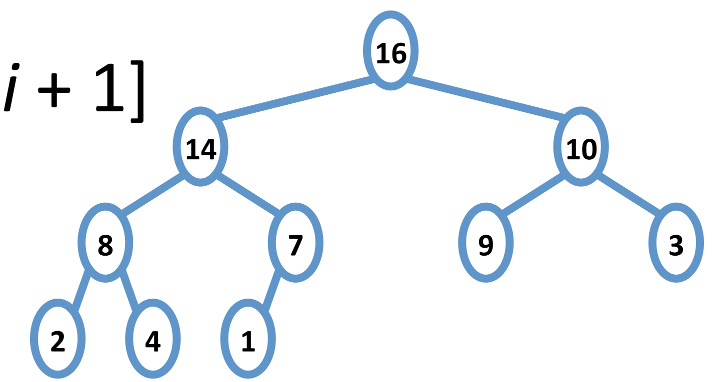
Heaps

- To represent a complete binary tree as an array:
  - The root node is A[1]
  - Node *i* is A[*i*]
  - The parent of node *i* is A[*i*/2] (note: integer divide)
  - The left child of node *i* is A[2*i*]
  - The right child of node *i* is A[2*i* + 1]

A =

16	14	10	8	7	9	3	2	4	1

=



Referencing Heap Elements

```
Parent(i) { return ⌊i/2⌋; }
Left(i) { return 2*i; }
right(i) { return 2*i + 1; }
```

Heap Property

- Heaps also satisfy the *heap property*:

  $A[\mathit{Parent}(i)] \geq A[i]$ for all nodes $i > 1$

  – In other words, the value of a node is at most the value of its parent

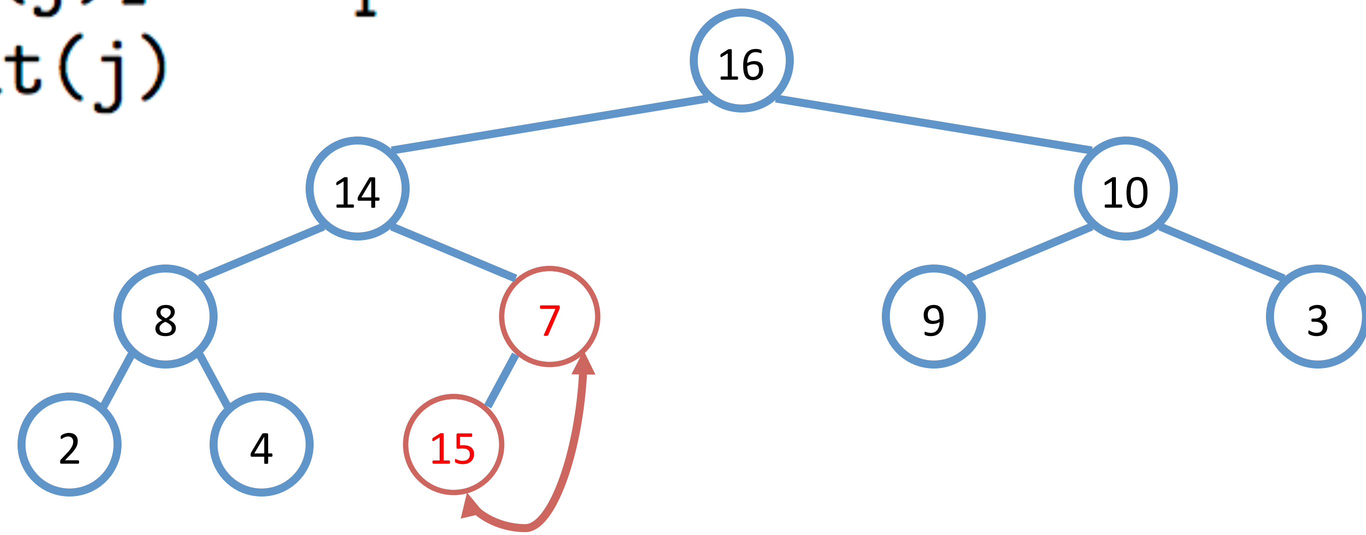
Insert

Pseudocode:

```
void insert(T[] A, T key)
  if heap-size(A) < A.length then
    A[heap-size(A)] = key
    heap-size(A) = heap-size(A) + 1
    The rest of this operation will be described in Step 2
  else
    throw new FullHeapException
  end if
```

Pseudocode for Step 2:

```
j = heap-size(A) − 1
while j > 0 and A[j] > A[parent(j)] do
    tmp = A[j]
    A[j] = A[parent(j)]
    A[parent(j)] = tmp
    j = parent(j)
end while
```

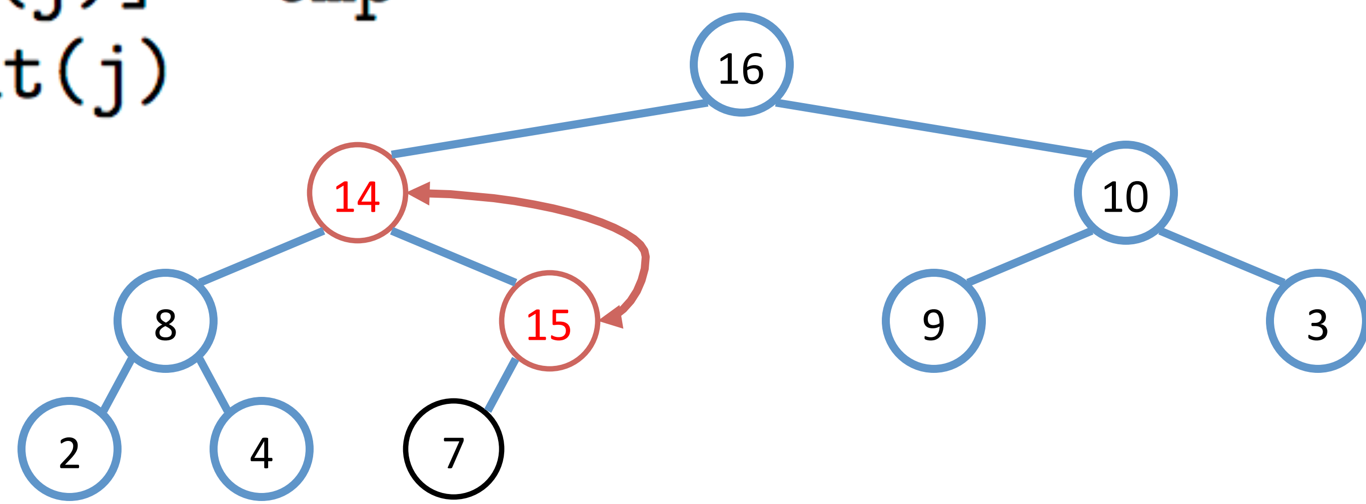


A =

16	14	10	8	7	9	3	2	4	15

Pseudocode for Step 2:

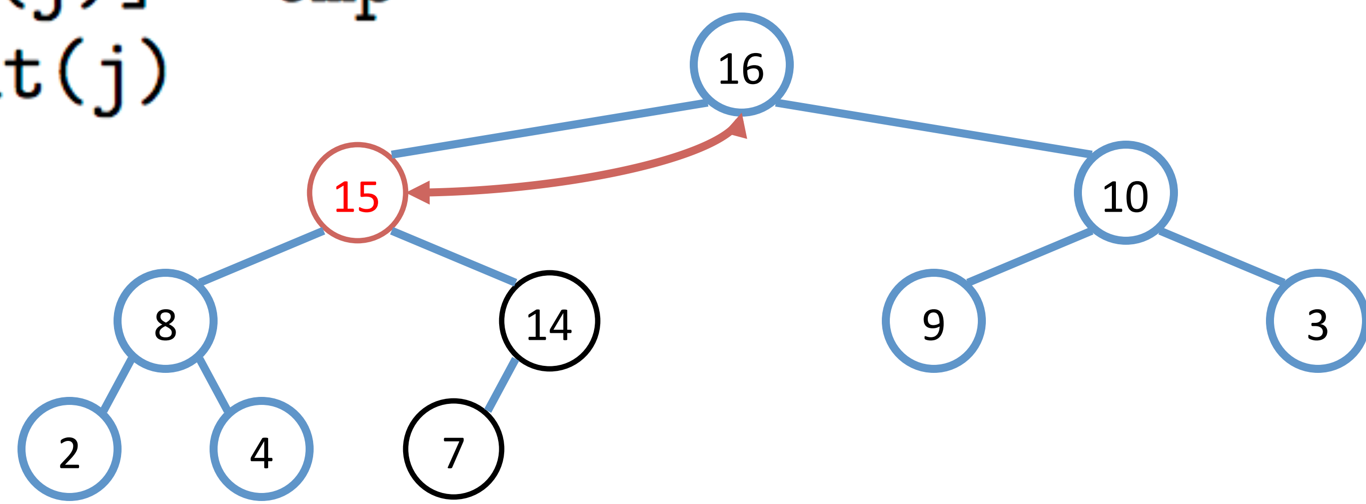
```
j = heap-size(A) − 1
while j > 0 and A[j] > A[parent(j)] do
    tmp = A[j]
    A[j] = A[parent(j)]
    A[parent(j)] = tmp
    j = parent(j)
end while
```



A =	16	14	10	8	15	9	3	2	4	7

Pseudocode for Step 2:

```
j = heap-size(A) − 1
while j > 0 and A[j] > A[parent(j)] do
    tmp = A[j]
    A[j] = A[parent(j)]
    A[parent(j)] = tmp
    j = parent(j)
end while
```

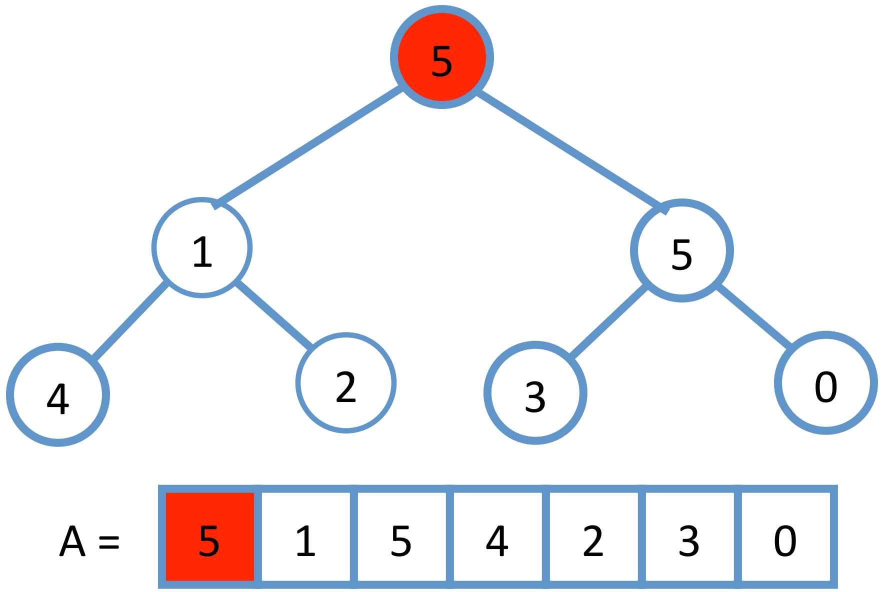


A =

16	15	10	8	14	9	3	2	4	7

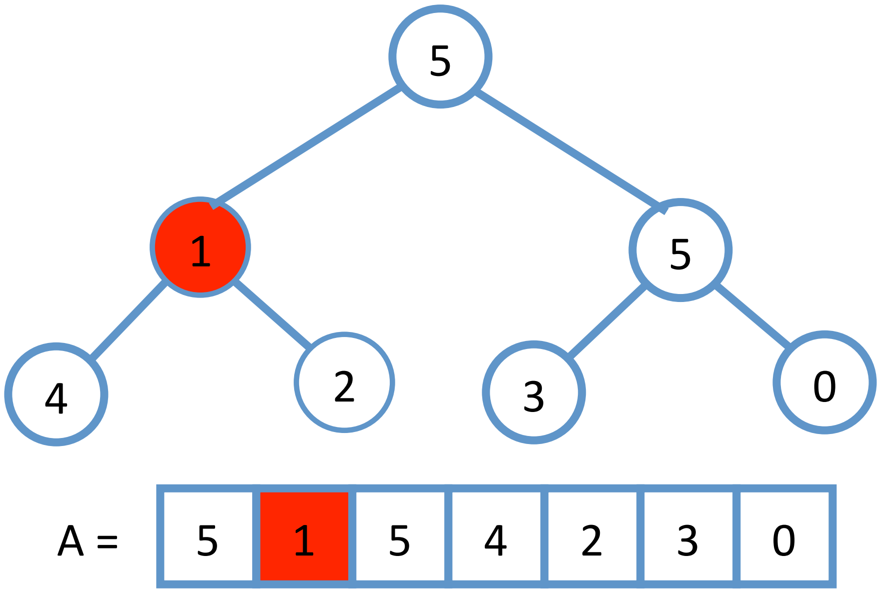
Heap Sort

```
void heapSort(T[] A)
  heap-size(A) = 1
  i = 1
  while i < A.length do
    insert(A, A[i])
    i = i + 1
  end while
  i = A.length − 1
  while i > 0 do
    largest = deleteMax(A)
    A[i] = largest
    i = i − 1
  end while
```

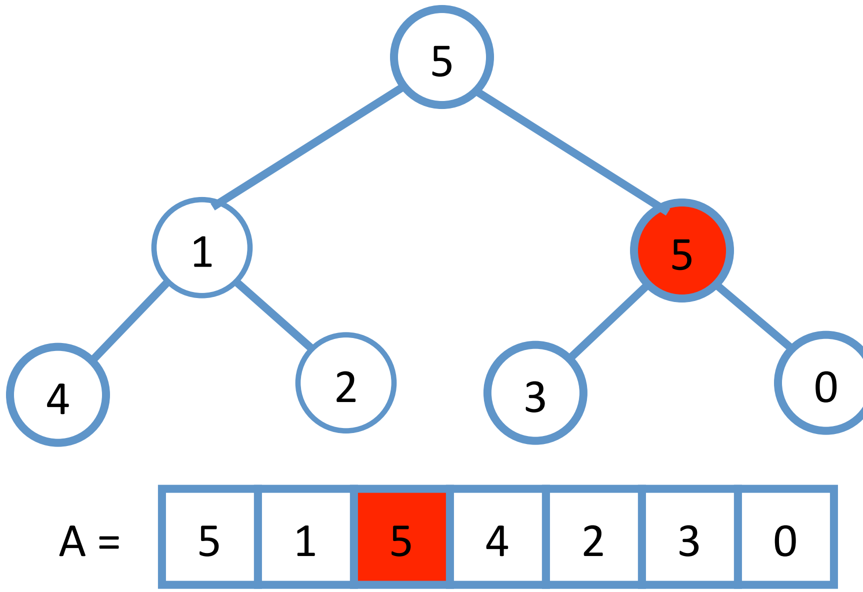


Heap Sort

```
void heapSort(T[] A)
  heap-size(A) = 1
  i = 1
  while i < A.length do
    insert(A, A[i])
    i = i + 1
  end while
  i = A.length − 1
  while i > 0 do
    largest = deleteMax(A)
    A[i] = largest
    i = i − 1
  end while
```



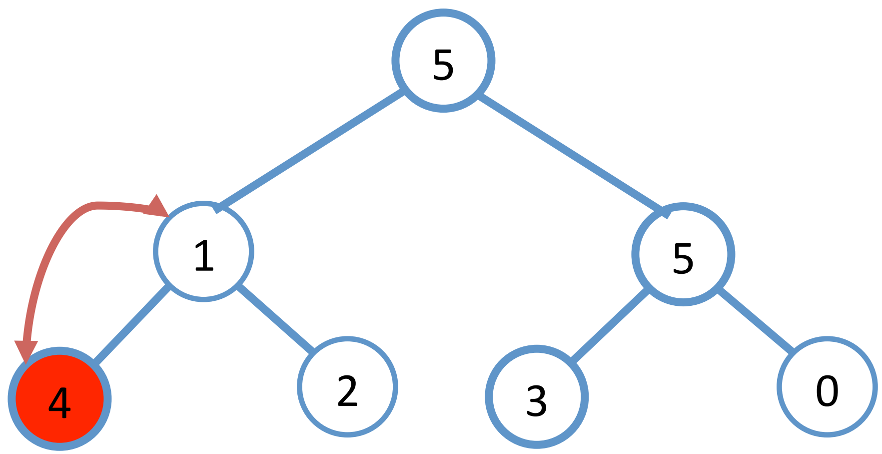
Heap Sort




```
void heapSort(T[] A)
  heap-size(A) = 1
  i = 1
  while i < A.length do
    insert(A, A[i])
    i = i + 1
  end while
  i = A.length − 1
  while i > 0 do
    largest = deleteMax(A)
    A[i] = largest
    i = i − 1
  end while
```

Heap Sort

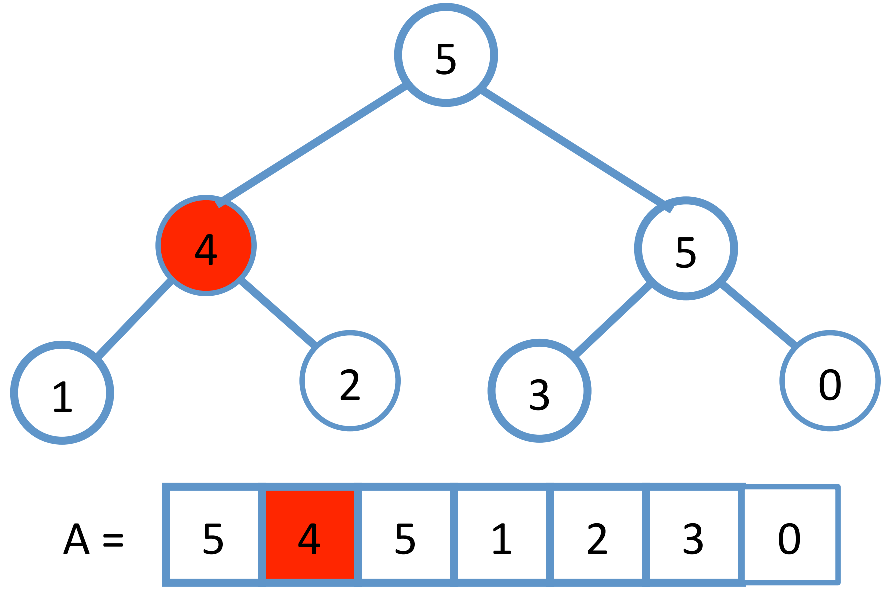
```
void heapSort(T[] A)
  heap-size(A) = 1
  i = 1
  while i < A.length do
    insert(A, A[i])
    i = i + 1
  end while
  i = A.length − 1
  while i > 0 do
    largest = deleteMax(A)
    A[i] = largest
    i = i − 1
  end while
```



A =	5	1	5	4	2	3	0

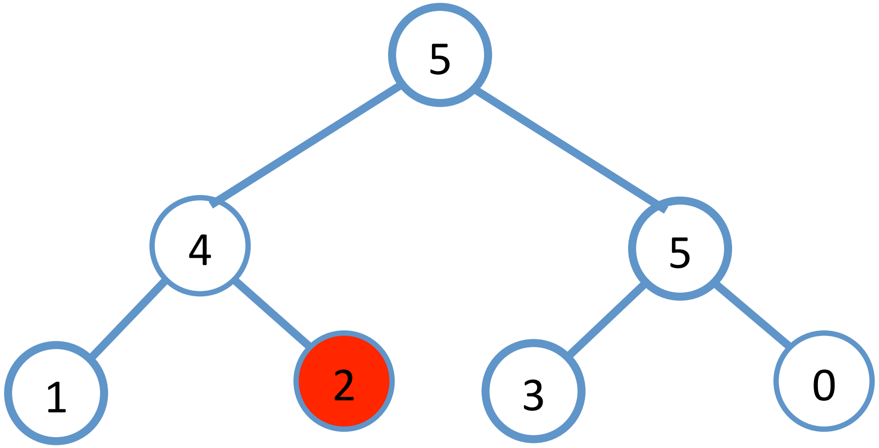
Heap Sort

```
void heapSort(T[] A)
  heap-size(A) = 1
  i = 1
  while i < A.length do
    insert(A, A[i])
    i = i + 1
  end while
  i = A.length − 1
  while i > 0 do
    largest = deleteMax(A)
    A[i] = largest
    i = i − 1
  end while
```



Heap Sort

```
void heapSort(T[] A)
  heap-size(A) = 1
  i = 1
  while i < A.length do
    insert(A, A[i])
    i = i + 1
  end while
  i = A.length − 1
  while i > 0 do
    largest = deleteMax(A)
    A[i] = largest
    i = i − 1
  end while
```

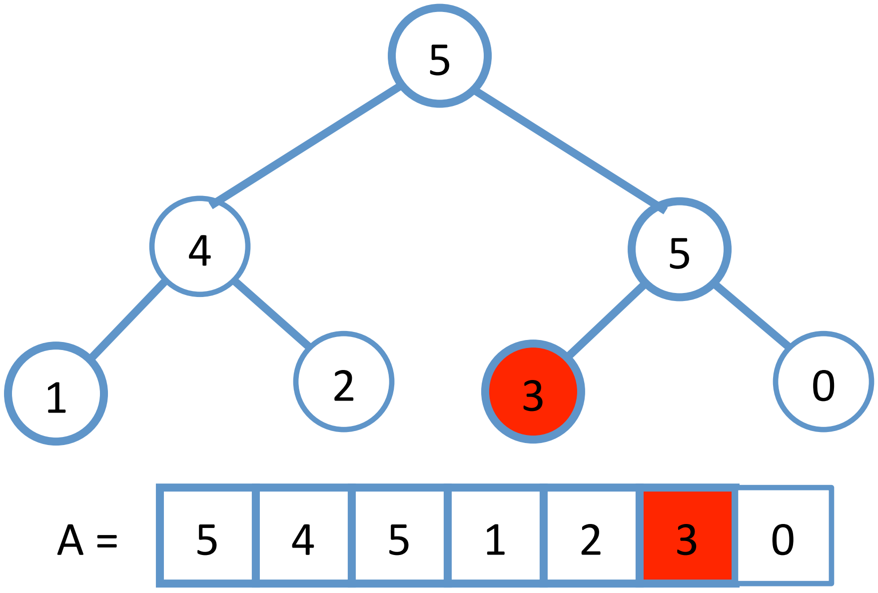


A =

5	4	5	1	2	3	0

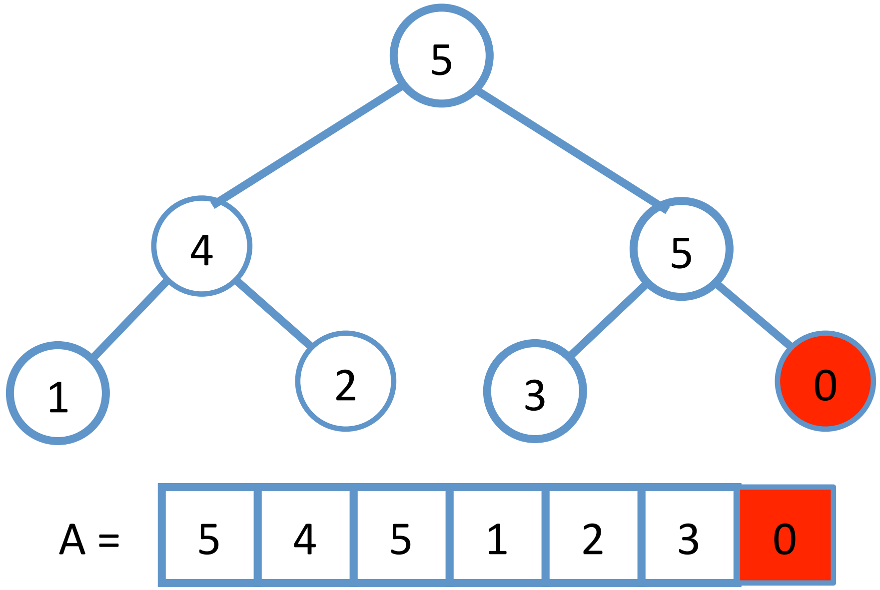
Heap Sort

```
void heapSort(T[] A)
  heap-size(A) = 1
  i = 1
  while i < A.length do
    insert(A, A[i])
    i = i + 1
  end while
  i = A.length − 1
  while i > 0 do
    largest = deleteMax(A)
    A[i] = largest
    i = i − 1
  end while
```



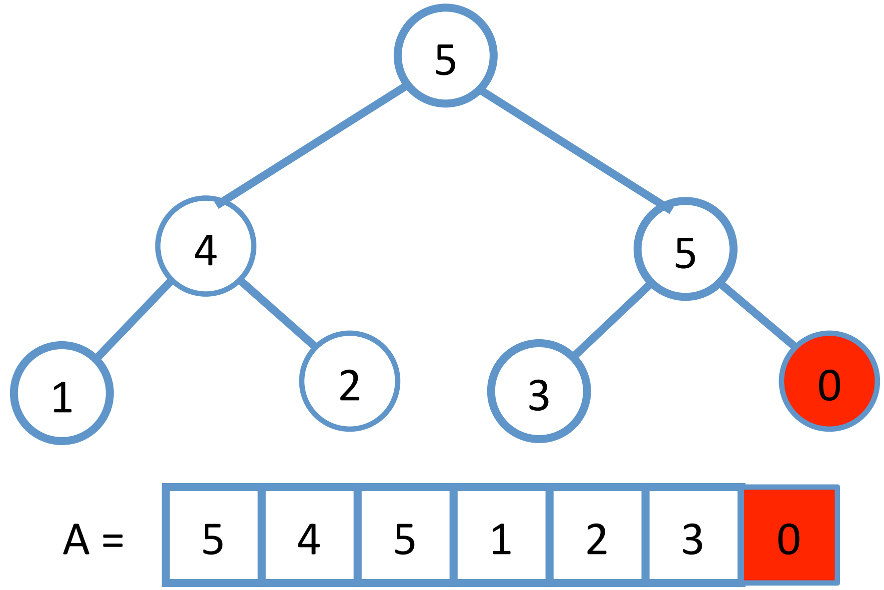
Heap Sort

```
void heapSort(T[] A)
  heap-size(A) = 1
  i = 1
  while i < A.length do
    insert(A, A[i])
    i = i + 1
  end while
  i = A.length − 1
  while i > 0 do
    largest = deleteMax(A)
    A[i] = largest
    i = i − 1
  end while
```



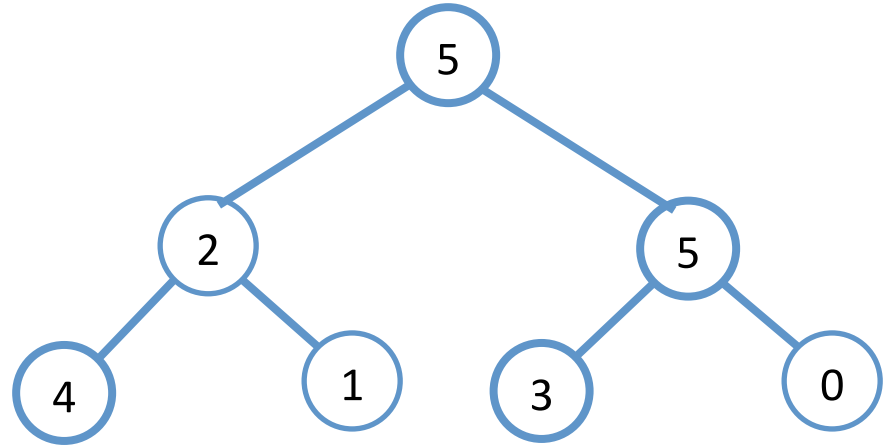
Heap Sort

```
void heapSort(T[] A)
  heap-size(A) = 1
  i = 1
  while i < A.length do
    insert(A, A[i])
    i = i + 1
  end while
  i = A.length − 1
  while i > 0 do
    largest = deleteMax(A)
    A[i] = largest
    i = i − 1
  end while
```

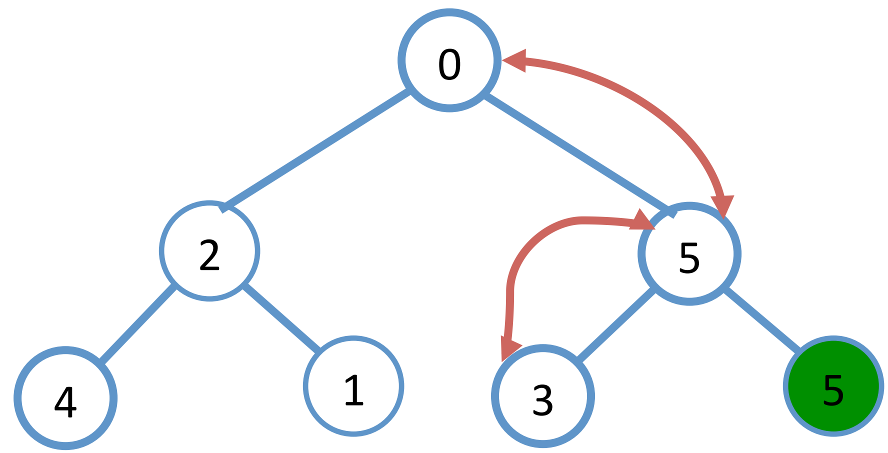


Heap Sort

```
void heapSort(T[] A)
  heap-size(A) = 1
  i = 1
  while i < A.length do
    insert(A, A[i])
    i = i + 1
  end while
  i = A.length − 1
  while i > 0 do
    largest = deleteMax(A)
    A[i] = largest
    i = i − 1
  end while
```

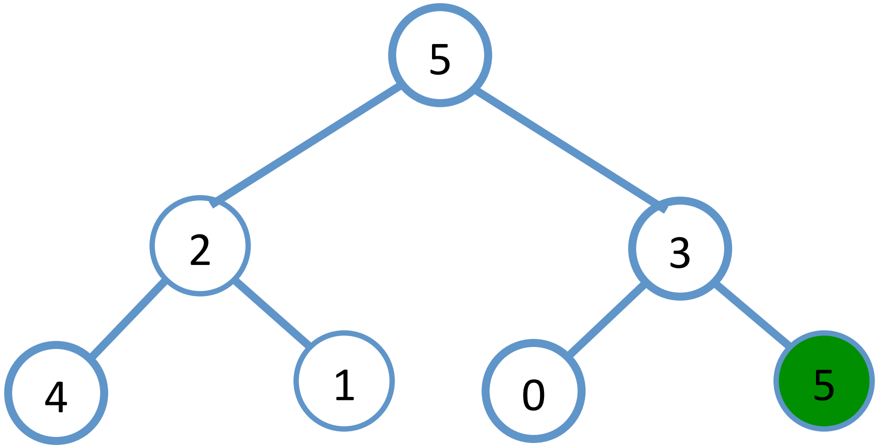


A =						
5	2	5	4	1	3	0

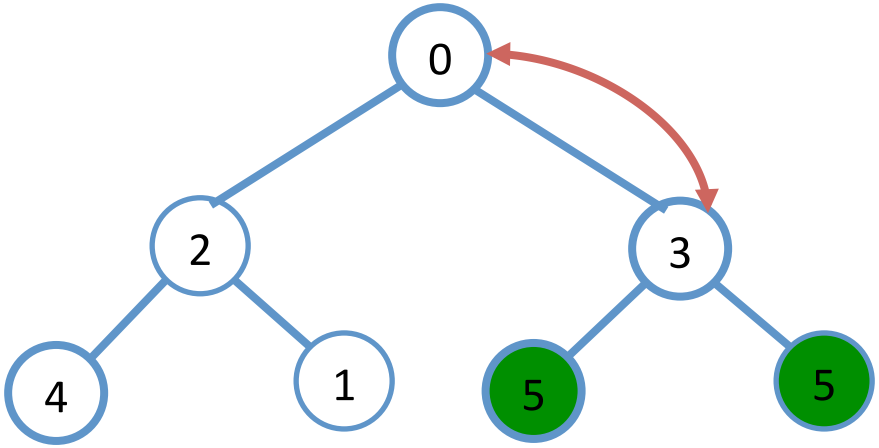


Heap Sort

```
void heapSort(T[] A)
  heap-size(A) = 1
  i = 1
  while i < A.length do
    insert(A, A[i])
    i = i + 1
  end while
  i = A.length − 1
  while i > 0 do
    largest = deleteMax(A)
    A[i] = largest
    i = i − 1
  end while
```



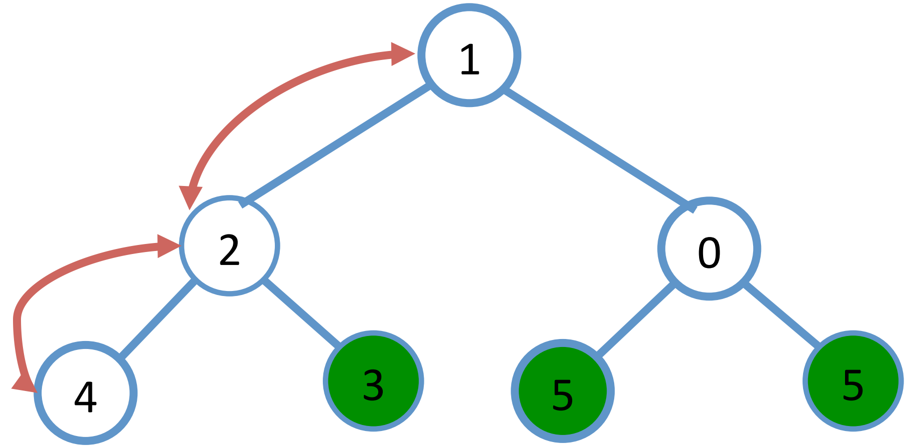
Heap Sort



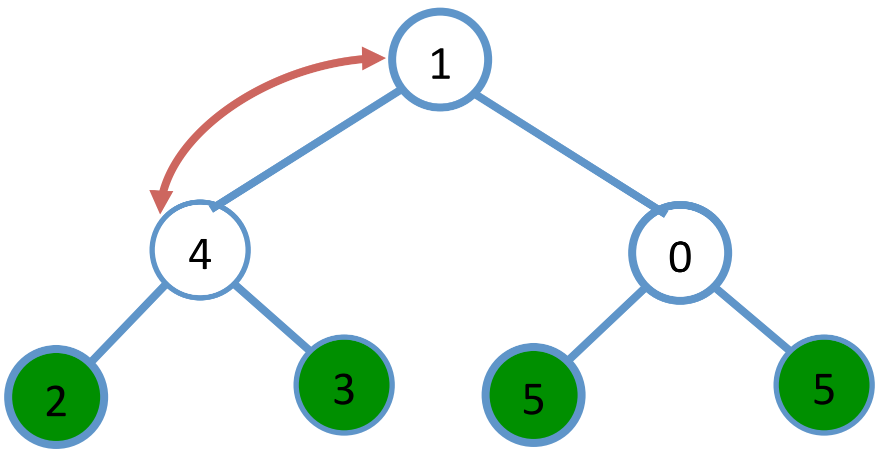

```
void heapSort(T[] A)
  heap-size(A) = 1
  i = 1
  while i < A.length do
    insert(A, A[i])
    i = i + 1
  end while
  i = A.length − 1
  while i > 0 do
    largest = deleteMax(A)
    A[i] = largest
    i = i − 1
  end while
```

Heap Sort

```
void heapSort(T[] A)
  heap-size(A) = 1
  i = 1
  while i < A.length do
    insert(A, A[i])
    i = i + 1
  end while
  i = A.length − 1
  while i > 0 do
    largest = deleteMax(A)
    A[i] = largest
    i = i − 1
  end while
```



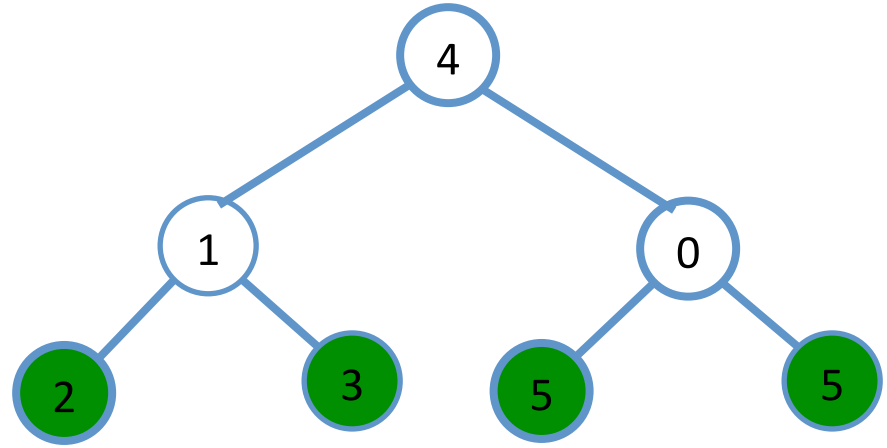
Heap Sort



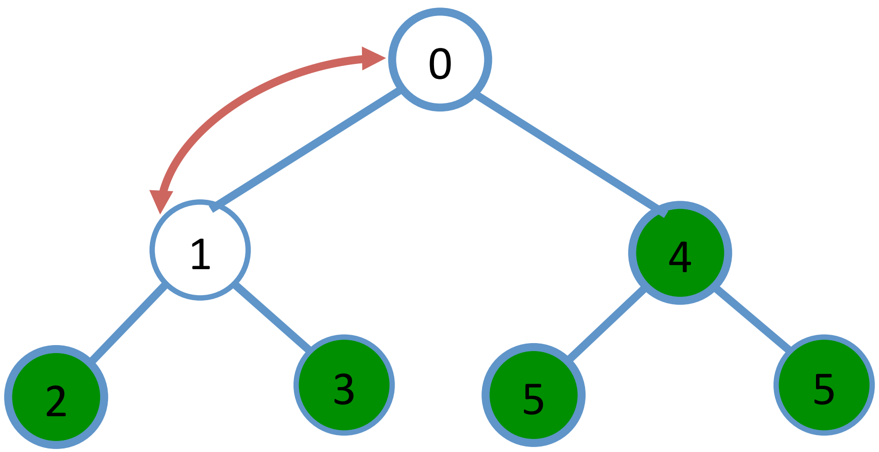

```
void heapSort(T[] A)
  heap-size(A) = 1
  i = 1
  while i < A.length do
    insert(A, A[i])
    i = i + 1
  end while
  i = A.length − 1
  while i > 0 do
    largest = deleteMax(A)
    A[i] = largest
    i = i − 1
  end while
```

Heap Sort

```
void heapSort(T[] A)
  heap-size(A) = 1
  i = 1
  while i < A.length do
    insert(A, A[i])
    i = i + 1
  end while
  i = A.length − 1
  while i > 0 do
    largest = deleteMax(A)
    A[i] = largest
    i = i − 1
  end while
```

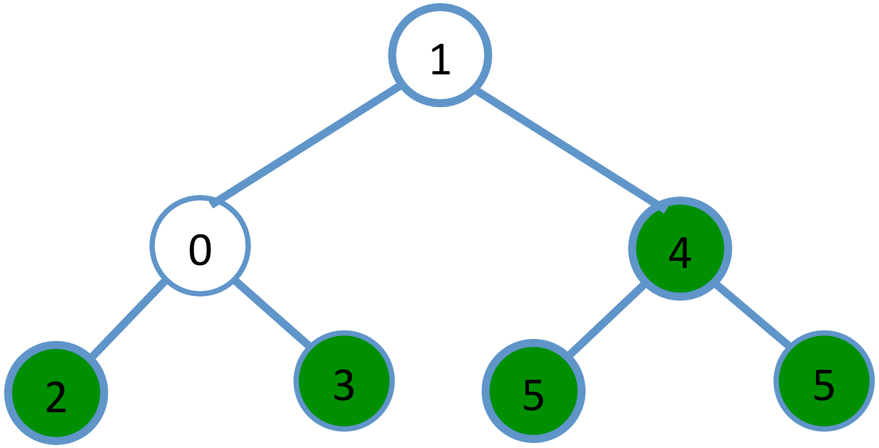


Heap Sort



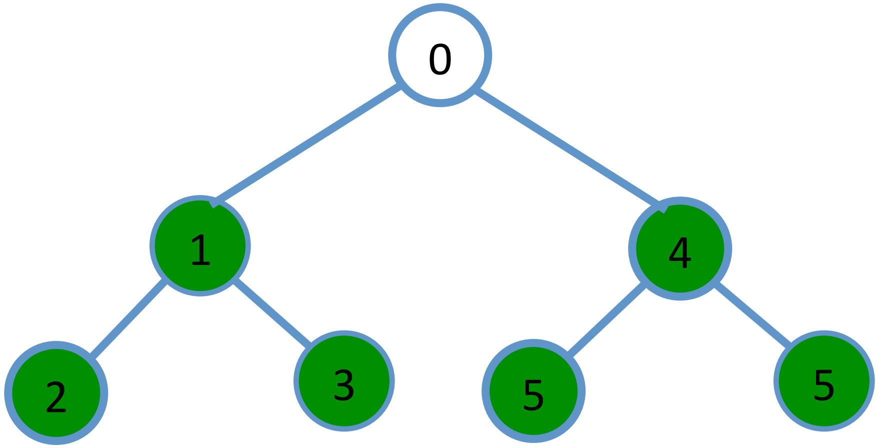

```
void heapSort(T[] A)
  heap-size(A) = 1
  i = 1
  while i < A.length do
    insert(A, A[i])
    i = i + 1
  end while
  i = A.length − 1
  while i > 0 do
    largest = deleteMax(A)
    A[i] = largest
    i = i − 1
  end while
```

Heap Sort



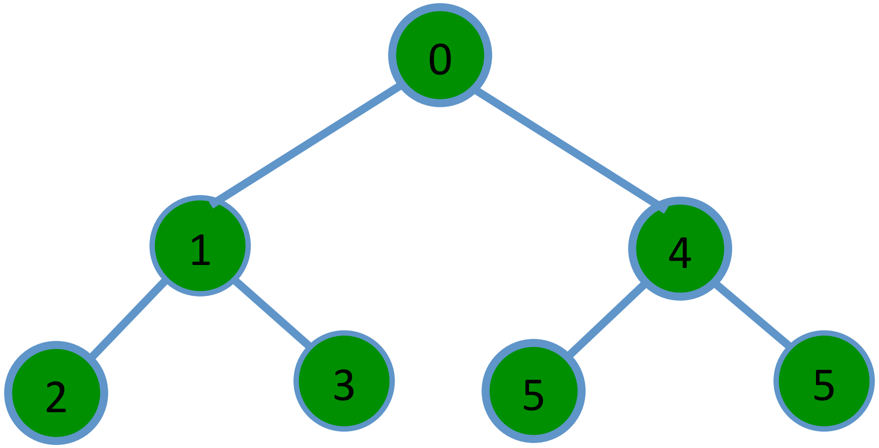
void **heapSort**(T[] A)
  heap-size(A) = 1
  i = 1
  while i < A.length **do**
    insert(A, A[i])
    i = i + 1
  end while
  i = A.length − 1
  while i > 0 **do**
    largest = **deleteMax**(A)
    A[i] = largest
    i = i − 1
  end while

Heap Sort




```
void heapSort(T[] A)
  heap-size(A) = 1
  i = 1
  while i < A.length do
    insert(A, A[i])
    i = i + 1
  end while
  i = A.length − 1
  while i > 0 do
    largest = deleteMax(A)
    A[i] = largest
    i = i − 1
  end while
```

Heap Sort




```
void heapSort(T[] A)
  heap-size(A) = 1
  i = 1
  while i < A.length do
    insert(A, A[i])
    i = i + 1
  end while
  i = A.length − 1
  while i > 0 do
    largest = deleteMax(A)
    A[i] = largest
    i = i − 1
  end while
```

Build Max Heap Recursively

- Build-Max-Heap(A,i)
  - N= A.length()
  - If Left_child(i)<=n then
    - Build-Max-Heap(A, Left_child(i))
  - If right_child(i)<=n then
    - Build-Max-Heap(A, right_child(i))
  - Max-Heapify(A, i)

- Max-Heapify(A, i)

set j equal to i the rest of it as follows:

```
while j < heap-size(A) do
  ℓ = left(j); r = right(j); largest = j
  if ℓ < heap-size(A) and A[ℓ] > A[largest] then
    largest = ℓ
  end if
  if r < heap-size(A) and A[r] > A[largest] then
    largest = r
  end if
  if largest ≠ j then
    tmp = A[j]; A[j] = A[largest]; A[largest] = tmp;
    j = largest
  else
    j = heap-size(A)
  end if
end while
```

Analyzing the time complexity of recursive Build-Max-Heap

- if $m \geq 2$ then $T(m)$ satisfies the following inequality:
  - $T(m) \leq T(m_L) + T(m_R) + c \log m$

- Prove that:
  - $T(m) \in O(m)$.
- Hint :
- It will be helpful to try to show that, for sufficiently large m,
- $T(m) \leq c_1\, m - c_2 \log m$