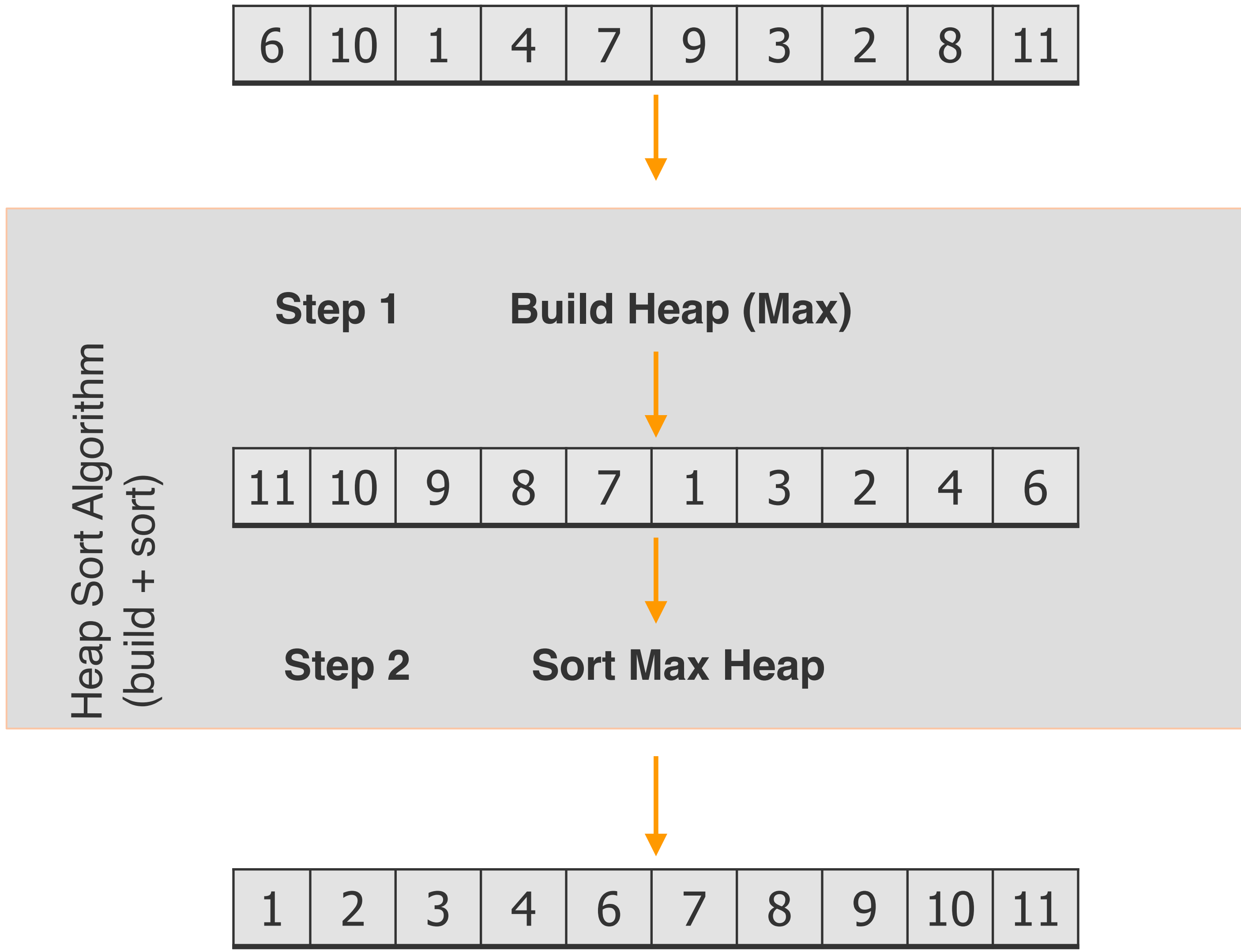
Heap Sort

Heapsort



6 10 1 4 7 9 3 2 8 11
Heap Sort Algorithm
(build + sort)
Step 1
Build Heap (Max)
11 10 9 8 7 1 3 2 4 6
Step 2
Sort Max Heap
1 2 3 4 6 7 8 9 10 11

Heapsort Algorithm

Function Heapsort(A)

```
Step 1   #Create max heap
         Build_Max_Heap from unordered array A
Step 2   # Finish sorting
         for i = n downto 2 do
                 exchange A[1] with A[i]
                 discard node i from heap (decrement heap size)
                 sift(A[1:i-1], 1) because new root may violate max heap property
```

Build Max Heap - Step 1

```
Function Build_Max_Heap(A)
    set heap size to the length of the array
    for j= n/2 down to 1 do
         sift(A, j)
```

Heap

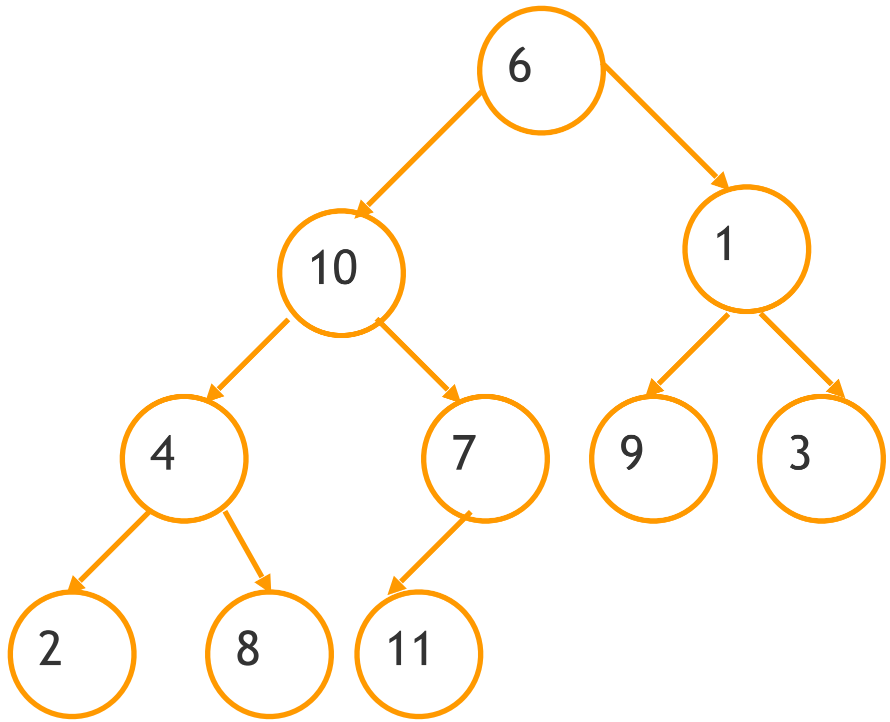
- The root of the tree is A[1], and given the index i of a node, we can easily compute the indices of its parent, left child, and right child:

```
function parent(i)
    return i/2

function left(i)
    return 2*i


function right(i)
    return 2 *i + 1
```

Max-Heapify (sift)

```
function sift(arr,i)
        n ← len(arr)          # array length
        l ← left(i)           # left node index
        r ← right(i)          # right node index

        if l <= n and arr[l] > arr[i] then
            largest ← l
        else
            largest ← i

        if r <= n and arr[r] > arr[largest] then
            largest ← r

        if largest != i then
            arr[i] ↔ arr[largest]
            sift(arr, largest)
        return arr
```

Start with an array (it is not a max heap)



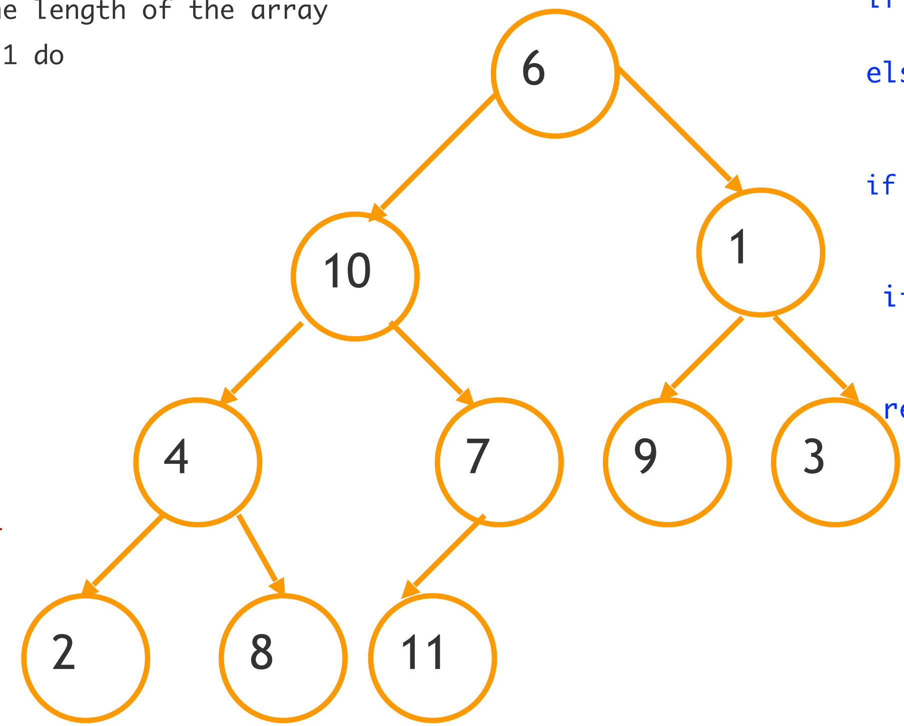
6	10	1	4	7	9	3	2	8	11

Exchange 7 and 11

```
Function Build_Max_Heap(A)
    set heap size to the length of the array
    for j= n/2 down to 1 do
        sift(A, j)
```

```
function parent(i)
    return i/2

function left(i)
    return 2*i

function right(i)
    return 2 *i + 1
```

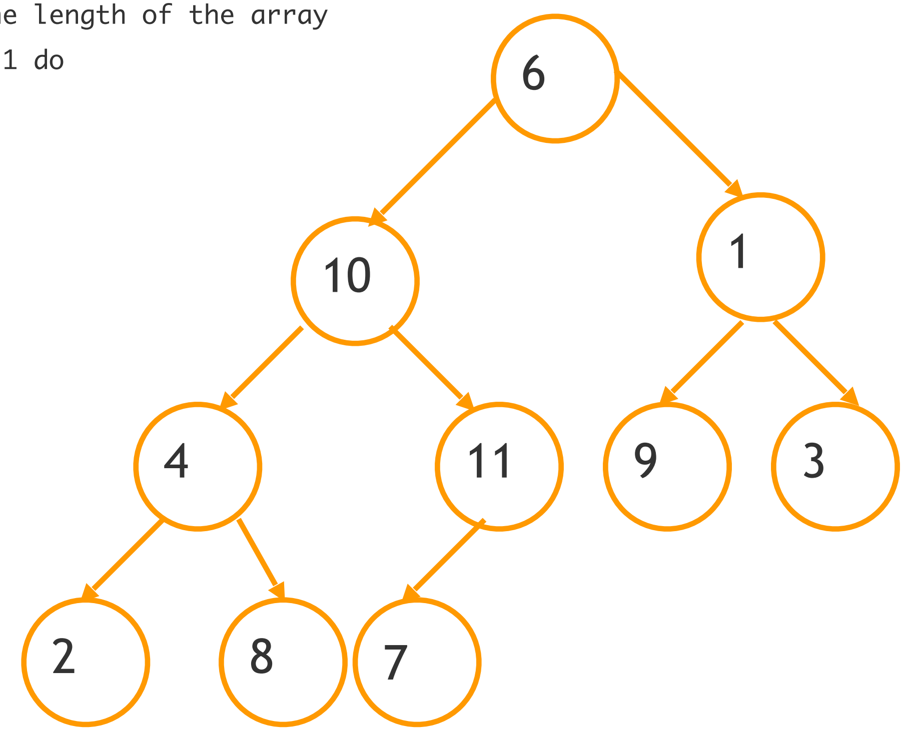
```
function sift(arr,i)
    n ← len(arr)
    l ← left(i)
    r ← right(i)

 if l <= n and arr[l] > arr[i] then
        largest ← l
 else
        largest ← i

 if r <= n and arr[r] > arr[largest] then
        largest = r

  if largest != i then
       arr[i] ↔ arr[largest]
       sift(arr,largest)
  return arr
```



6	10	1	4	7	9	3	2	8	11
				j					

Exchange 4 and 8

```
Function Build_Max_Heap(A)
    set heap size to the length of the array
    for j= n/2 down to 1 do
         sift(A, j)
```

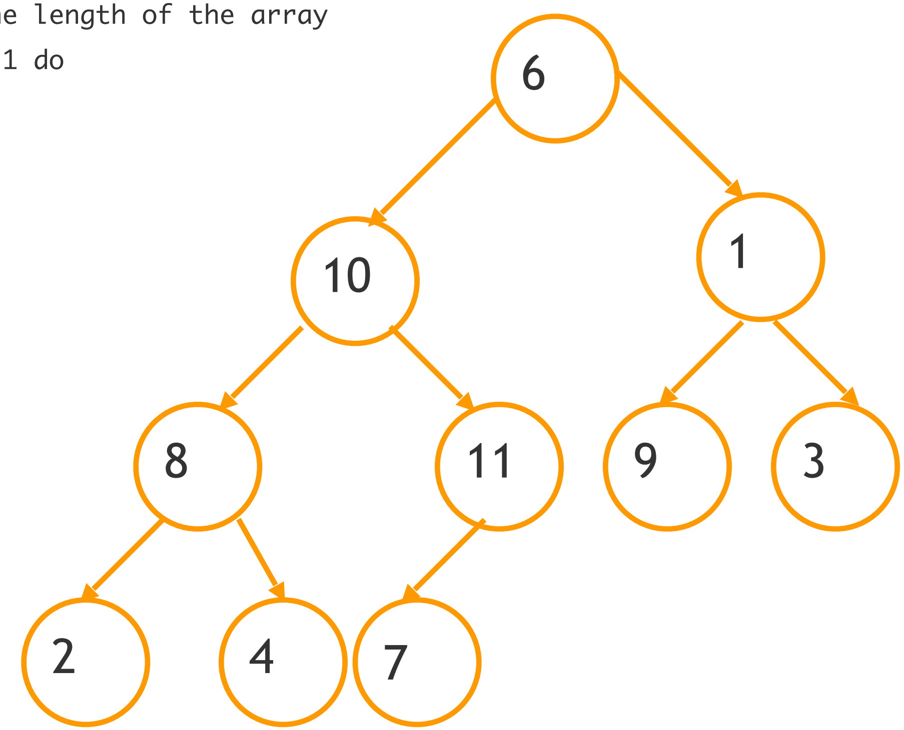


6	10	1	4	11	9	3	2	8	7

j

Exchange 9 and 1

```
Function Build_Max_Heap(A)
    set heap size to the length of the array
    for j= n/2 down to 1 do
         sift(A, j)
```

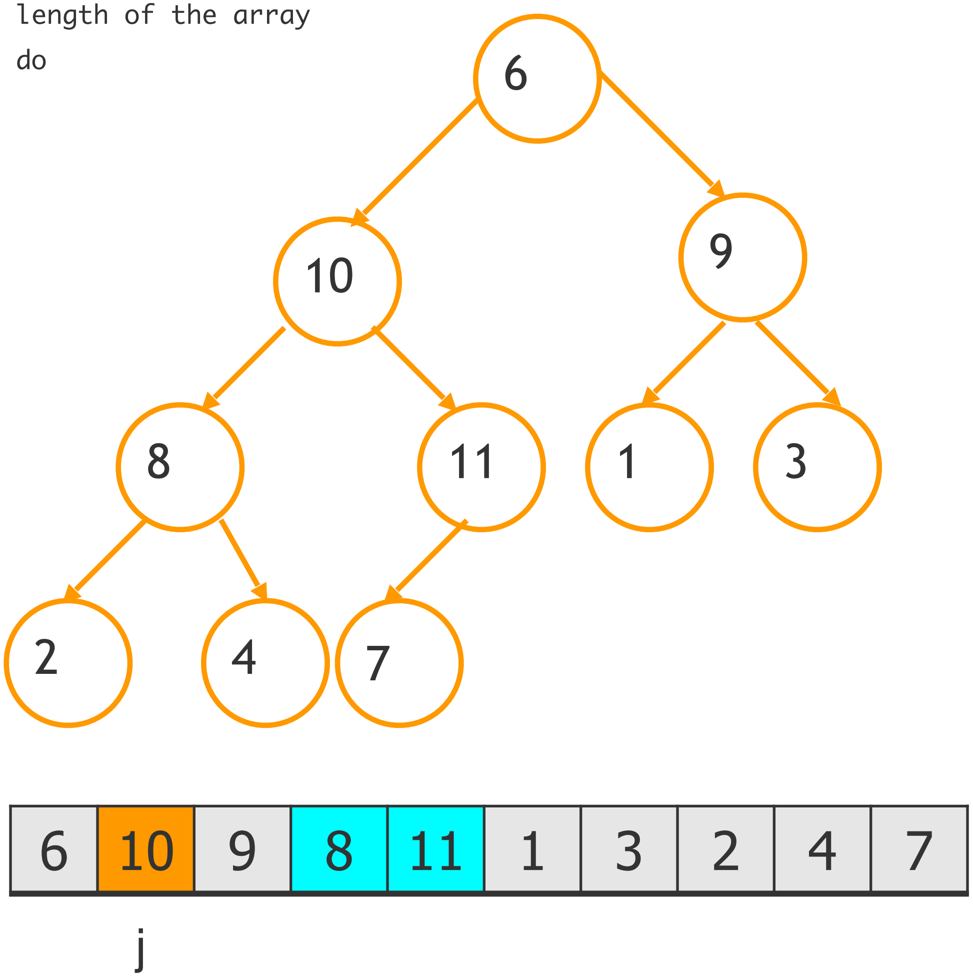


6	10	1	8	11	9	3	2	4	7

j

Exchange 10 and 11

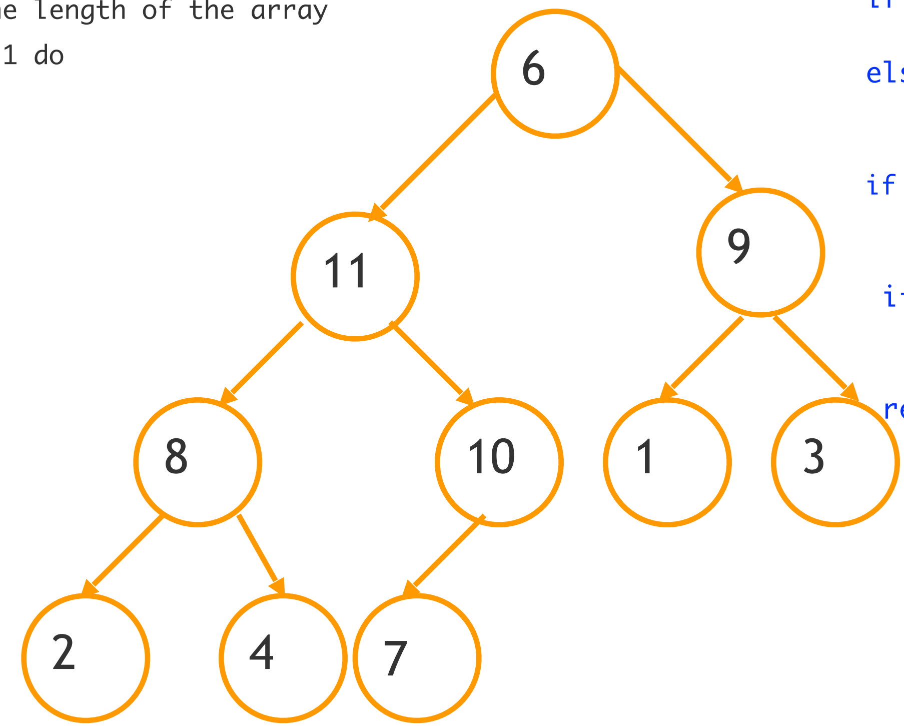
```
Function Build_Max_Heap(A)
    set heap size to the length of the array
    for j= n/2 down to 1 do
        sift(A, j)
```



Exchange 6 and 11

```
Function Build_Max_Heap(A)
    set heap size to the length of the array
    for j= n/2 down to 1 do
         sift(A, j)
```

```
function sift(arr,i)
        n ← len(arr)
        l ← left(i)
        r ← right(i)

    if l <= n and arr[l] > arr[i] then
            largest ← l
    else
            largest ← i

    if r <= n and arr[r] > arr[largest] then
            largest ← r

     if largest != i then
           arr[i] ↔ arr[largest]
           sift(arr,largest)
     return arr
```



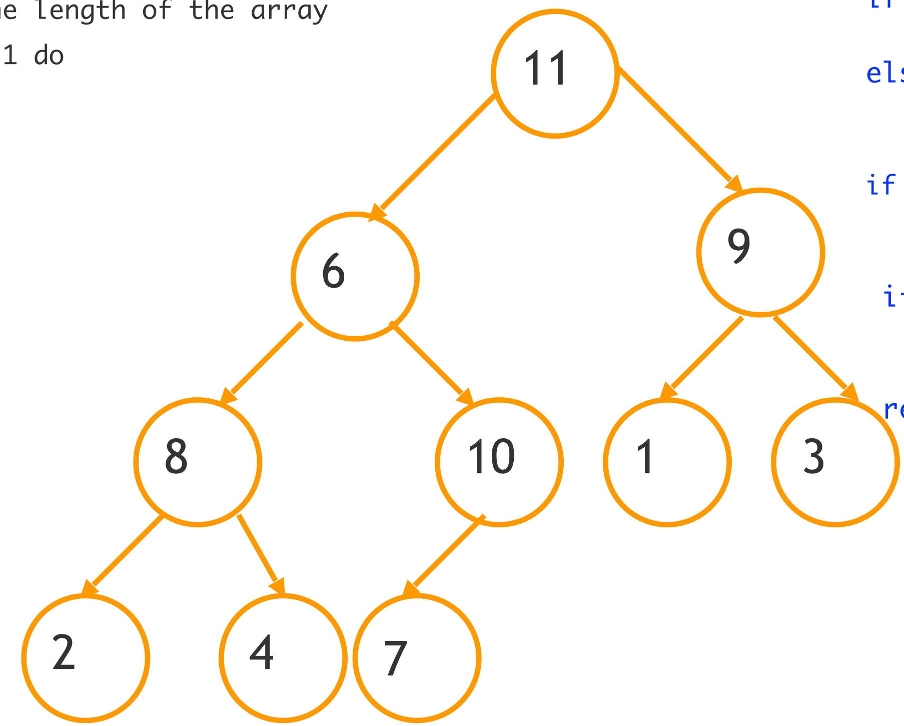
6	11	9	8	10	1	3	2	4	7

j

Exchange 6 and 10

```
Function Build_Max_Heap(A)
    set heap size to the length of the array
    for j= n/2 down to 1 do
        sift(A, j)
```

```
function sift(arr,i)
        n ← len(arr)
        l ← left(i)
        r ← right(i)

    if l <= n and arr[l] > arr[i] then
            largest ← l
    else
            largest ← i

    if r <= n and arr[r] > arr[largest] then
            largest ← r

    if largest != i then
            arr[i] ↔ arr[largest]
            sift(arr,largest)
    return arr
```

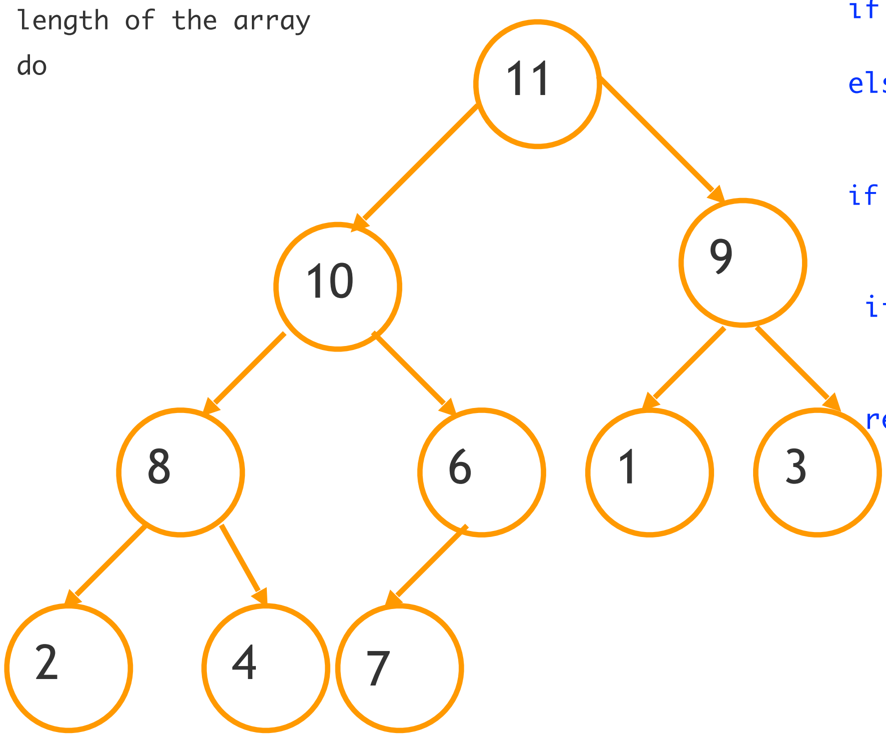


11	6	9	8	10	1	3	2	4	7
j	i			largest					

Exchange 6 and 7

```
function sift(arr,i)
      n ← len(arr)
      l ← left(i)
      r ← right(i)

   if l <= n and arr[l] > arr[i] then
         largest ← l
   else
         largest ← i

   if r <= n and arr[r] > arr[largest] then
         largest ← r

    if largest != i then
        arr[i] ↔ arr[largest]
        sift(arr,largest)
    return arr
```

```
Function Build_Max_Heap(A)
    set heap size to the length of the array
    for j= n/2 down to 1 do
         sift(A, j)
```



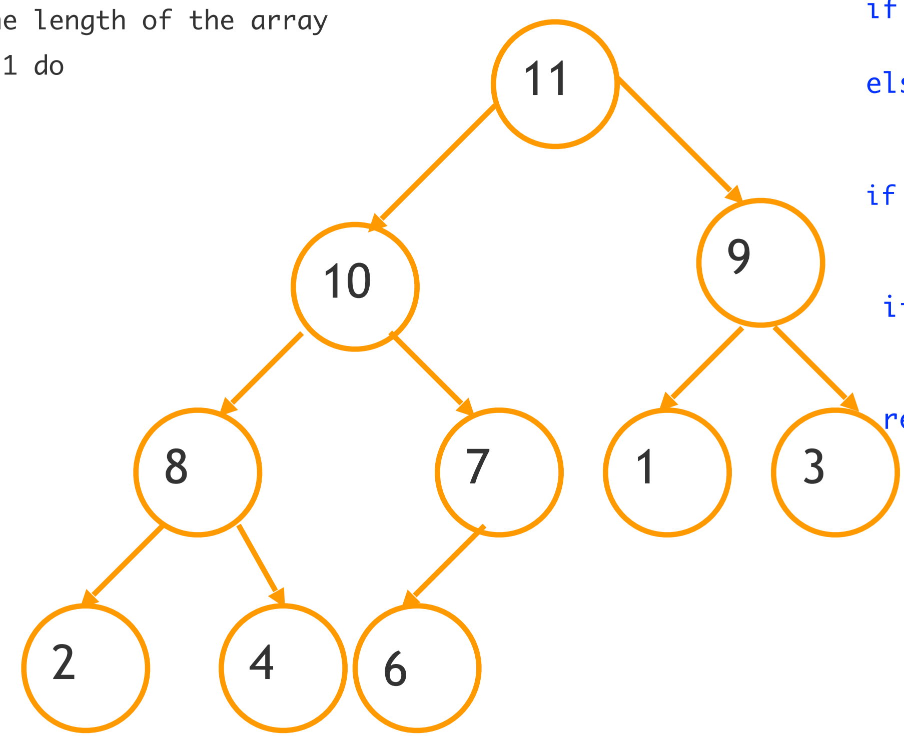
11	10	9	8	6	1	3	2	4	7

j　　　　i　　　　largest

max_heapify

```
Function Build_Max_Heap(A)
    set heap size to the length of the array
    for j= n/2 down to 1 do
         sift(A, j)
```

```
function sift(arr,i)
        n ← len(arr)
        l ← left(i)
        r ← right(i)

    if l <= n and arr[l] > arr[i] then
            largest ← l
    else
            largest ← i

    if r <= n and arr[r] > arr[largest] then
            largest ← r

     if largest != i then
           arr[i] ↔ arr[largest]
           sift(arr,largest)
     return arr
```



11	10	9	8	7	1	3	2	4	6

j　　　　　　　　　　　　　　　　　　　　　i largest

Sorting

Function Heapsort(A)

```
#Create max heap
Build_Max_Heap from unordered array A
# Finish sorting
for i = n downto 2 do
    exchange A[1] with A[i]
    discard node i from heap (decrement heap size)
    sift(A[1:i-1],1) because new root may violate max heap property
```

Exchange 11 and 6

Function Heapsort(A)

#Create max heap

Build_Max_Heap from unordered array A

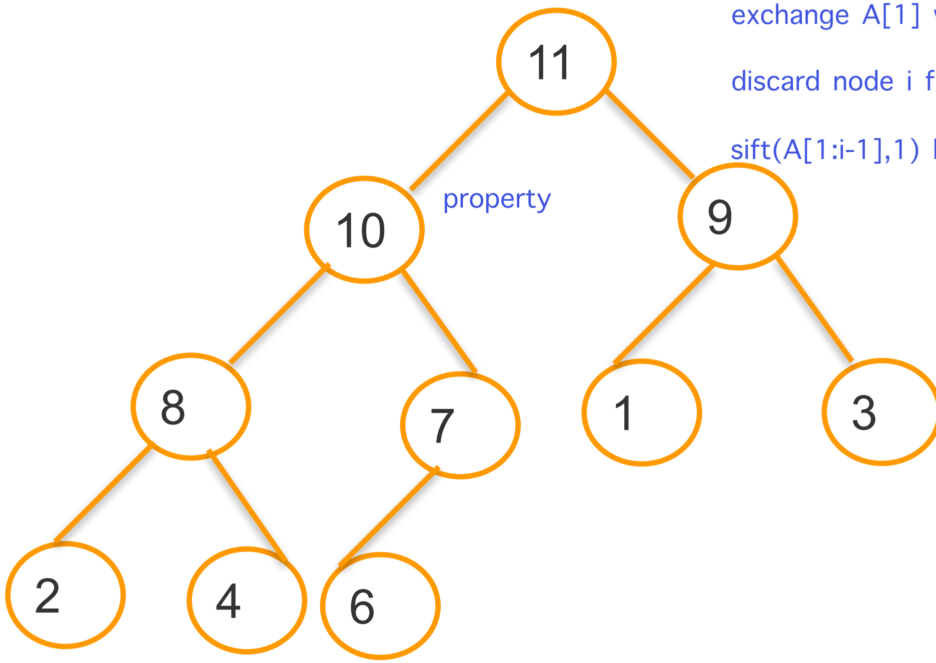
Finish sorting

for i = n downto 2 do

exchange A[1] with A[i]

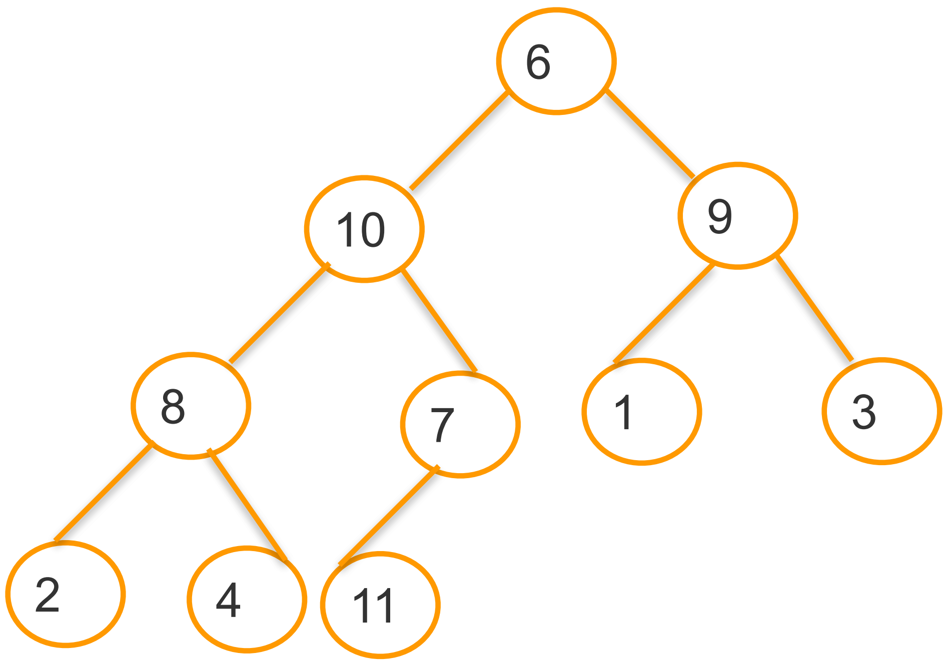
discard node i from heap (decrement heap size)

sift(A[1:i-1],1) because new root may violate max heap property



11	10	9	8	7	1	3	2	4	6

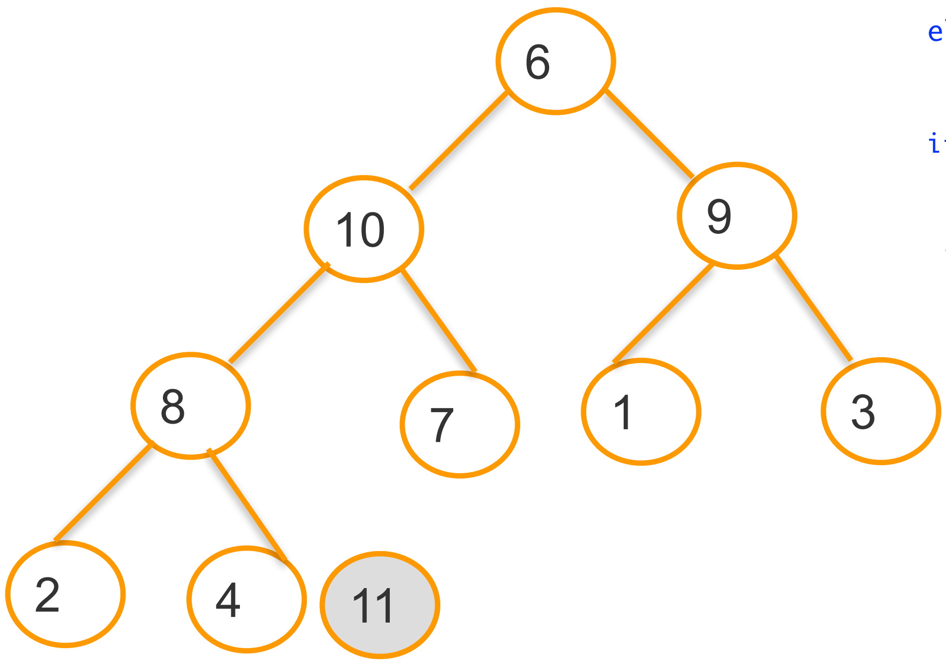
Remove 11 from the heap



6	10	9	8	7	1	3	2	4	11

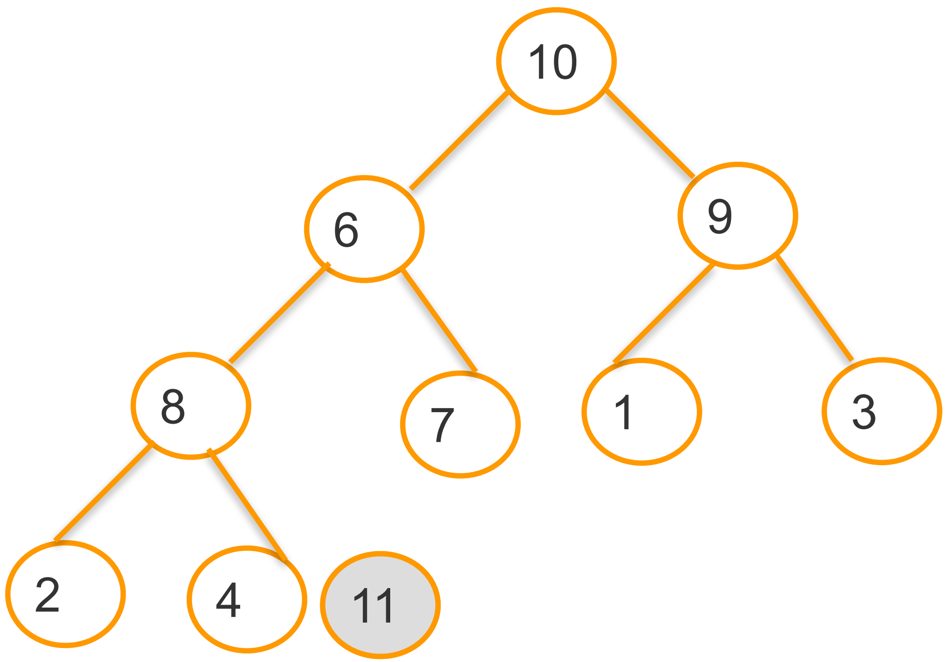
Swap 6 and 10

```
function sift(arr,i)
        n ← len(arr)
        l ← left(i)
        r ← right(i)

    if l <= n and arr[l] > arr[i] then
            largest ← l
    else
            largest ← i

    if r <= n and arr[r] > arr[largest] then
            largest ← r

     if largest != i then
          arr[i] ↔ arr[largest]
          sift(arr,largest)
     return arr
```



6	10	9	8	7	1	3	2	4	11

Exchange 6 and 8



10	6	9	8	7	1	3	2	4	11

Exchange 10 and 4

Function Heapsort(A)

#Create max heap

Build_Max_Heap from unordered array A

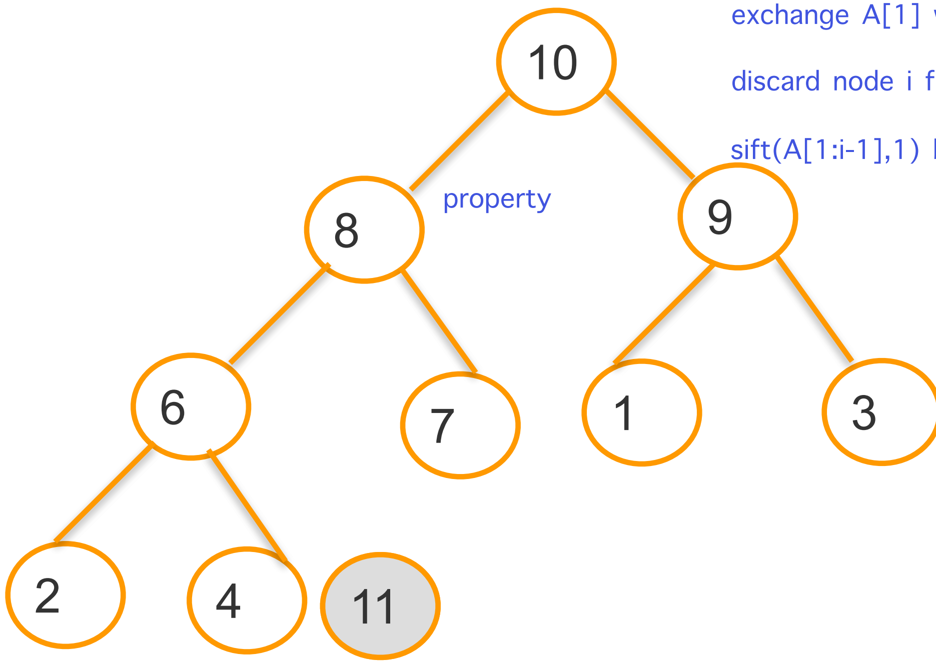
Finish sorting

for i = n downto 2 do

exchange A[1] with A[i]

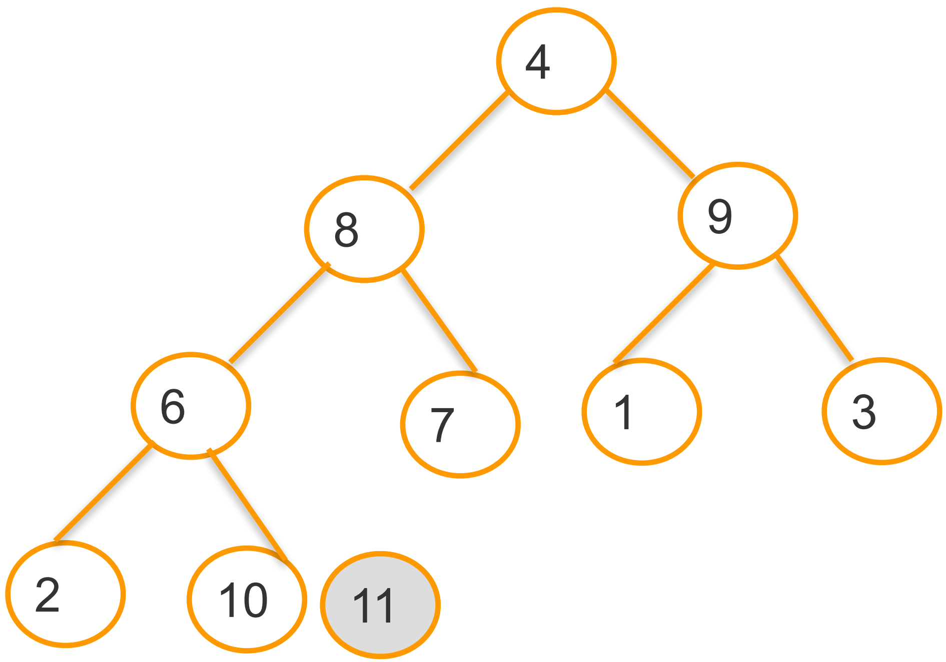
discard node i from heap (decrement heap size)

sift(A[1:i-1],1) because new root may violate max heap property



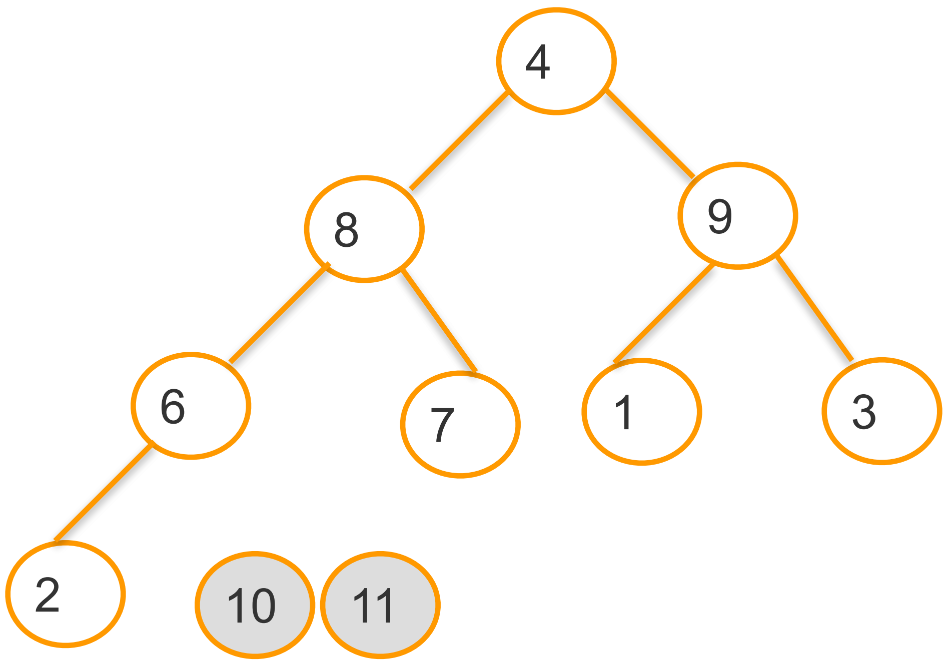
10	8	9	6	7	1	3	2	4	11

Remove 10 from the heap



4	8	9	6	7	1	3	2	10	11

Exchange 4 and 9



4	8	9	6	7	1	3	2	10	11

Exchange 9 and 2

Function Heapsort(A)

#Create max heap

Build_Max_Heap from unordered array A

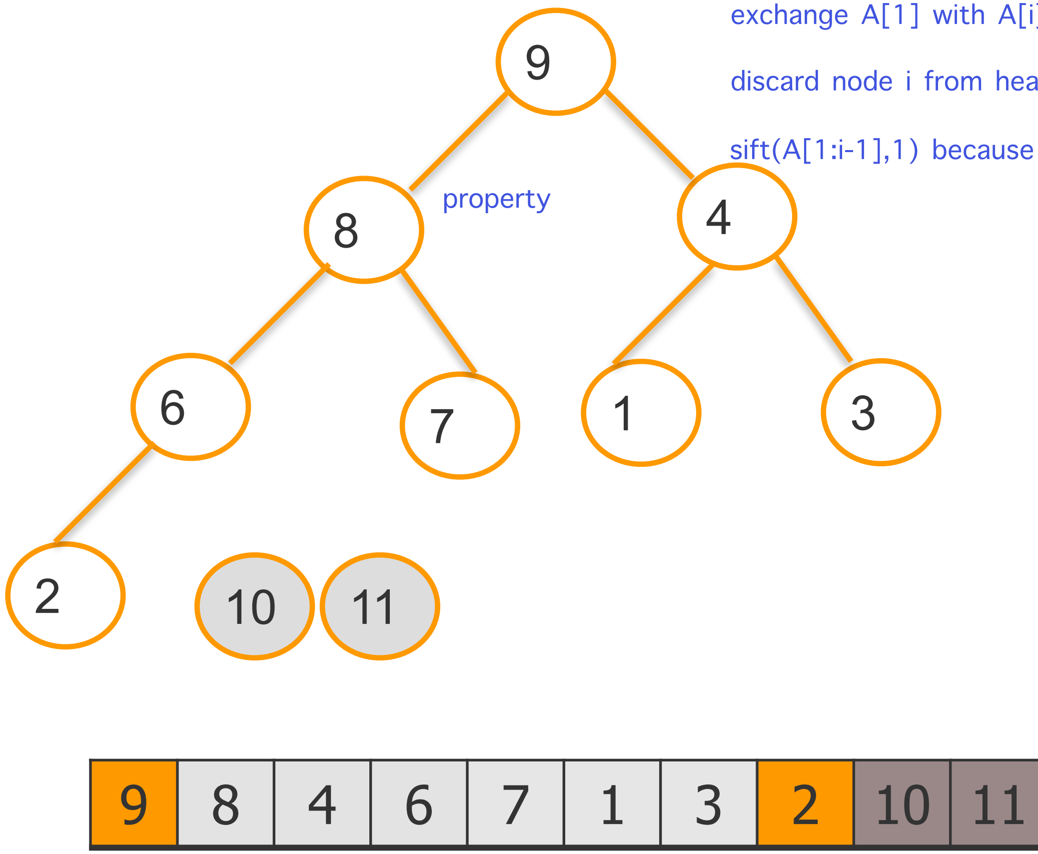
Finish sorting

for i = n downto 2 do

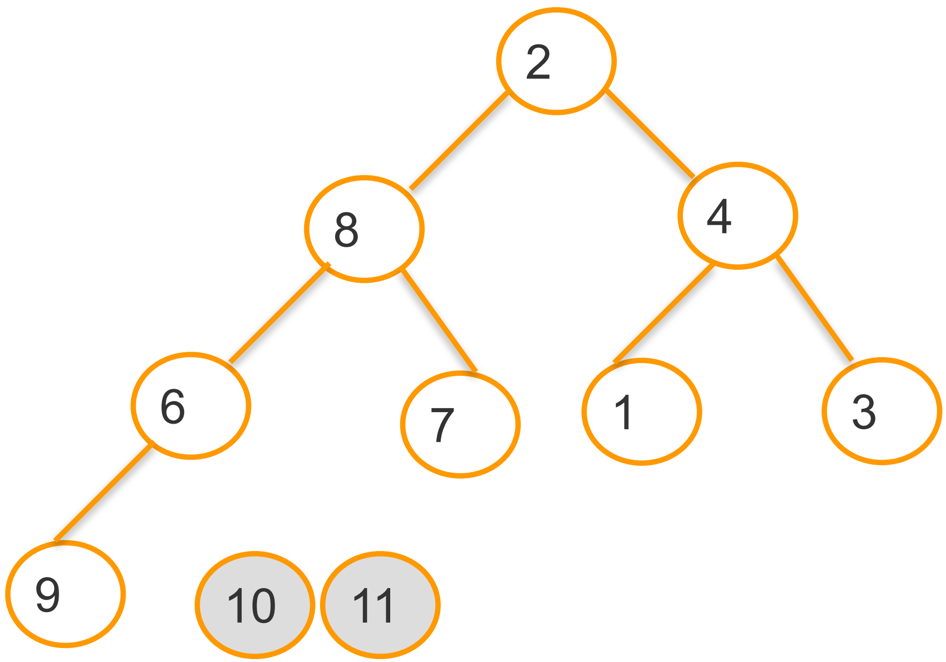
exchange A[1] with A[i]

discard node i from heap (decrement heap size)

sift(A[1:i-1],1) because new root may violate max heap property

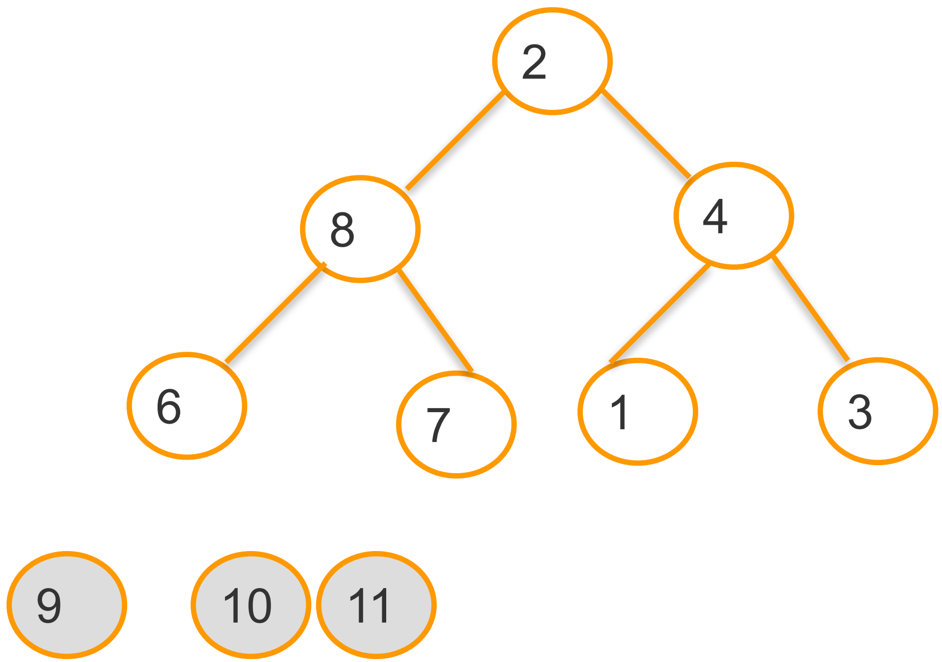


Remove 9 from the heap



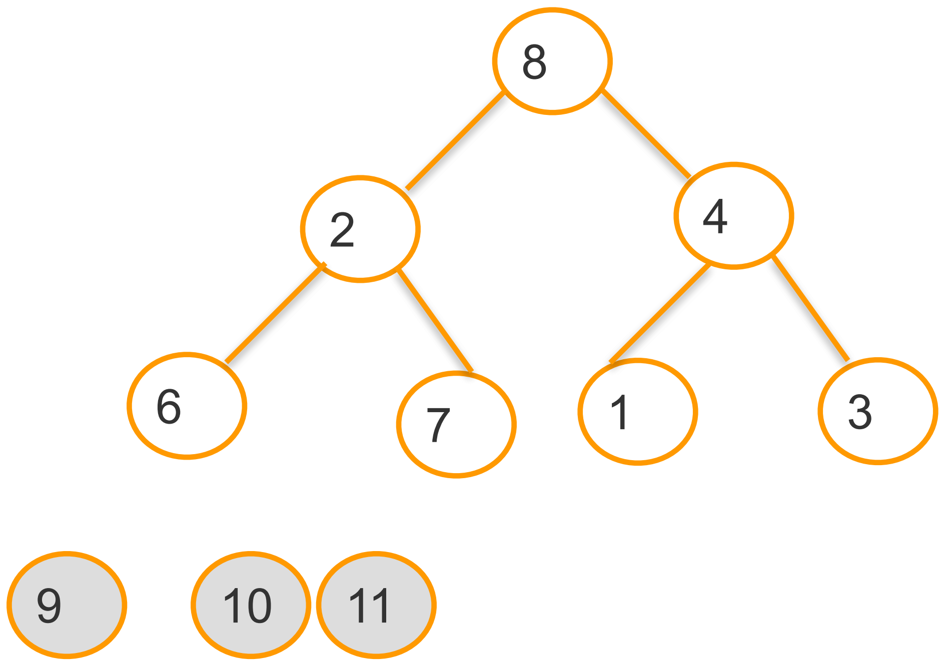
2	8	4	6	7	1	3	9	10	11

Exchange 2 and 8



2	8	4	6	7	1	3	9	10	11

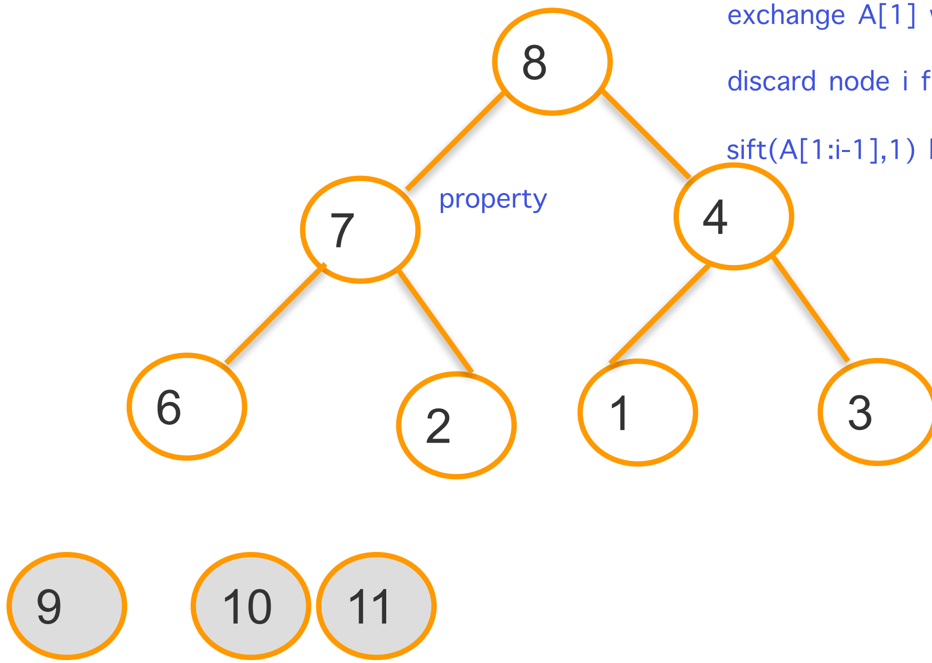
Exchange 2 and 7



8	2	4	6	7	1	3	9	10	11

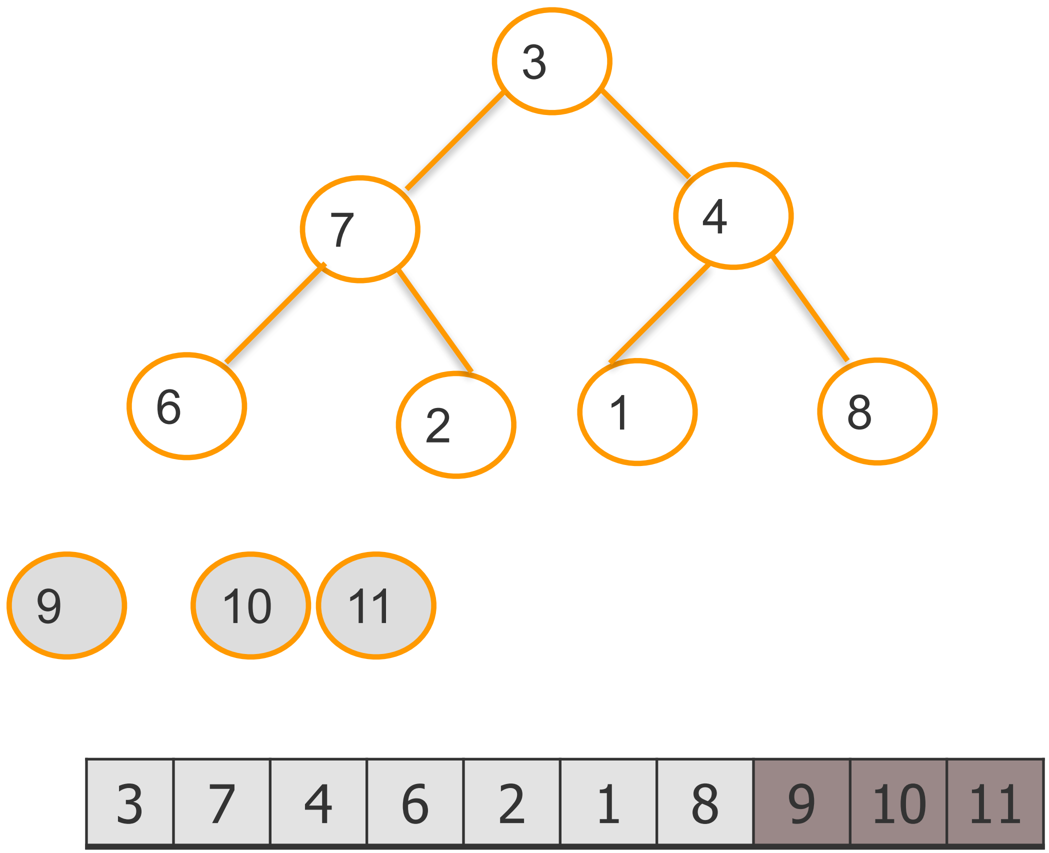
Exchange 8 and 3

```
Function Heapsort(A)
    #Create max heap
    Build_Max_Heap from unordered array A
    # Finish sorting
    for i = n downto 2 do
        exchange A[1] with A[i]
        discard node i from heap (decrement heap size)
        sift(A[1:i-1],1) because new root may violate max heap property
```

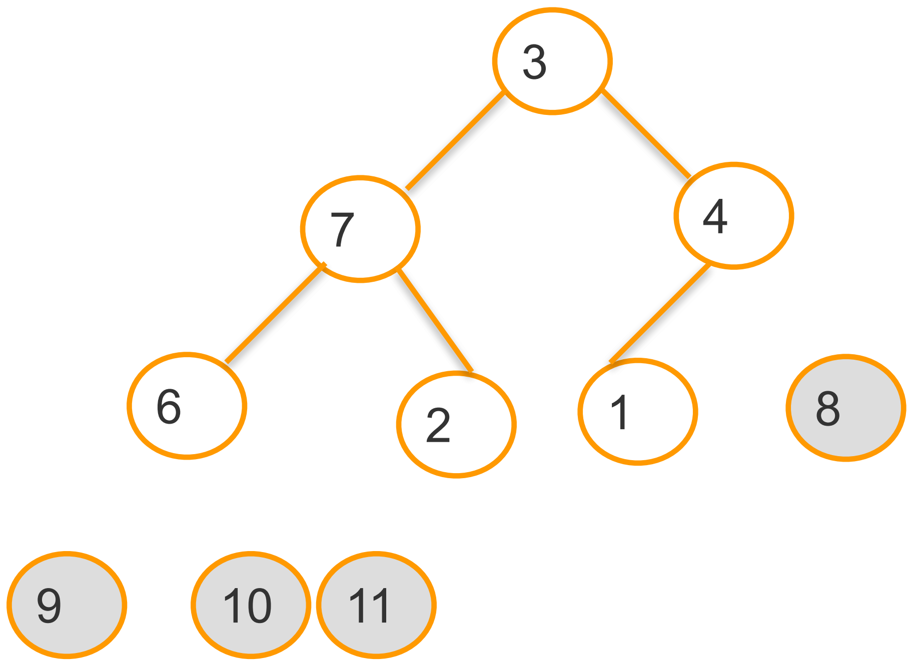


8	7	4	6	2	1	3	9	10	11

Remove 8 from the heap

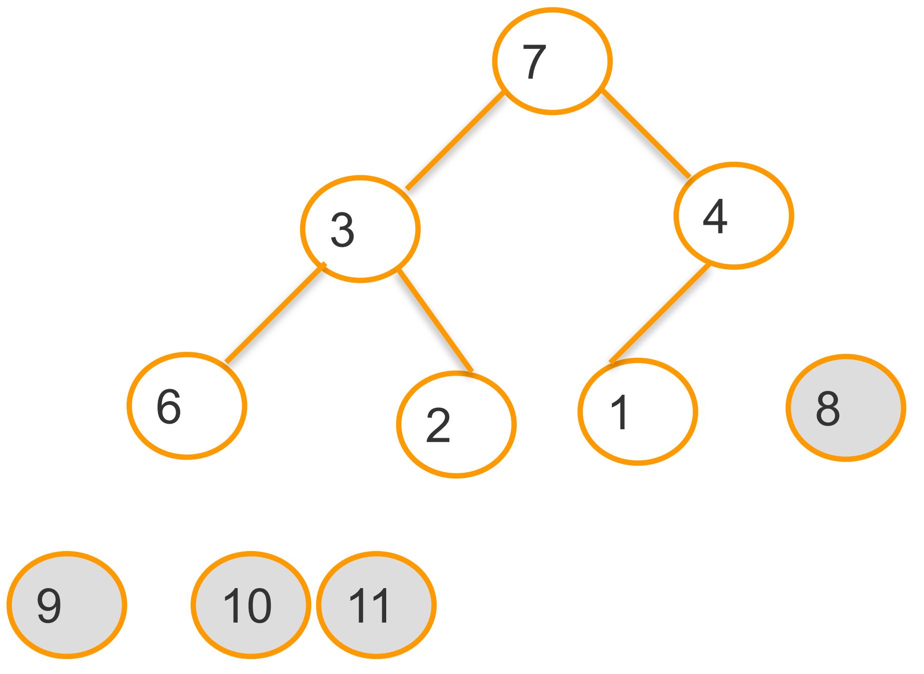


Exchange 3 and 7



3	7	4	6	2	1	8	9	10	11

Exchange 3 and 6



7	3	4	6	2	1	8	9	10	11

Exchange 1 and 7

Function Heapsort(A)

#Create max heap

Build_Max_Heap from unordered array A

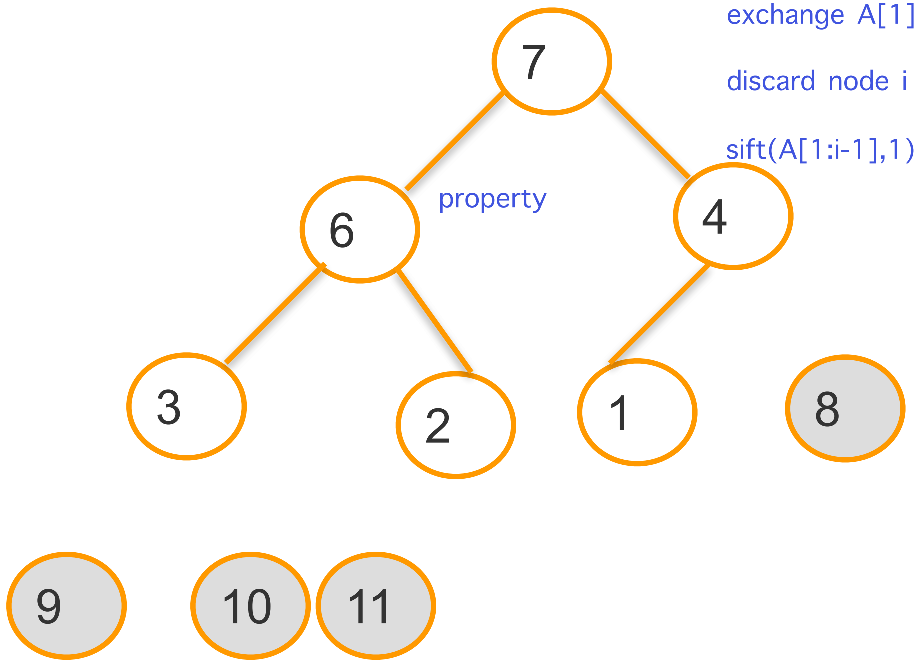
Finish sorting

for i = n downto 2 do

exchange A[1] with A[i]

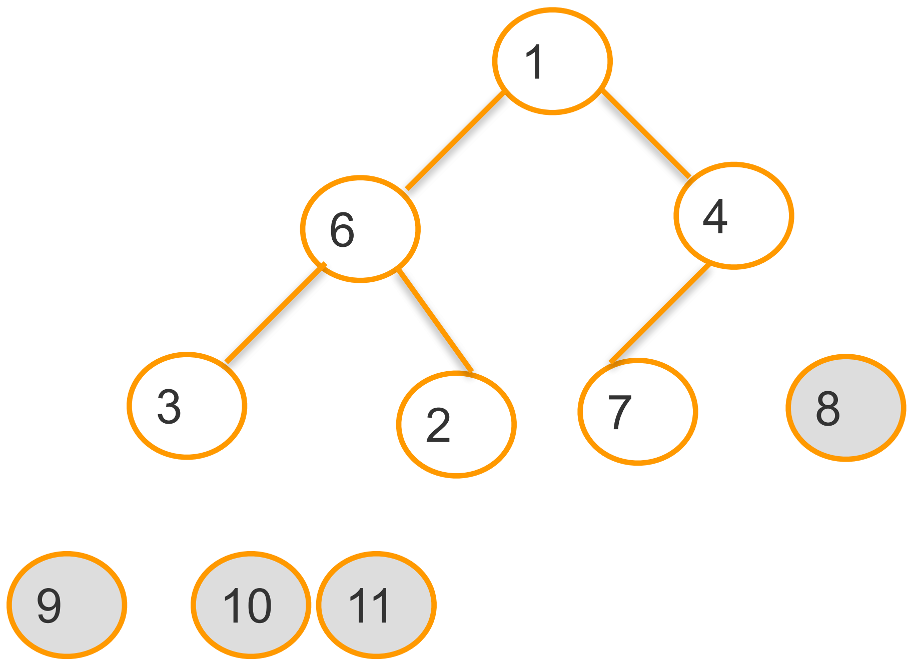
discard node i from heap (decrement heap size)

sift(A[1:i-1],1) because new root may violate max heap property



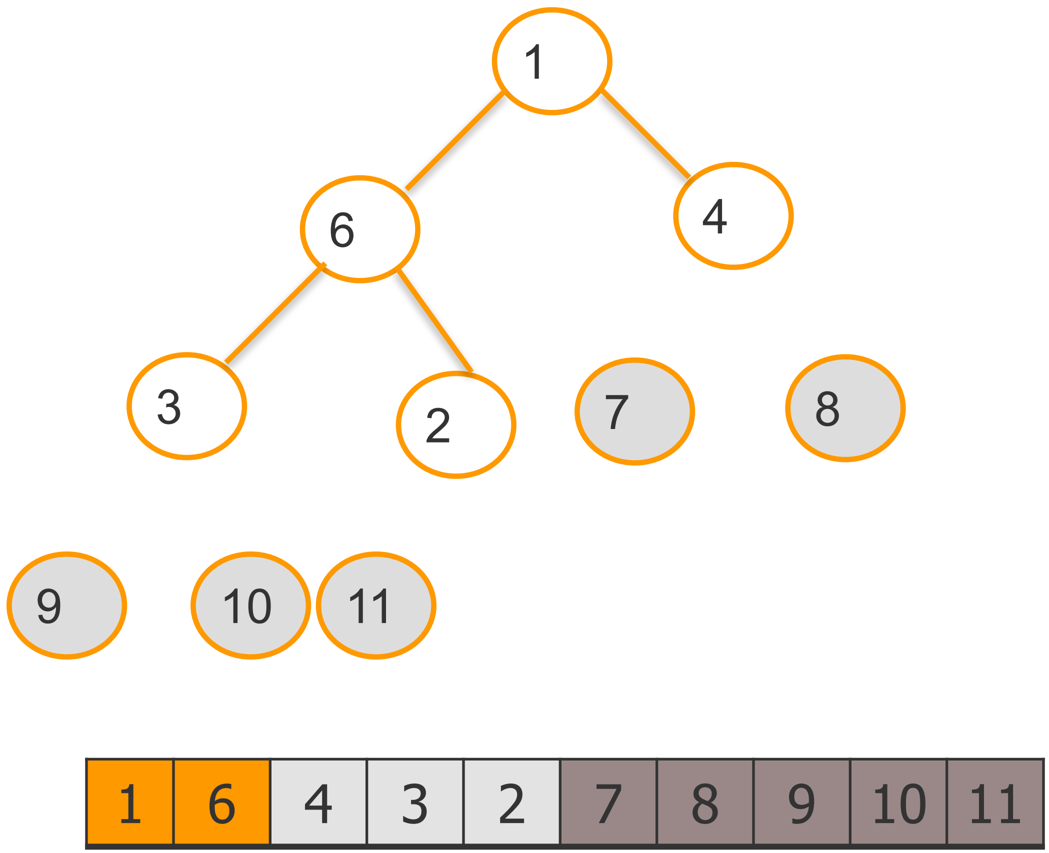
7	6	4	3	2	1	8	9	10	11

Remove 7 from the heap

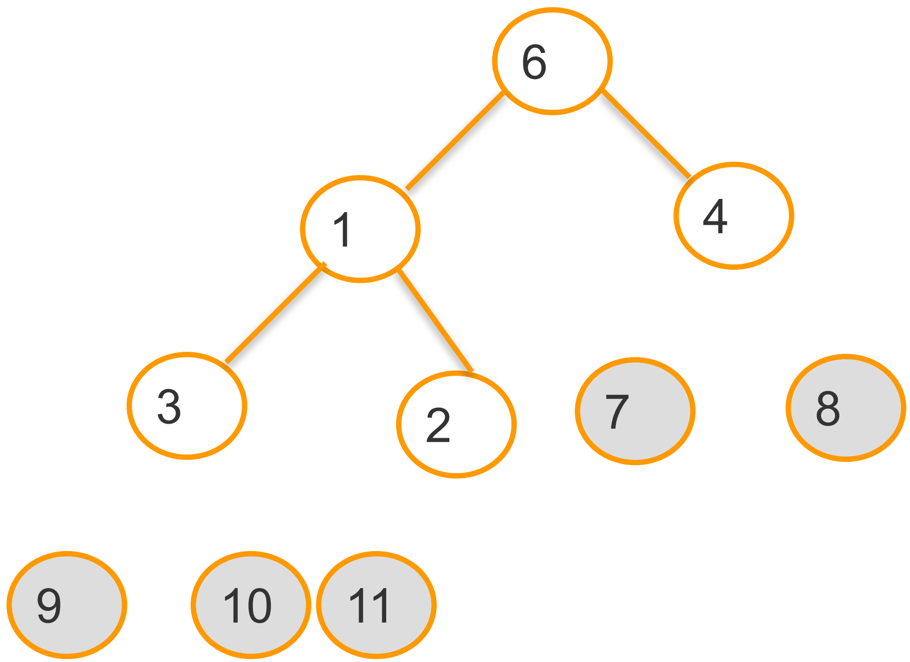


1	6	4	3	2	7	8	9	10	11

Exchange 1 and 6



Exchange 1 and 3



6	1	4	3	2	7	8	9	10	11

Exchange 6 and 2 and remove from the heap

Function Heapsort(A)

#Create max heap

Build_Max_Heap from unordered array A

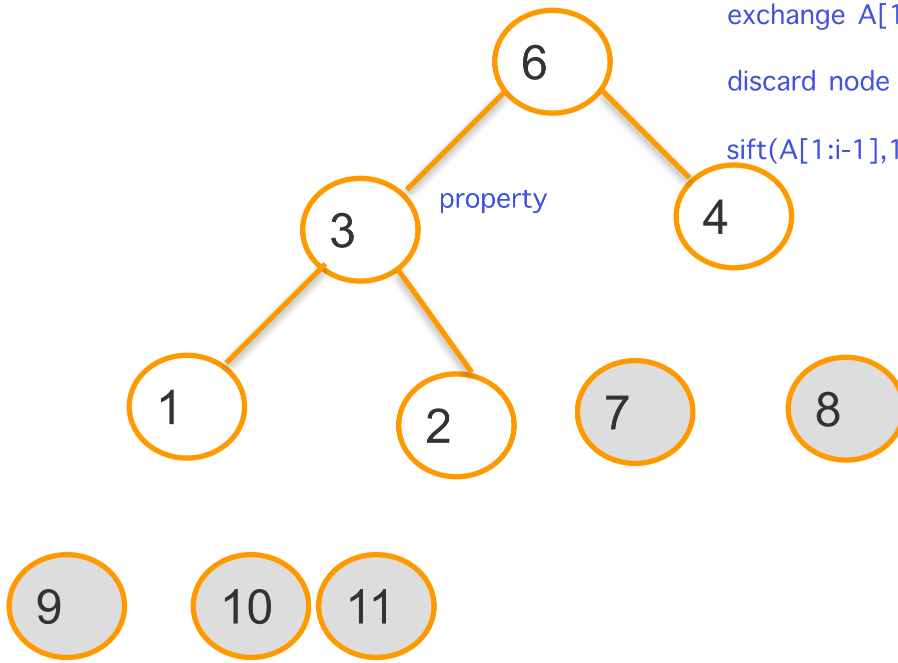
Finish sorting

for i = n downto 2 do

exchange A[1] with A[i]

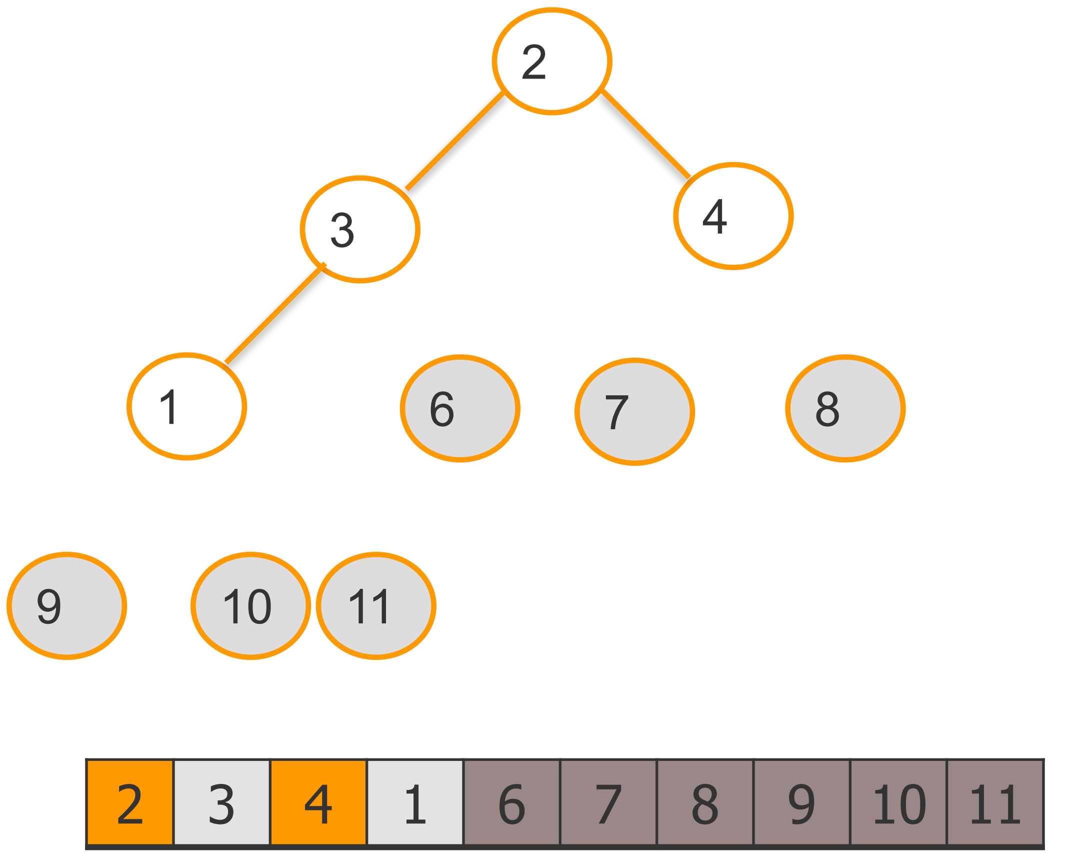
discard node i from heap (decrement heap size)

sift(A[1:i-1],1) because new root may violate max heap property



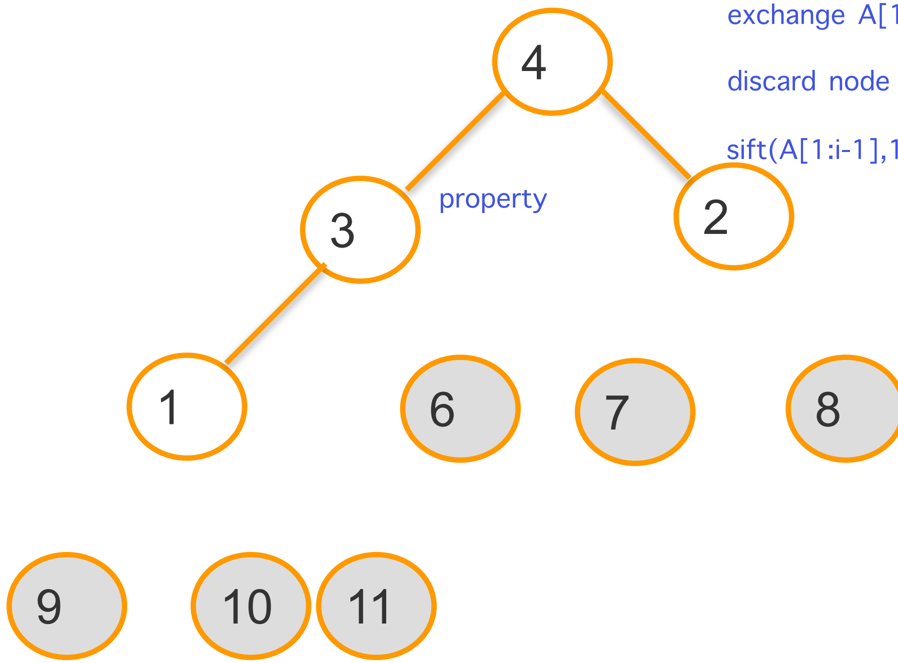
6	3	4	1	2	7	8	9	10	11

Exchange 4 and 2



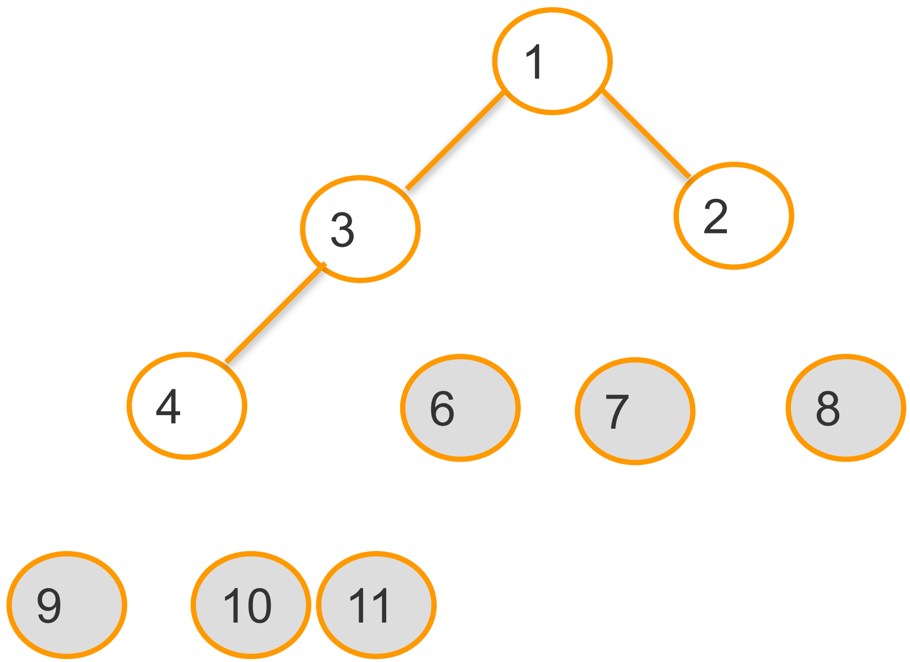
Exchange 4 and 1

```
Function Heapsort(A)
    #Create max heap
    Build_Max_Heap from unordered array A
    # Finish sorting
    for i = n downto 2 do
        exchange A[1] with A[i]
        discard node i from heap (decrement heap size)
        sift(A[1:i-1],1) because new root may violate max heap property
```



4	3	2	1	6	7	8	9	10	11

Remove 4, exchange 1 and 3



1	3	2	4	6	7	8	9	10	11

Exchange 2 and 3, and remove 3 from heap

Function Heapsort(A)

#Create max heap

Build_Max_Heap from unordered array A

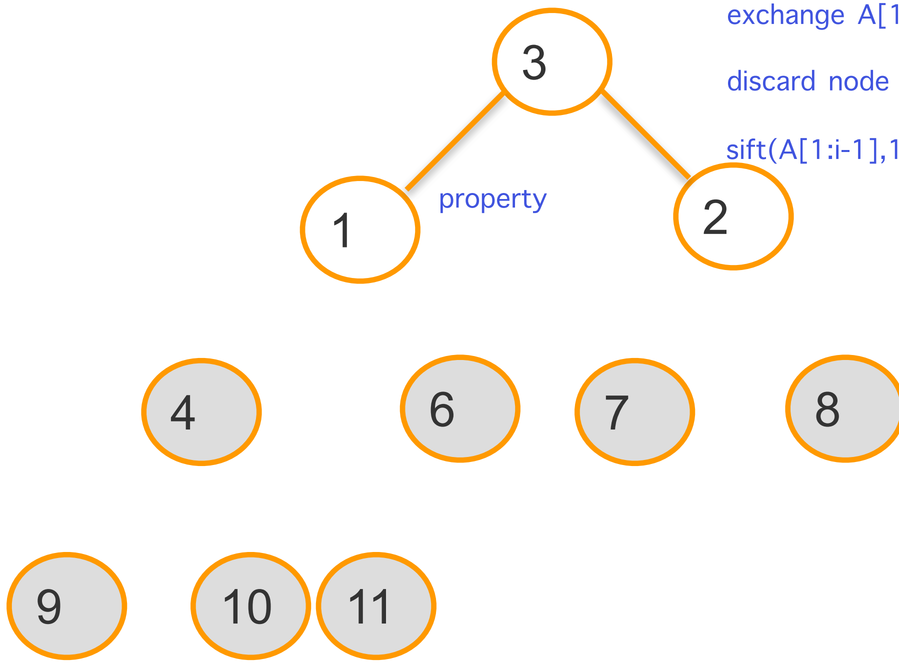
Finish sorting

for i = n downto 2 do

exchange A[1] with A[i]

discard node i from heap (decrement heap size)

sift(A[1:i-1],1) because new root may violate max heap property



3	1	2	4	6	7	8	9	10	11

Exchange 1 and 2 and remove from heap

Function Heapsort(A)

#Create max heap

Build_Max_Heap from unordered array A

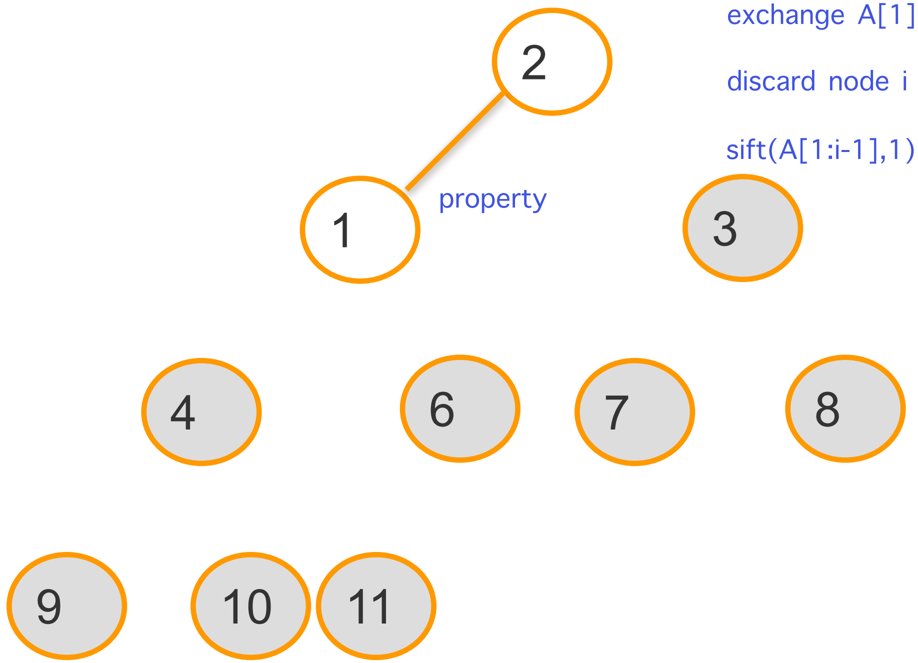
Finish sorting

for i = n downto 2 do

exchange A[1] with A[i]

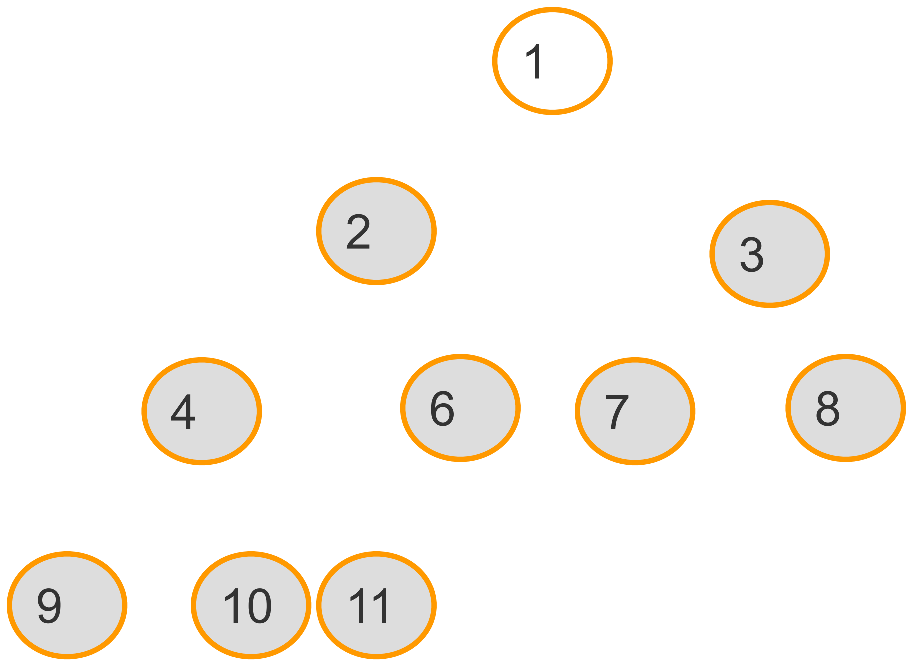
discard node i from heap (decrement heap size)

sift(A[1:i-1],1) because new root may violate max heap property



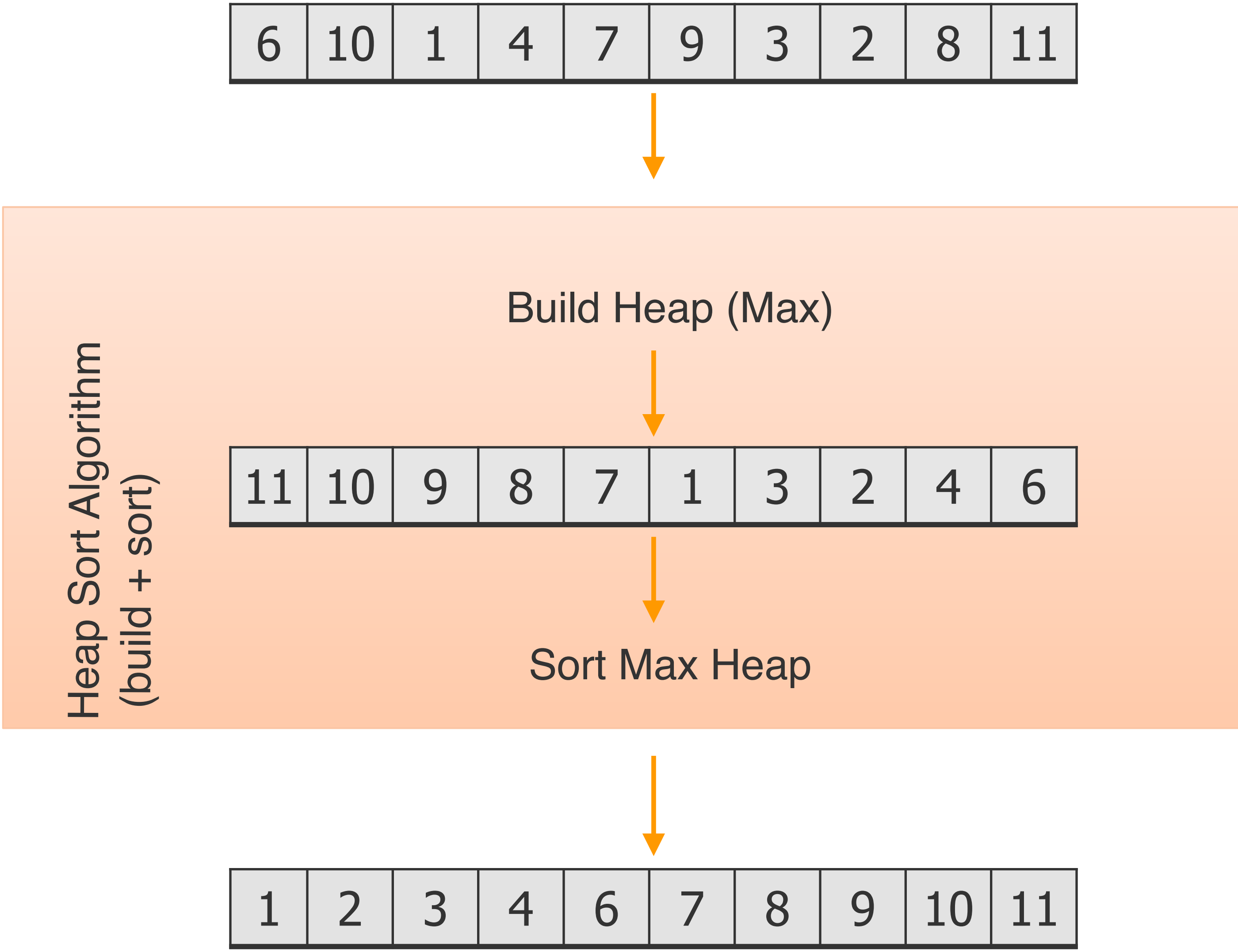
2	1	3	4	6	7	8	9	10	11

The array is sorted



1	2	3	4	6	7	8	9	10	11

Sorted Output



Heapsort Algorithm

```
Function Heapsort(A)
    #Create max heap
    Build_Max_Heap from unordered array A
    # Finish sorting
    for i = n downto 2 do
        exchange A[1] with A[i]
        discard node i from heap (decrement heap size)
        sift(A[1:i-1], 1) because new root may violate max heap property
```

Build Max Heap

```
Function Build_Max_Heap(A)
    set heap size to the length of the array
    for j= n/2 down to 1 do
         sift(A, j)
```

Heap

- The root of the tree is A[1], and given the index i of a node, we can easily compute the indices of its parent, left child, and right child:

```
function parent(i)
    return i/2

function left(i)
    return 2*i


function right(i)
    return 2 *i + 1
```

Max-Heapify (sift)

```
function sift(arr,i)
        n ← len(arr)
        l ← left(i)
        r ← right(i)

        if l <= n and arr[l] > arr[i] then
            largest ← l
        else:
            largest ← i

        if r <= n and arr[r] > arr[largest] then
            largest ← r

        if largest != i then
            arr[i] ↔ arr[largest]
            sift(arr, largest)
        return arr
```