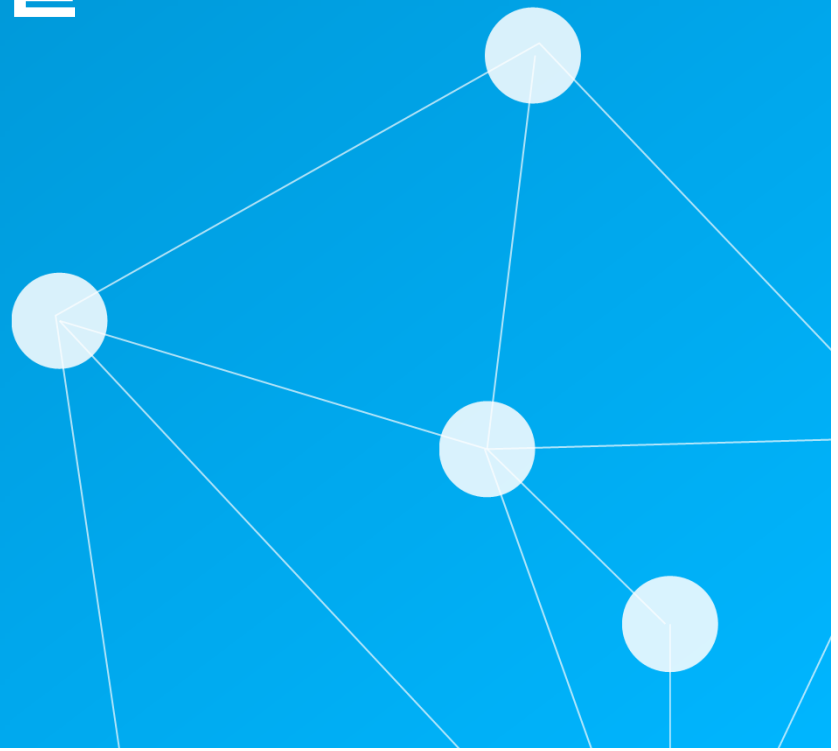
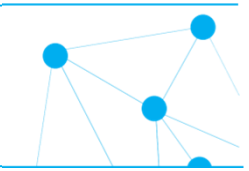

파이썬
자료구조

정렬



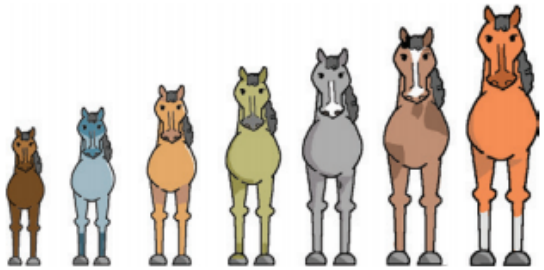
정렬이란?



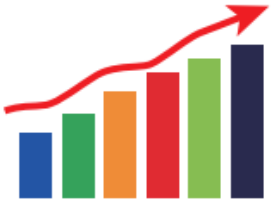
- 정렬이란?
- 용어들
- 정렬 알고리즘 종류

정렬이란?

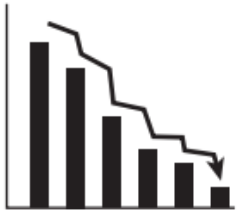
- 데이터를 순서대로 재배열하는 것
  - 가장 기본적이고 중요한 알고리즘
  - 비교할 수 있는 모든 속성들은 정렬의 기준이 될 수 있다
  - 오름차순(ascending order)과 내림차순(descending order)



경주마의 정렬(키 순)



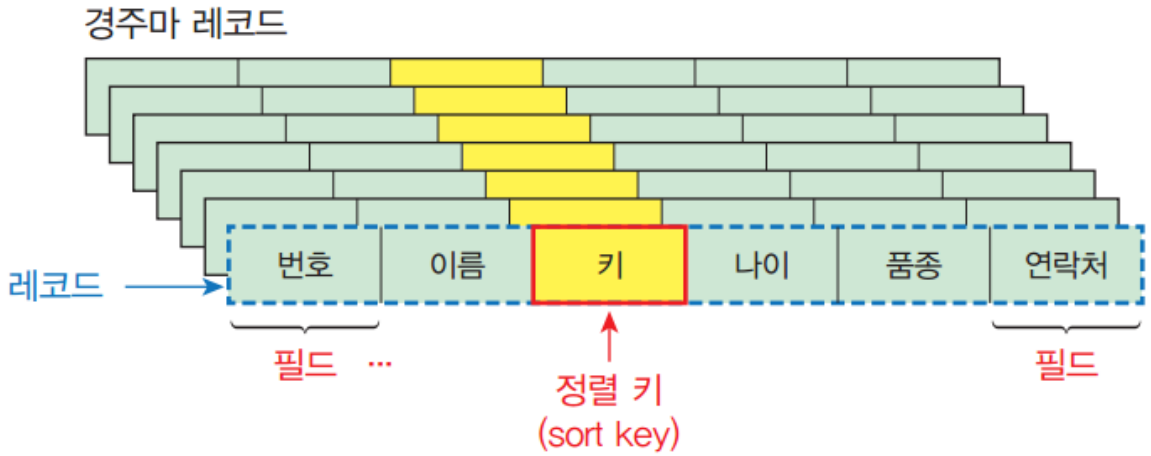
오름차순정렬



내림차순정렬

용어들

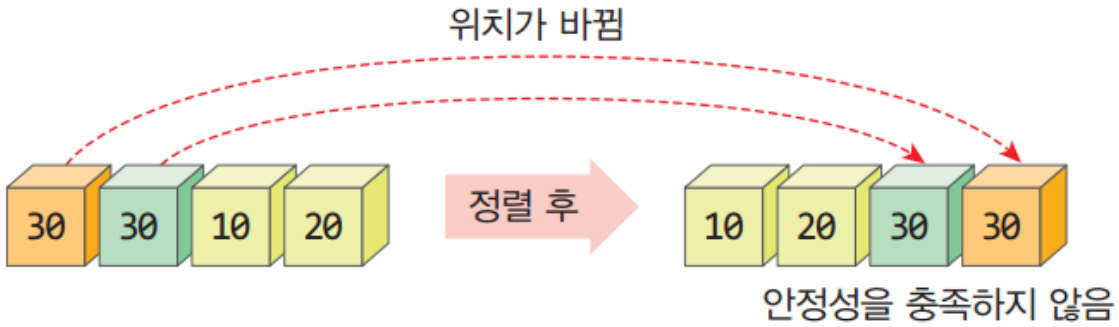
- 레코드: 정렬시켜야 될 대상
  - 여러 개의 필드(field)로 이루어짐
  - 정렬 키(sort key): 정렬의 기준이 되는 필드



- 정렬이란 레코드들을 키(key)의 순서로 재배열하는 것

정렬 알고리즘 종류

- 정렬 장소에 따른 분류
  - 내부(internal) 정렬: 모든 데이터가 메인 메모리
  - 외부(external) 정렬: 외부 기억 장치에 대부분의 레코드
- 단순하지만 비효율적인 방법
  - 삽입, 선택, 버블정렬 등
- 복잡하지만 효율적인 방법
  - 퀵, 힙, 병합, 기수정렬, 팀 등
- 정렬 알고리즘의 안정성(stability)



간단한 정렬 알고리즘

- 선택 정렬(selection sort)
- 삽입 정렬(insertion sort)
- 버블 정렬(bubble sort)

선택 정렬(selection sort)

제일작은 원소 찾아서 맨 앞으로보냄

- 오른쪽 리스트에 서 가장 작은 숫자를 선택하여 왼쪽 리스트의 맨 뒤로 이동하는 작업을 반복 선택정렬

정렬 된(왼쪽) 리스트	정렬 안 된(오른쪽) 리스트	설명
[ ]	[5,3,8,4,9,1,6,2,7]	초기상태
[1]	[5,3,8,4,9,6,2,7]	1선택 및 이동
[1,2]	[5,3,8,4,9,6,7]	2선택 및 이동
[1,2,3]	[5,8,4,9,6,7]	3선택 및 이동
…	…	4~8 선택 및 이동
[1,2,3,4,5,6,7,8,9]	[ ]	9선택 및 이동

선택 정렬 알고리즘

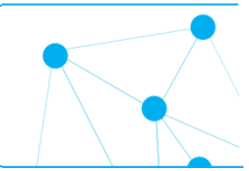

```
def selection_sort(A) :
    n = len(A)
    for i in range(n-1) :
        least = i;
        for j in range(i+1, n) :
            if (A[j]<A[least]) :
                least = j
        A[i], A[least] = A[least], A[i]
        printStep(A, i + 1);
```

- 시간 복잡도

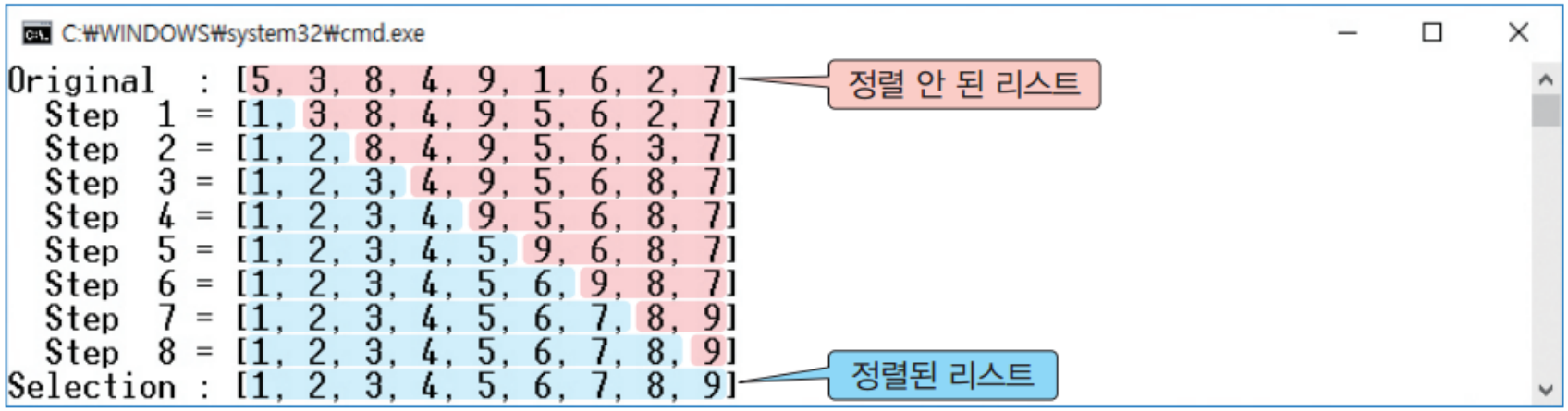
$$(n-1)+(n-2)+\cdots+1=n(n-1)/2=O(n^2)$$

  - 알고리즘이 간단, 자료 이동 횟수가 미리 결정됨

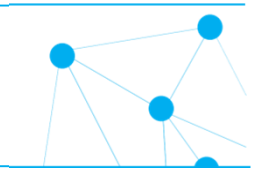
테스트 프로그램



```
data = [ 5, 3, 8, 4, 9, 1, 6, 2, 7 ]
print("Original  : ", data)
selection_sort(data)
print("Selection : ", data)
```



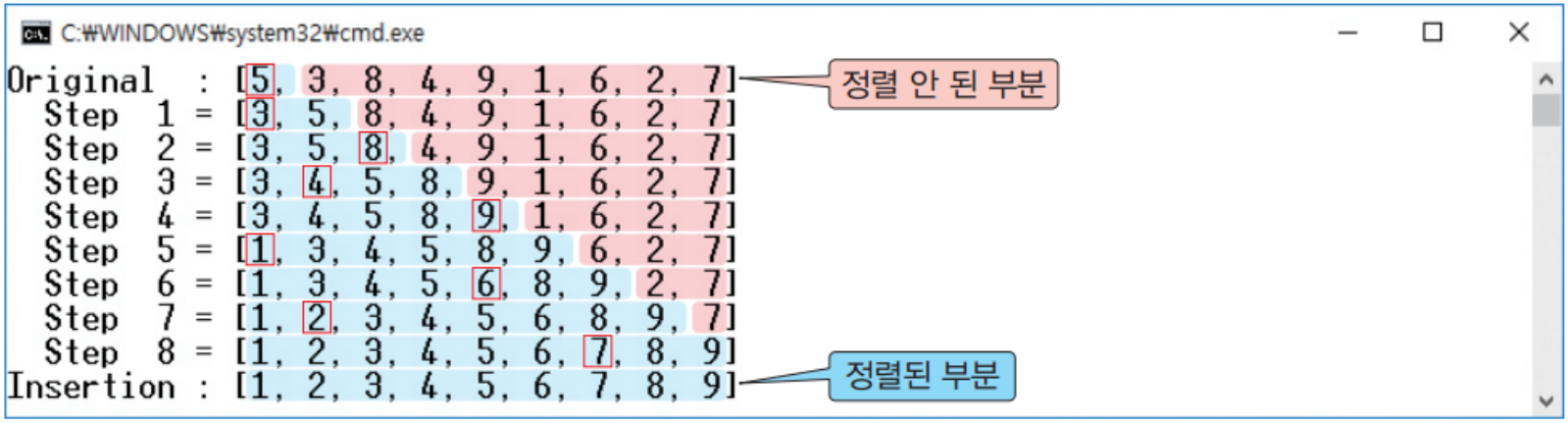
삽입 정렬(insertion sort)



- 정렬되어 있는 부분에 새로운 레코드를 올바른 위치에 삽입하는 과정 반복

삽입 정렬 알고리즘


```
def insertion_sort(A) :
    n = len(A)
    for i in range(1, n) :
        key = A[i]
        j = i-1
        while j>=0 and A[j] > key :
            A[j + 1] = A[j]
            j -= 1
        A[j + 1] = key
        printStep(A, i)
```



삽입 정렬 분석

- 복잡도 분석
  - 최선의 경우 O(n) : 이미 정렬되어 있는 경우: 비교: *n-1* 번
  - 최악의 경우 $O(n^2)$: 역순으로 정렬되어 있는 경우
    - 모든 단계에서 앞에 놓인 자료 전부 이동
    - 비교: $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$
    - 이동: $\frac{n(n-1)}{2} + 2(n-1) = O(n^2)$
  - 평균의 경우 $O(n^2)$
- 특징
  - 많은 이동 필요 → 레코드가 큰 경우 불리
  - 안정된 정렬방법
  - 대부분 정렬되어 있으면 매우 효율적

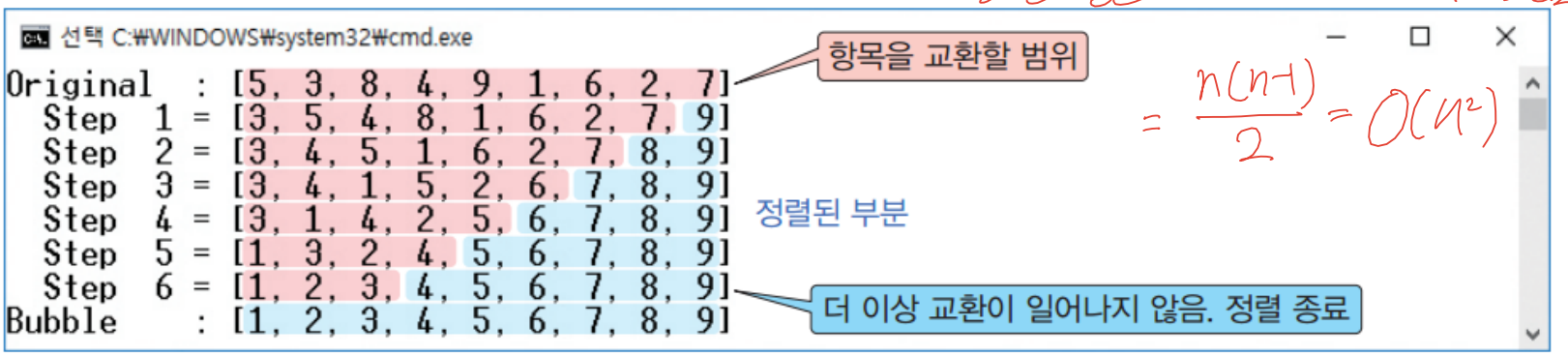
- 이미 정렬된 파일의 뒷부분에 소량의 신규 데이터를 추가하여 정렬하는 경우(입력이 거의 정렬된 경우) 우수한 성능을 보임
- 입력 크기가 작은 경우에도 매우 좋은 성능을 보임
  삽입 정렬은 재귀 호출을 하지 않으며, 프로그램도 매우 간단하기 때문
- 삽입 정렬은 합병정렬이나 퀵정렬과 함께 사용되어 실질적으로 보다 빠른 성능에 도움을 줌
  - 단, 이론적인 수행 시간은 향상되지 않음

버블 정렬 (bubble sort)

- 기본 전략
  - 인접한 2개의 레코드를 비교하여 순서대로 서로 교환
  - 비교-교환 과정을 리스트의 전체에 수행(스캔)
    - 한번의 스캔이 완료되면 리스트의 오른쪽 끝에 가장 큰 레코드
  - 끝으로 이동한 레코드를 제외하고 다시 스캔 반복

버블 정렬 알고리즘


```
def bubble_sort(A) :
    n = len(A)
    for i in range(n-1, 0, -1) :
        bChanged = False
        for j in range (i) :
            if (A[j]>A[j+1]) :
                A[j], A[j+1] = A[j+1], A[j]
                bChanged = True

        if not bChanged: break;
        printStep(A, n - i);
```



$= \frac{n(n-1)}{2} = O(n^2)$

버블정렬 분석

- 비교 횟수(최상, 평균, 최악의 경우 모두 일정)

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

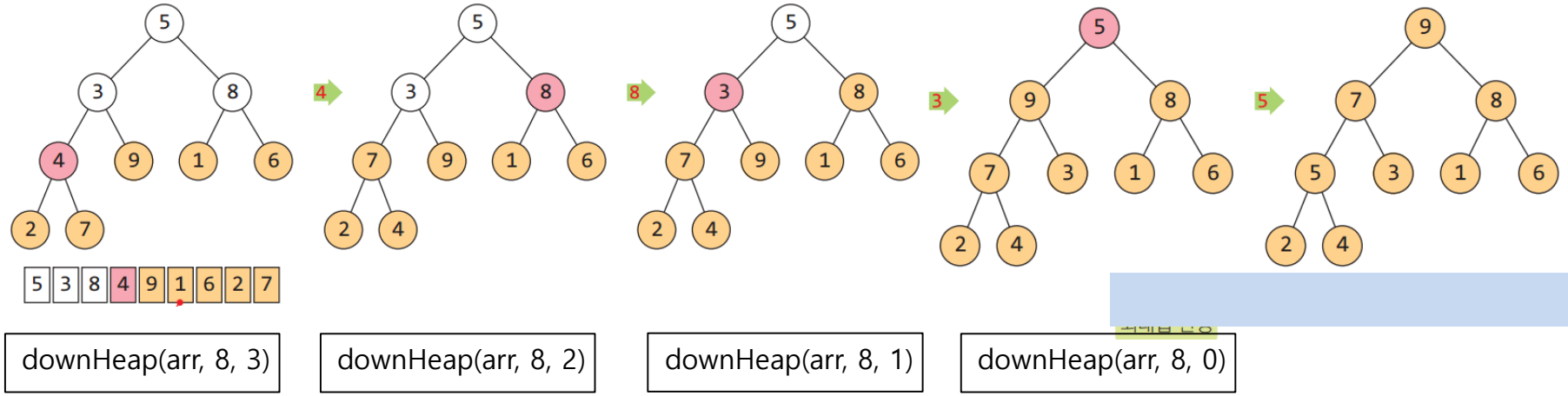
- 이동 횟수
  - 역순으로 정렬된 경우(최악): 이동 횟수 = 3 * 비교 횟수
  - 이미 정렬된 경우(최선의 경우) : 이동 횟수 = 0
  - 평균의 경우 : $O(n^2)$

- 레코드의 이동 과다
  - 이동연산은 비교연산 보다 더 많은 시간이 소요됨

힙 정렬

- 단계1: 리스트를 최대힙으로 만듦
- 단계2: 최대힙을 정렬된 리스트로 만듦

- Heapify: 최대 힙을 만드는 과정

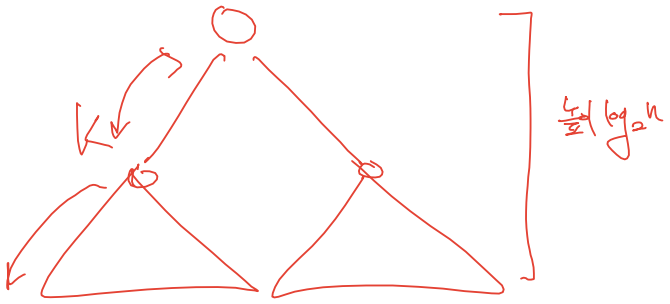


Step 1: 배열(리스트)을 최대힙으로 만듦

$O(n \log n)$

◆ ***downHeap*** 을 이용하여 리스트를 최대힙으로 만든다.

```
n = len(A)
    for i in range(n//2-1, -1, -1): // step 1
            downHeap(A,  i,  n);
```



0까지 감소함.

수행시간 = 높이 $\log_2 n$

$O(\frac{n}{2}(\log_2 n))$

18

Step 2: 최대힙을 정렬된 배열로 만듬

단계 1에서 만든 최대힙(배열)으로부터 다음 과정을 반복하여 정렬함 (n은 원소 개수)

last : 힙의 마지막 노드의 위치

Initially, the heap is in arr[0 .. *last*], where *last* = *n* – 1.

1) 힙의 루트에 있는 원소 arr[0] (즉, 최대 원소)와 마지막 원소 arr[*last*]를 교환한다.
2) 힙의 크기를 1 줄인다. 루트가 0인 semiheap에 대하여 ***downHeap***를 호출하여 힙으로 만든다.
3) *last* = 0이 될 때까지 위의 단계 1-2를 반복한다.

Step 2: 최대힙을 정렬된 배열로 만듬

```
heap_size = n
for last in range(n-1, 0, -1)
# move the largest item in the A[0 .. last], to the
#  beginning of the sorted region, A[last+1 .. n–1], and
#  increase the sorted region. That is,  swap A[0] and A[ last ].
    A[0], A[last] = A[last], hA[0]
# transform the semiheap in A[0 .. Last-1] into a heap.
    heap_size -= 1;
    downHeap(A,  0,  heap_size )
```

$\leq \log_2 n$

Step 2: 최대힙을 정렬된 배열로 만듬

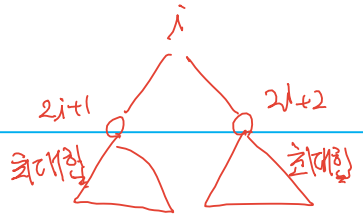
단계 1에서 만든 최대힙(배열)으로부터 다음 과정을 반복하여 정렬함 (n은 원소 개수)

last : 힙의 마지막 노드의 위치

Initially, the heap is in arr[0 .. *last*], where *last* = *n* – 1.

1) 힙의 루트에 있는 원소 arr[0] (즉, 최대 원소)와 마지막 원소 arr[*last*]를 교환한다.
2) 힙의 크기를 1 줄인다. 루트가 0인 semiheap에 대하여 ***downHeap***를 호출하여 힙으로 만든다.
3) *last* = 0이 될 때까지 위의 단계 1-2를 반복한다.

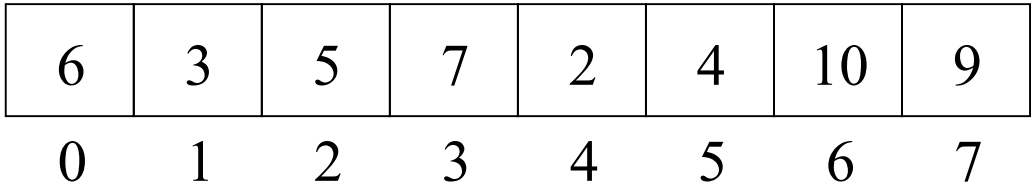
리스트를 힙으로 만듬

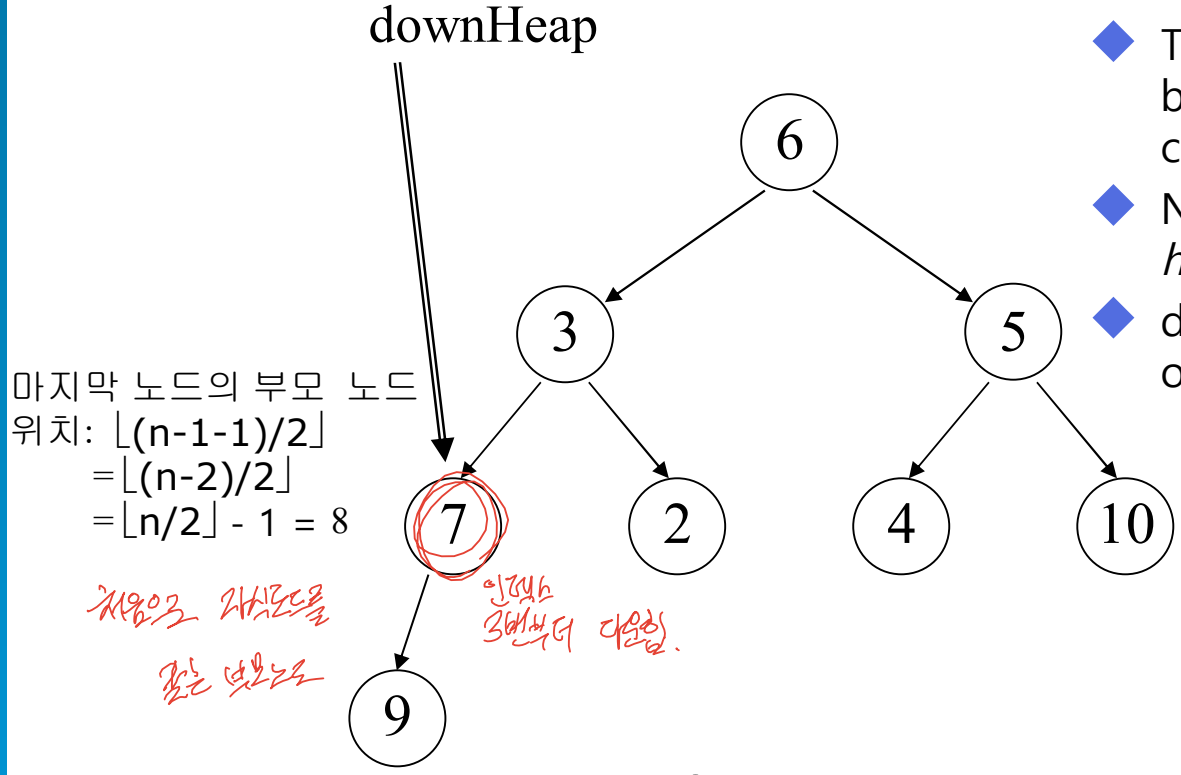


```
def downHeap(arr, n, i):  # heapify(혹은 rebuildHeap)라고도 함
    if n == 0:
        return None
    current = i
    value = arr[i]  # 노드 i의 값을 value에 저장
    while (2*current + 1 < n):   # curren가 leaf가 아니면
# 두 자식 노드 중 큰 값의 노드를 largerChild
        largerChild = 2*current + 1
        if (largerChild + 1) < n and arr[largerChild + 1] > arr[largerChild]:
            largerChild += 1

        if value < arr[largerChild]: # largerChild의 값이 크면
            arr[current] = arr[largerChild]
            current = largerChild          # current를 largerChild로 내림
        else:
            break

    arr[current] = value

def makeHeap(arr):
    n = len(arr)
    for i in range(n//2 -1, -1, -1):
        downHeap(arr, n, i)
```

```
def heapSort(arr):
    n = len(arr)
# 단계 1
    makeHeap(arr)
# 단계 2
    for last in range(n-1, 0, -1):
        arr[last], arr[0] = arr[0], arr[last]
        downHeap(arr, i, 0)
```

Step 1: Transform an Array Into a Heap Using downHeap - Example

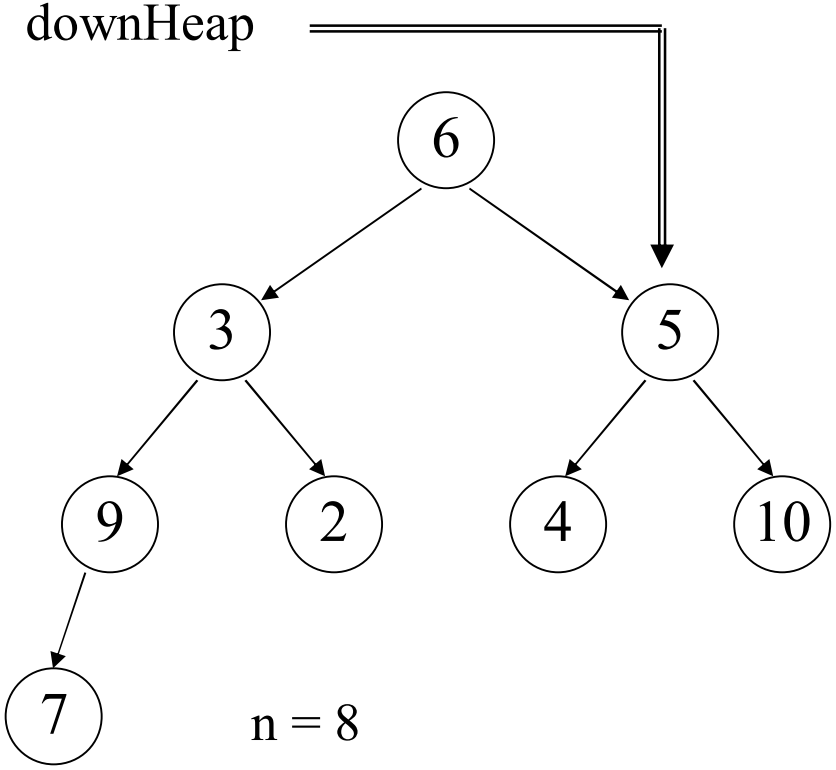




- The items in the array, above, can be considered to be stored in the complete binary tree shown at left.
- Note that leaves 9, 10, 4, and 2 are *heaps.*
- downHeap is invoked on the *parent* of the last node (=9) in the array.

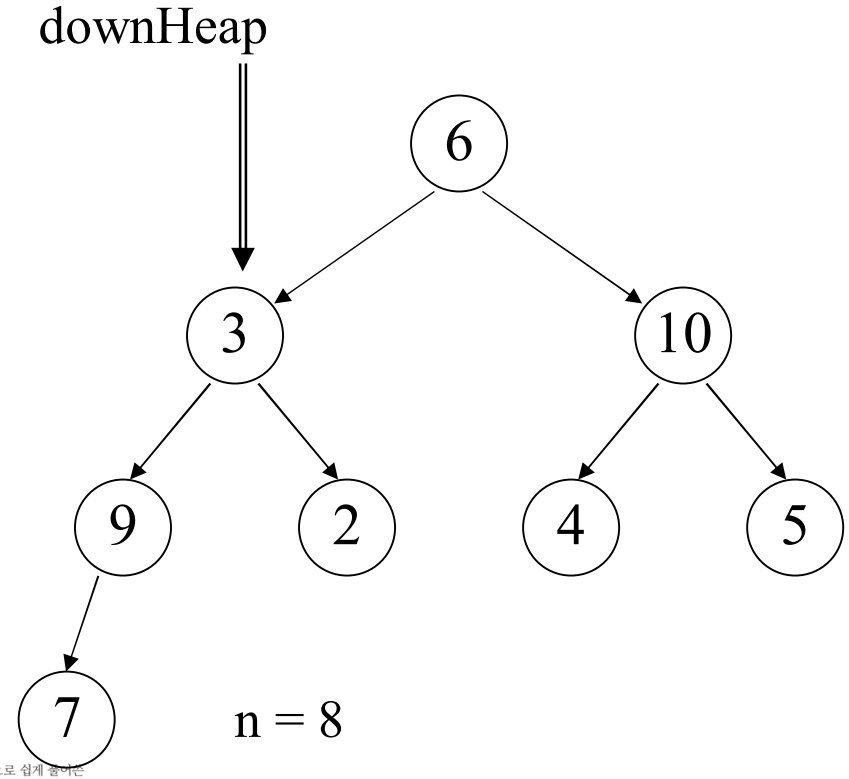
Step 1: Transform an Array Into a Heap Using downHeap - Example (Cont'd)

6	3	5	9	2	4	10	7
0	1	2	3	4	5	6	7



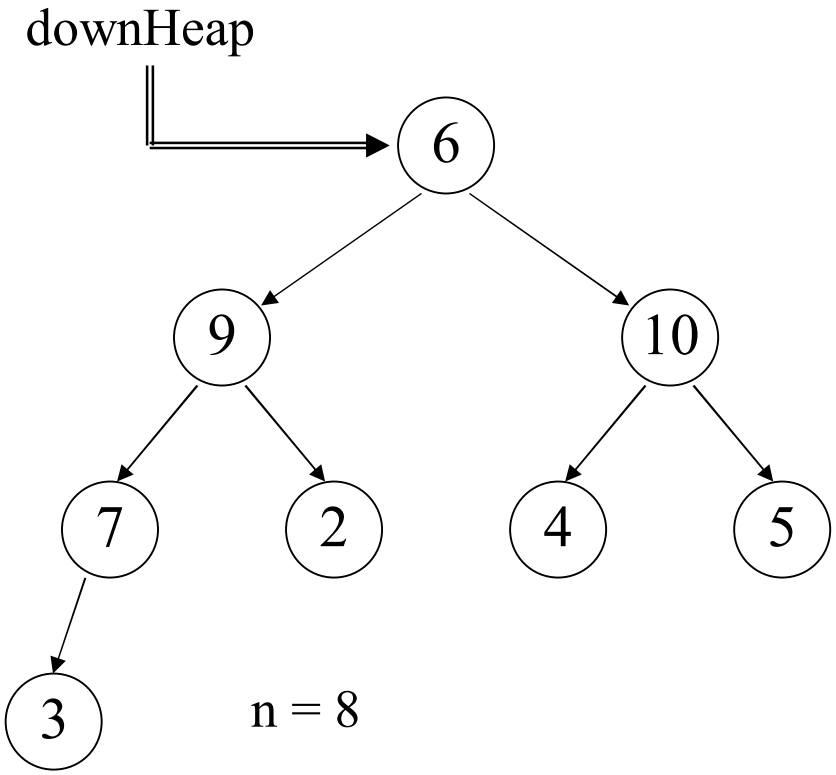
Step 1: Transform an Array Into a Heap Using downHeap - Example (Cont'd)

6	3	10	9	2	4	5	7
0	1	2	3	4	5	6	7



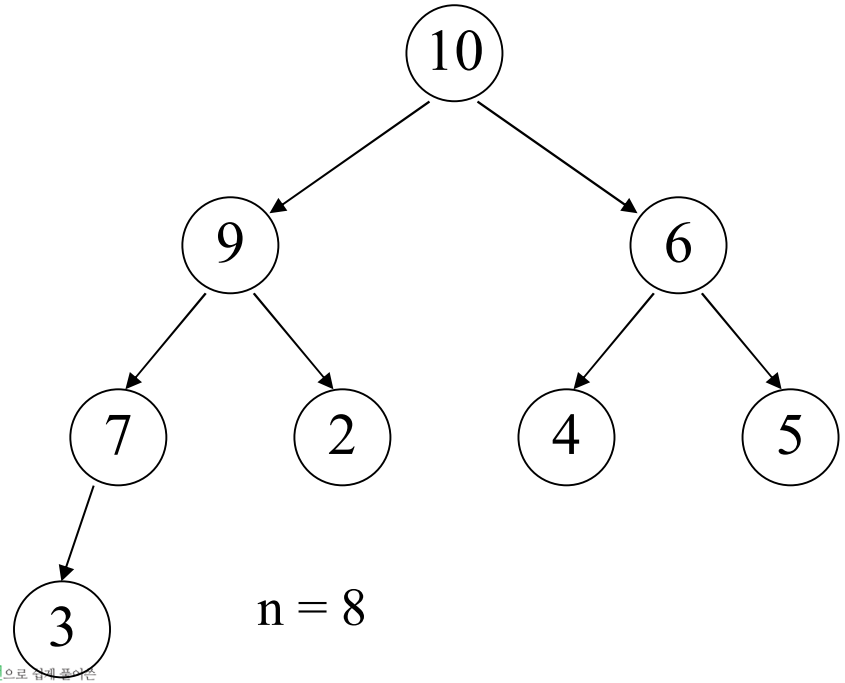
Step 1: Transform an Array Into a Heap Using downHeap - Example (Cont'd)

6	9	10	7	2	4	5	3
0	1	2	3	4	5	6	7



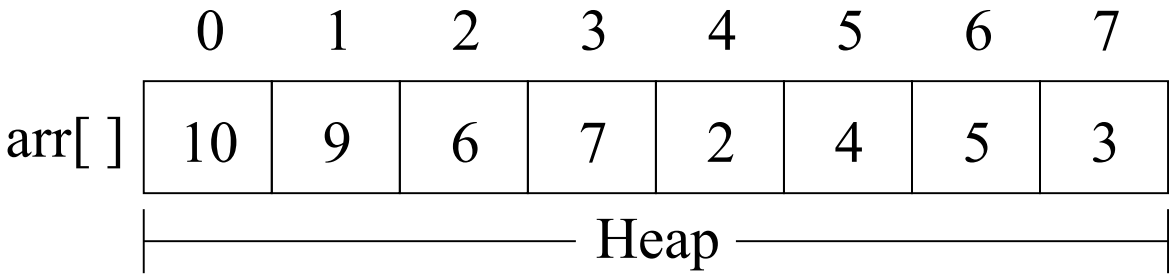
Step 1: Transform an Array Into a Heap Using downHeap - Example (Cont'd)

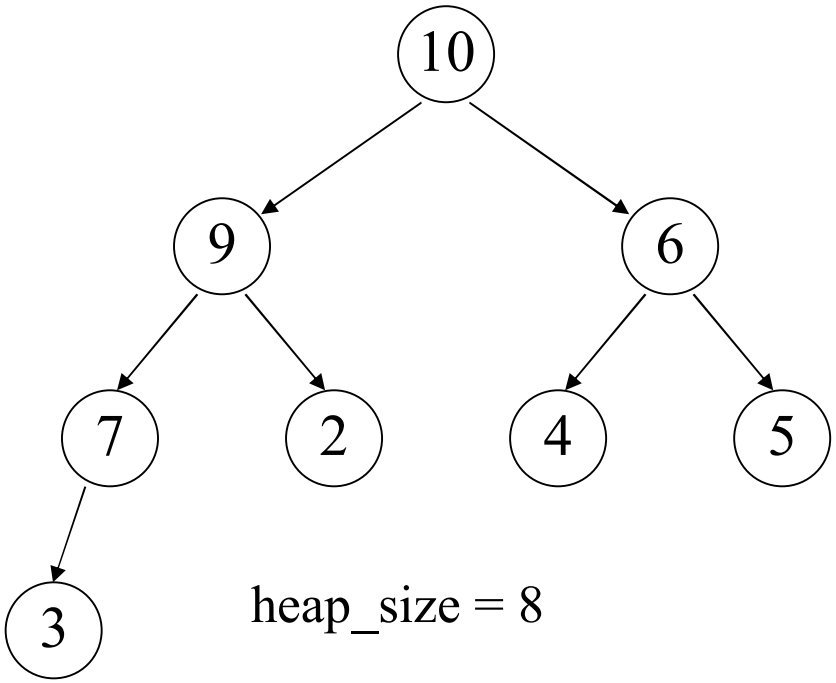
10	9	6	7	2	4	5	3
0	1	2	3	4	5	6	7



- Note that node 10 is now the root of a *heap*.
- The transformation of the *array* into a *heap* is complete.

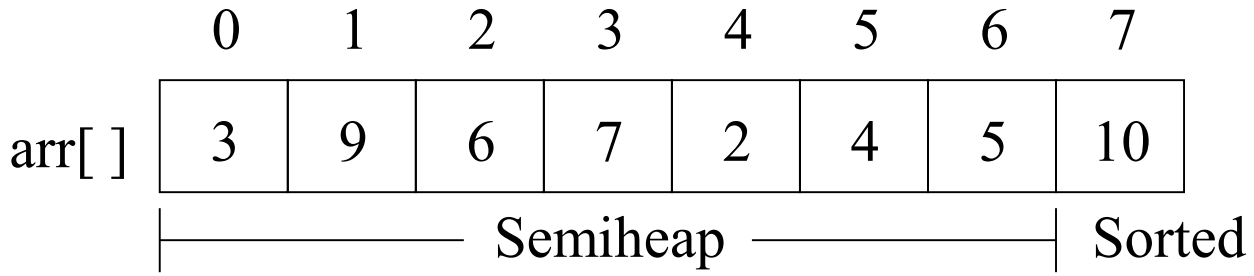
Step 2: Transform a Heap Into a Sorted Array: *Example*

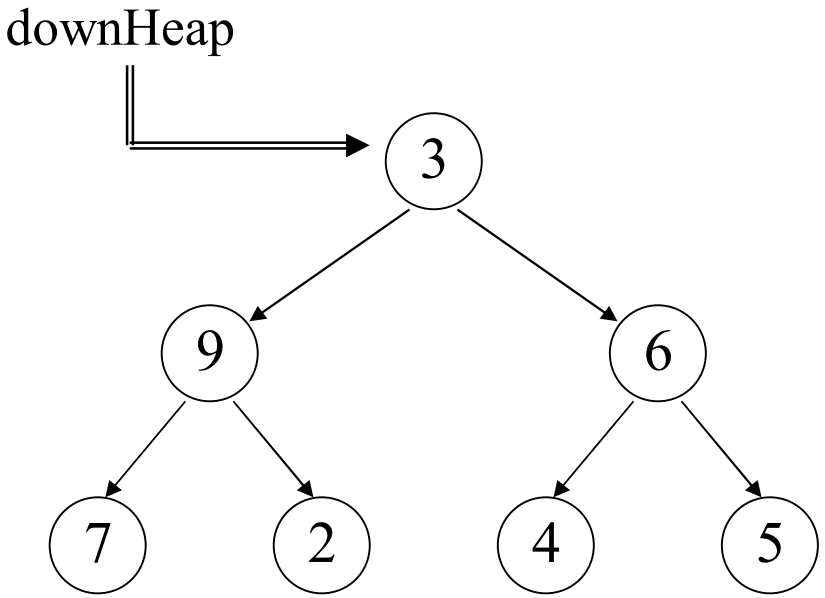




- We start with the heap that we formed from an unsorted array.
- The heap is in arr[0..7] and the sorted region is empty.
- We move the largest item in the heap to the beginning of the sorted region by swapping arr[0] with arr[7].

Step 2: Transform a Heap Into a Sorted Array: *Example* (Cont'd)

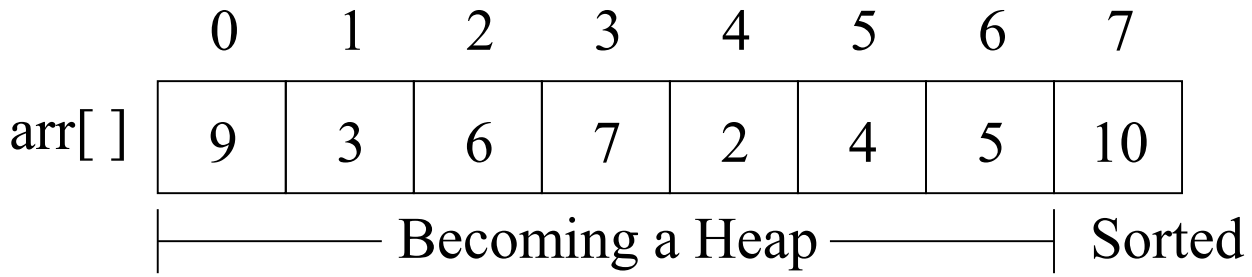


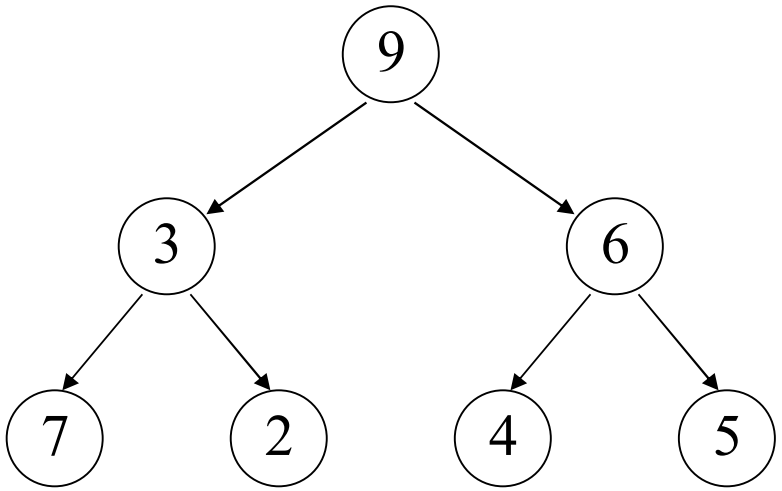


heap_size = 7

- arr[0..6] now represents a semiheap.
- arr[7] is the sorted region.
- Invoke downHeap on the semiheap rooted at arr[0].

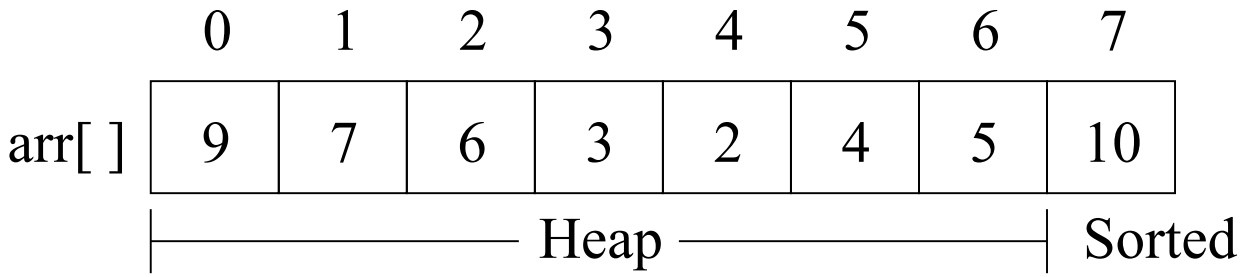
Step 2: Transform a Heap Into a Sorted Array: *Example* (Cont'd)

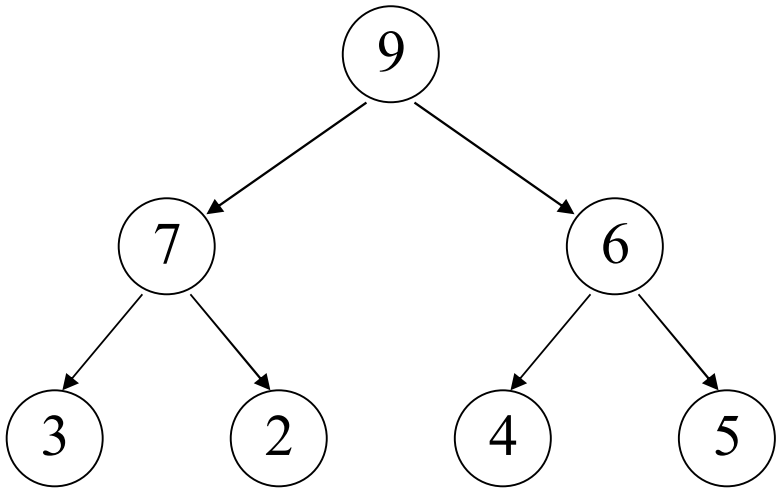




heap_size = 7

Step 2: Transform a Heap Into a Sorted Array: *Example* (Cont'd)

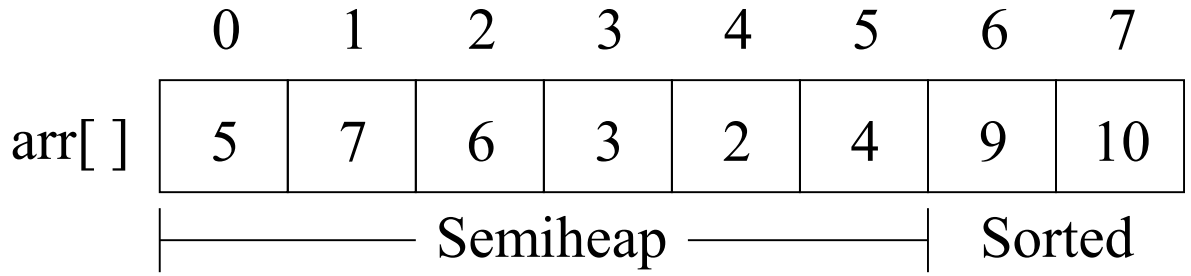


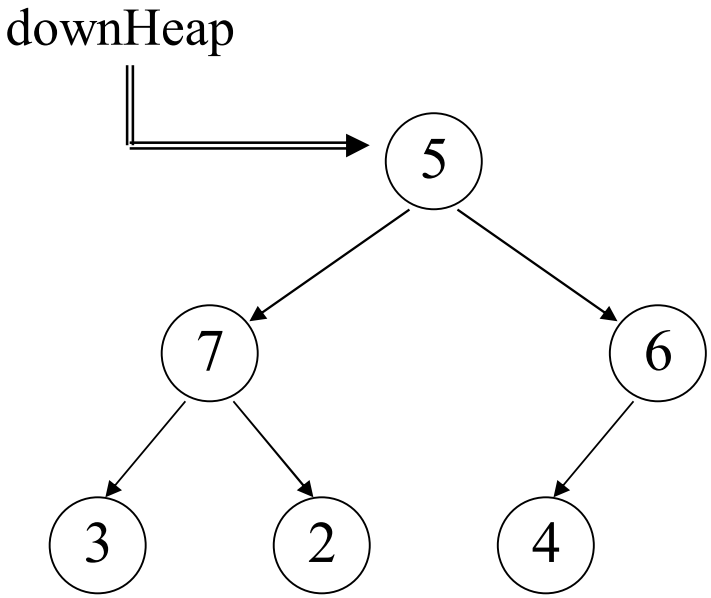


heap_size = 7

- arr[0] is now the root of a heap in arr[0..6].
- We move the largest item in the heap to the beginning of the sorted region by swapping arr[0] with arr[6].

Step 2: Transform a Heap Into a Sorted Array: *Example* (Cont'd)

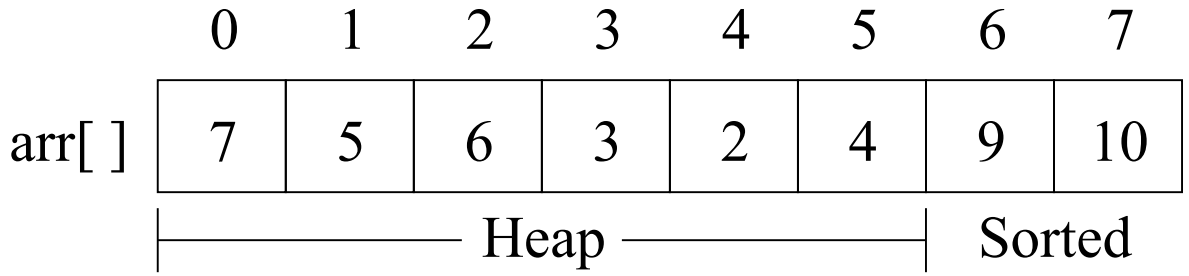


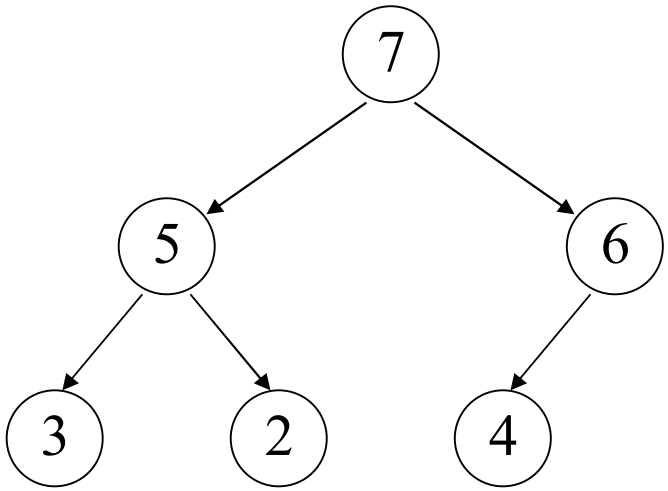


heap_size = 6

- arr[0..5] now represents a semiheap.
- arr[6..7] is the sorted region.
- Invoke downHeap on the semiheap rooted at arr[0].

Step 2: Transform a Heap Into a Sorted Array: *Example* (Cont'd)

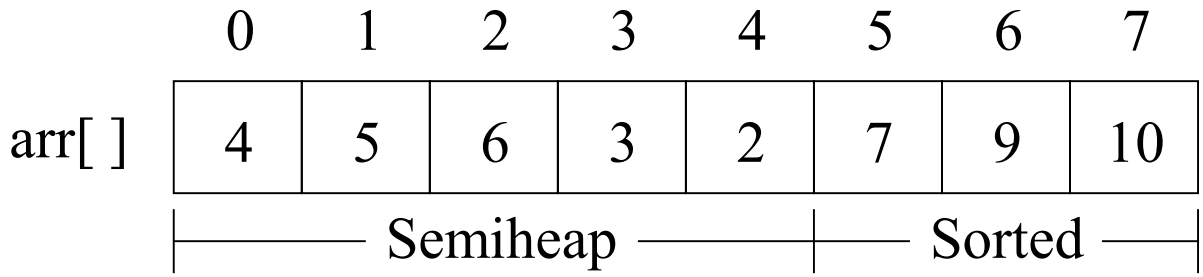


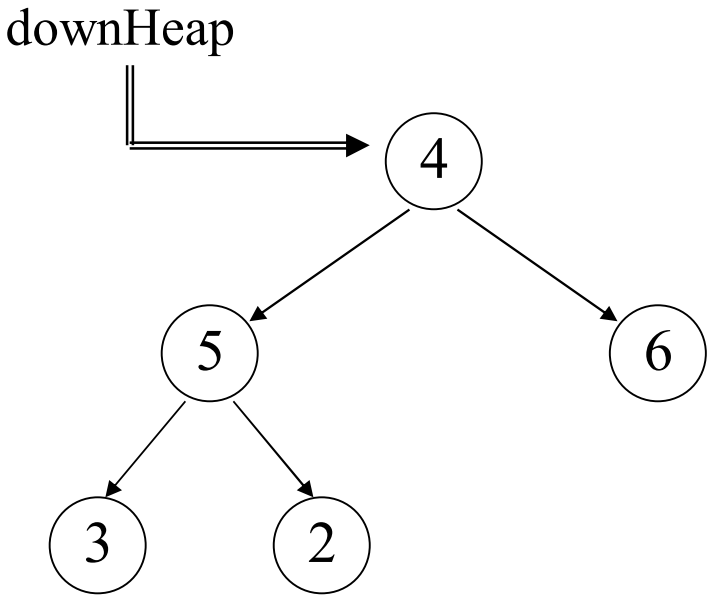


heap_size = 6

- arr[0] is now the root of a heap in arr[0..5].
- We move the largest item in the heap to the beginning of the sorted region by swapping arr[0] with arr[5].

Step 2: Transform a Heap Into a Sorted Array: *Example* (Cont'd)

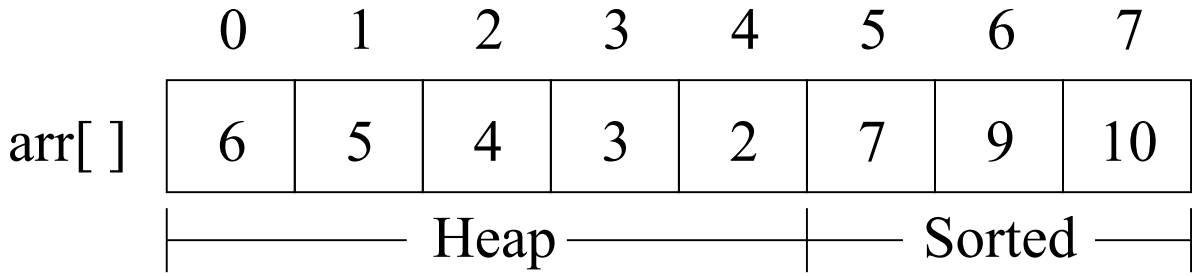


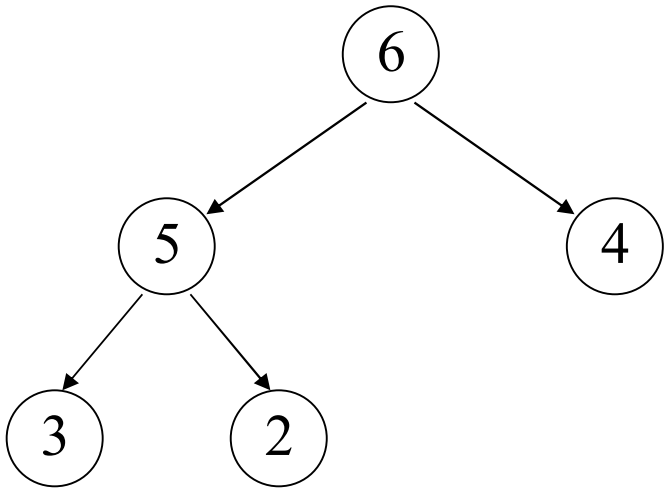


heap_size = 5

- arr[0..4] now represents a semiheap.
- arr[5..7] is the sorted region.
- Invoke downHeap on the semiheap rooted at arr[0].

Step 2: Transform a Heap Into a Sorted Array: *Example* (Cont'd)





heap_size = 5

- arr[0] is now the root of a heap in arr[0..4].
- We move the largest item in the heap to the beginning of the sorted region by swapping arr[0] with arr[4].

Heapsort 수행시간 분석

단계 1:

. 대략적인 수행시간

downHeap을 대략 n/2번 수행

downHeap 수행시간: O(log n)

단계 1의 수행시간: O(n log n)

. 보다 정확한 수행시간 분석

O(n)

단계 2:

- downHeap(A, 0, heap_size) : O(log n) 시간
- downHeap(A, 0, hezp_size)을 (*n*-*1*)번 호출
- step 2의 수행시간은 O(*n* log *n*)

단계 1 + 단계 2: heap sort의 수행시간은 O(n log n)이다.