자료구조: 2022년 1학기 [강의]

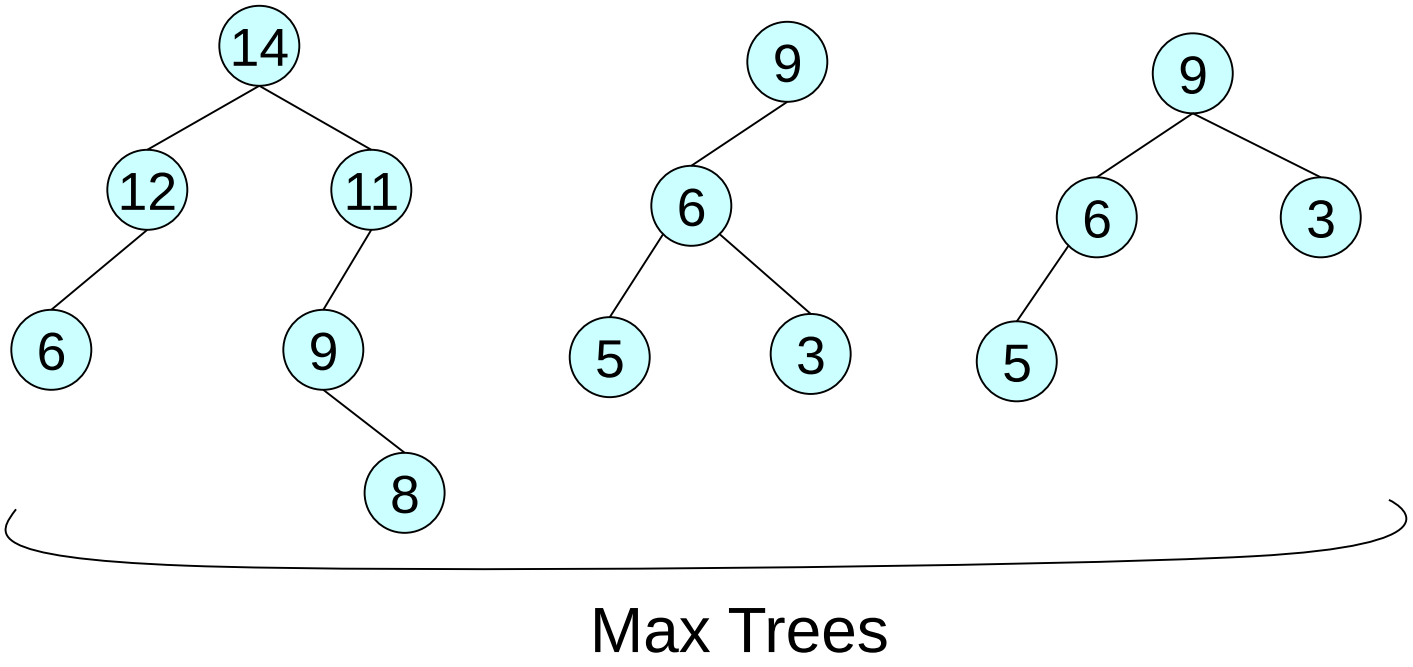
Priority Queue

강지훈
jhkang@cnu.ac.kr
충남대학교 컴퓨터융합학부

힢 (Heap)

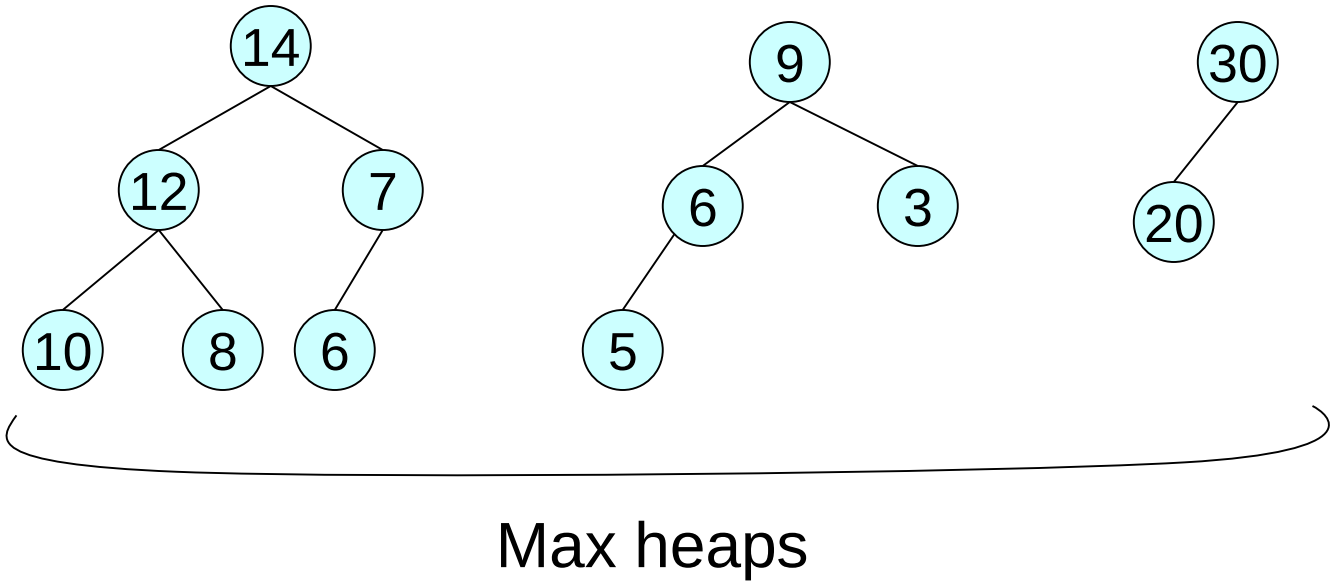
최대 트리 (Max Tree)

- 각 노드의 키 값이 자식 노드의 키 값보다 작지 않다



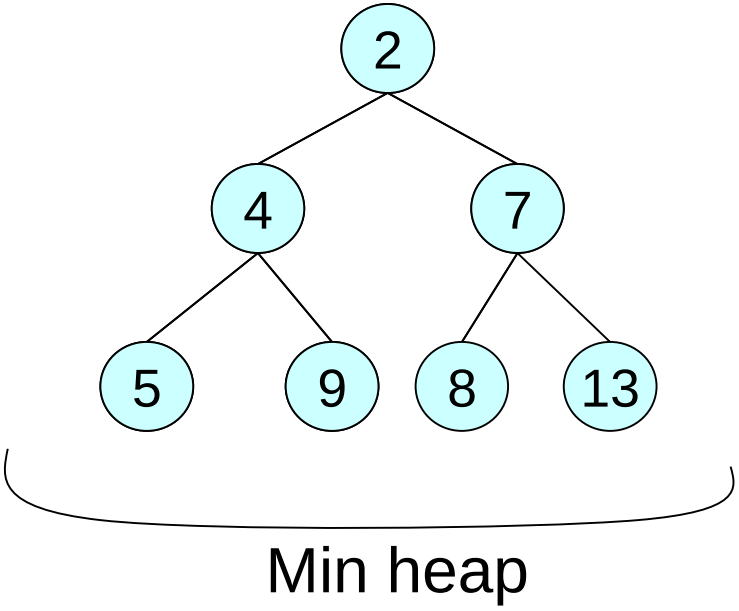
최대 힢 (Max Heap)

- 최대 트리 이면서 완전이진트리



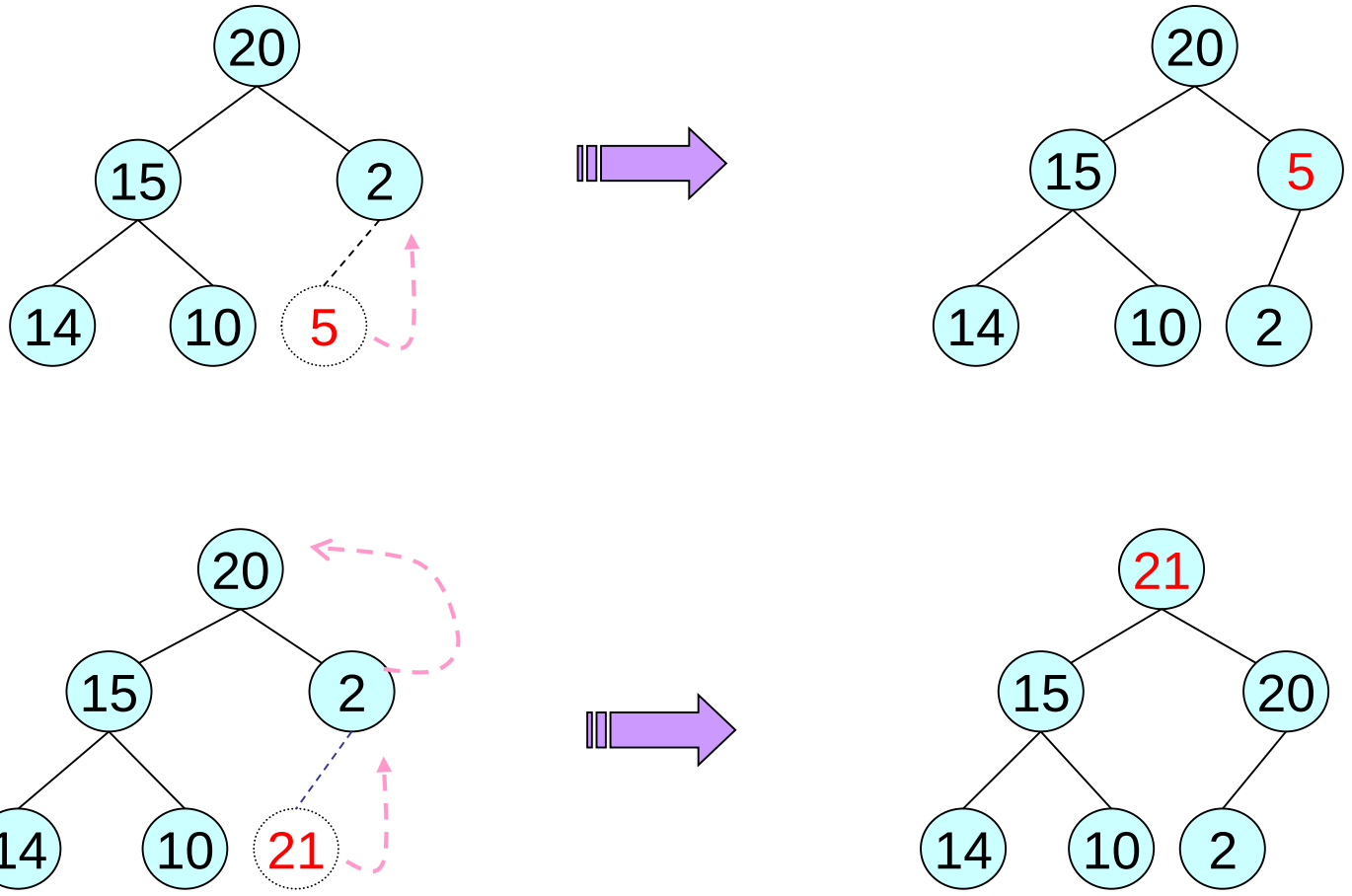
최소 힢 (Min Heap)

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



Max Heap 에서의 삽입/삭제

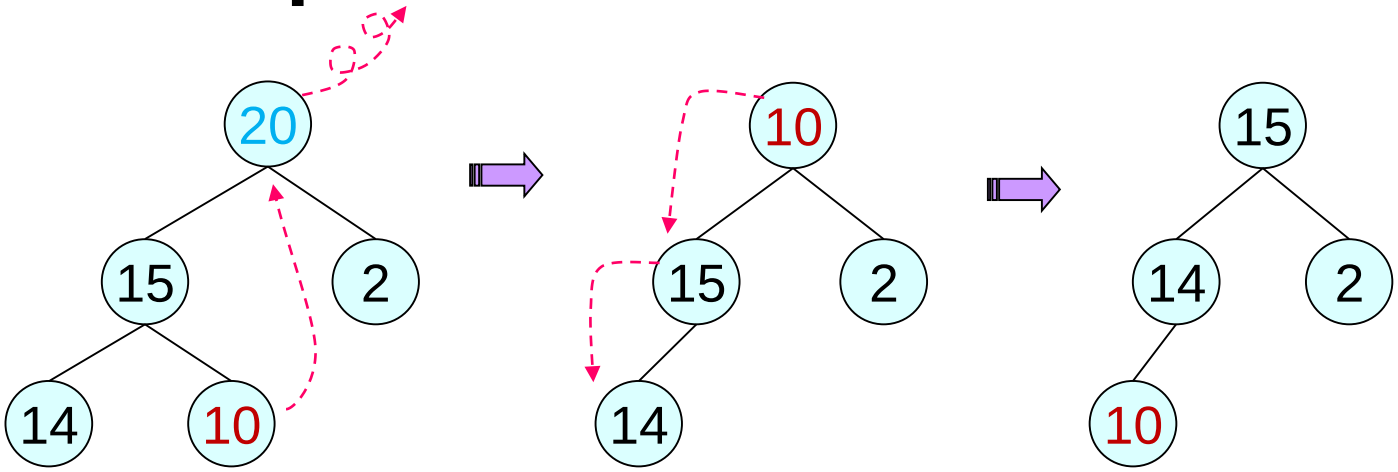
Max Heap: 삽입



삽입의 분석

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

Max Heap: 삭제



- 삭제 과정
  - Max element 를 얻기 위해 Root 를 삭제한다.
    - 이 삭제된 root 가 함수의 return 값이다.
    - 이 행위에 의해 Heap 의 모습이 망가진다. 복구가 필요하다.
  - Heap 의 마지막 원소를 떼어내어 root 로 가져온다.
    - 이 행위는 Complete Binary Tree 의 모습을 유지하게 한다.
  - Child 가 없을 때까지 다음을 반복한다: Heap 을 복구.
    - Child 가 존재하면 그 중에서 큰 key 값을 갖는 child 를 찾는다.
    - Parent 가 child 보다 key 값이 작지 않으면 반복을 종료한다.
    - Child 를 위로 올리고, parent 를 아래로 내려 보낸다.
  - 삭제된 root 를 return 한다.

- 시간복잡도 : $O(\log_2 n)$

우선순위 큐
(Priority Queue)

우선순위 큐 (Priority Queue)

기본 행위

- 임의의 우선순위를 갖는 원소를 삽입
- 가장 높은 우선순위를 갖는 원소를 삭제
  - 응용에 따라 가장 낮은 우선순위의 원소를 삭제

다양한 구현이 가능

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

힙: 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 힙	**$O(\log_2 n)$**	**$O(\log_2 n)$**	Θ(n)	$O(\log_2 n)$	O(n)

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

Abstract Class "PriorityQueue"

PriorityQueue<T>의 선언

```
public abstract class  PriorityQueue<T>
{
    public              PriorityQueue() ;

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

    public abstract  T    max () ;

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

Class "PriorityQueueByMaxHeap"의 구현

PriorityQueueByMaxHeap: 비공개 인스턴스 변수

```
public class  PriorityQueueByMaxHeap<T extends Comparable<T>>
      extends PriorityQueue<T>
{
    // 비공개 상수
    private static final int     DEFAULT_CAPACITY = 100 ;
    private static final int     ROOT = 1 ;

    // 비공개 멤버 변수
    private int                  _capacity ;
    private int                  _size ;
    private T[]                  _heap ;
    ......
```

PriorityQueueByMaxHeap: Getter/Setter

```
public class  PriorityQueueByMaxHeap<T extends Comparable<T>> extends PriorityQueue<T>
{
    // 비공개 상수
    private static final int    DEFAULT_CAPACITY = 100 ;
    private static final int    ROOT = 1 ;

    // 비공개 멤버 변수
    private int                 _capacity ;
    private int                 _size ;
    private T[]                 _heap ;

    // Getter/Setter
    private int  capacity () {
        return  this._capacity ;
    }
    private void  setCapacity (int newCapacity) {
        this._capacity = newCapacity ;
    }
    @Override
    public int  size () {
        return  this._size ;
    }
    private void  size (int newSize) {
        this._size = newSize ;
    }
    private T[]  heap() {
        return  this._heap ;
    }
    private void  setHeap (T[] newHeap) {
        this._heap = newHeap ;
    }
    ......
```

구현: 생성자

```
public class  PriorityQueueByMaxHeap<T extends Comparable<T>>
     extends PriorityQueue<T>
{
    // 비공개 상수
    ……

    // 비공개 멤버 변수
    ……

    // 생성자
    @SuppressWarnings("unchecked")
    public  PriorityQueueByMaxHeap ()
    {
        this.setCapacity (PriorityQueueByMaxHeap.DEFAULT_CAPACITY) ;
        this.setHeap ( (T[]) new Comparable[this.capacity()+1] ) ;
        this.setSize (0) ;
    }
```

상태 알아보기

```
public class  PriorityQueueByMaxHeap<T extends Comparable<T>>
     extends  PriorityQueue<T>
{
    ......

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

    @Override
    public boolean  isFull()
    {
        return  ( this.size() == this.capacity() ) ;
    }

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

PriorityQueue : 내용 알아보기

```
public class  PriorityQueueByMaxHeap<T extends Comparable<T>>
     extends  PriorityQueue<T>
{
    .....

    @Override
    public T  max()
    {
        if ( this.isEmpty() ) {
            return   null ;
        }
        else {
            return   this.heap()[PriorityQueueByMaxHeap.ROOT] ;
        }
    }
```

PriorityQueue : add()

```
@Override
public boolean    add (T anElement)
{
    if ( this.isFull() ) {
        return  false ;
    }
    else {
        this.setSize (this.size()+1) ;
        int  i = this.size() ;
        while ( (i > PriorityQueueByMaxHeap.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()

```
@Override
public T  removeMax()
{
	if ( this.isEmpty() ) {
		return  null ;
	}
	T  rootElement = this.heap()[PriorityQueueByMaxHeap.ROOT] ;
	this.setSize (this.size()-1) ;
	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 = PriorityQueueByMaxHeap.ROOT ;
		int  biggerChild ;
		while ( (parent*2) <= this.size() ) {
			// child 가 존재. left, right 중에서 더 큰 key 값을 갖는 child 를 biggerChild 로 한다.
			biggerChild = parent * 2 ;
			if ( (biggerChild < this.size()) && (this.heap()[biggerChild].compareTo(this.heap()[biggerChild+1]) < 0) ) {
				biggerChild++ ; // right child 가 존재하고, 그 값이 더 크므로, right child 를 biggerChild 로 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 ;
}
```

End of

"Priority Queue"