COSE213: Data Structure

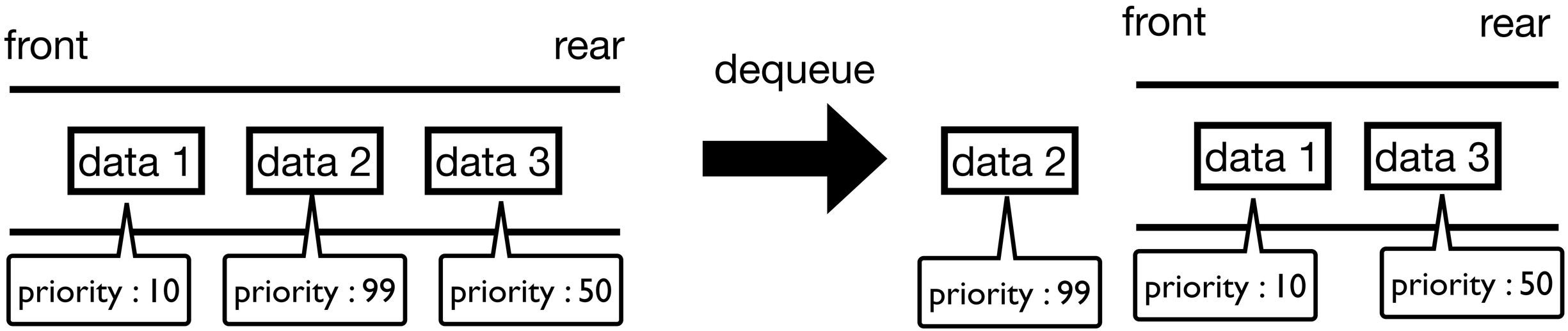
Lecture 10 - 힙 (Heap)

Minseok Jeon

2024 Fall

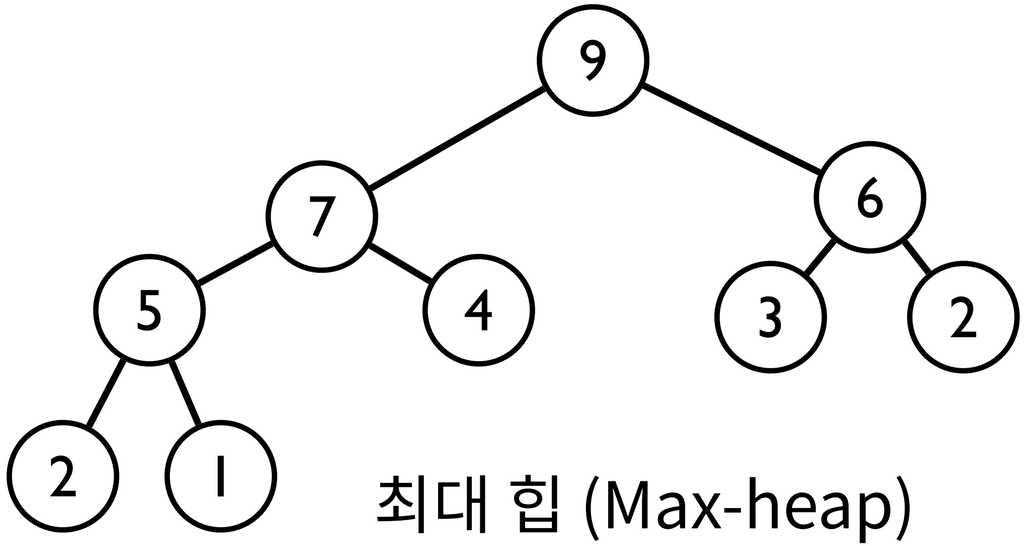
문제: 우선순위 큐 (Priority Queue)

- 우선순위가 높은 데이터가 먼저 처리되어야 하는 경우

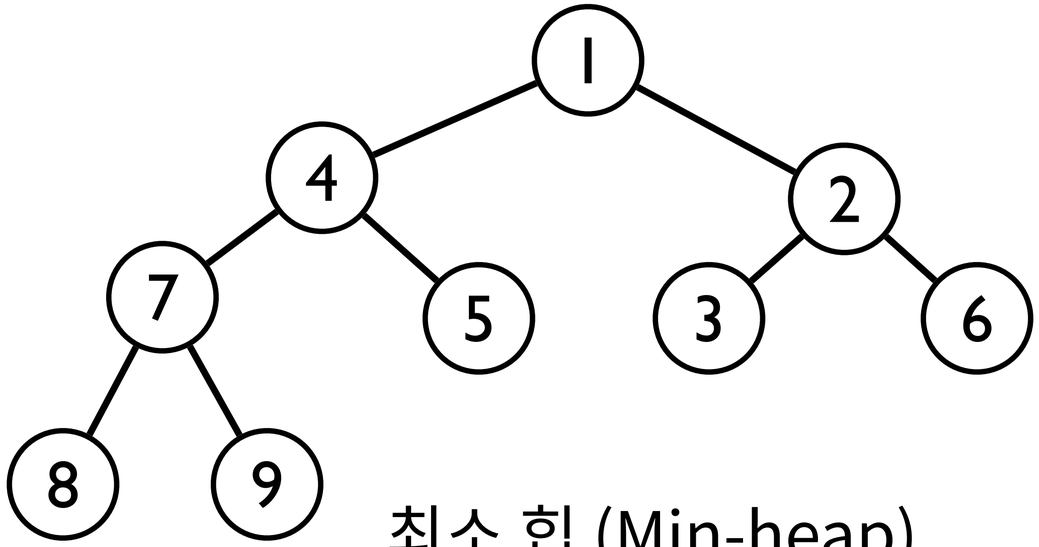


해결책: 힙(Heap) 자료구조

- 최대 힙 (Max-heap): 부모 노드의 키 값이 자식노드들의 키값보다 항상 크거나 같음
  - 루트 노드가 가장 큰 키값을 가짐
- 최소 힙 (Min-heap): 부모 노드의 키 값이 자식노드들의 키값보다 항상 작거나 같음
  - 루트 노드가 가장 작음 키값을 가짐
- 힙 자료구조는 완전 이진트리임



최대 힙 (Max-heap)

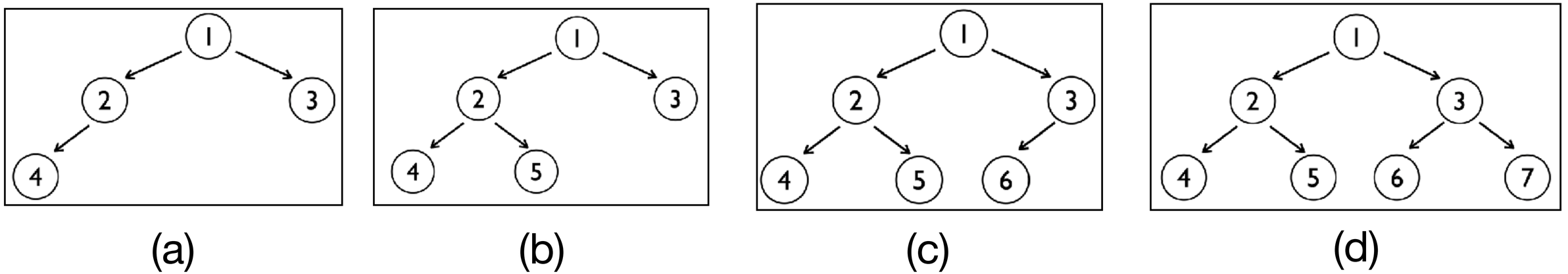


최소 힙 (Min-heap)

완전 이진트리 (Complete Binary Tree)

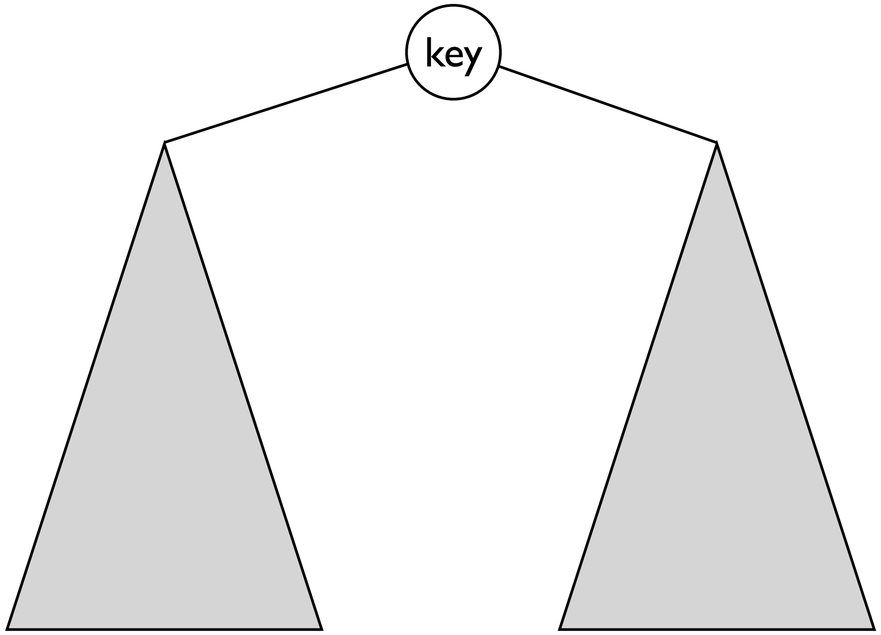
(1) 높이가 k인 트리에서 레벨 1부터 $k-1$까지는 노드가 모두 채워져 있고

(2) 마지막 레벨 k에서는 왼쪽부터 오른쪽으로 노드가 순서대로 채워져 있는 이진트리



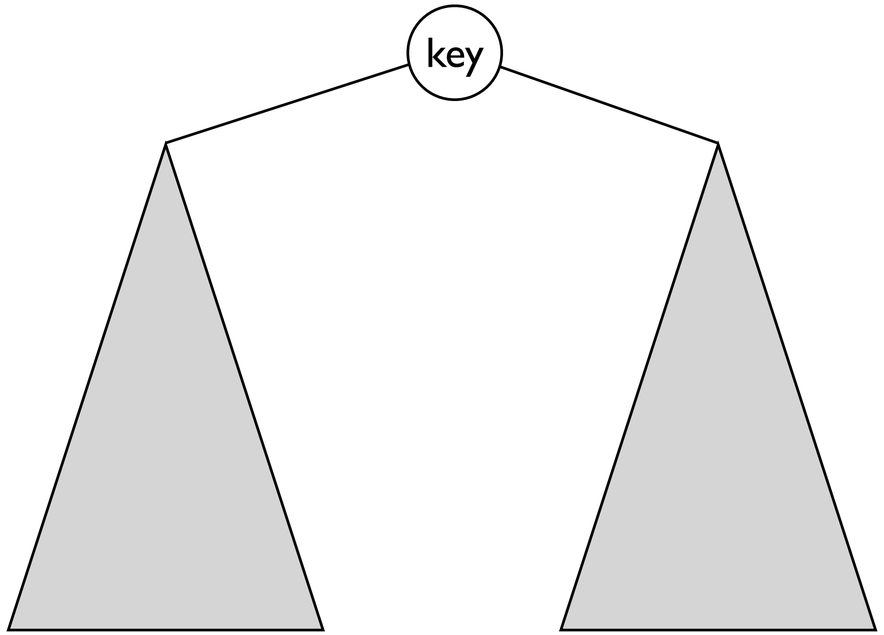
높이가 3인 완전 이진트리들

최대 힙 (Max-Heap)과 최소힙 (Min-Heap)



왼쪽 서트리의 노드들 ≤ key　　오른쪽 서트리의 노드들 ≤ key

최대 힙



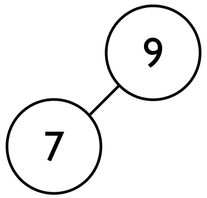
왼쪽 서트리의 노드들 ≥ key　　오른쪽 서트리의 노드들 ≥ key

최소 힙

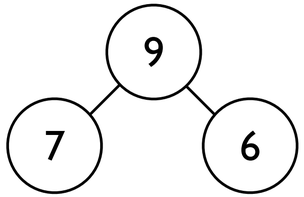
Example

- (a) ~ (i)를 최대힙인 것과 아닌 것으로 분류하시오

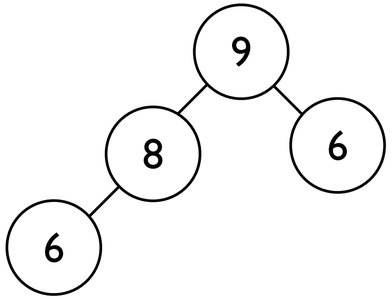
(a)



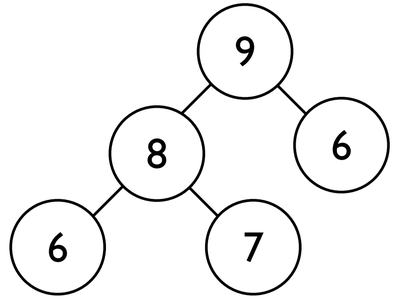
(b)



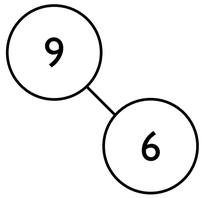
(c)



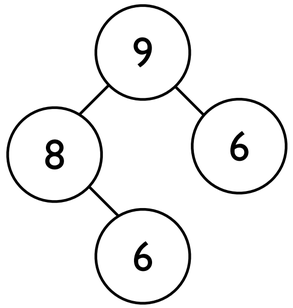
(d)



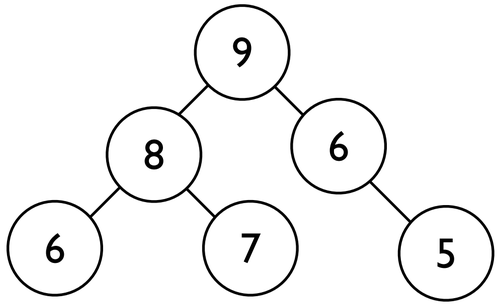
(e)



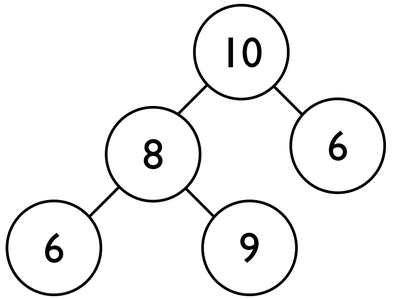
(f)



(g)



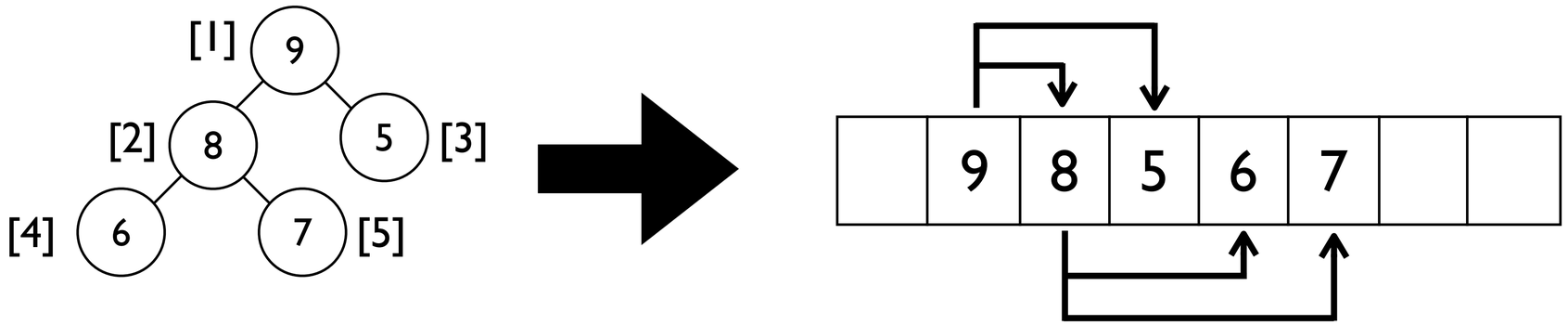
(h)



(i)

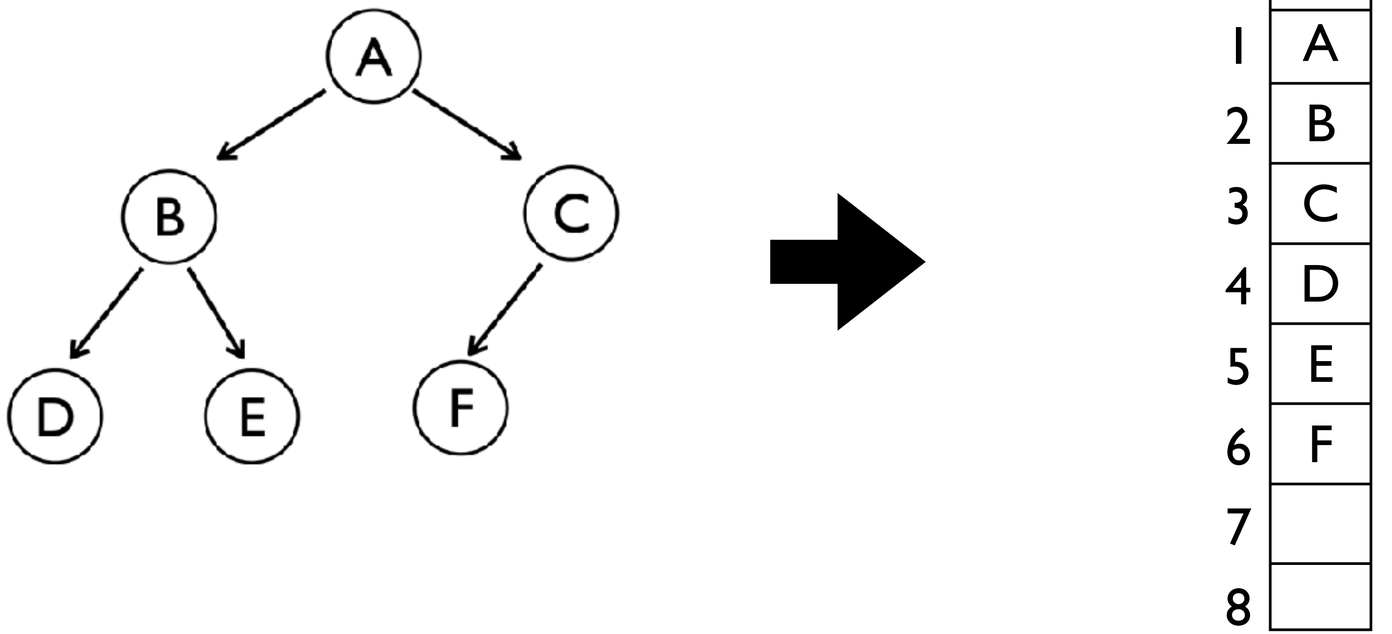
힙(Heap)의 구현

- 힙(Heap)은 완전 이진트리이므로 일반적으로 배열을 사용하여 구현함
  - 왼쪽 자식노드의 인덱스 = (부모 노드의 인덱스)*2
  - 오른쪽 자식노드의 인덱스 = (부모 노드의 인덱스)*2 + 1
  - 인덱스가 i인 노드의 부모노드의 인덱스 = ⌊i/2⌋
  - 왼쪽 자식 노드의 인덱스가 i일 때 오른쪽 자식노드의 인덱스 = i + 1



이진트리의 구현

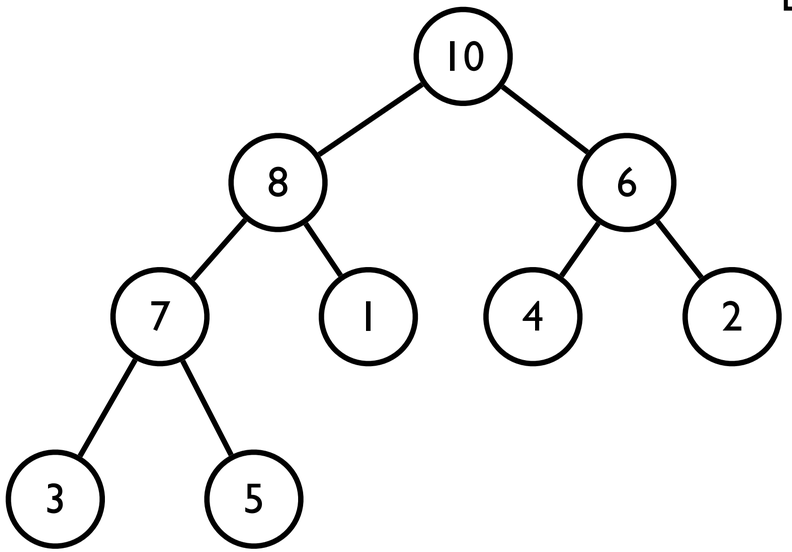
- 배열(Array) 표현법
  - 이진트리가 완전 이진트리라고 가정하고 높이가 k이면 $2^k - 1$개의 연속적인 공간으로 표현



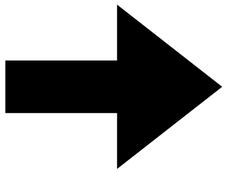
힙(Heap)의 구현

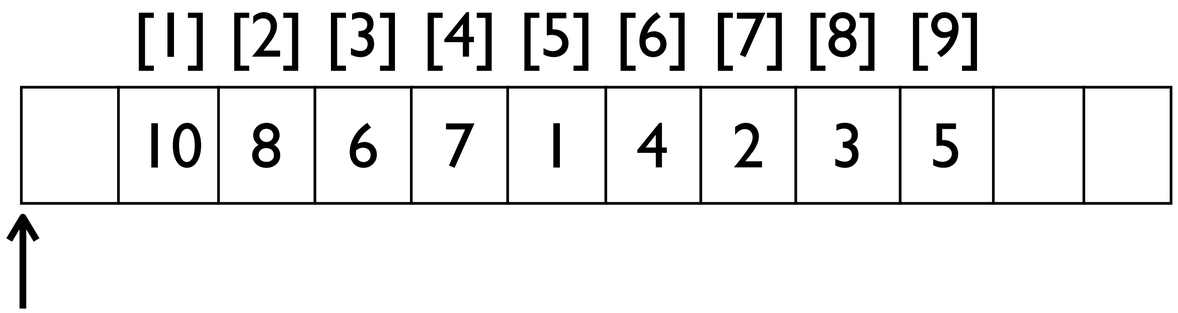
- 힙(Heap)은 완전 이진트리이므로 일반적으로 배열을 사용하여 구현함

```
typedef struct {
    int size;
    int capacity;
    int* arr;
} Heap;
```



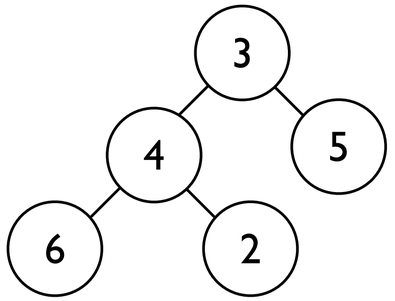
최대 힙 (Max-heap)





arr
size = 9
capacity = 12

완전이진트리를 힙 자료구조로 바꾸기

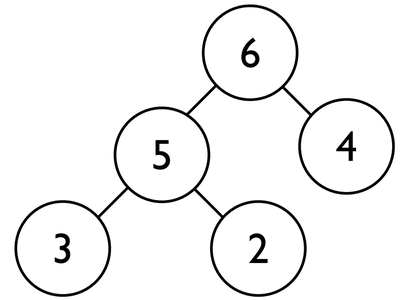


	3	4	5	6	2	

완전이진 트리 (arr)

```
procedure buildHeap1(arr)
  heap ← allocateHeap()
  heap.arr ← arr
  heap.size ← length(arr)
  heap.capacity ← maxCapacity()
  for i = 1 to heap.size do
    heapifyUp(heap, i)
  end for
  return heap
```

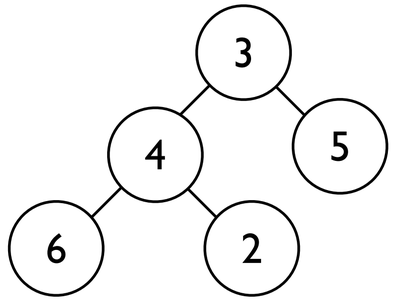
heap.capacity =maxCapacity()
heap.size = 5



	6	5	4	3	2	

heap.arr

완전이진트리를 힙 자료구조로 바꾸기

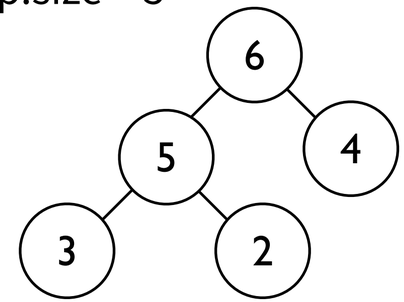


	3	4	5	6	2	

완전이진 트리 (arr)

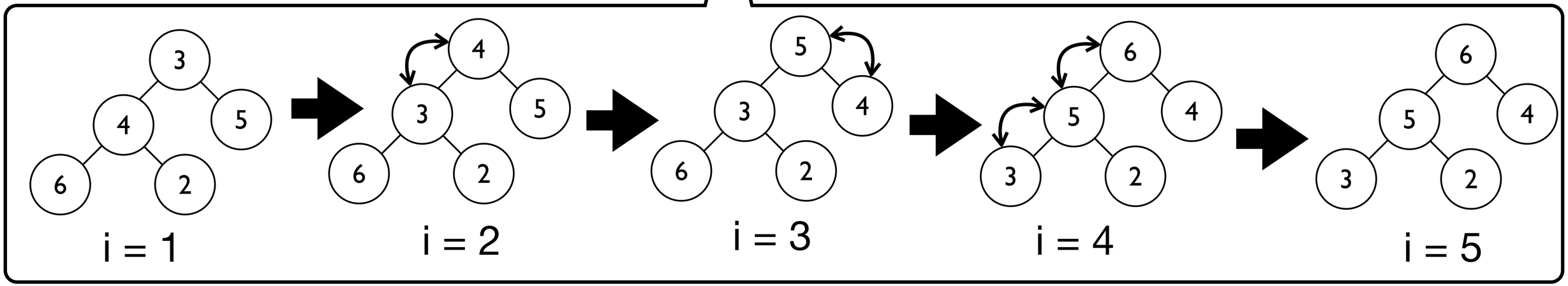
```
procedure buildHeap1(arr)
  heap ← allocateHeap()
  heap.arr ← arr
  heap.size ← length(arr)
  heap.capacity ← maxCapacity()
  for i = 1 to heap.size do
    heapifyUp(heap, i)
  end for
  return heap
```

heap.capacity =maxCapacity()
heap.size = 5



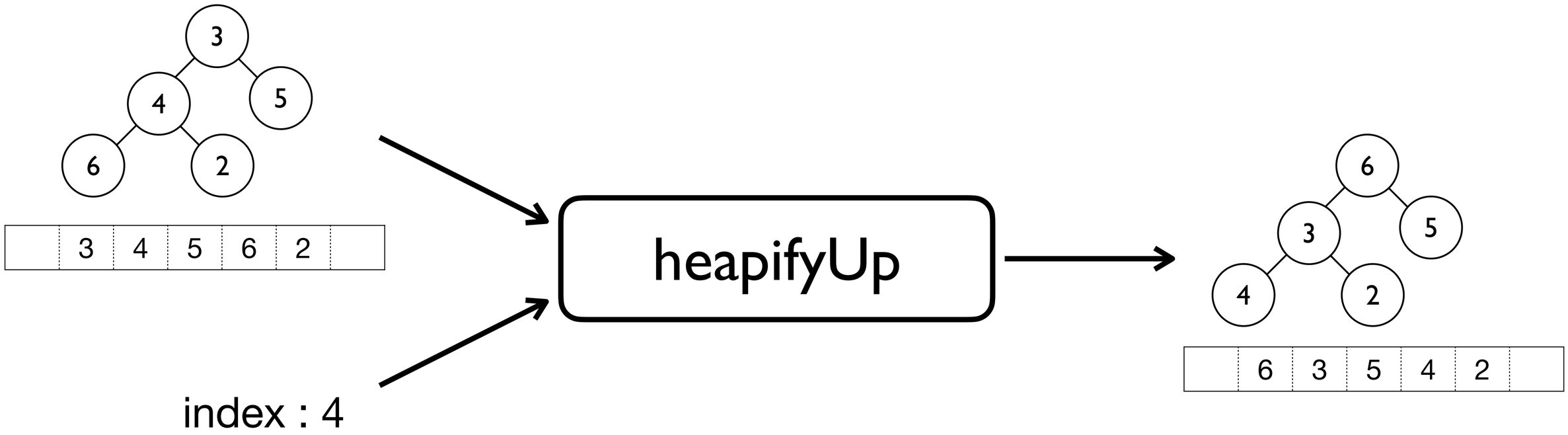
	6	5	4	3	2	

heap.arr



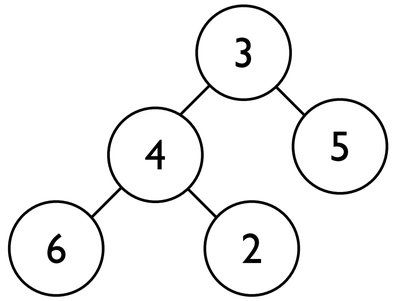
힙으로 재배열 (Heapify)

- heapifyUp: 주어진 이진트리를 힙 구조로 재배열하는 것을 heapify라 한다.



힙으로 재배열 (Heapify)

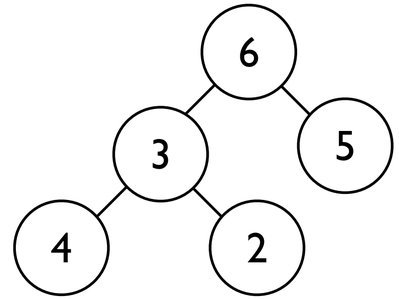
- heapifyUp: 주어진 이진트리를 힙 구조로 재배열하는 것을 heapify라 한다.



	3	4	5	6	2	

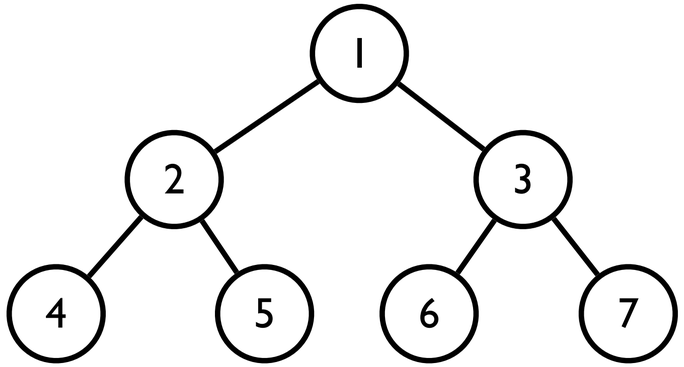
```
procedure heapifyUp(heap, index)
  if index > 1 then
    parent ← ⌊index / 2⌋
    if (heap.arr[index] > heap.arr[parent]) then
      temp ← heap.arr[parent]
      heap.arr[parent] ← heap.arr[index]
      heap.arr[index] ← temp
      heapifyUp(heap, parent)
  end if
```

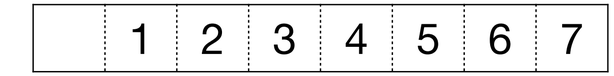
index : 4



	6	3	5	4	2	

완전이진트리를 힙 자료구조로 바꾸기



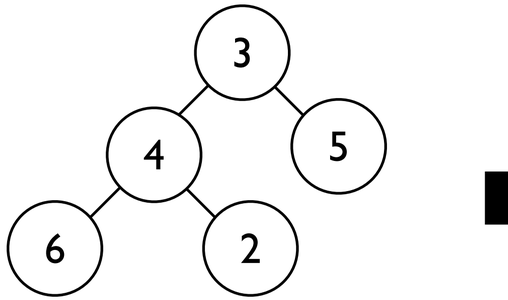



```
procedure buildHeap1(arr)
  heap ← allocateHeap()
  heap.arr ← arr
  heap.size ← length(arr)
  heap.capacity ← maxCapacity()
  for i = 1 to heap.size do
    heapifyUp(heap, i)
  end for
  return heap
```

i=7

i=1 i=2 i=3 i=4 i=5 i=6

완전이진트리를 힙 자료구조로 바꾸기

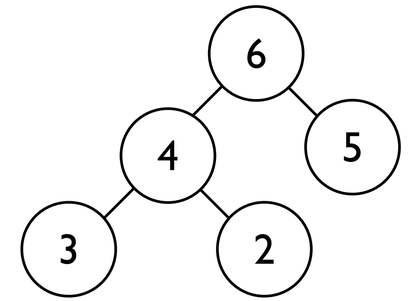


	3	4	5	6	2	

완전이진 트리 (arr)

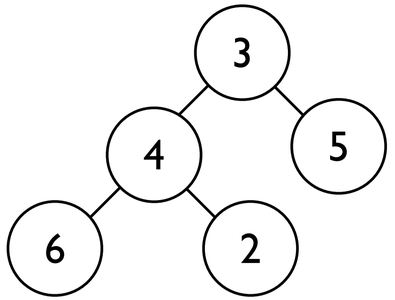
```
procedure buildHeap2(arr)
  heap ← allocateHeap()
  heap.arr ← arr
  heap.size ← length(arr)
  heap.capacity ← maxCapacity()
  for i = heap.size to 1 do
    heapifyDown(heap, i)
  end for
  return heap
```

heap.capacity =maxCapacity()
heap.size = 5



	6	5	4	3	2	

완전이진트리를 힙 자료구조로 바꾸기

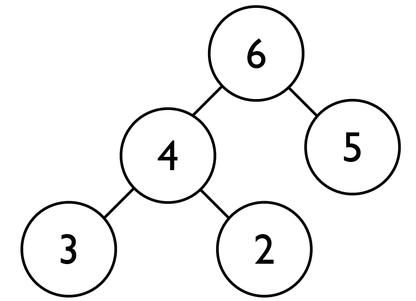


	3	4	5	6	2	

완전이진 트리 (arr)

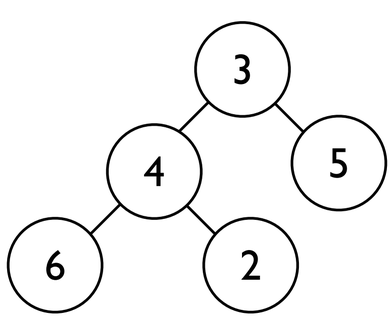
```
procedure buildHeap2(arr)
  heap ← allocateHeap()
  heap.arr ← arr
  heap.size ← length(arr)
  heap.capacity ← maxCapacity()
  for i = heap.size to 1 do
    heapifyDown(heap, i)
  end for
  return heap
```

heap.capacity =maxCapacity()
heap.size = 5

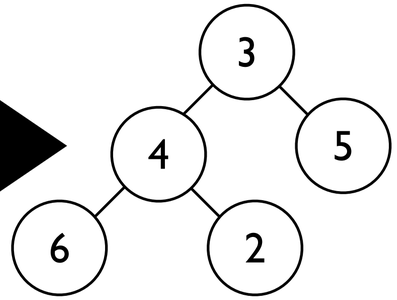


	6	5	4	3	2	

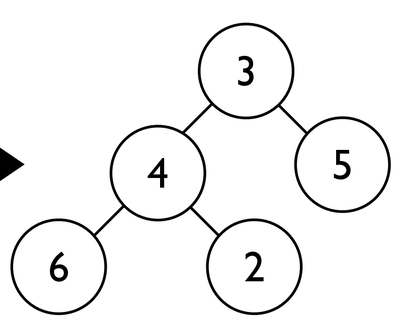
heap.arr



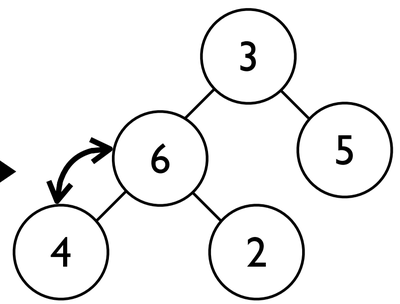
i = 5



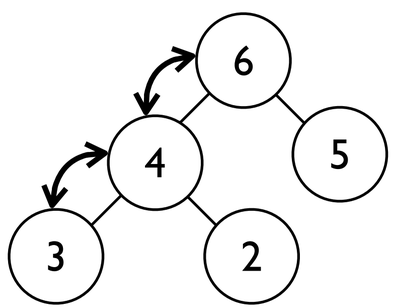
i = 4



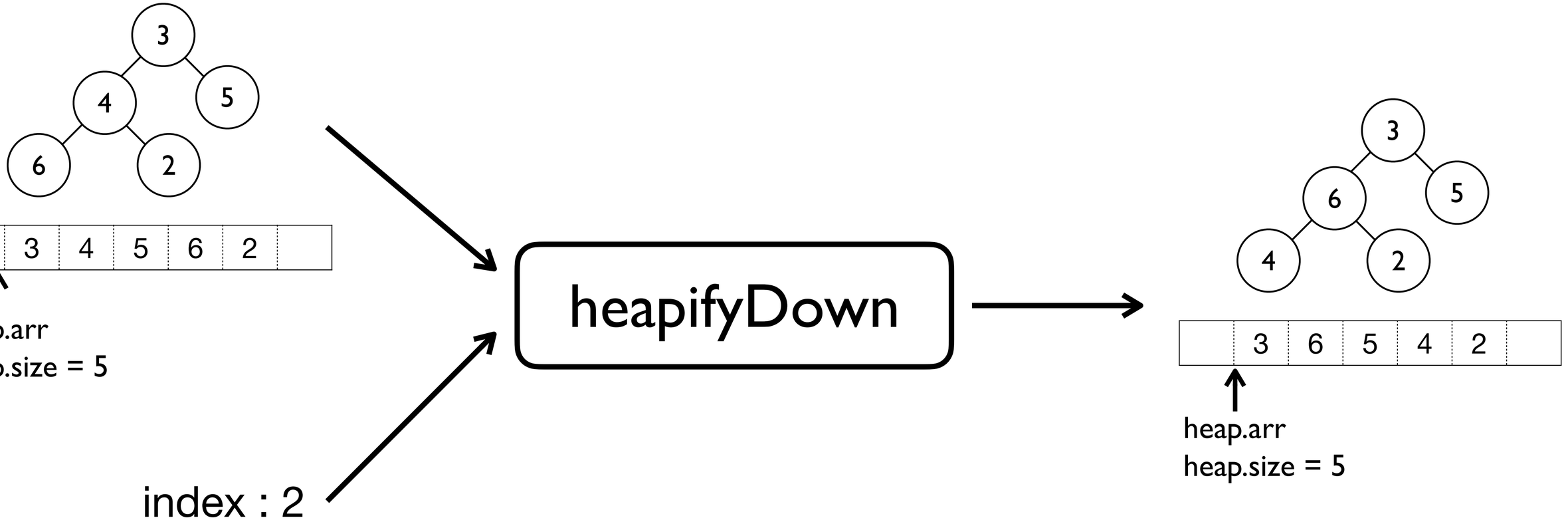
i = 3

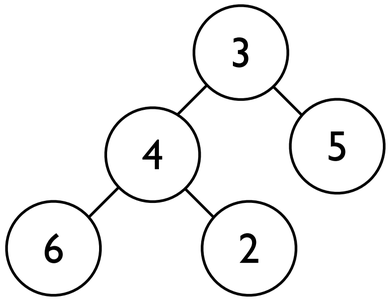


i = 2



i = 1


3
4
5
6
2
3 4 5 6 2
heap.arr
heap.size = 5
index : 2
heapifyDown
3
6
5
4
2
3 6 5 4 2
heap.arr
heap.size = 5



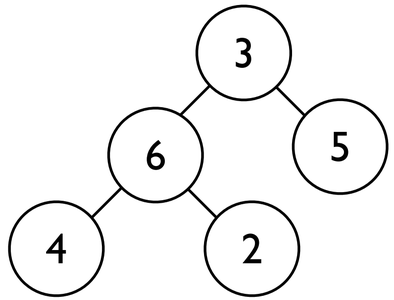
	3	4	5	6	2	

heap.arr

heap.size = 5

index : 2

```
procedure heapifyDown(heap, index)
  left ← index * 2
  right ← index * 2 + 1
  largest ← index
  if left ≤ heap.size then
    if heap.arr[index] > heap.arr[left] then
      largest ← left
    end if
  end if
  if right ≤ heap.size then
    if heap.arr[index] > heap.arr[right] then
      largest ← right
    end if
  end if
  if largest ≠ index then
    temp ← heap.arr[index]
    heap.arr[index] ← heap.arr[largest]
    heap.arr[largest] ← temp
    heapifyDown(heap, largest)
  end if
```

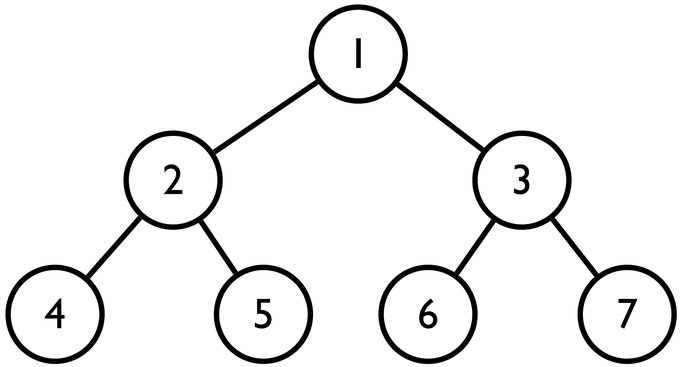


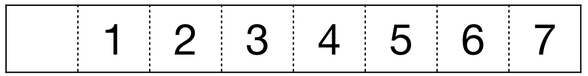
	3	6	5	4	2	

heap.arr

heap.size = 5

완전이진트리를 힙 자료구조로 바꾸기






```
procedure buildHeap2(arr)
  heap ← allocateHeap()
  heap.arr ← arr
  heap.size ← length(arr)
  heap.capacity ← maxCapacity()
  for i = heap.size to 1 do
    heapifyDown(heap, i)
  end for
  return heap
```

i=1

i=7 i=6 i=5 i=4 i=3 i=2

힙 (Heap) 자료구조

- 힙 (heap)자료구조는 다음과 같은 기능들 제공함
  - create(): 힙 자료구조를 생성 후 반환함
  - insert(heap, data): 힙의 특성을 유지하면서 새로운 데이터 data를 추가함
  - deleteRoot(heap): 힙의 특성을 유지하면서 루트 노드를 삭제 및 반환함

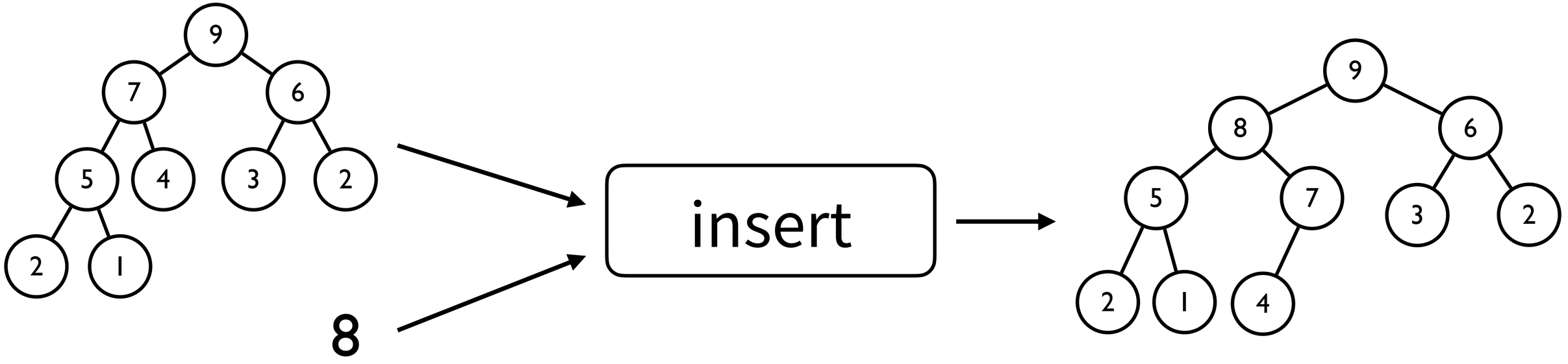
create

- create: 힙 자료구조를 생성 후 반환함

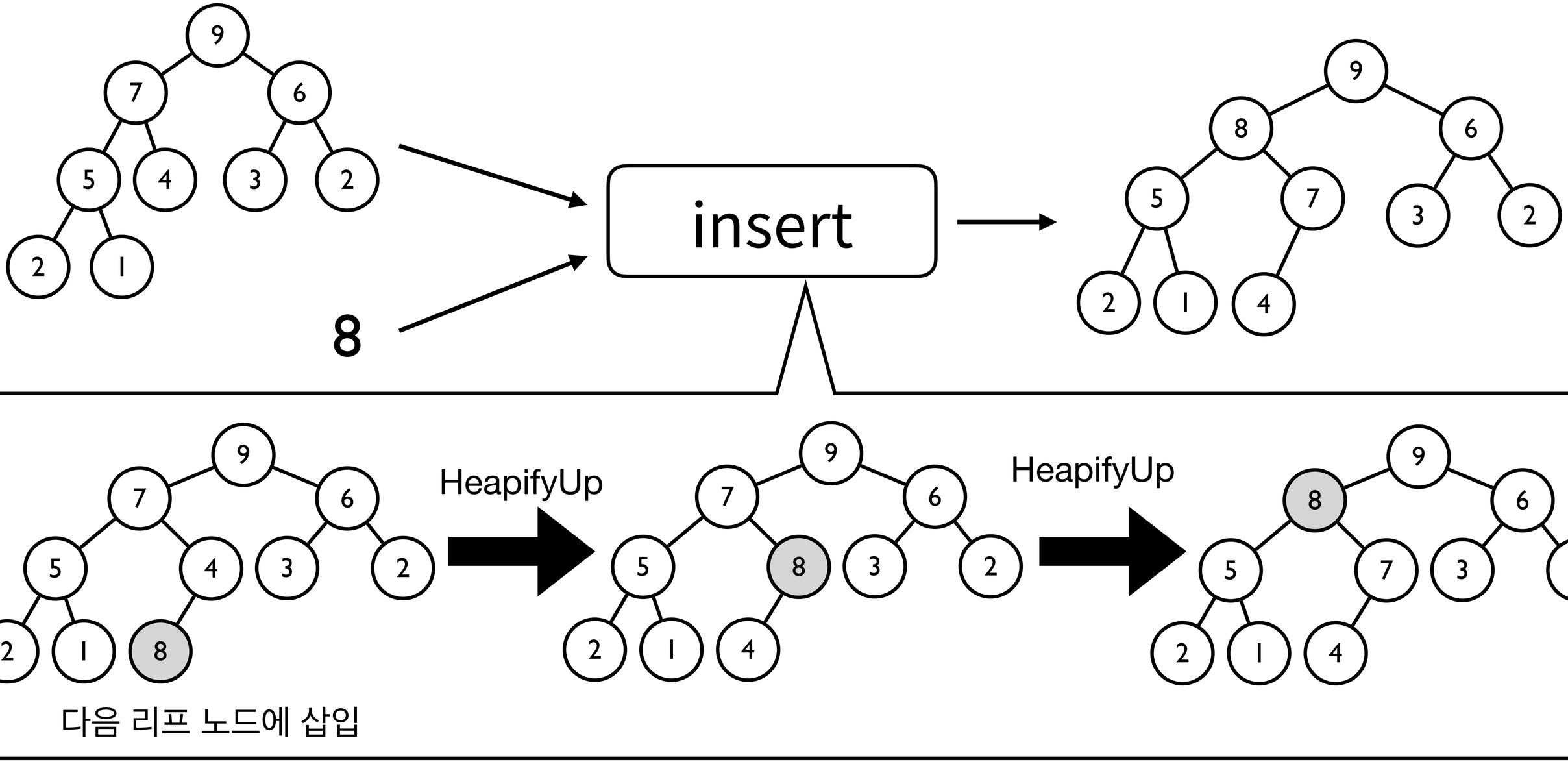
```
procedure create()
  heap ← allocateHeap()
  heap.arr ← allocateArray()
  heap.size ← 0
  heap.capacity ← maxCapacity()
  return heap
```

insert

- insert: 힙의 특성을 유지하면서 새로운 데이터 data를 추가함

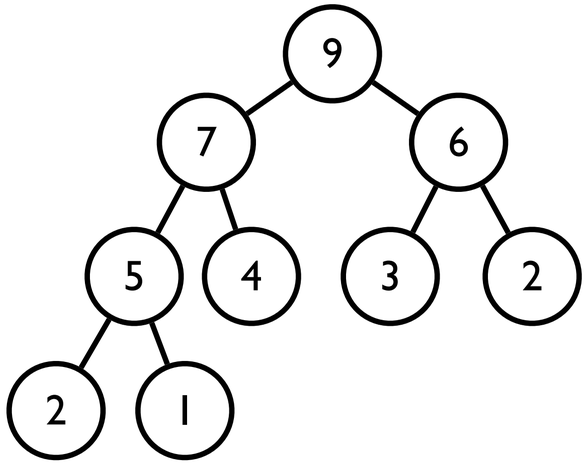


insert


9
7
6
5
4
3
2
2
1
8
insert
9
8
6
5
7
3
2
2
1
4
9
7
6
5
4
3
2
2
1
8
다음 리프 노드에 삽입
HeapifyUp
9
7
6
5
8
3
2
2
1
4
HeapifyUp
9
8
6
5
7
3
2
2
1
4

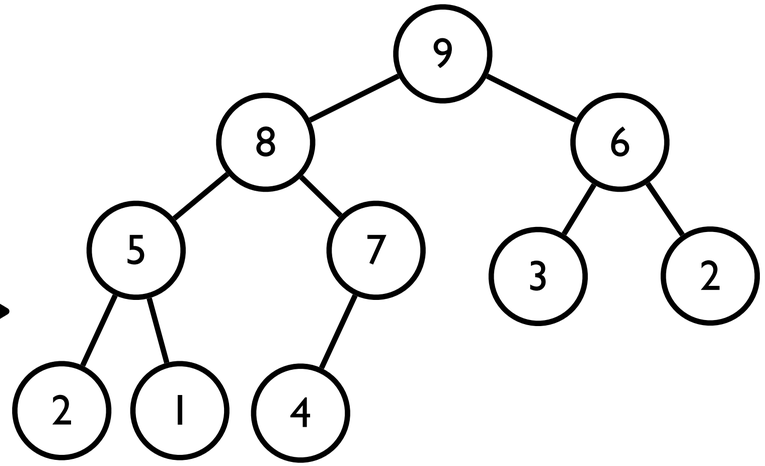
insert

- insert: 힙의 특성을 유지하면서 새로운 데이터 data를 추가함



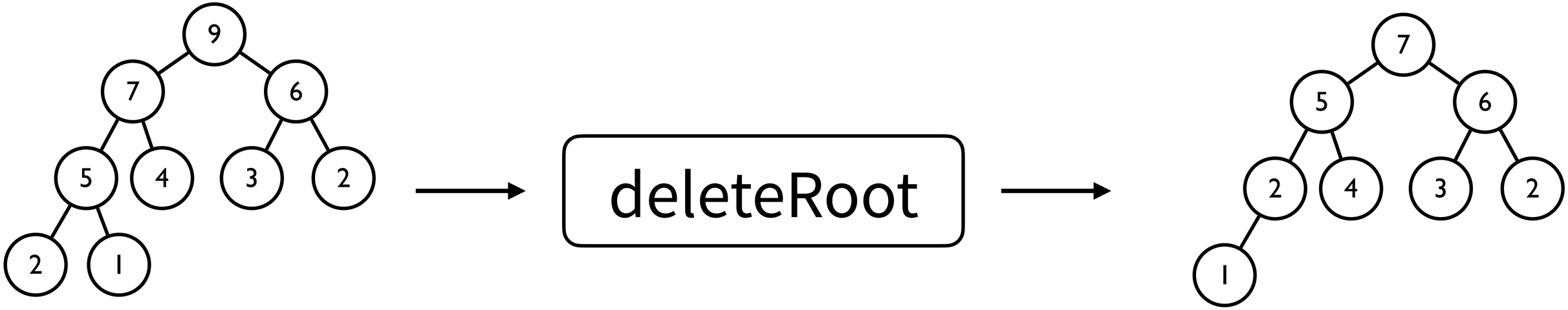
8

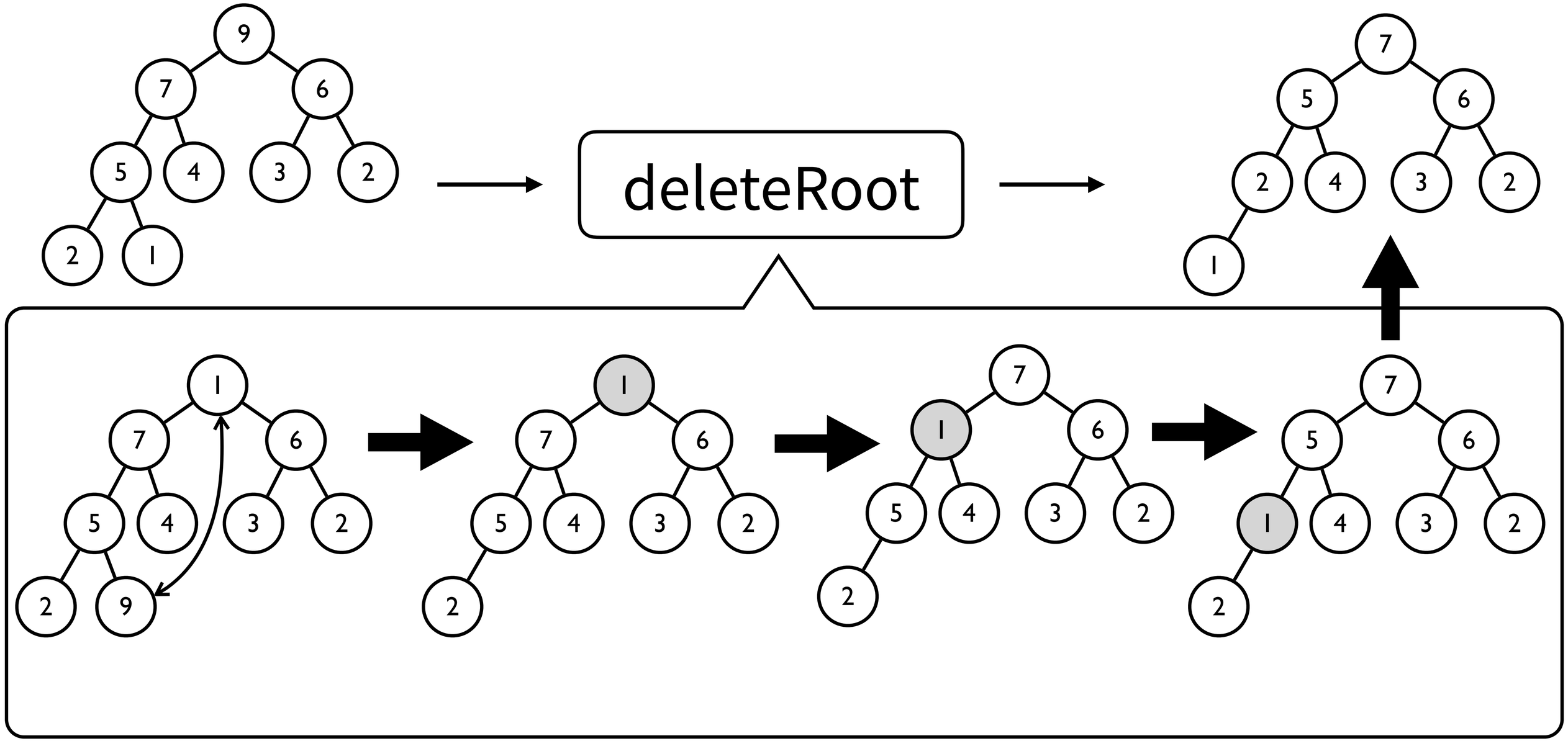
```
procedure insert(heap, data)
  if heap.size = heap.capacity then
    return error()
  end if
  heap.size ← heap.size + 1
  heap.arr[heap.size] ← value
  heapifyUp(heap, heap.size)
  return item
```



deleteRoot

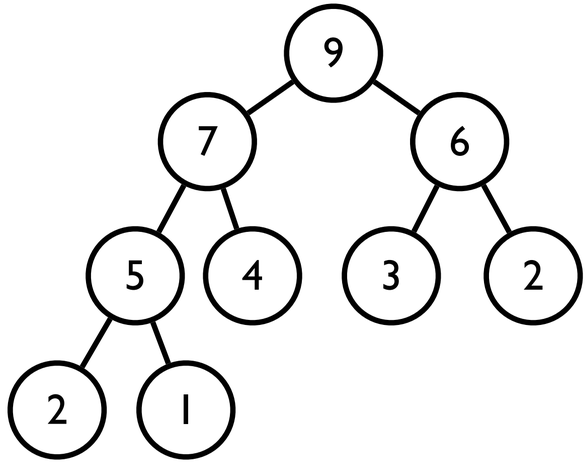
- deleteRoot : 힙의 루트(root) 노드를 삭제 후 반환함 (삭제 후에도 힙의 성질을 만족해야 함)




9
7
6
5
4
3
2
2
1
deleteRoot
7
5
6
2
4
3
2
1
1
7
6
5
4
3
2
2
9
1
7
6
5
4
3
2
2
7
1
6
5
4
3
2
2
7
5
6
1
4
3
2
2

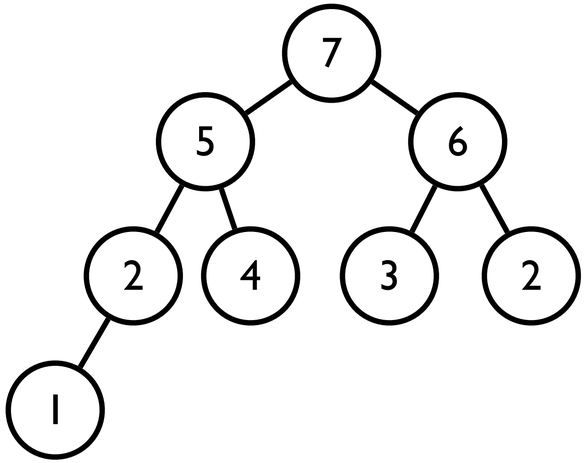
deleteRoot

- deleteRoot : 힙의 루트(root) 노드를 삭제 후 반환함 (삭제 후에도 힙의 성질을 만족해야 함)



힙(heap)

```
procedure deleteRoot(heap)
  if heap.size = 0 then
    return error()
  end if
  item ← heap.arr[1]
  heap.arr[1] ← heap.arr[heap.size]
  heap.size ← heap.size - 1
  heapifyDown(heap, 1)
  return item
```



삽입 후의 힙

힙(heap) 자료구조의 구현

- 힙(heap)은 다음과 같은 기능을 제공함 (힙의 추상 자료형)
  - `Heap* create()` : 비어있는 힙을 생성 후 반환함
  - `void insert(Heap* heap, int value)` : 힙에 새로운 데이터 value를 추가함
  - `int deleteRoot(Heap* heap)`: 힙의 루트노드를 제거 후 루트노드의 값을 반환함
  - `void traversal(Heap* heap)`: 힙을 구성하는 노드들의 값들을 출력함
  - `void destroy(Heap* heap)`: 힙이 사용하고 있는 메모리를 반환함

힙(heap) 자료구조의 구현

- 힙(heap)은 다음과 같은 정보를 가지는 자료구조

```
typedef struct {
    int size;
    int capacity;
    int* arr;
} Heap;
```

- `Heap* create()` : 비어있는 힙을 생성 후 반환함

```
Heap* create() {
    Heap* heap = (Heap*)malloc(sizeof(Heap));
    heap->size = 0;
    heap->capacity = 100;
    heap->arr = (int*)malloc(heap->capacity * sizeof(int));
    return heap;
}
```

힙(heap) 자료구조의 구현

- `void insert(Heap* heap, int value)` : 힙에 새로운 데이터 value를 추가함

```c
void insert(Heap* heap, int value) {
    if (heap->size >= heap->capacity) {
        printf("Error: Heap is full!\n");
        return;
    }
    heap->size = heap->size + 1;
    heap->arr[heap->size] = value;
    heapifyUp(heap, heap->size);
}
```

```c
void heapifyUp(Heap *heap, int index) {
    if (index <= 1) {
        return;
    }
    int parent = index / 2;
    if (heap->arr[parent] < heap->arr[index]) {
        int temp = heap->arr[parent];
        heap->arr[parent] = heap->arr[index];
        heap->arr[index] = temp;
        heapifyUp(heap, parent);
    }
}
```

힙(heap) 자료구조의 구현

- `int deleteRoot(Heap* heap)`: 힙의 루트노드를 제거 후 루트노드의 값을 반환함

```
int deleteRoot(Heap *heap) {
    if (heap->size <= 0) {
        printf("Heap is empty!\n");
        return -1;
    }
    int max = heap->arr[1];
    heap->arr[1] = heap->arr[heap->size];
    heap->size--;
    heapifyDown(heap, 1);
    return max;
}
```

힙(heap) 자료구조의 구현

```
void heapifyDown(Heap *heap, int index) {
    int left = 2 * index;
    int right = 2 * index + 1;
    int largest = index;

    if (left <= heap->size && heap->arr[left] > heap->arr[largest]) {
        largest = left;
    }
    if (right <= heap->size && heap->arr[right] > heap->arr[largest]) {
        largest = right;
    }
    if (largest != index) {
        int temp = heap->arr[index];
        heap->arr[index] = heap->arr[largest];
        heap->arr[largest] = temp;
        heapifyDown(heap, largest);
    }
}
```

힙(heap) 자료구조의 구현

- `void traversal(Heap* heap)`: 힙을 구성하는 노드들의 값들을 출력함

```c
void traversal(Heap* heap) {
    printf("Heap elements: ");
    for (int i = 1; i <= heap->size; i++) {
        printf("%d ", heap->arr[i]);
    }
    printf("\n");
}
```

- `void destroy(Heap* heap)`: 힙이 사용하고 있는 메모리를 반환함

```c
void destroy(Heap* heap) {
    free(heap->arr);
    free(heap);
}
```

Example

```c
#include "Heap.h"
#include <stdio.h>


int main() {
    Heap* heap = create();

    insert(heap, 10);
    insert(heap, 20);
    insert(heap, 15);
    insert(heap, 30);
    insert(heap, 40);

    printf("After inserting elements:\n");
    traversal(heap);

    printf("Extracted max: %d\n", deleteRoot(heap));
    printf("After extracting max:\n");
    traversal(heap);

    printf("Extracted max: %d\n", deleteRoot(heap));
    printf("After extracting max:\n");
    traversal(heap);

    destroy(heap);

    return 0;
}
```

힙 자료구조의 응용

(1) k번째로 큰 수 찾기 (select k)

- 정렬되지 않은 배열에서 k번째 큰 수 찾기

(2) 우선 순위 큐 (priority queue)

- 우선 순위가 높은 데이터가 먼저 나가는 자료구조

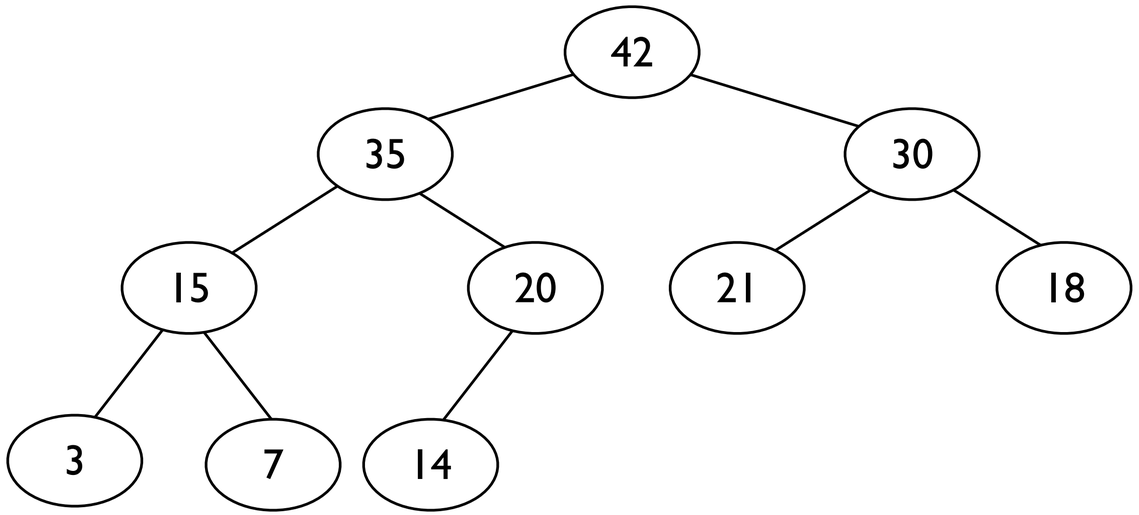
(3) 힙 정렬 (heap sort)

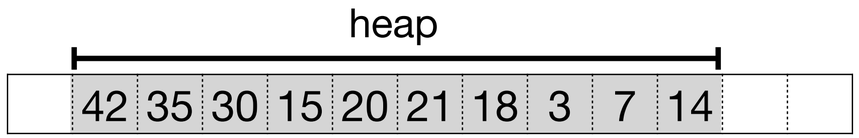
힙 자료구조의 응용 1 : k번째 큰 수 찾기(select k)

- 정렬되지 않은 배열에서 k번째 큰 수 찾기

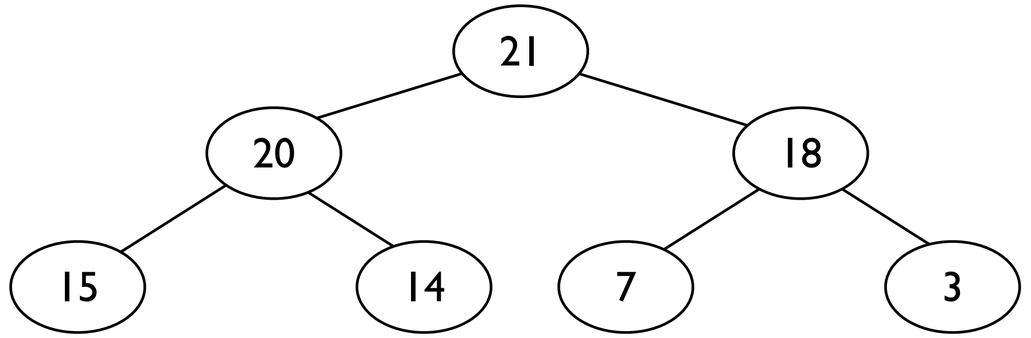
  (1) 배열을 최대 힙(max-heap)으로 바꿈

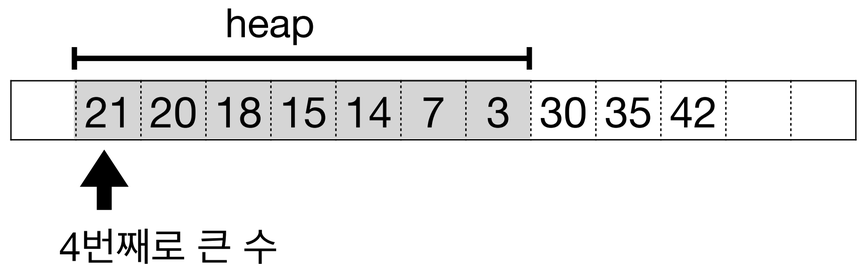
  (2) k-1번 delete 연산을 실행후 힙(heap)의 루트(root)노드를 반환





k=4





힙 자료구조의 응용 1 : k번째 큰 수 찾기(select k)

```c
#include "Heap.h"
#include <stdio.h>

int selectK(int arr[], int arr_size, int k) {
    Heap* heap = create();
    for (int i = 0; i < arr_size; i++) {
        insert(heap, arr[i]);
    }
    for (int i = 1; i < k; i++) {
        deleteRoot(heap);
    }
    int result = deleteRoot(heap);
    destroy(heap);
    return result;
}

int main() {
    int arr[] = {10, 20, 15, 30, 40};
    int k = 3;
    int result = selectK(arr, 5, k);
    printf("The %d-th largest element is %d\n", k, result);
    return 0;
}
```

힙 자료구조의 응용 2 : 우선 순위 큐 (Priority Queue)

- 큐(Queue) 자료구조
  - 먼저 들어온 데이터가 먼저 나가는 (FIFO) 자료구조
- 우선순위 큐 (Priority Queue) 자료구조
  - 우선 순위가 높은 데이터가 먼저 나가는 자료구조

```
typedef struct {
    int size;
    int capacity;
    int* arr;
} PriorityQueue;
```

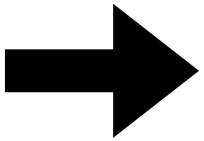
```
procedure enqueue(pqueue, data)
  if pqueue.size = pqueue.capacity then
    return error()
  end if
  pqueue.size ← pqueue.size + 1
  pqueue.arr[heap.size] ← value
  heapifyUp(pqueue, pqueue.size)
  return item
```

```
procedure dequeue(pqueue)
  if pqueue.size = 0 then
    return error()
  end if
  item ← pqueue.arr[1]
  pqueue.arr[1] ← pqueue.arr[heap.size]
  pqueue.size ← pqueue.size - 1
  heapifyDown(pqueue, 1)
  return item
```

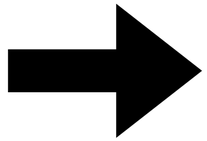
힙 자료구조의 응용 3 : 힙 정렬(heap sort)

arr

4	6	1	2	5	3	8



```
procedure heapsort(arr)
  heap ← create()
  for i = 0 to length(arr) -1 do
    insert(heap, arr[i])
  end for
  for i = length(arr) -1 to 0 do
    arr[i] = delete(heap)
  end for
 return arr
```



arr

8	6	5	4	3	2	1

힙 자료구조의 응용 3 : 힙 정렬(heap sort)

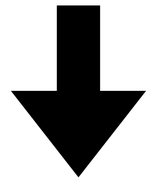
```c
#include "Heap.h"
#include <stdio.h>
#include <stdlib.h>

void myheapsort(int arr[], int size) {
    Heap* heap = create();
    for (int i = 0; i < size; i++) {
        insert(heap, arr[i]);
    }
    for (int i = 0; i < size; i++) {
        arr[i] = deleteRoot(heap);
    }
    destroy(heap);
}

int main() {
    int arr[] = {4, 6, 1, 2, 5, 3, 8};
    int size = sizeof(arr) / sizeof(arr[0]);
    myheapsort(arr, size);
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

arr

4	6	1	2	5	3	8



myheapsort(arr,7)

arr

8	6	5	4	3	2	1

힙 자료구조의 응용 3 : 힙 정렬(heap sort)

- 힙 정렬은 시간 복잡도가 $O(n \log n)$인 정렬 알고리즘

```
procedure heapsort(arr)
  heap ← create()
  for i = 0 to length(arr) -1 do
    insert(heap, arr[i])
  end for
  for i = length(arr) -1 to 0 do
    arr[i] = delete(heap)
  end for
 return arr
```

VS

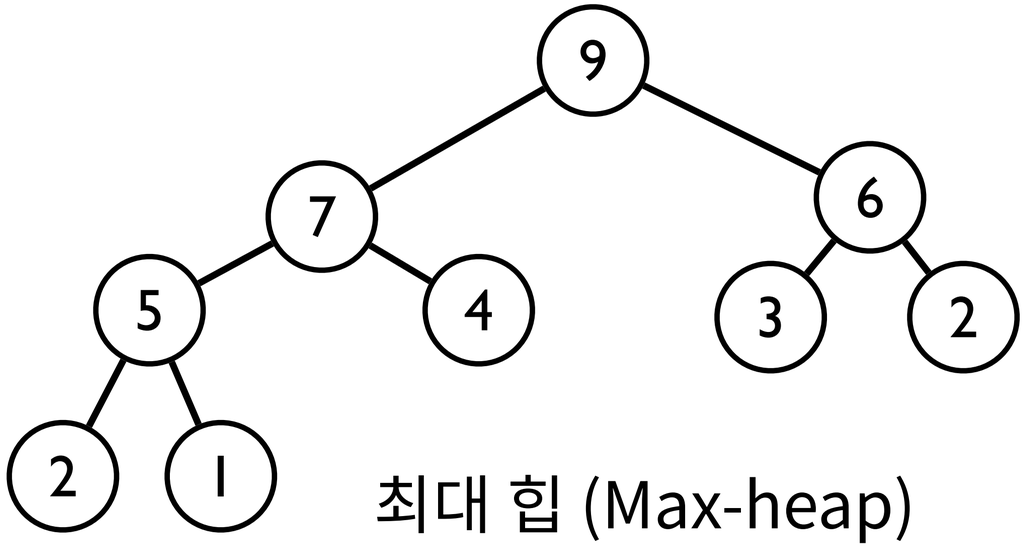
```
procedure bubblesort(arr)
  for i = 0 to length(arr) -1 do
    for j = 0 to length(arr) - i -1 do
      if (arr[j] > arr[j+1]) then
        swap(arr[j], arr[j+1])
    end for
  end for
return arr
```

시간 복잡도 : $O(n \log n)$

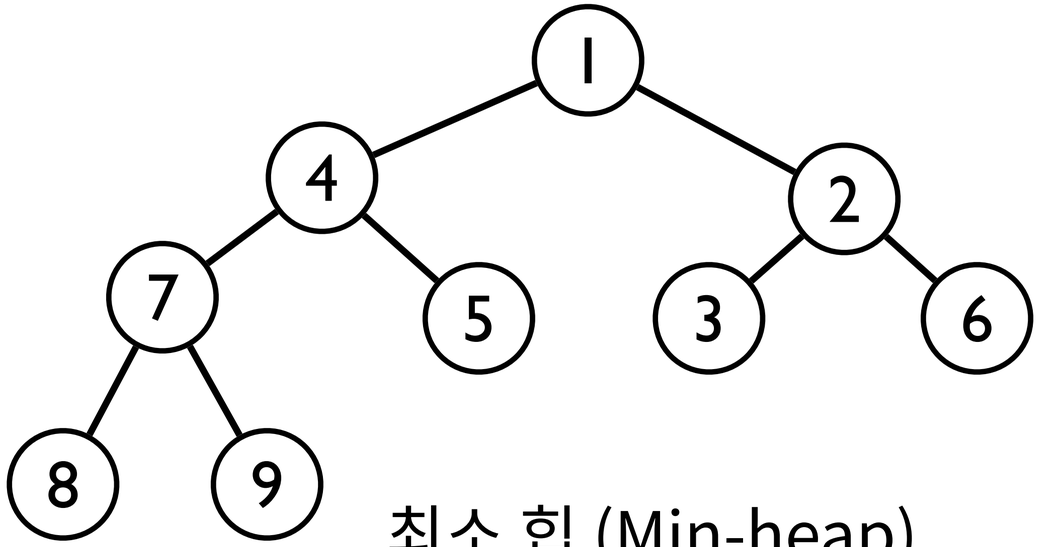
시간 복잡도 : $O(n^2)$

마무리

- Heap: 데이터의 최대값 또는 최소값들을 위주로 다루어야 할 때 적합한 자료구조
  - 최대 힙 (Max-heap): 부모 노드의 키 값이 자식노드들의 키값보다 항상 크거나 같음
    - 루트 노드가 가장 큰 키값을 가짐
  - 최소 힙 (Min-heap): 부모 노드의 키 값이 자식노드들의 키값보다 항상 작거나 같음
    - 루트 노드가 가장 작은 키값을 가짐



최대 힙 (Max-heap)



최소 힙 (Min-heap)