Data Structure

(Java programming)

Chapter 08.

Contents

- List-Based PQ
- Heap

- ***List-Based PQ***

List-Based PQ

Priority Queue

- A ***priority queue*** stores a collection of *prioritized* elements
  - Arbitrary insertions, but
  - *Priority-ordered* removals
- Exposes no notion of "position"
  - The element with *first priority* can be removed *anytime*
  - This is the fundamental difference from the position-based ADTs (e.g., Stacks, Queues, Deques, or Node lists)
- We need a special property "***key***" to prioritize the elements in a priority queue (PQ for short, henceforth)
  - The ***key*** is assigned by a user or application
  - No need to be unique, and even can be changed
  - No need to be a single number, but can be a complex property

List-Based PQ

Priority Queue ADT

- A (***key***, ***value***) pair inserted into a PQ is called "***entry***"
  - In a PQ, an element is referred as a "***value***", so we refer to an entry of PQ as a (***key***, ***value***) pair
- Main operations of the PQ ADT
  - entry **insert**(***k***, ***x***)
    Inserts an entry with key ***k*** and value ***x*** into a PQ and returns the entry
  - entry **removeMin**()
    Removes and returns an entry with the *smallest* key (i.e., an entry whose key is less than or equal to that of every other entry in the PQ)
- Additional operations
  - entry **min**()
    Returns, but does not remove, an entry with the smallest key
  - **size**(), **isEmpty**()
    The same as the other ADTs

• List-Based PQ

Priority Queue Operation Example

Operation	Output	Priority Queue
insert$(5,A)$	$e_1[=(5,A)]$	$\{(5,A)\}$
insert$(9,C)$	$e_2[=(9,C)]$	$\{(5,A),(9,C)\}$
insert$(3,B)$	$e_3[=(3,B)]$	$\{(3,B),(5,A),(9,C)\}$
insert$(7,D)$	$e_4[=(7,D)]$	$\{(3,B),(5,A),(7,D),(9,C)\}$
min()	e_3	$\{(3,B),(5,A),(7,D),(9,C)\}$
removeMin()	e_3	$\{(5,A),(7,D),(9,C)\}$
size()	3	$\{(5,A),(7,D),(9,C)\}$
removeMin()	e_1	$\{(7,D),(9,C)\}$
removeMin()	e_4	$\{(9,C)\}$
removeMin()	e_2	$\{\}$

$e_{1\ldots n}$ = address of elements and element

• List-Based PQ

List-Based PQ Structure

- Implementation with an *unsorted* list S

- Fast insertions and slow removals
  - **insert** operation
    - Create a new entry object e
    - Just add e to the end of the list S
  - **removeMin** operation
    - Inspect all the entries in S to find an entry e_{min} with the smallest key using the given comparator
    - Remove and return e_{min}

- Implementation with a *sorted* list S

- Slow insertions and fast removals
  - **insert** operation
    - Create a new entry object e
    - Scan thru the list S to find the proper position p for e
    - Insert e into S at position p (an entry with the smallest key is always at the first position)
  - **removeMin** operation
    - Just remove and return the entry stored at the first position of S

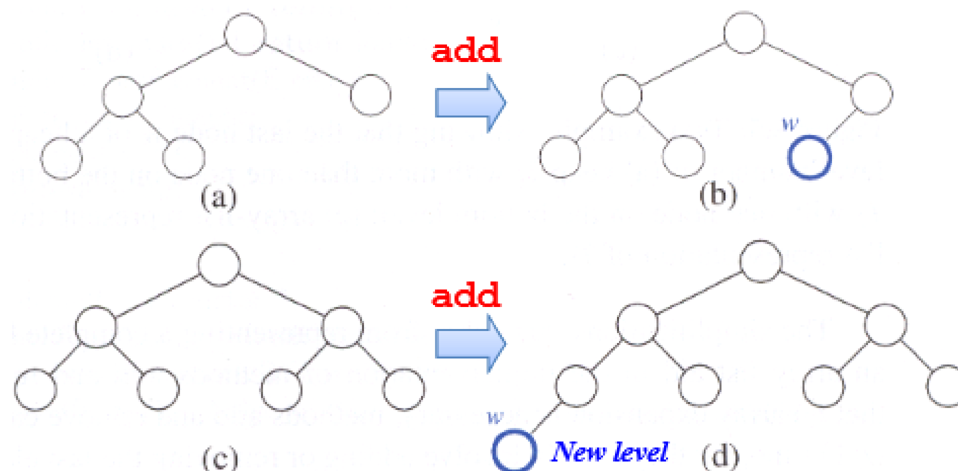
- ***Heap***

• Heap

Complete Binary Tree ADT

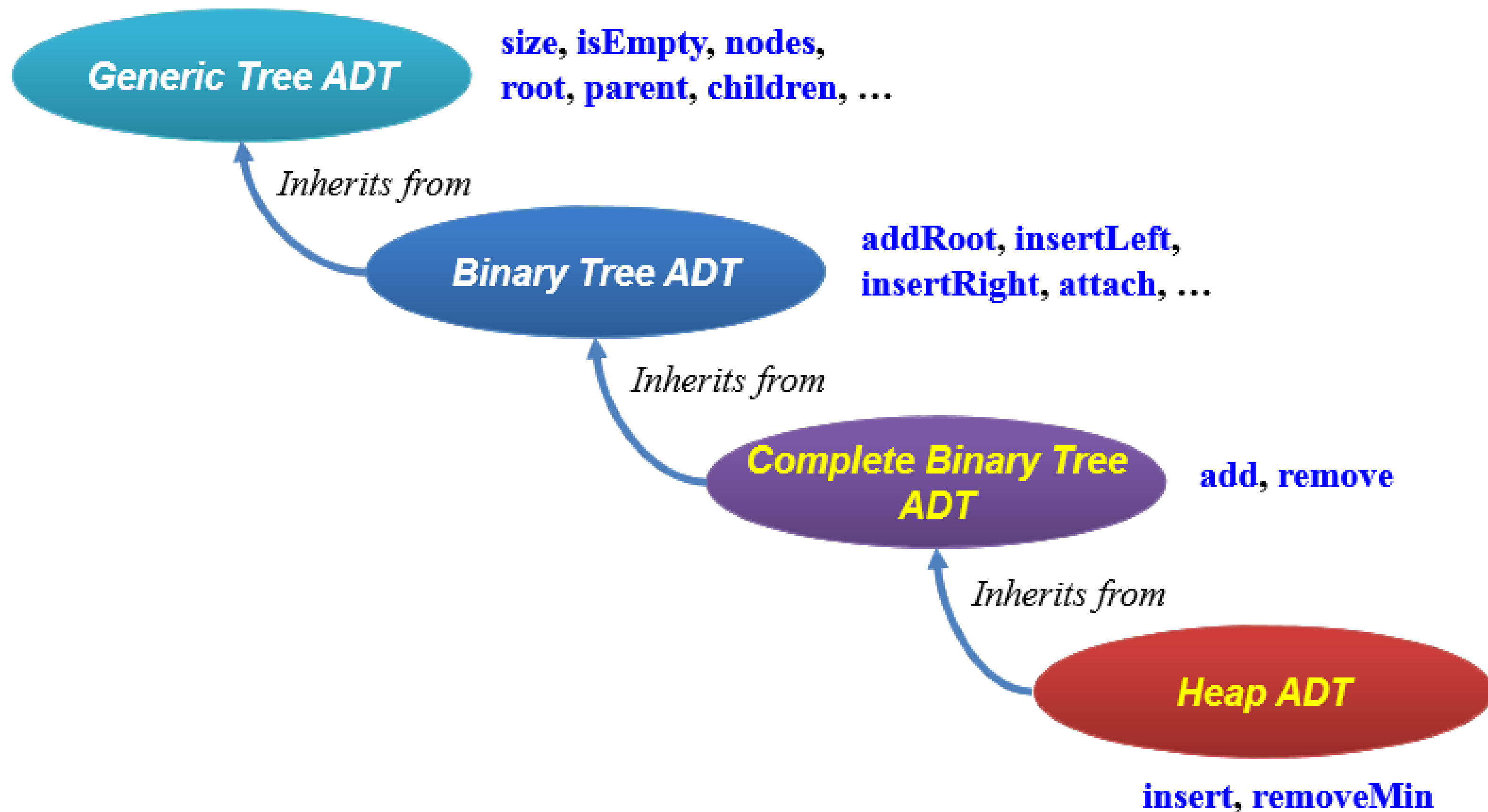
- An ADT of a complete binary tree ***T*** inherits all the operations from the binary tree ADT and supports the following additional operations
  - node **add**(object ***e***): add to ***T*** and return a new last node ***w*** storing element ***e***, thus, the resulting tree is also a complete binary tree with the last node ***w***
  - object **remove**(): remove the last node of ***T*** and return its element

❖ *Insertions and removals of a complete binary tree always occur at the* ***last node***



• Heap

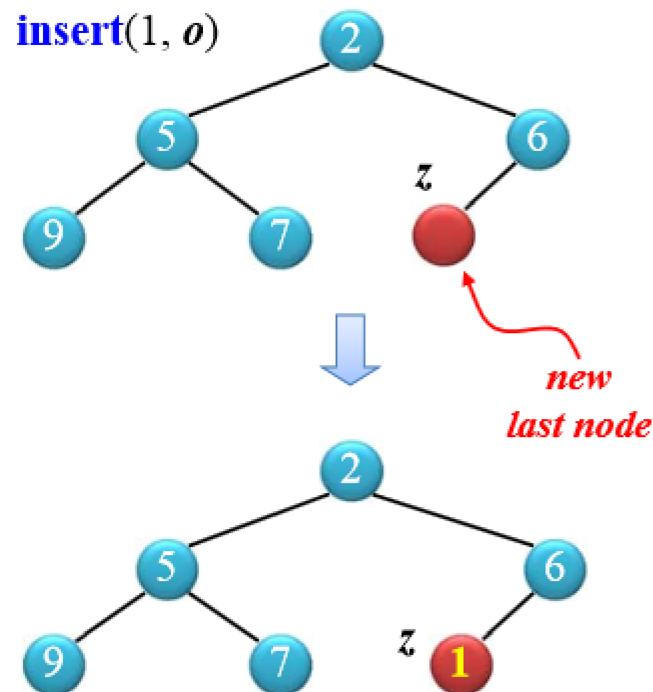
Relationship among the Heap ADT and Other Tree ADTs



• Ex1-Heap Data Structure

Insertion into a Heap

- Operation **insert**(k, o) of the PQ ADT corresponds to the insertion of a key k to the heap T
- The insertion algorithm consists of three steps
  1) Add a new last node z to T with **add** operation of the heap T
  2) Store entry (k, o) at z
  3) Restore the heap-order property with the ***up-heap bubbling*** algorithm



• Heap

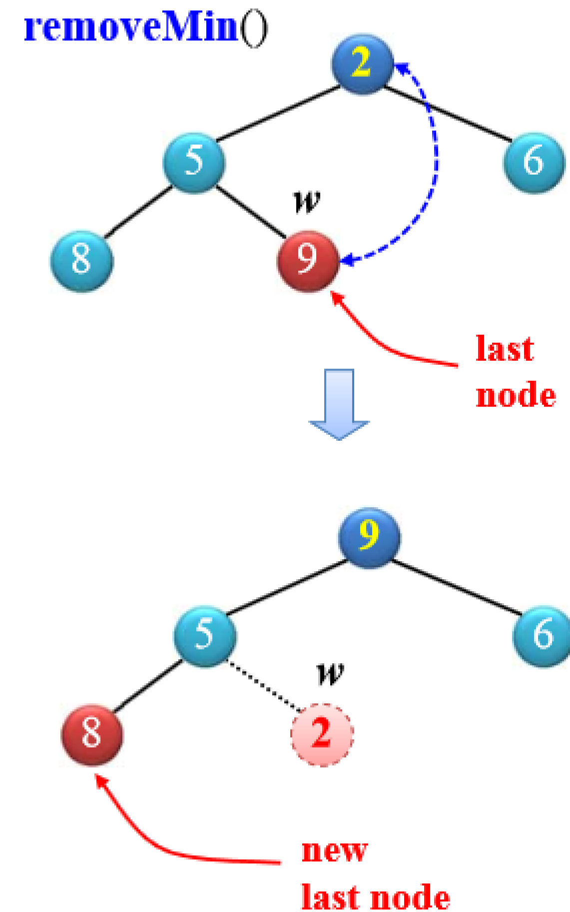
Up-heap Bubbling

- After the insertion of a new node k, the heap-order property may be violated
- ***Up-heap bubbling*** algorithm restores the heap-order property by swapping k along an upward path from the new node to the root
- Up-heap bubbling terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k
- Since a heap has height of $O(\log n)$, up-heap bubbling runs in $O(\log n)$ time

• Heap

Removal from a Heap

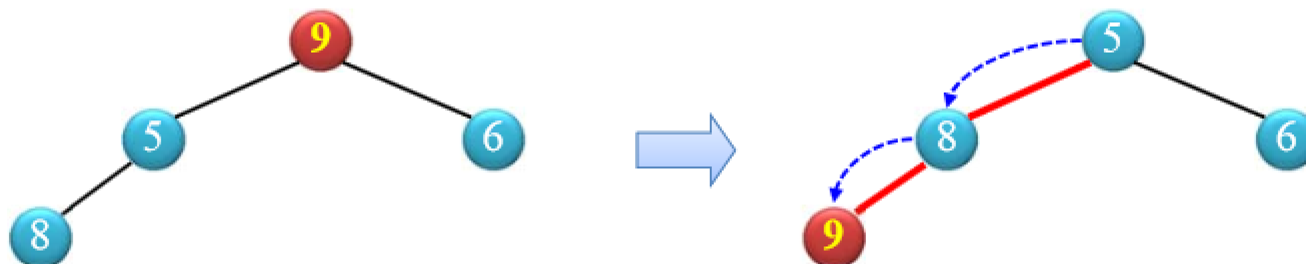
- Operation **removeMin** of the PQ ADT corresponds to the removal of the root node from the heap T
  - The minimum key of a heap is always stored at the root
- The removal algorithm consists of three steps
  1) Replace the root entry of T with the entry of the last node w
  2) Remove w with **remove** operation of the heap T
  3) Restore the heap-order property with the ***down-heap bubbling*** algorithm



• Heap

Removal from a Heap

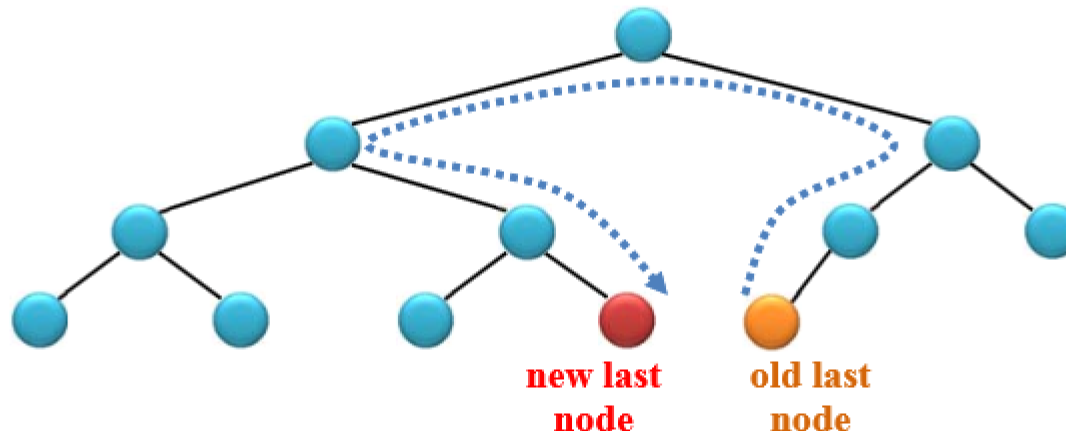
- After replacing the root key with the key of the last node, the heap-order property may be violated
- ***Down-heap bubbling*** algorithm restores the heap-order property by swapping the root key k along a downward path to a leaf (following the *smaller* child)
- Down-heap bubbling terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k
- Since a heap has height of $O(\log n)$, down-heap bubbling runs in $O(\log n)$ time



• Heap

Updating the Last Node

- After removing the last node, we must find a new last node
  1) Start from the old last node
  2) Go up until the subtree containing the old node is the right child, or the root is reached
  3) Go to the left child and then go down right until a leaf is reached
     – If the root is reached from left, just go down right until a leaf is reached
- The new last node can be found by traversing a path of at most (2 * depth) nodes
  – I.e., performance of updating the last node is $O(2 * \log n) = O(\log n)$



- # Heap

Heap.java

```
public class Heap {
    private int[] data;
    private int size;
    private int maximumSize;
    public Heap() {
        this.maximumSize = 127;
        this.data = new int[maximumSize];
        this.size = 0;

    }
    public Heap(int maximumSize) {
        if(maximumSize < 1) {
            this.maximumSize = 127;
        }
        else {
            this.maximumSize = maximumSize;
        }
        this.size = 0;
        this.data = new int[this.maximumSize];
    }
    public boolean isEmpty() {
        return size==0;
    }
    public boolean isFull() {
        return this.size==this.maximumSize;
    }
    public void clear() {
        data = null;
        size = 0;
    }
```

```
public void insert(int newValue) {
    int pointer;
    if(isFull()) {
        System.out.println("이미 가득 차있습니다");
    }
    else {
        data[size] = newValue;
        pointer = size;
        size ++;
        while(pointer > 0 && data[pointer] < data[(pointer-1)/2]) {
            int temp = data[pointer];
            data[pointer] = data[(pointer-1)/2];
            data[(pointer-1)/2] = temp;
            pointer = (pointer-1)/2;
        }
    }
}
public int remove() {
    int retValue;
    if(isEmpty()) {
        System.out.println("힙이 비어있습니다");
        return 0;
    }
    else {
        retValue = data[0];
        size--;
        data[0] = data[size];
        data[size] = 0;
        downBubbling();
        return retValue;
    }
}
```

• Heap

Heap.java

```
public void downBubbling() {
    int pointer = 0;
    int childValue = 0;
    while(pointer*2+2 < size) {
        if(data[pointer*2+1] < data[pointer*2+2]) {
            childValue = 1;
        }
        else {
            childValue = 2;
        }
        int temp = data[pointer];
        data[pointer] = data[pointer*2+childValue];
        data[pointer*2+childValue] = temp;
        pointer = pointer*2+childValue;
    }
}
public void printAll() {
    int pointer = 0;
    while(pointer < size) {
        System.out.print(data[pointer] + " ");
        pointer++;
    }
    System.out.println();
}
```

Main.java

```
public class Main {
    public static void main(String[] args) {
        Heap heap = new Heap();
        heap.insert(1);
        heap.insert(5);
        heap.insert(11);
        heap.insert(21);
        heap.insert(15);
        heap.printAll();

        heap.remove();
        heap.printAll();
    }
}
```

```
<terminated> Main (3) [Java Application] C:
1 5 11 21 15
5 15 11 21
```

Submission

e-mail : 2017kudatastructure@gmail.com

Attach methods :

1. Create zip file (java project folder)
2. Modify the file names :
   ID_name_datastructure[Classcode].dat
   ex) 201173378_이명재_datastructure[2403-1].dat
3. e-mail title :
   datastructure_name_chapter
   ex) datastructure_최수용_chapter8[2404-1]

Classcode

2403-1 **[A-1반]**	2403-2 **[A-2반]**	
2404-1 **[B-1반]**	2404-2 **[B-2반]**	2405 **[C반]**