

트리 2

-우선순위 큐와 힙-

HaRim Jung, Ph.D.

Visiting Professor / Senior Researcher

SKKU Institute for Convergence / Convergence Research Institute

Sungkyunkwan University, Korea

우선순위 큐 (1/9)

□ 우선순위 큐(Priority Queue) 개요

- 우선순위 큐

- **임의의 기준을 중심**으로 가장 높은 우선순위를 가지는 항목의 삭제 및 반환을 보장하는 큐(자료구조)

- 스택·큐도 일종의 우선순위 큐

- 스택: **시간 중심**으로 가장 마지막으로 삽입된 항목이 가장 높은 우선순위를 가지므로, 삽입된 시간이 최근 시간일수록 높은 우선순위를 부여
 - 큐: **시간 중심**으로 가장 먼저 삽입된 항목이 가장 높은 우선순위를 가지므로, 삽입된 시간이 이른 시간일수록 더 높은 우선순위를 부여
 - 즉, 스택과 큐는 **우선순위 큐의 특수한 형태**로서 시간에 그 우선순위를 부여한 것임

- 응급실에서 환자 치료의 예

- 스택: 나중에 도착한 환자를 먼저 치료
 - 큐: 먼저 도착한 환자를 먼저 치료
 - **우선순위 큐**: 위급한 환자를 먼저 치료



참고: 주우석, "IT CookBook, C-C++로 배우는 자료구조론," 한빛미디어(주), 2014

우선순위 큐 (2/9)

□ 우선순위 큐의 구현

• 우선순위 큐에 적용 가능한 연산

- `PriorityQueue()`: 새로운 우선순위 큐 생성
- `enqueue(item)`: 기존 Rear 위치에 item 삽입
- `dequeue()`: 우선순위가 가장 높은 item **삭제 및 반환**
- ⋮

• Python 리스트를 이용한 구현 생각해봅시다: 정렬된 동적 배열을 사용한다면?

- Python 리스트를 이용한 (혹은 Queue 클래스 정의를 통한) 큐 구현과 유사
 - 빈(empty) Python 리스트 생성하여 `PriorityQueue()` 구현 **시간복잡도: $O(1)$**
 - `append(item)` 메소드로 `enqueue(item)` 구현 **시간복잡도: $O(1)$**
 - Python 리스트에 저장되어 있는 항목 중 우선순위가 가장 높은 항목을 **탐색 후** 해당 항목을 **삭제 및 반환**함으로써 `dequeue()` 구현

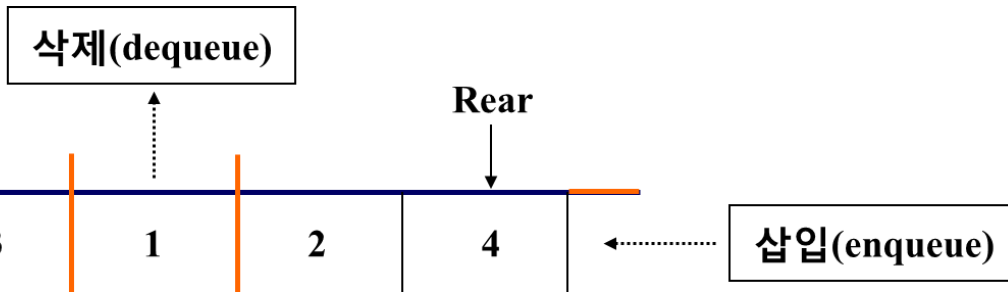
항목의 삽입 위치는 맨 끝, 삭제 위치는 우선순위가 가장 높은 항목의 위치로 제한하는 것이 더 바람직하므로 **Python 리스트를 이용한 PriorityQueue 클래스도 정의해 보기 바람**

- 매우 간단하므로 본 강의 자료에서는 생략

시간복잡도: $O(N)$

□ 탐색: $O(N)$

□ 삭제: 자리이동으로 인한 $O(N)$



항목의 값이 작을수록 높은 우선순위를 가진다고 가정

```
1 priority_queue = []
2 priority_queue.append(3)
3 priority_queue.append(1)
4 priority_queue.append(2)
5 priority_queue.append(4)
6 smallest = priority_queue[0]
7 for i in priority_queue:
8     if i < smallest:
9         smallest = i
10 index = priority_queue.index(smallest)
11 result = priority_queue.pop(index)
12 print(result)
```

우선순위 큐 (3/9)

□ 우선순위 큐의 구현 contd.

- 정렬된 단순 연결 리스트를 이용한 구현

– PriorityQueue 클래스 정의

```
1 from node import Node
2 class PriorityQueue:
3
4     def __init__(self):
5         self.head = None
6
7     def enqueue(self, item):
8         current = self.head
9         previous = None
10        stop = False
11        while current != None and not stop:
12            if current.get_item() > item:
13                stop = True
14            else:
15                previous = current
16                current = current.get_next()
17        temp = Node(item)
18        if previous == None:
19            temp.set_next(self.head)
20            self.head = temp
21        else:
22            temp.set_next(current)
23            previous.set_next(temp) 계속
```

```
25     def dequeue(self):
26         if self.head == None:
27             return None
28         else:
29             temp = self.head
30             dequeued_item = temp.get_item()
31             self.head = self.head.get_next()
32             return dequeued_item
33
```

우선순위 큐 (4/9)

□ 우선순위 큐의 구현 contd.


- 정렬된 단순 연결 리스트를 이용한 구현 contd.

– `PriorityQueue()` 시간복잡도: $O(1)$

```
4 def __init__(self):  
5     self.head = None
```

- 빈 우선순위 큐 생성

» 빈 (정렬된) 단순 연결 리스트 생성과 동일하므로 첫 번째 노드의 참조를 저장하는 변수인 `head`를 `None`으로 설정

`q = PriorityQueue()` 
`q.head`

– `enqueue()` 시간복잡도: $O(N)$

```
7 def enqueue(self, item):  
8     current = self.head  
9     previous = None  
10    stop = False  
11    while current != None and not stop:  
12        if current.get_item() > item:  
13            stop = True  
14        else:  
15            previous = current  
16            current = current.get_next()
```

```
17     temp = Node(item)  
18     if previous == None:  
19         temp.set_next(self.head)  
20         self.head = temp  
21     else:  
22         temp.set_next(current)  
23         previous.set_next(temp)  
24
```

- (정렬된 단순 연결 리스트와 동일) 연결 리스트의 (오름차순) 정렬된 순서에 맞는 위치에 `item` 삽입

우선순위 큐 (5/9)

□ 우선순위 큐의 구현 contd.

- 정렬된 단순 연결 리스트를 이용한 구현 contd.

– enqueue() contd.

- (정렬된 단순 연결 리스트와 동일) 연결 리스트의 (오름차순) 정렬된 순서에 맞는 위치에 item 삽입

```
7  def enqueue(self, item):
8      current = self.head
9      previous = None
10     stop = False
11     while current != None and not stop:
12         if current.get_item() > item:
13             stop = True
14         else:
15             previous = current
16             current = current.get_next()
```

```
17     temp = Node(item)
18     if previous == None:
19         temp.set_next(self.head)
20         self.head = temp
21     else:
22         temp.set_next(current)
23         previous.set_next(temp)
24
```

- » 라인 8: 삽입할 위치에 있는 기존 노드 Nold를 찾는 순회를 위한 지역 변수 current에 head가 참조하고 있는 노드 할당
- » 라인 9: 새로운 노드 Nnew를 삽입 시 필요한 Nold 이전 노드를 할당하기 위한 previous 지역 변수 생성 및 None 할당
- » 라인 10: 삽입할 위치(i.e., 기존 노드 Nold) 탐색 성공 여부 확인을 위한 stop 지역 변수 선언 및 False 할당
- » 라인 11: 삽입할 위치를 찾을 때까지 while-루프 실행
- » 라인 12-13: if current가 참조하고 있는 노드 N이 저장하는 item의 값이 삽입하고자 하는 item의 값보다 크다면, i.e., N이 Nold라면, stop에 True 할당(삽입할 위치 탐색 성공)

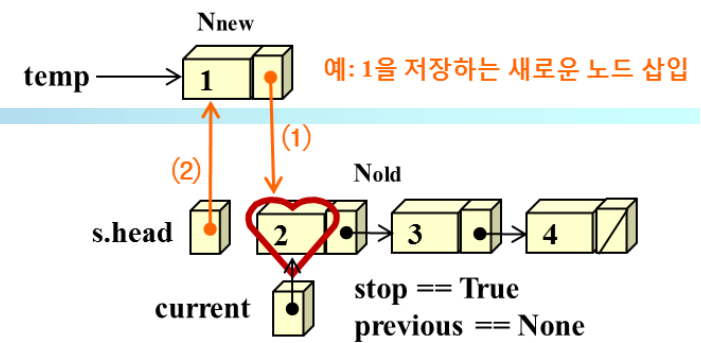
우선순위 큐 (6/9)

□ 우선순위 큐의 구현 contd.

- 정렬된 단순 연결 리스트를 이용한 구현 contd.

– enqueue() contd.

- (정렬된 단순 연결 리스트와 동일) 연결 리스트의 (오름차순) 정렬된 순서에 맞는 위치에 item 삽입



라인 18-20의 예

```
7 def enqueue(self, item):
8     current = self.head
9     previous = None
10    stop = False
11    while current != None and not stop:
12        if current.get_item() > item:
13            stop = True
14        else:
15            previous = current
16            current = current.get_next()
```

```
17 temp = Node(item)
18 if previous == None:
19     temp.set_next(self.head)
20     self.head = temp
21 else:
22     temp.set_next(current)
23     previous.set_next(temp)
24
```

- » 라인 15-16: else, previous에 N을 할당하고 current에 N의 다음 노드를 할당
- » 라인 17: 삽입하고자 하는 item을 저장하는 새로운 노드 N_{new} 생성 후 지역 변수 temp에 할당
- » 라인 18-20: if previous에 None이 할당되어 있다면, current가 참조하고 있는 노드 N_{old}는 head가 참조하고 있는 노드
이므로(즉, None이거나 첫 노드의 item부터 N_{new}의 item보다 큰 경우), (1) temp에 할당된 N_{new}가 N_{old}를 참조하게 하고,
(2) head는 N_{new}를 참조하게 함 (순서 중요)

우선순위 큐 (7/9)

□ 우선순위 큐의 구현 contd.

- 정렬된 단순 연결 리스트를 이용한 구현 contd.

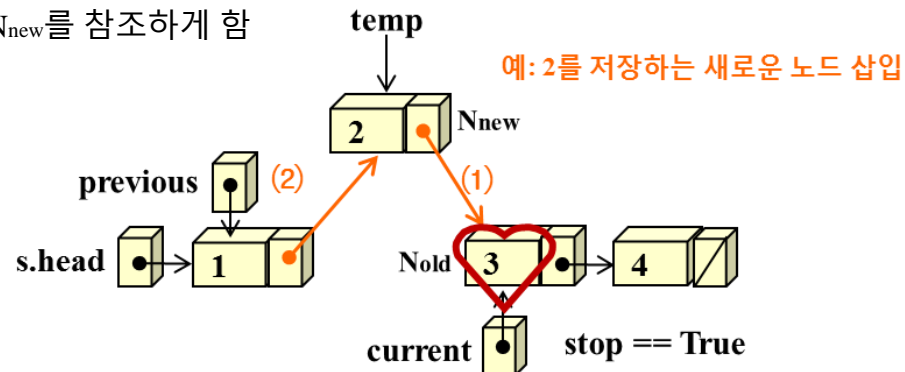
- enqueue() contd.

- (정렬된 단순 연결 리스트와 동일) 연결 리스트의 (오름차순) 정렬된 순서에 맞는 위치에 item 삽입

```
7 def enqueue(self, item):
8     current = self.head
9     previous = None
10    stop = False
11    while current != None and not stop:
12        if current.get_item() > item:
13            stop = True
14        else:
15            previous = current
16            current = current.get_next()
```

```
17    temp = Node(item)
18    if previous == None:
19        temp.set_next(self.head)
20        self.head = temp
21    else:
22        temp.set_next(current)
23        previous.set_next(temp)
24
```

- » 라인 21-23: else, (1) temp에 할당된 N_{new} 가 current에 할당된 노드(i.e., N_{old})를 참조하게 하고, (2) previous에 할당된 노드(i.e., N_{old} 이전 노드)는 N_{new} 를 참조하게 함



우선순위 큐 (8/9)

□ 우선순위 큐의 구현 contd.

- 정렬된 단순 연결 리스트를 이용한 구현 contd.

- dequeue() **시간복잡도: O(1)**

- 연결 리스트의 첫 노드 삭제 및 해당 노드가 저장하고 있는 item 반환

```
25 def dequeue(self):
26     if self.head == None:
27         return None
28     else:
29         temp = self.head
30         dequeued_item = temp.get_item()
31         self.head = self.head.get_next()
32         return dequeued_item
```

```
34 if __name__ == "__main__":
35     q = PriorityQueue()
36     q.enqueue(3)
37     q.enqueue(1)
38     q.enqueue(2)
39     q.enqueue(4)
40     print(q.dequeue())
41     print(q.dequeue())
42     print(q.dequeue())
43     print(q.dequeue())
44     print(q.dequeue())
```

```
1
2
3
4
None
```

결과

일련의 우선순위 큐 연산과 출력

- » 라인 26-27: if, 빈 연결 리스트라면 None 반환
- » 라인 28-32: else, 연결 리스트의 첫 번째 노드 N 삭제 및 N에 저장된 item 반환이므로 N을 삭제하기 전에 변수 temp가 N을 참조하도록 하고(라인 29), 변수 dequeued_item에 N에 저장된 item을 참조하도록 하고(라인 30), 그 후 head가 N 다음 노드를 참조하게 함으로써 N을 삭제하고(라인 31), 마지막으로 item 반환(라인 32)

우선순위 큐 (9/9)

□ 우선순위 큐의 구현과 enqueue·dequeue 연산의 시간 복잡도 비교

구현	enqueue	dequeue
Python 리스트(동적 배열)	$O(1)$: 첫 항목 삽입	$O(N)$: 삭제 항목 탐색과 자리 이동
정렬된 동적 배열	$O(N)$: 삽입 항목 위치 탐색과 자리 이동	$O(1)$: 마지막 항목 삭제 및 반환
정렬된 단순 연결 리스트	$O(N)$: 삽입 위치 탐색	$O(1)$: (head가 가리키는) 첫 노드 삭제 및 반환
힙(Heap)	$O(\log N)$	$O(\log N)$

NOTE: 위의 시간 복잡도 비교는 본 강의에서 사용한 구현 방법에 따른 비교이므로 본 강의에서 사용한 구현 방법과 다르다면 위의 비교와 다른 결과도 존재할 수 있음.
단, 힙을 이용한 우선순위 큐 구현 시 enqueue·dequeue의 시간 복잡도는 반드시 $O(\log N)$

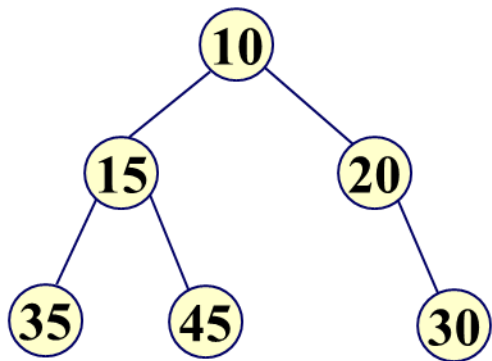
힙(Heap)을 이용한 우선순위 큐 구현의 Motivation

- 우선순위 큐는 현재 우선순위가 가장 높은 항목을 하나씩 삭제 및 반환하는 목적으로 사용
- 현재 우선순위가 가장 높은 항목을 하나씩 삭제 및 반환할 경우 모든 항목을 정렬시킬 필요가 없음

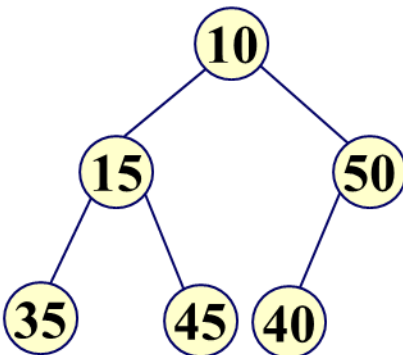
힙 (1/17)

□ 힙(Heap)

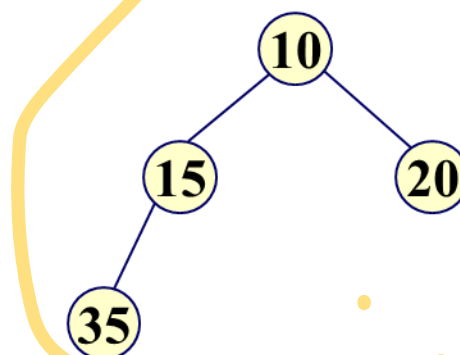
- 힙은 완전 이진 트리로서 부모 노드 키 값의 우선순위가 자식 노드 키 값의 우선순위보다 높은(힙속성: **Heap Property**) 자료 구조
 - 힙은 완전 이진 트리의 형태와 노드의 키 값에 대한 힙의 조건(힙 속성)을 반드시 유지해야 함
- 키 값이 작을수록 높은 우선순위를 가진다고 가정했을 때, 다음 중 어느 트리가 힙일까?



①



②



③

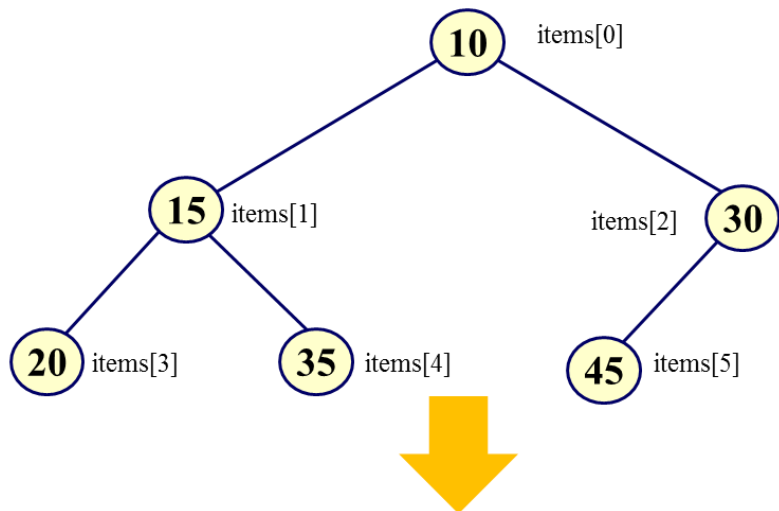
- 힙의 종류
 - **MAX 힙**: 키 값이 큰 항목이 우선순위가 높은 것으로 간주하므로 루트 노드의 키가 가장 큼
 - **MIN 힙**: 키 값이 작은 항목이 우선순위가 높은 것으로 간주하므로 루트 노드의 키가 가장 작음(본 강의는 MIN 힙에 초점을 맞춤)

힙 (2/17)

부모 노드와 자식 노드 간의 관계는 Python 리스트의 인덱스 연산으로 알 수 있음(이전 강의 참조)

□ 힙의 구현

- Python 리스트(동적 배열)로 구현



INDEX:	0	1	2	3	4	5
	10	15	30	20	35	45

- items[i]에 저장되어 있는 노드의 왼쪽 자식 노드는 $T[2 \cdot i + 1]$ 에 위치
- items[i]에 저장되어 있는 노드의 오른쪽 자식 노드는 $T[2 \cdot (i + 1)]$ 에 위치
- items[i]에 저장되어 있는 노드의 부모 노드는 $T[(i - 1) // 2]$ 에 위치(단, $i > 0$)
- Python 리스트(혹은 동적 배열)로 구현한 힙의 노드 총 수가 N일 때, 단말 노드들 중 첫 번째 단말 노드의 인덱스 값을 알 수 있음

$\lfloor \frac{N}{2} \rfloor$, 즉, $(N // 2)$ 부터 $N - 1$ 까지의 인덱스 값을 가지는 노드들이 단말 노드
예를 들어, 옆의 힙에서 단말 노드는 items[3], items[4], items[5]

- 힙에 적용 가능한 주요 연산

- BinaryHeap(array = []): 빈 힙 생성 혹은 item들의 array를 담고 있는 Python 리스트 생성
- insert(key): 힙에 새 노드 삽입(NOTE: 노드가 하나의 값, i.e., 하나의 속성을 가지고 있다고 가정), 우선순위 큐의 enqueue()
- extract_min(): 힙에서 루트 노드 삭제 및 (키 값) 반환, 우선순위 큐의 dequeue()
- size(): 힙의 사이즈 반환
- build_heap(array): 입력 받은 item들의 array를 힙으로 변환

힙 (3/17)

□ 힙의 구현 contd.

● 힙 객체를 위한 클래스 정의

```
1 class BinaryHeap:
2     def __init__(self, array = []):
3         self.items = array
4
5     def size(self):
6         return len(self.items)
7
8     def swap(self, i, j):
9         self.items[i], self.items[j] = self.items[j], self.items[i]
10
11    def insert(self, key):
12        self.items.append(key)
13        self.upheap(self.size() - 1)
14
15    def extract_min(self):
16        if self.size() == 0:
17            print("Heap is empty.")
18            return None
19        minimum = self.items[0]
20        self.swap(0, -1)
21        del self.items[-1]
22        self.downheap(0)
23        return minimum
24
25    def downheap(self, i):
26        while 2*i + 1 <= self.size()-1:
27            k = 2*i + 1
28            if k < self.size()-1 and self.items[k] > self.items[k+1]:
29                k += 1
30            if self.items[i] < self.items[k]:
31                break
32            self.swap(i, k)
33            i = k
```

계속

```
35    def upheap(self, i):
36        while i > 0 and self.items[(i-1)//2] > self.items[i]:
37            self.swap(i, (i-1)//2)
38            i = (i-1)//2
39
40    def build_heap(self):
41        for i in range(len(self.items)//2 - 1, -1, -1):
42            self.downheap(i)
43
44    def print_heap(self):
45        for i in range(0, self.size()):
46            print(self.items[i], end = ' ')
47        print("\nSize of Heap = ", self.size())
```

힙 (4/17)

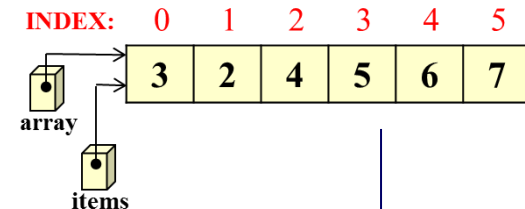
□ 힙의 구현 contd.

- `BinaryHeap(array = [])` Python의 경우 시간복잡도: $O(1)$

```
2 def __init__(self, array = []):
3     self.items = array
```

- 빈 힙 생성 혹은 Python 리스트 형태로 입력 받은 item들의 array를 담고 있는 Python 리스트 생성
 - 인자가 없다면 Python 리스트를 이용하는 빈 힙 생성 후 멤버 변수 items가 참조하게 함
 - Python 리스트 형태로 item들의 array를 인자로 받으면 멤버 변수 items가 이를 참조하게 함

```
array = [3, 2, 4, 5, 6, 7]
bheap = BinaryHeap(array)
```



- `size()` 시간복잡도: $O(1)$

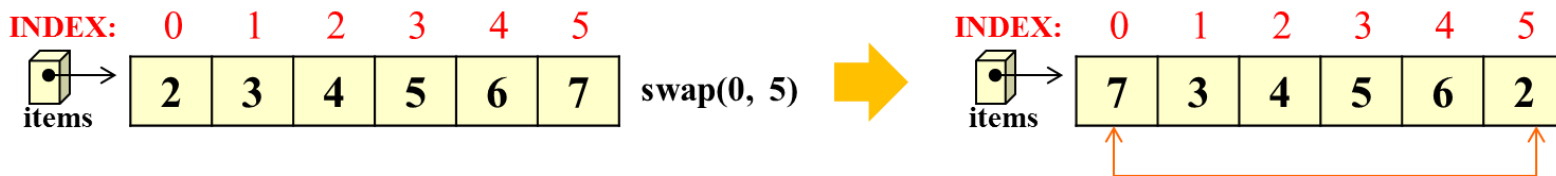
```
5 def size(self):
6     return len(self.items)
```

- 힙의 사이즈 반환

- `swap(i, j)` 시간복잡도: $O(1)$

```
8 def swap(self, i, j):
9     self.items[i], self.items[j] = self.items[j], self.items[i]
```

- 힙의 두 노드 위치를 바꿈



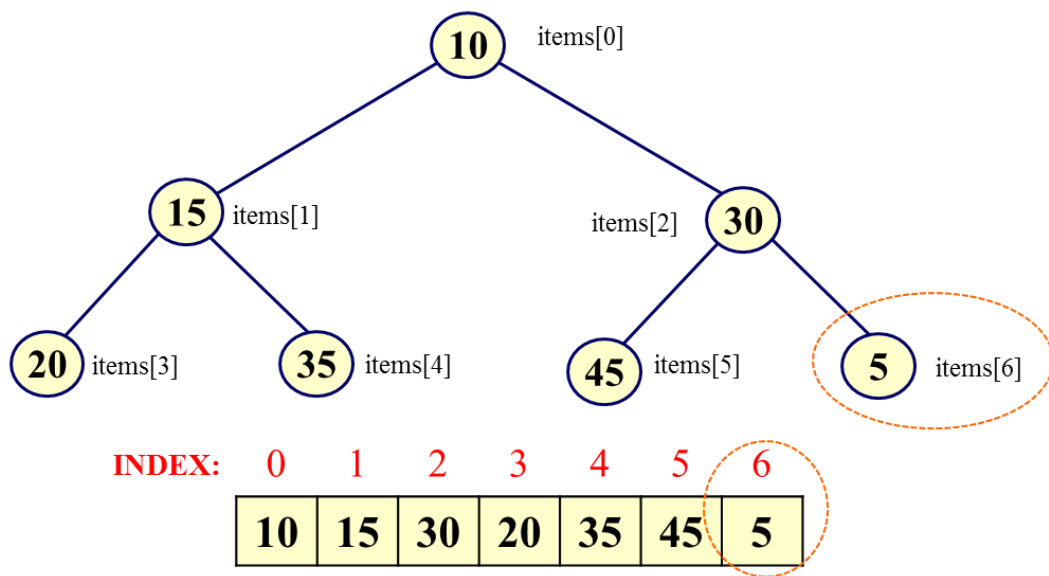
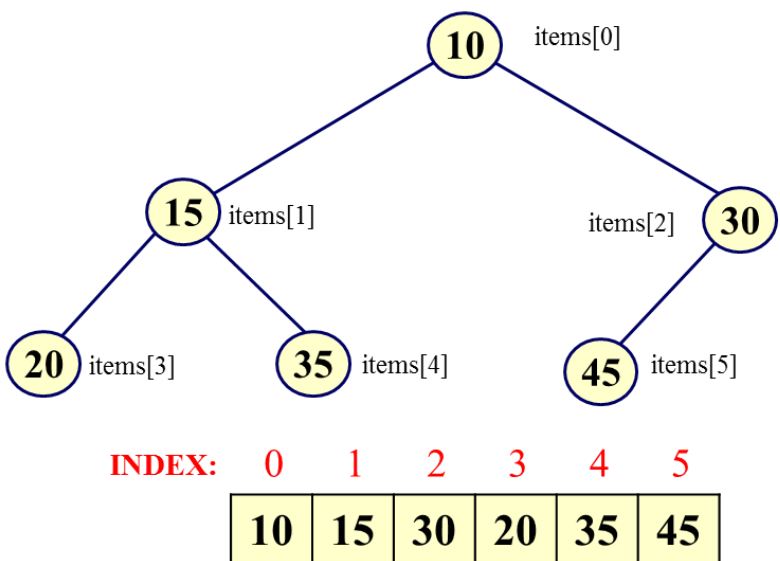
힙 (5/17)

□ 힙의 구현 contd.

-힙 삽입 알고리즘-

1. **완전 이진 트리의 형태를 유지하기 위해** 힙의 마지막 위치에 있는 노드(즉, Python 리스트의 마지막 항목)의 바로 다음 위치에 새로운 노드 **N**을 저장
2. 마지막 노드(i.e., N)로부터 **루트 노드 방향으로 올라가면서**(i.e., 레벨을 감소 시키면서) (1) 부모 노드와 키 값을 비교하고, (2) 부모 노드의 키 값이 더 크면 위치를 서로 교환하는 연산을 **힙 속성이 만족될 때까지 반복** 진행
 - 2의 과정은 단말 노드로부터 위로 올라가면서 수행되므로 **Upheap**이라고 부름

예: 힙에 새로운 노드 5를 삽입하는 경우

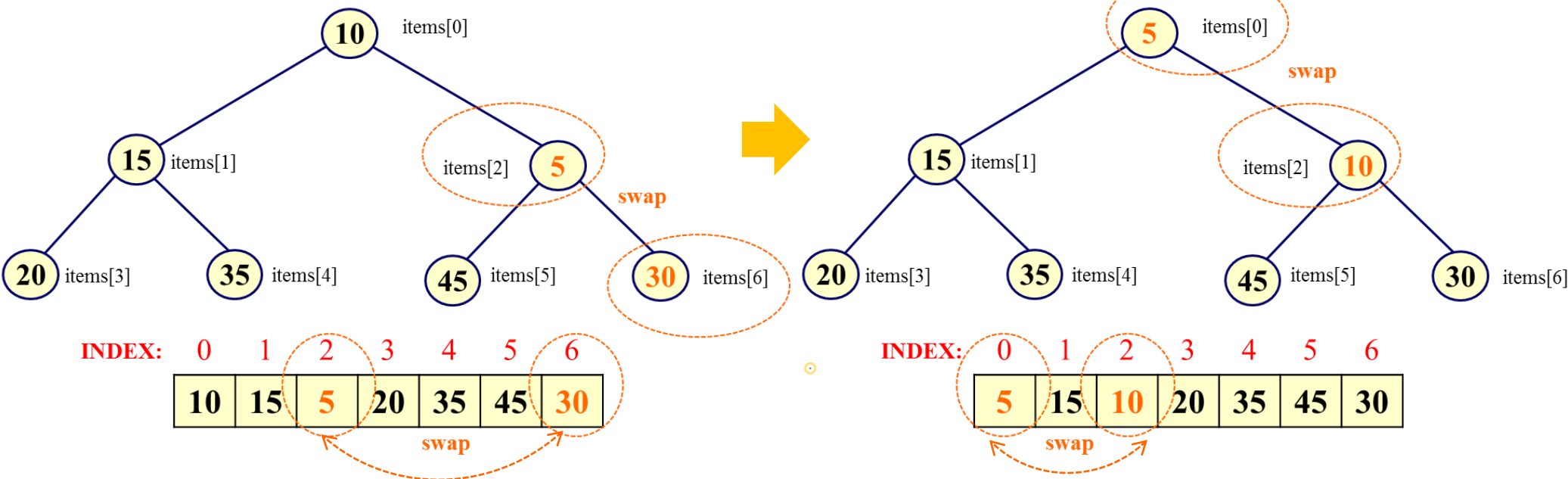


(1) 마지막 노드 45 다음 위치에 새로운 노드 5 삽입

힙 (6/17)

□ 힙의 구현 contd.

예: 힙에 새로운 노드 5를 삽입하는 경우 contd.



(2) 5의 부모 노드 30과 키 값 비교 후 5의 키 값이 더 작으므로 위치 교환(swap) (3) 5의 부모 노드 10과 키 값 비교 후 5의 키 값이 더 작으므로 위치 교환(swap)

[힙의 삽입]

참고: 주우석, "IT CookBook, C-C++로 배우는 자료구조론," 한빛미디어(주), 2014

“신입사원이 오면 일단 말단 자리에 앉힌 다음, 능력껏 위로 진급시키는 것(Upheap, Promotion)”

[연습 1] 다음에 주어진 키들이 순서대로 empty인 MIN 힙에 삽입되었을 경우 결과를 그리시오.

80, 40, 70, 30, 60, 20, 50, 10

[연습 2] 다음에 주어진 키들이 순서대로 empty인 MAX 힙에 삽입되었을 경우 결과를 그리시오.

10, 50, 20, 60, 30, 70, 40, 80

힙 (7/17)

□ 힙의 구현 contd.

- `insert(key)` 시간복잡도: $O(\log N)$

Proof:

최악의 경우 힙의 높이 $h = \lceil \log_2(N+1) \rceil - 1$ 만큼 `upheap(swap)`

```
11 def insert(self, key):
12     self.items.append(key)
13     self.upheap(self.size() - 1)
```

- 힙에 새로운 노드 삽입
 - 라인 12: 완전 이진 트리의 형태를 유지하기 위해 새로운 노드 N 을 Python 리스트 `items`의 맨 끝에 저장
 - 라인 13: 위로 올라가며(레벨을 감소시키며) 힙 속성을 회복하기 위해 **upheap 메소드 호출** (인자는 N 의 인덱스 값)
- `upheap(i)`: 아래서 위로 올라가며 힙 속성을 회복

```
35 def upheap(self, i):
36     while i > 0 and self.items[(i-1)//2] > self.items[i]:
37         self.swap(i, (i-1)//2)
38         i = (i-1)//2
```

- 라인 36: Python 리스트 `items`의 인덱스 i 번째에 있는 노드(현재 노드)가 (1) 루트 노드가 아니고, (2) 자신의 부모 노드 키 값이 자신의 키 값보다 클 때까지 `while`-루프를 진행
- 라인 37: 현재 노드와 부모 노드의 위치를 교환
- 라인 38: 인덱스 값 i 를 위치 교환이 끝난 현재 노드의 인덱스 값으로 변경(레벨을 감소시킴으로써 위로 한 단계 올라가서 반복)

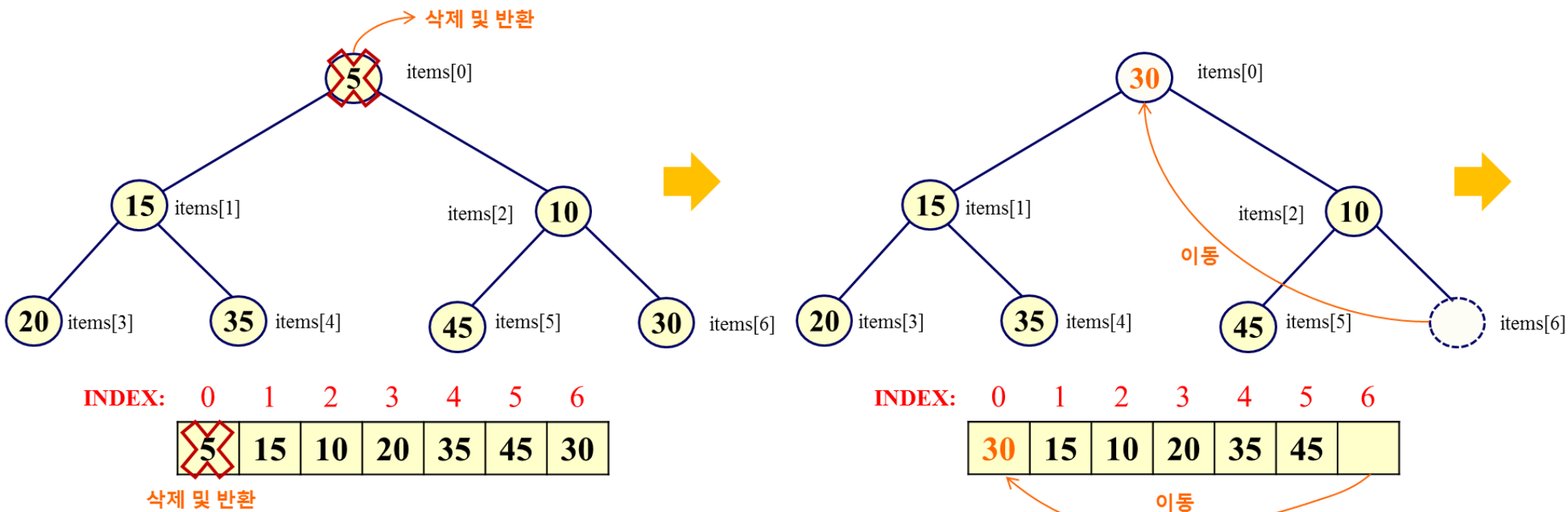
힙 (8/17)

□ 힙의 구현 contd.

-힙 삭제 및 반환 알고리즘-

1. 완전 이진 트리의 형태를 유지하기 위해 **루트 노드 삭제 및 반환** 후 힙의 가장 마지막 위치의 노드 **N**을 루트 노드 위치로 이동시킴
2. 루트 노드(i.e., N)로부터 시작하여 단말 노드 방향으로 내려가면서 (1) 자식 노드들 중에서 더 작은 키 값을 가진 자식 노드(승자)와 키 값을 비교하고, (2) 자식 노드의 키 값이 더 작으면 위치를 서로 교환하는 연산을 **힙 속성이 만족될 때까지 반복** 진행
 - 2의 과정은 루트 노드로부터 아래로 내려가면서 수행되므로 **Downheap**이라고 부름

예: 힙에서 루트 노드 5를 삭제하는 경우

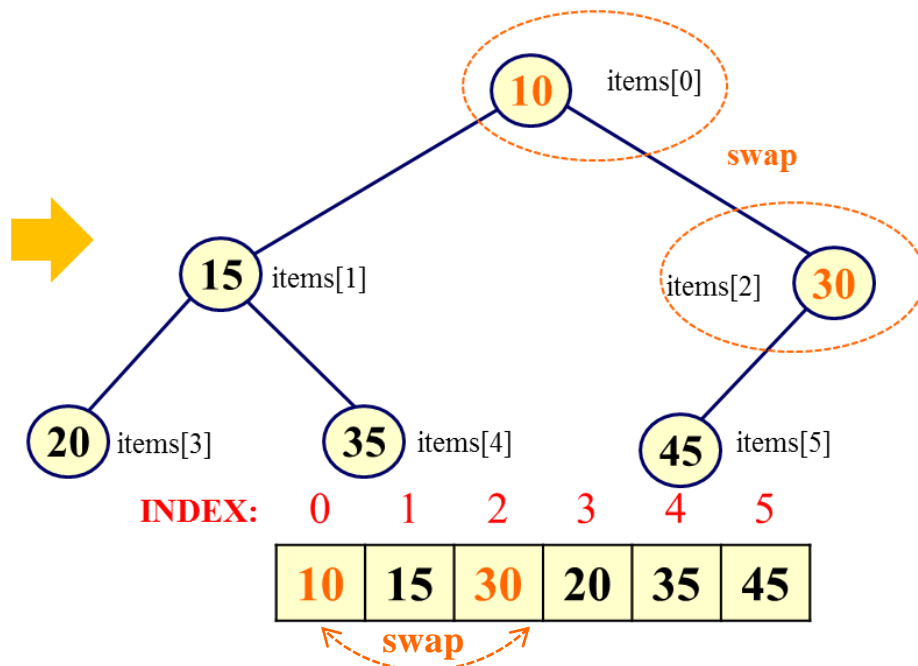
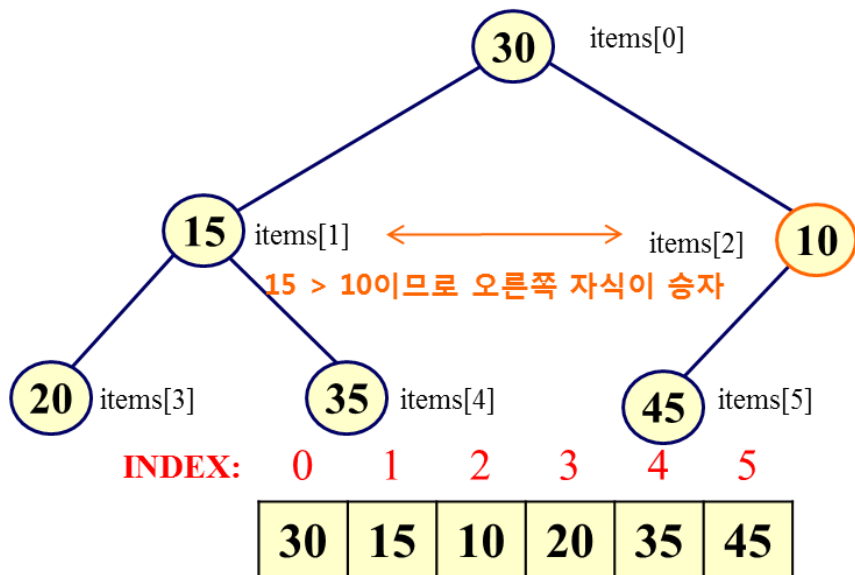


(1) 마지막 노드 30을 루트 노드 위치로 이동시킴

힙 (9/17)

□ 힙의 구현 contd.

예: 힙에서 루트 노드 5를 삭제하는 경우 contd.



(2) 5자식 노드들 중 키 값이 10인 오른쪽 자식 노드가 승자

(3) 오른쪽 자식 노드 10과 키 값 비교 후 10의 키 값이 더 크므로 위치 교환(swap)

[힙의 삭제 및 반환]

참고: 주우석, "IT CookBook, C-C++로 배우는 자료구조론," 한빛미디어(주), 2014

“사장자리가 비면 말단 사원을 그 자리에 앉힌 다음, 바로 아래 부하직원보다는 능력이 좋다고 판단될 때까지 강등시키는 것(Downheap, Demotion)”

[연습 1] 다음의 최소힙에서 삭제 및 반환 연산을 수행한 후의 최소힙을 그리시오.

10, 20, 30, 40, 35, 70, 50, 80, 45

힙 (10/17)

□ 힙의 구현 contd.

Proof:

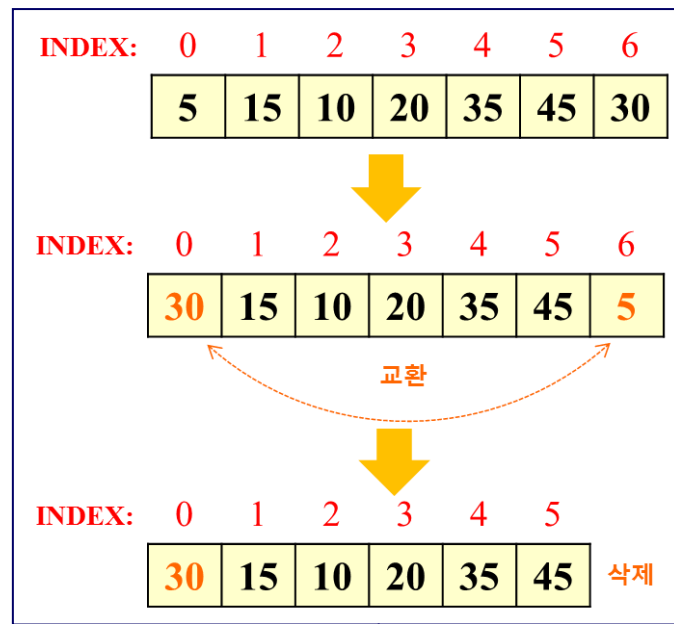
최악의 경우 힙의 높이 $h = \lceil \log_2(N+1) \rceil - 1$ 만큼 `downbheap(swap)`

- `extract_min()` 시간복잡도: $O(\log N)$

```
15 def extract_min(self):
16     if self.size() == 0:
17         print("Heap is empty.")
18         return None
19     minimum = self.items[0]
20     self.swap(0, -1)
21     del self.items[-1]
22     self.downheap(0)
23     return minimum
```

– 힙에서 루트 노드 삭제 및 (키 값) 반환

- 라인 16-18: 힙이 empty이면 “Heap is empty.” 출력 후 None 반환
- 라인 19: 키 값 반환을 위한 지역 변수 minimum에 루트 노드를(루트 노드의 키 값을) 할당
- 라인 20-21: 루트 노드 R과 힙의 가장 마지막에 있는 노드 N의 위치 교환 후 R 삭제
- 라인 22: 아래로 내려가며 힙 속성을 회복하기 위해 **downheap 메소드 호출** (인자는 N의 인덱스 값)
- 라인 23: minimum에 할당한 값을 반환
- 라인 41-42: self.items의 마지막 비단말 노드를 시작으로 루트노드까지의 각 노드를 루트로 하는 서브 트리에 대해 `downheap()` 메소드를 호출하여 힙 속성을 충족시킴



힙 (11/17)

□ 힙의 구현 contd.

- `extract_min()` contd.

- `downheap(i)`: 위에서 아래로 내려가며 힙 속성을 회복

```
25     def downheap(self, i):
26         while 2*i + 1 <= self.size()-1:
27             k = 2*i + 1
28             if k < self.size()-1 and self.items[k] > self.items[k+1]:
29                 k += 1
30             if self.items[i] < self.items[k]:
31                 break
32             self.swap(i, k)
33             i = k
```

- 라인 26: Python 리스트 `items`의 인덱스 `i`번째에 있는 노드(현재 노드)의 왼쪽 자식이 힙에 존재할 때까지 `while`-루프를 진행
- 라인 27: 왼쪽 자식의 인덱스 값을 지역 변수 `k`에 할당
- 라인 28-29: 왼쪽과 오른쪽 자식의 승자를 결정한 후, `k`에 승자의 인덱스 값을 할당
 - » (1) `k == self.size() - 1`이라면 오른쪽 자식이 존재하지 않는다는 의미이므로 라인 29를 실행하지 않으며, (2) `self.items[k] < self.items[k+1]`이라면 왼쪽 자식이 승자이므로 라인 29를 실행하지 않으며, (3) `self.items[k] == self.items[k+1]`라면 승자가 없으므로 라인 29를 실행하지 않음
- 라인 30-31: 현재 노드(인덱스 `i`번째 노드)가 자식 승자보다 작으면(우선순위가 높으면), 루프를 중단
- 라인 32: 현재 노드와 승자인 자식 노드의 위치를 교환
- 라인 33: 인덱스 값 `i`를 위치 교환이 끝난 현재 노드의 인덱스 값으로 변경(레벨을 증가시킴으로써 아래로 한 단계 올라가서 반복)

힙 (12/17)

□ 힙의 구현 contd.

-Python 리스트를 힙으로 만들기(Heapify)-

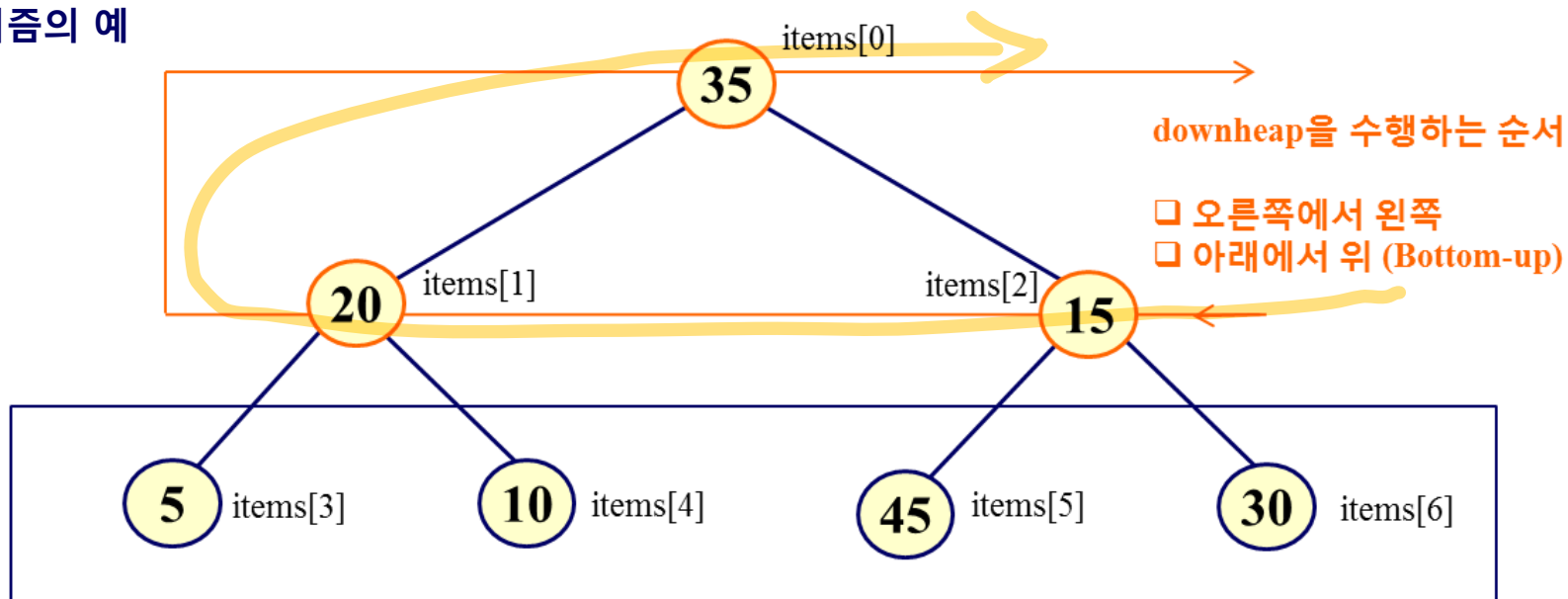
1. 하향식(Top-down) 알고리즘

- Python 리스트 items의 각 항목을 차례로 빈 힙에 삽입(insert 메소드 사용)
- Python 리스트 items의 사이즈가 N이라고 했을 때, 각 항목을 하나 삽입할 때의 시간 복잡도가 $O(\log N)$ 이고 N 개의 항목을 삽입해야 하므로 **하향식 알고리즘의 시간 복잡도는 $O(N \log N)$**

2. 상향식(Bottom-up) 알고리즘

- 사이즈가 N인 Python 리스트 items 내의 각 항목이 임의의 순서로 저장되어 있을 때, (1) 이를 이진 트리로 간주하고, (2) items[N//2 - 1] 위치의 노드(i.e., 마지막 비단말 노드)를 시작으로 items[0] 위치의 노드(i.e., 루트 노드)까지의 각 노드를 루트 노드로 하는 서브 트리에 대해 downheap() 메소드를 사용하여 힙 속성을 충족시킴 (downheap을 수행하는 순서는 오른쪽에서 왼쪽 · 아래에서 위 순서) → **시간 복잡도는 $O(N)$**

상향식 알고리즘의 예

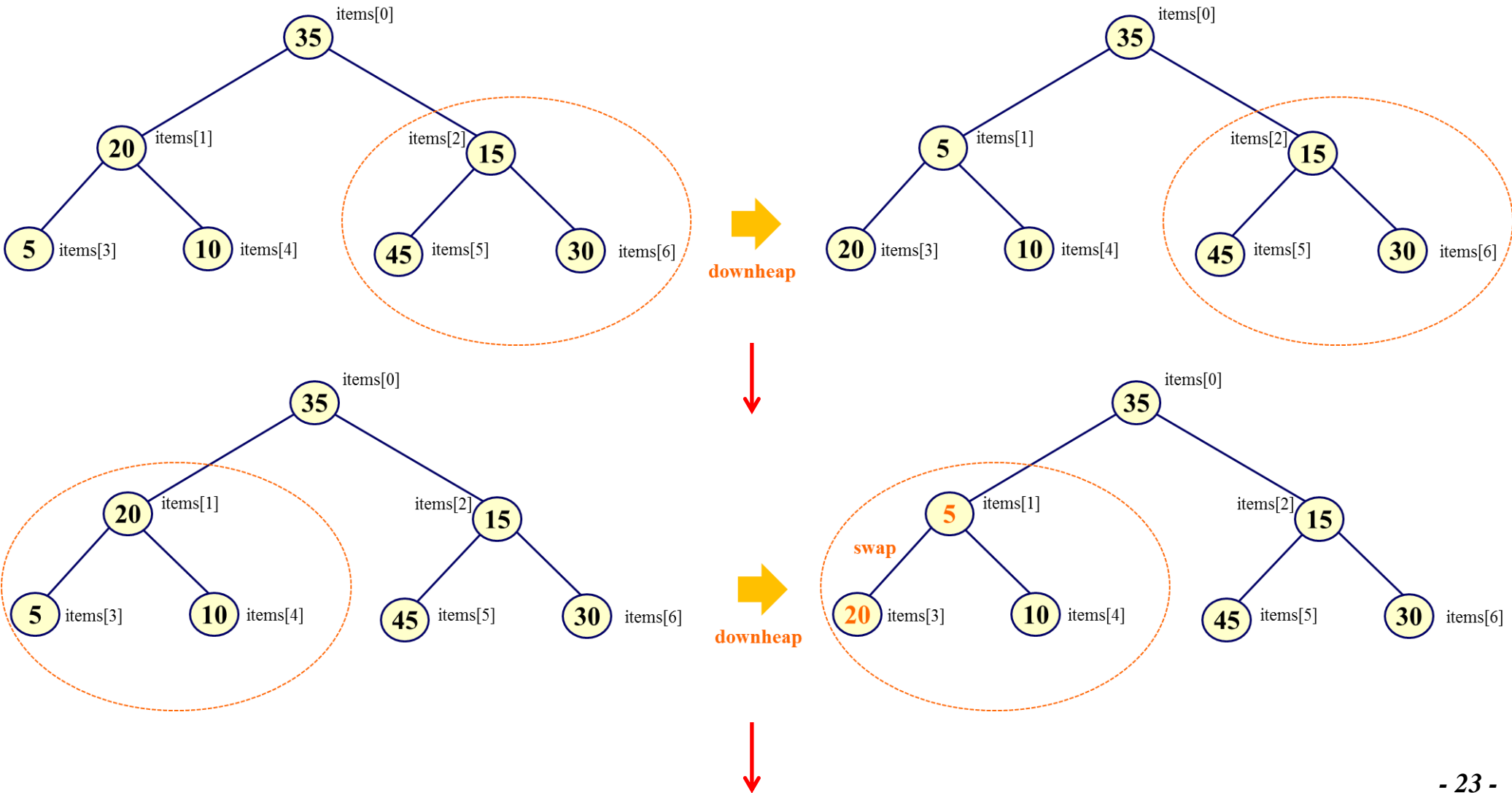


각 단말 노드를 루트로 하는 서브 트리에 대해 downheap을 수행하지 않는 이유는 이미 heap이기 때문

힙 (13/17)

□ 힙의 구현 contd.

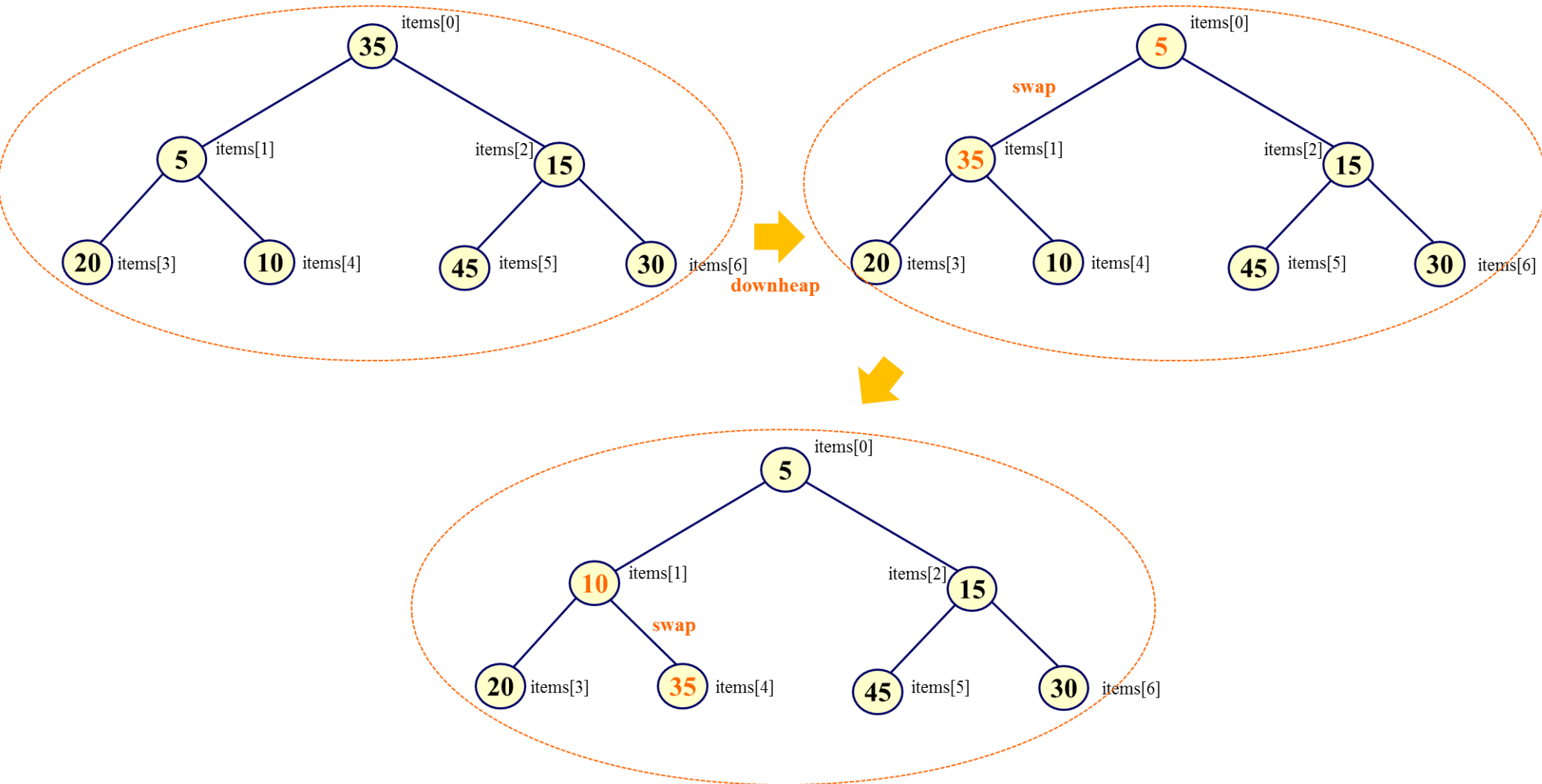
상향식 알고리즘의 예 contd.



힙 (14/17)

□ 힙의 구현 contd.

상향식 알고리즘의 예 contd.



힙 (15/17)

□ 힙의 구현 contd.

- `build_heap(array)` **Heapify, 시간복잡도: O(N)**

```
40 def build_heap(self):
41     for i in range(len(self.items)//2 - 1, -1, -1):
42         self.downheap(i)
```

– 입력 받은 item들의 array를 힙으로 변환

- 라인 41-42: `self.items`의 마지막 비단말 노드를 시작으로 루트노드까지의 각 노드를 루트로 하는 서브 트리에 대해 `downheap()` 메소드를 호출하여 힙 속성을 충족시킴

– 시간 복잡도 분석

- 노드의 수가 N(분석의 단순성을 위해 $N = 2^k - 1$ 로 가정, k는 임의의 상수)인 힙의 각 층에 있는 노드의 수를 살펴보면:

» 최하위 층(= 최대 레벨)인 0 층의 노드 수는 $\lceil \frac{N}{2^1} \rceil$

» 1 층의 노드 수는 $\lceil \frac{N}{2^2} \rceil$

» 2 층의 노드 수는 $\lceil \frac{N}{2^3} \rceil$

⋮

» 최상위 층(= 최소 레벨)인 h 층의 노드 수는 $\lceil \frac{N}{2^{h+1}} \rceil$

맨 아래 0 층은 본인 자신이 힙 속성을 만족하므로 비교 및 교환 0번

다음 1 층 노드는 자식과 비교 및 교환 최대 1번

- Heapify는 h = 1인 경우부터 시작하여 최상위 층의 루트 노드까지 각 노드를 루트로 하는 서브 트리에 대해 `downheap`을 수

행하므로 $f(N) = 0 \cdot \frac{N}{2^1} + 1 \cdot \frac{N}{2^2} + 2 \cdot \frac{N}{2^3} + 3 \cdot \frac{N}{2^4} + \dots = N \cdot (0 + 1 \cdot \frac{1}{4} + 2 \cdot \frac{1}{8} + 3 \cdot \frac{1}{16} + \dots) = N \cdot \sum_{h=0}^{\infty} \frac{h}{2^{h+1}} = N \cdot \sum_{h=0}^{\infty} \frac{h}{2 \cdot 2^h}$ Big O 표기법이므로 삭제 가능 - 25 -

힙 (16/17)

□ 힙의 구현 contd.

- `build_heap(array)` contd.

- 시간 복잡도 분석 contd.

$$N \cdot \sum_{h=0}^{\infty} \frac{h}{2^h} = 0 \cdot \frac{N}{2^0} + 1 \cdot \frac{N}{2^1} + 2 \cdot \frac{N}{2^2} + 3 \cdot \frac{N}{2^3} + \dots = N \cdot (0 + 1 \cdot \frac{1}{2} + 2 \cdot \frac{1}{4} + 3 \cdot \frac{1}{8} + \dots) \quad \dots \text{식 1}$$

- 식 1에 1/2를 곱하면,

$$\frac{1}{2} N \cdot \sum_{h=0}^{\infty} \frac{h}{2^h} = 0 \cdot \frac{N}{2 \cdot 2^0} + 1 \cdot \frac{N}{2 \cdot 2^1} + 2 \cdot \frac{N}{2 \cdot 2^2} + 3 \cdot \frac{N}{2 \cdot 2^3} + \dots = N \cdot (0 + 1 \cdot \frac{1}{4} + 2 \cdot \frac{1}{8} + 3 \cdot \frac{1}{16} + \dots) \quad \dots \text{식 2}$$

- 식 1에서 식 2를 빼면,

$$\frac{1}{2} N \cdot \sum_{h=0}^{\infty} \frac{h}{2^h} = N \cdot \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots \right) = N$$

Proof:

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots = \sum_{n=1}^{\infty} \left(\frac{1}{2} \right)^n = \frac{\frac{1}{2}}{1 - \frac{1}{2}} = 1$$

» 양 변에 2를 곱하면 $N \cdot \sum_{h=0}^{\infty} \frac{h}{2^h} = 2N$

» 따라서 시간 복잡도는 $O(N)$

힙 (17/17)

□ 힙의 구현 contd.

- print_heap()

```
44 def print_heap(self):
45     for i in range(0, self.size()):
46         print(self.items[i], end = ' ')
47         print("\nSize of Heap = ", self.size())
```

- 힙의 노드 및 사이즈 출력

- 일련의 힙 연산·출력 및 결과

```
49 if __name__ == "__main__":
50     array = [3, 2, 4, 5, 6, 7]
51     bheap = BinaryHeap(array)
52     bheap.build_heap(array)
53     bheap.print_heap()
54     bheap.insert(1)
55     bheap.insert(9)
56     bheap.insert(11)
57     bheap.insert(19)
58     bheap.print_heap()
59     print(bheap.extract_min())
60     print(bheap.extract_min())
61     bheap.print_heap()
```

일련의 힙 연산과 출력

```
2 3 4 5 6 7
Size of Heap = 6
1 3 2 5 6 7 4 9 11 19
Size of Heap = 10
1
2
3 5 4 9 6 7 19 11
Size of Heap = 8
```

결과