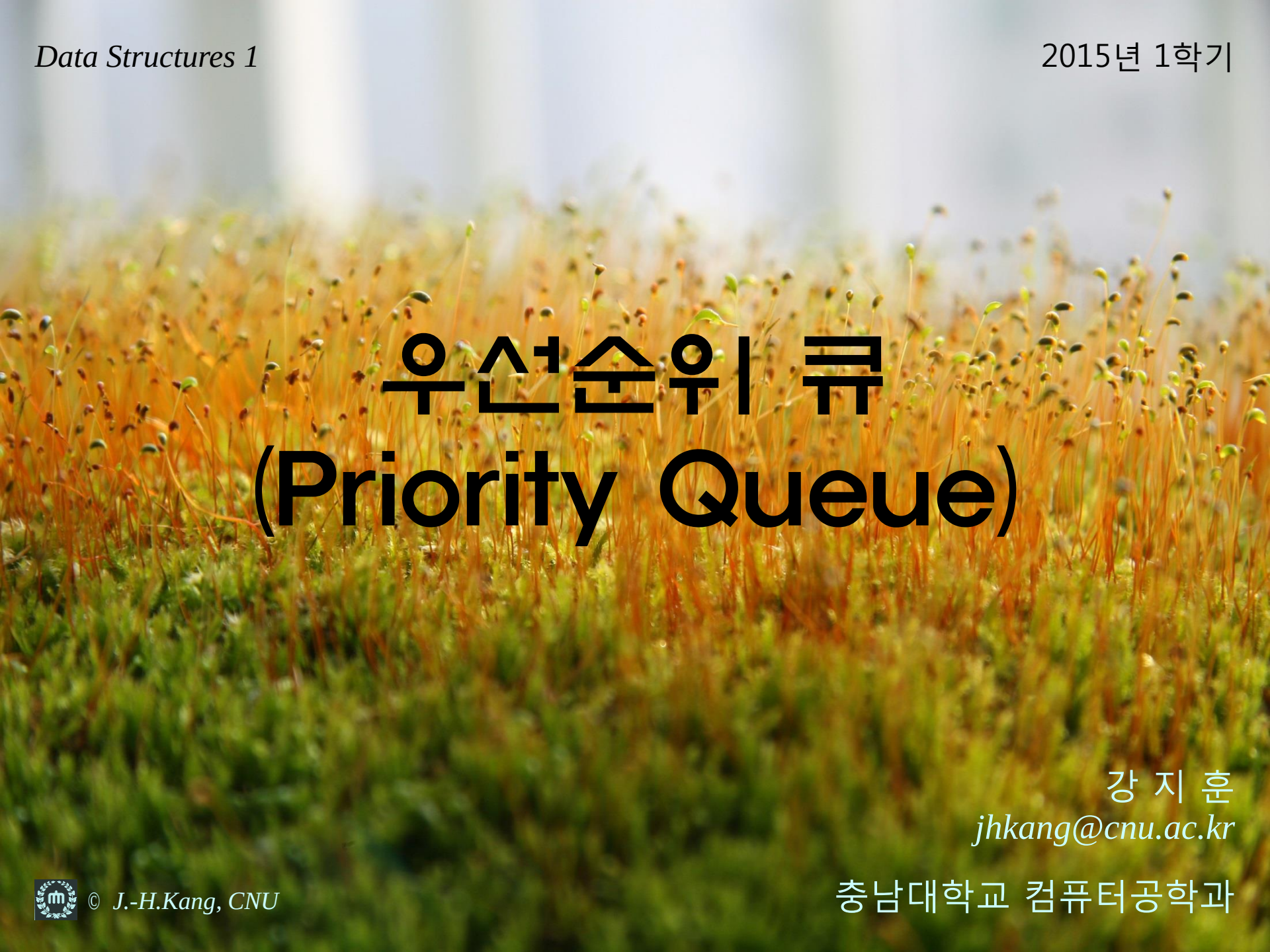


우선순위 큐
(Priority Queue)

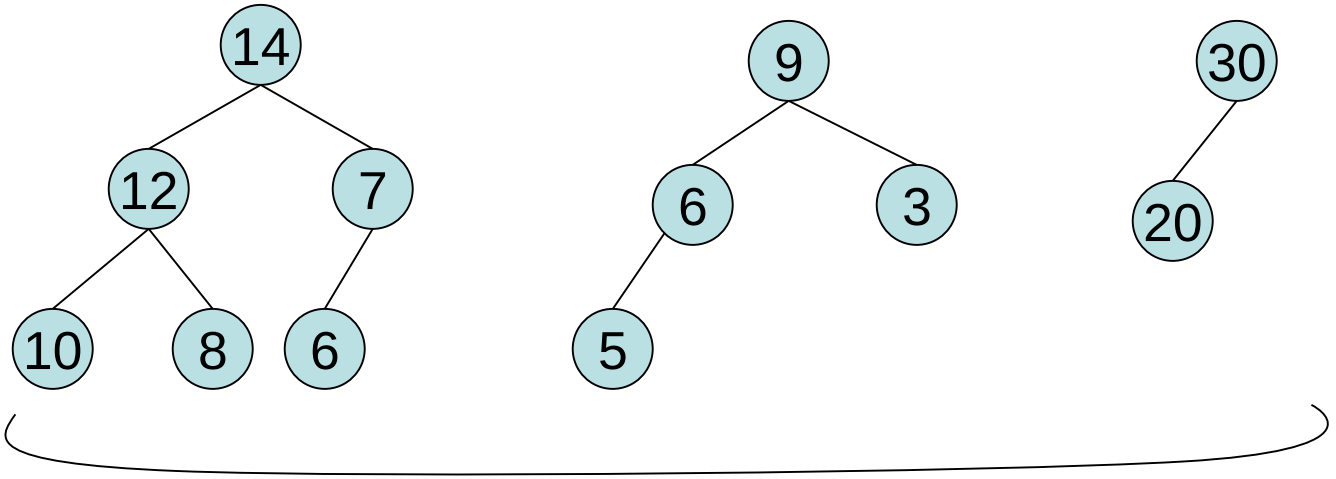
강 지 훈
jhkang@cnu.ac.kr

충남대학교 컴퓨터공학과

힢 (Heaps)

최대 힢 (Max Heaps)

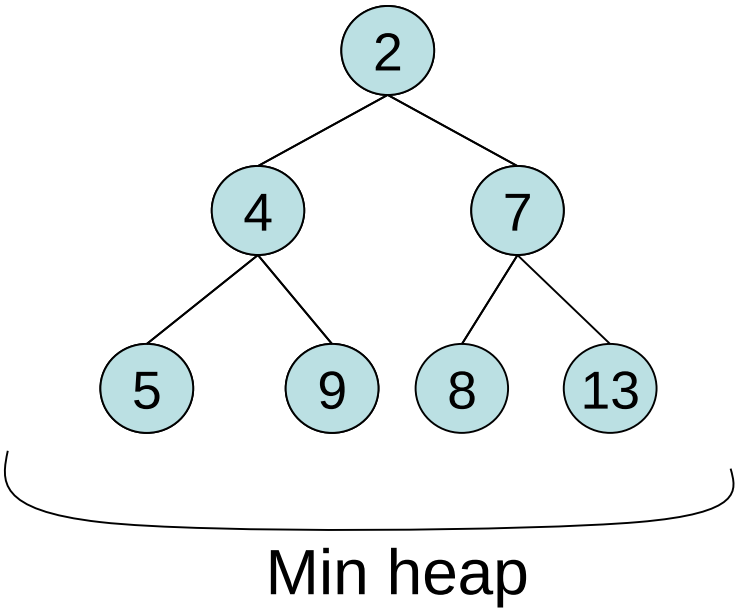
- 최대 트리 (max tree)
  - 각 노드의 키 값이 자식 노드의 키 값보다 작지 않다
- 최대 힢 (max heap)
  - 최대 트리 이면서 완전이진트리



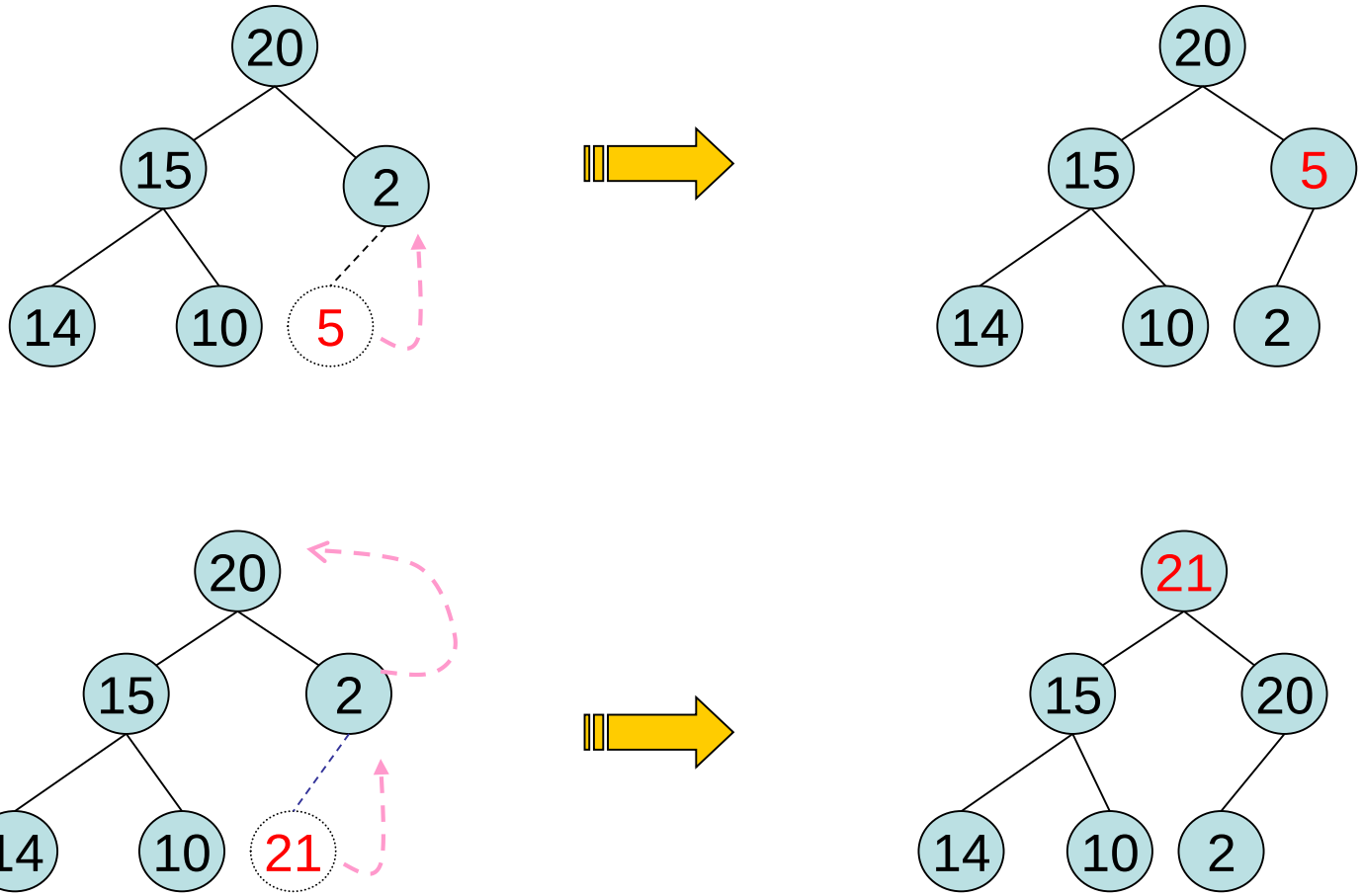
Max heaps

최소 힢 (Min Heaps)

- 최소 트리 (min tree)
  - 각 노드의 키 값이 자식 노드의 키 값보다 작지 않다
- 최소 힢 (min heap)
  - 최소 트리 이면서 완전이진트리



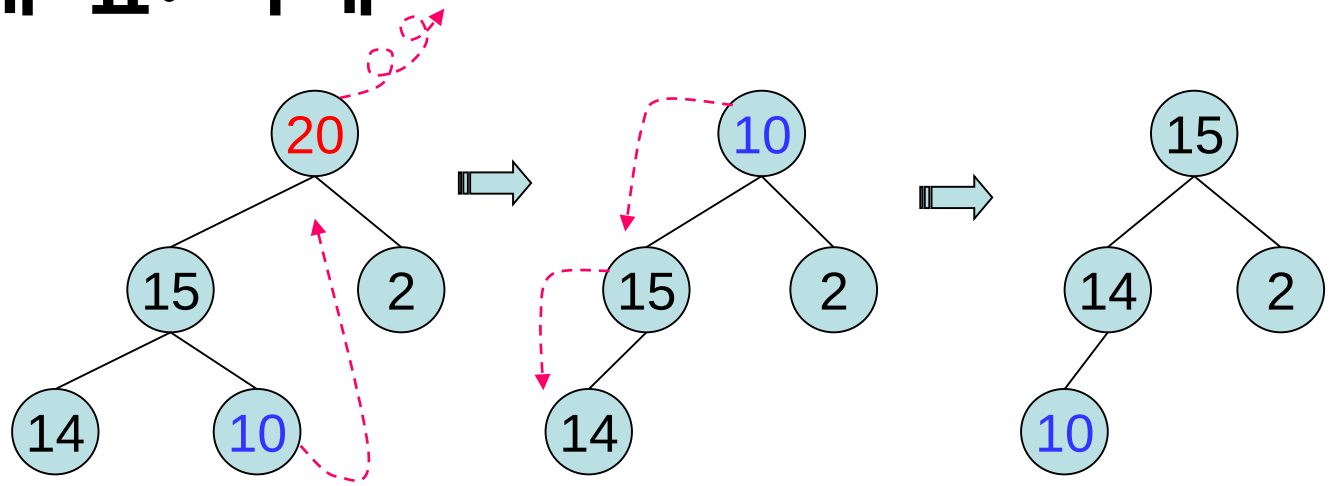
최대 힙: 삽입



삽입의 분석

- n개의 노드를 갖는 완전이진트리의 높이: $\lceil \log_2(n+1) \rceil$.
- While 반복 회수: $\lceil \log_2(n+1) \rceil$
- 시간 복잡도: $O(\log_2 n)$.

최대 힙: 삭제



- 삭제 과정
  - Root를 삭제한다. (이 삭제된 root가 함수의 return 값이다.)
  - 마지막 원소를 떼어내어 root로 가져온다.
  - child가 없을 때까지 다음을 반복한다.
    - child가 존재하면 그 중에서 큰 key 값을 갖는 child를 찾는다.
    - parent가 child보다 key 값이 작지 않으면 반복을 종료한다.
    - child를 위로 올리고, parent를 아래로 내려 보낸다.
  - 삭제된 root를 return 한다.

- 시간복잡도 : $O(\log_2 n)$

Class “MaxHeap”

MaxHeap의 공개함수

- MaxHeap 객체 사용법
  - public MaxHeap() ;
  - public boolean isEmpty();
  - public boolean isFull();
  - public int size();
  - public T max();
  - public void add(T newElement);
  - public T removeMax();
  - public void clear();

우선순위 큐 (PriorityQueue)

우선순위 큐 (Priority Queues)

- 기본 행위
  - 임의의 우선순위를 갖는 원소를 삽입
  - 가장 높은 (또는 가장 낮은) 우선순위를 갖는 원소를 삭제

- 다양한 구현이 가능

구현 방법	삽입	최대값 삭제
정렬되지 않은 배열리스트	$\Theta(1)$	$\Theta(n)$
정렬되지 연결리스트	$\Theta(1)$	$\Theta(n)$
정렬된 배열리스트	$O(n)$	$\Theta(1)$
정렬된 연결리스트	$O(n)$	$\Theta(1)$
최대 힢	$O(\log_2 n)$	$O(\log_2 n)$

힢: 모든 행위가 효율적인가?

- 구현 방법에 따른, 행위 별 시간복잡도

구현 방법	삽입	최대값 삭제	최소값 삭제	임의 원소 삭제 (찾은 후)	검색
정렬되지 않은 배열리스트	**Θ(1)**	Θ(n)	Θ(n)	O(n)	O(n)
정렬되지 않은 연결리스트	**Θ(1)**	Θ(n)	Θ(n)	**O(1)**	O(n)
정렬된 배열리스트 (오름차순)	O(n)	**Θ(1)**	Θ(n)	O(n)	**O($\log_2$n)**
정렬된 연결리스트 (내림차순)	O(n)	**Θ(1)**	Θ(n)	**O(1)**	O(n)
이진검색트리 (평균적으로)	**O($\log_2$n)**	**O($\log_2$n)**	**O($\log_2$n)**	O($\log_2$n)	**O($\log_2$n)**
최대 힢	**O($\log_2$n)**	**O($\log_2$n)**	Θ(n)	O($\log_2$n)	O(n)

- 임의의 원소를 삽입/삭제/검색한다면?
- 이진검색 트리를 우선순위 큐의 구현에 사용한다면?

Class "PriorityQueue"

PriorityQueue의 공개함수

- PriorityQueue 객체 사용법

```
public          PriorityQueue() ;

public boolean  isEmpty () ;
public boolean  isFull () ;
public int      size () ;

public T        max ();

public void     add (T anElement) ;
public T        removeMax ();
```

PriorityQueue의 구현: 비공개 인스턴스 변수

```
public class PriorityQueue <T>
{
    // 비공개 멤버 변수
    private static final int    DEFAULT_CAPACITY = 100 ;
    private static final int    ROOT = 1 ;
    private int                 _maxSize ;
    private int                 _size ;
    private <T>[]               _heap ;
    ……
```

Class "PriorityQueue"의 구현: 생성자

```
public class PriorityQueue<T>
{
    // 비공개 멤버 변수
    ……

    // 생성자
    public PriorityQueue ()
    {
        this._heap = (T[]) new Object[PriorityQueue.DEFAULT_CAPACITY+1] ;
        this._maxSize = PriorityQueue.DEFAULT_CAPACITY ;
        this._size = 0 ;
    }
```

PriorityQueue : 상태 알아보기

```
public calss PriorityQueue<T>
{
    ......

    // 상태 알아보기
    public boolean  isEmpty()
    {
        return  (this._size == 0) ;
    }

    public boolean  isFull()
    {
        return  (this._size == this._maxSize ) ;
    }

    public int size ()
    {
        return  this._size ;
    }
```

PriorityQueue : 내용 알아보기

```
public calss PriorityQueue<T>
{
    ......

    public T  max()
    {
        if ( this.isEmpty() ) {
            return   null ;
        }
        else {
            return   this._heap[PriorityQueue.ROOT] ;
        }
    }
```

PriorityQueue : add()

```
public boolean    add (T anElement)
{
    if ( this.isFull() ) {
        return   false ;
    }
    else {
        this._size++;
        int   i = this._size ;
        while ( (i > PriorityQueue.ROOT) &&
                  (anElement.comapreTo(this._heap[i/2]) > 0) )
        {
            this._heap[i] = this._heap[i/2] ;
            i /= 2 ; // ( i = i /2 ;)
        }
        this._heap[i] = anElement ;
        return   true ;
    }
}
```

PriorityQueue : removeMax()

```
public T  removeMax()
{
      if ( this.isEmpty() ) {
            return  null ;
      }
      T  rootElement = this._heap[PriorityQueue.ROOT] ;
      this._size-- ;
      if ( this._size > 0 ) {
            // 삭제 한 후에 적어도 하나의 원소가 남아 있다.
            // 그러므로 마지막 위치 (this._size+1)의 원소를 떼어내어,
            // root 위치 (1)로부터 아래쪽으로 새로운 위치를 찾아 내려간다.
            T  lastElement = this._heap[this._size+1] ;
            int  parent = PriorityQueue.ROOT ;
            int  biggerChild ;
            while ( (parent*2) <= this._size ) {
                  // child 가 존재. left, right 중에서 더 큰 key 값을 갖는 child를 biggerChild로 한다.
                  biggerChild = parent * 2 ;
                  if ( (biggerChild < this._size) && (this._heap[biggerChild].compareTo(this._heap[biggerChild+1]) < 0) ) {
                        biggerChild++ ; // right child가 존재하고, 그 값이 더 크므로, right child를 biggerChild로 한다.
                  }
                  if ( lastElement.compareTo(this._heap[biggerChild]) >= 0  ) {
                        break ; // lastElement 는 더 이상 아래로 내려갈 필요가 없다. 현재의 parent 위치에 삽입하면 된다.
                  }
                  // child 원소를 parent 위치로 올려 보낸다. child 위치는 새로운 parent 위치가 된다.
                  this._heap[parent] = this._heap[biggerChild] ;
                  parent = biggerChild ;
            }  // end while
            this._heap[parent] = lastElement ;
      }
      return  rootElement ;
}
```

실습 : 우선순위 큐의 성능 비교

실습: 우선순위 큐의 성능 비교

- 우선순위 큐를 다음의 5가지로 구현하여 성능을 비교한다.
  - Unsorted Array
  - Sorted Array (Increasing Order)
  - Unsorted Linked List
  - Sorted Linked List (Decreasing Order)
  - Max Heap
- 데이터의 크기를 변화시켜 결과를 비교한다.
  - 구현에 따른 성능 측정 비교
  - 데이터 크기에 따른 성능 측정 비교

실습: 우선순위 큐의 성능 비교

- 입력
  - 없음
  - 필요한 데이터는 프로그램에서 생성
- 출력 : 성능 측정 결과
  - 데이터 크기 변화에 따른 성능 측정 결과

- 데이터 크기
  - 2000, 4000, 6000, 8000, 10000
- 데이터 생성
  - Random number를 생성하여 사용한다.
  - 데이터에 중복이 없게 만든다.

"우선순위 큐" [끝]