Lecture 14

Priority Queues and Heaps

Hyojin Sung

For the second half of the semester

Advanced data structures

- Priority queues, heaps, graphs, …

For the second half of the semester

Advanced data structures

- Priority queues, heaps, graphs, …

Advanced algorithms

- Shortest paths, minimum spanning tree, dynamic programming, greedy algorithms, …

For the second half of the semester

Advanced data structures

- Priority queues, heaps, graphs, …

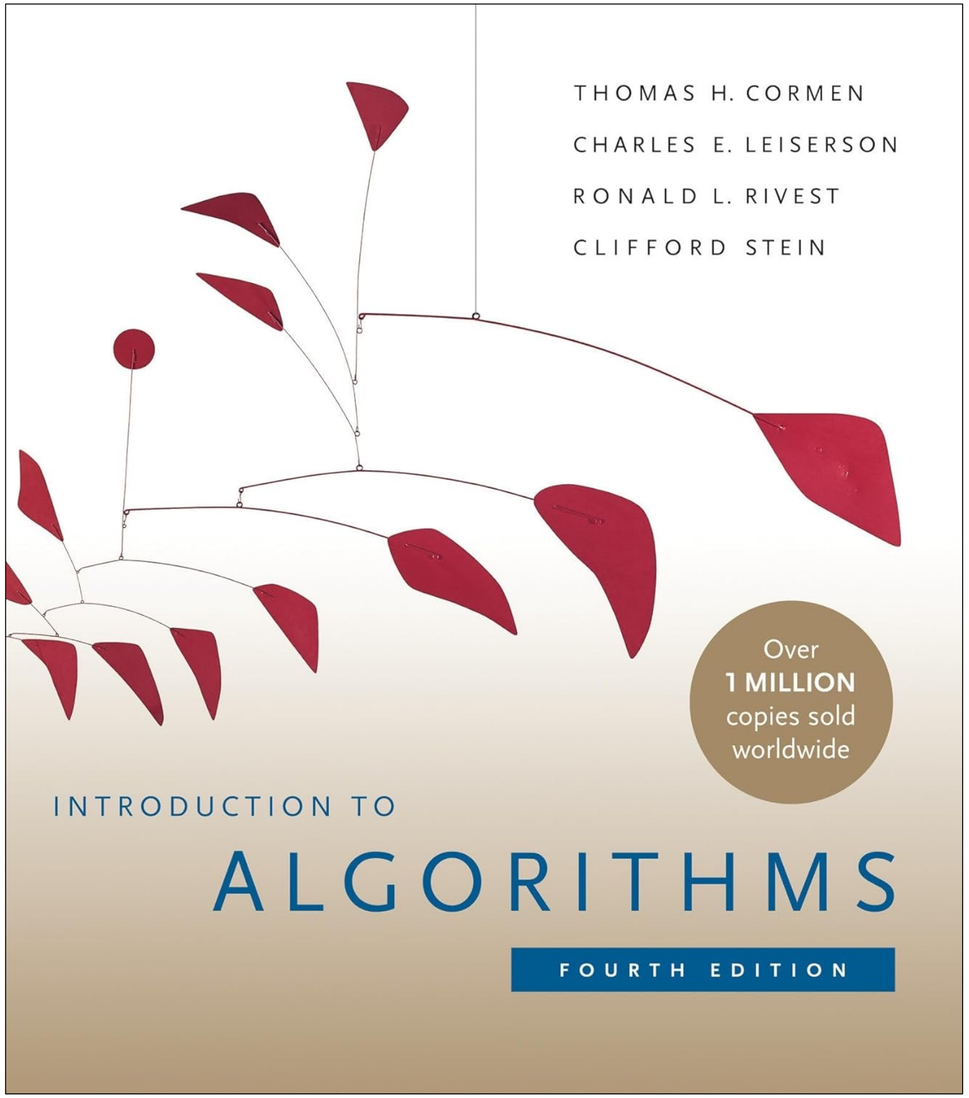
Advanced algorithms

- Shortest paths, minimum spanning tree, dynamic programming, greedy algorithms, …

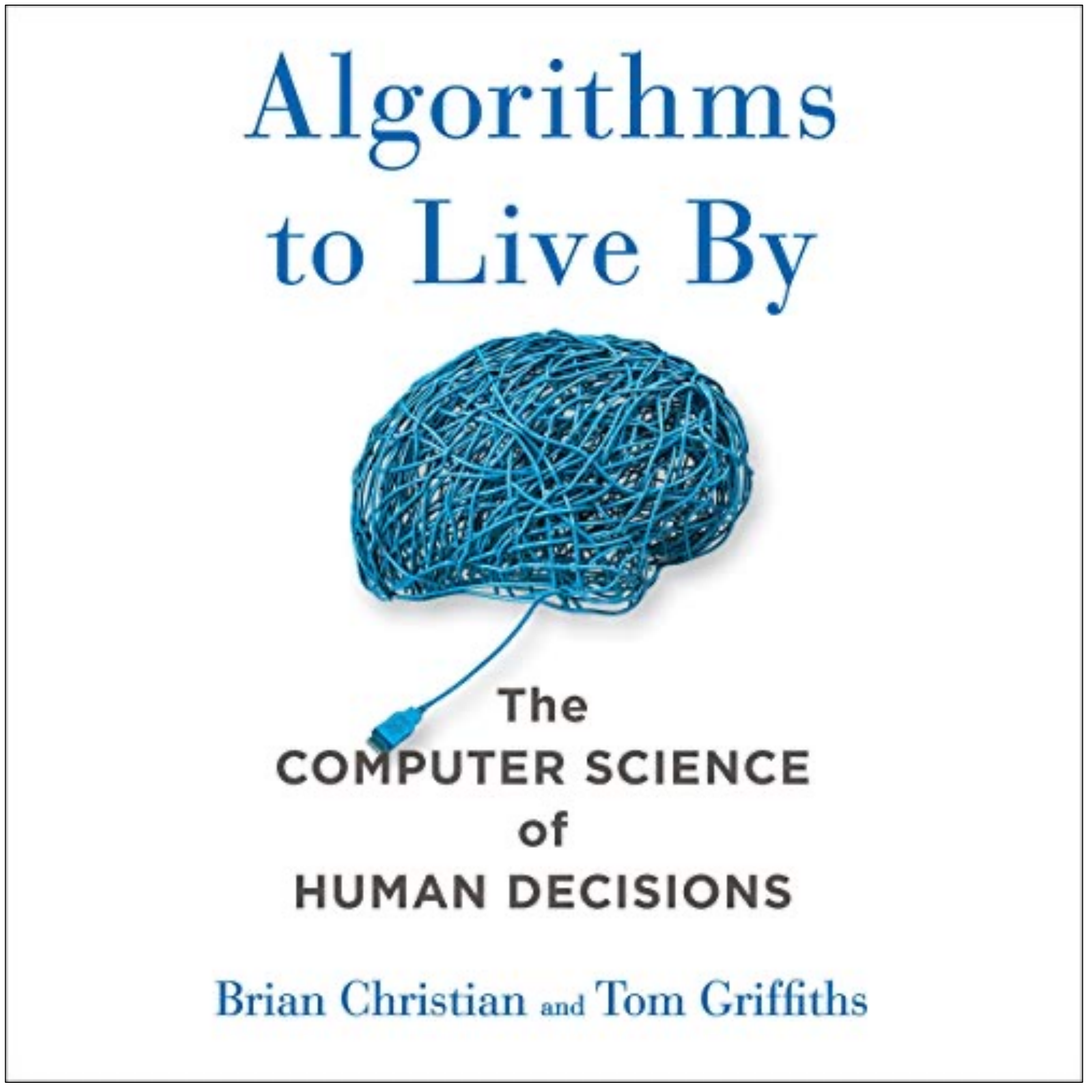
Machine learning concepts

- K-nearest neighbors, logistic regression, multi-level perceptron, ….

For Reference



For Fun


Algorithms
to Live By
The
COMPUTER SCIENCE
of
HUMAN DECISIONS
Brian Christian and Tom Griffiths

Upcoming Schedule

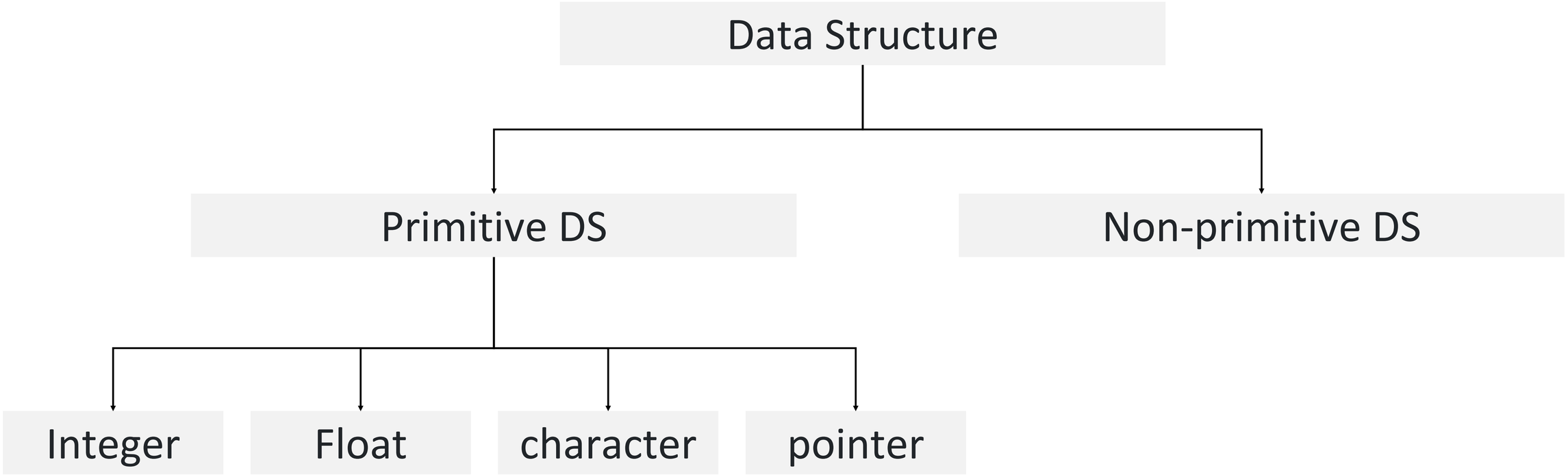
- 4/23 (Tue)
  - Midterm (Building 43 Room 101)

Lecture Overview

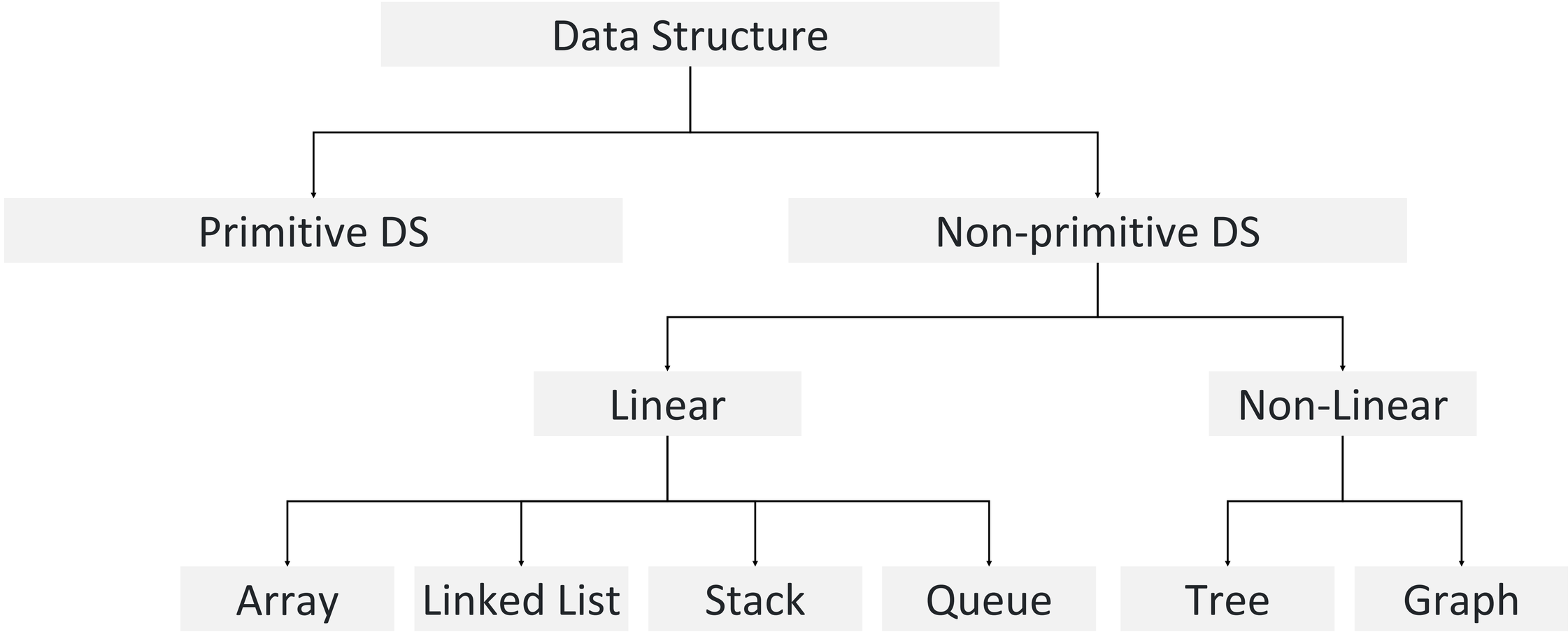
- Review: data structures, ADT
- Priority Queues
- Heaps

Review: Data Structures

Classification of Data Structure



Classification of Data Structure



Abstract Data Types (ADT)

- Abstract Data Types

데이터를 가지고 "뭘 할지만을 정의
실제로 이 데이터 structure 가 어떻게 구현 되는지는 알필요는 없음. (abstract)

  - A definition for expected operations and behaviors
  - Defines what operations are to be performed, but not how

  → abstract, implementation-independent

  - E.g., list, map, stack, queue

- **Data structure implements an ADT**
  - Describes exactly how the operations are performed

Case Study: The List ADT

- List stores a ordered sequence of information
  - Each item is accessible by an index
  - Lists have a variable size as items can be added and removed

List ADT

state

Set of ordered items
Count of items

behavior

get(index) return item at index
set(item, index) replace item at index
append(item) add item to end of list
insert(item, index) add item at index
delete(index) delete item at index
size() count of items

supported operations:

- **get(index):** returns the item at the given index
- **set(value, index):** sets the item at the given index to the given value
- **append(value):** adds the given item to the end of the list
- **insert(value, index):** insert the given item at the given index maintaining order
- **delete(index):** removes the item at the given index maintaining order
- **size():** returns the number of elements in the list

Case Study: List Implementations

List ADT

state

Set of ordered items
Count of items

behavior

get(index) return item at index
set(item, index) replace item at index
append(item) add item to end of list
insert(item, index) add item at index
delete(index) delete item at index
size() count of items

Array

Linked list

Design Decisions

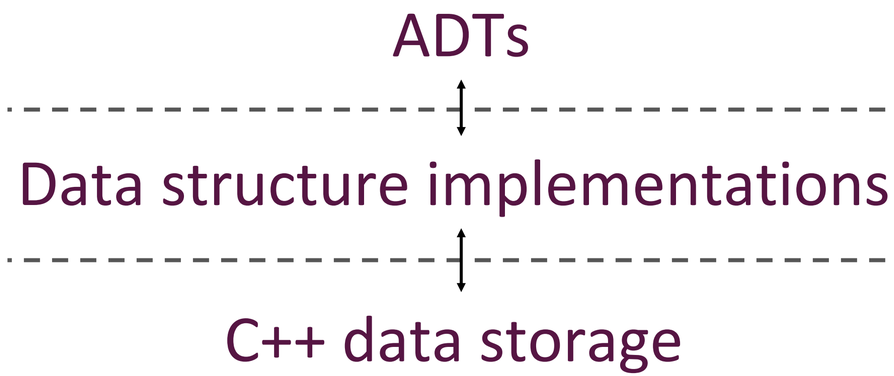
- For every ADT, there are different ways to impelement them
- Why?

  추상화의 level의 차이 때문에 가능한 것.
  같은 data structure도 저차원에서 어떻게 구현되는냐에 따라 달라진다.

  ex) list구현 : array로? linked_list로 구현?

Design Decisions

- For every ADT, there are different ways to impelement them
- Why?
  - The abstraction gap
- Trade-offs to consider
  - Memory vs. speed
  - General/reusability vs. specific/specialized
  - Robustness vs. performance
    일부 data structure은 병렬로 처리를 가능하게 하면서, robusness를 희생하면서 성능을 높일 수도 잇음.
  - …



Case Study: Stacks

- A collection based on the principle of adding elements and retrieving them in the opposite order
  - Last-In, First-Out (“LIFO”)
  - Elements are stored in order of insertion
  - Users can only add/remove/examine the last element added (the “top”)

Stack ADT

state

Set of ordered items
Number of items

behavior

push(item) add item to top
pop() return and remove item at top
peek() look at item at top
size() count of items
isEmpty() count of items is 0?

supported operations:

- **push(item)**: Add an element to the top of stack
- **pop()**: Remove the top element and returns it
- **peek()**: Examine the top element without removing it
- **size():** how many items are in the stack?
- **isEmpty():** false if there are 1 or more items in stack, true otherwise

보통은 array로 구현하는 것이 효율적.
linked_list은 마지막까지의 접근을 항상 해야함.

Array Stack

```
class Stack {
 public:
        int *array;
        int top;
        int maxSize;

        Stack (int max) {
                top = 0;
                maxSize = max;
                array = new int[max];
        }
        void push(int item) {
                if (size == maxSize – 1) return;
                array[top++] = item;
                return SUCCESS;
        }
        int pop() {
                if (isEmpty()) return ERROR;
                return array[top--];
        }
        int peek() {
                if (isEmpty()) return ERROR;
                return array[top];
        }
        int isEmpty() {
                return top == 0;
        }
        int size() {
                return top;
        }
}
```

Array Stack

```
class Stack {
 public:
        int *array;
        int top;
        int maxSize;

        Stack (int max) {
                top = 0;
                maxSize = max;
                array = new int[max];
        }
        int push(int item) {
                if (size == maxSize – 1) return ERROR;
                array[top++] = item;
                return SUCCESS;
        }
        int pop() {
                if (isEmpty()) return ERROR;
                return array[top--];
        }
        int peek() {
                if (isEmpty()) return ERROR;
                return array[top];
        }
        int isEmpty() {
                return top == 0;
        }
        int size() {
                return top;
        }
}
```

Array Stack

```
class Stack {
 public:
        int *array;
        int top;
        int maxSize;

        Stack (int initial) {
                top = 0;
                maxSize = initial;
                array = new int[initial];
        }
        int push(int item) {
                if (size == maxSize – 1) {
                    maxSize = 2*maxSize;
                    new_array = new int[maxSize];
                    // copy all array data to new_array
                    delete array;
                    array = new_array;
                  }
                array[top++] = item;
                return SUCCESS;
        }
        int pop() {
                if (isEmpty()) return ERROR;
                return array[top--];
        }
        int peek() {
                if (isEmpty()) return ERROR;
                return array[top];
        }
        int isEmpty() {
                return top == 0;
        }
        int size() {
                return top;
        }
}
```

Linked Stack

```
struct Node {
        int data;
        Node *next;

        Node(int data, Node *next): data(data), next(next) {}
        Node(): data(0), next(nullptr) {}
}
```

Linked Stack

```
class Stack {
 public:
        Node *front;
        int size;

        Stack () : front(nullptr), size(0) {}

        void push(int item) {
                Node *newNode = new Node(item, front);
                front = newNode;
                size++;
        }
        int pop() {
                Node *tmp = front;
                int val;
                if (isEmpty()) return ERROR;
                val = front->data;
                front = front->next;
                delete tmp;
                size--;
                return val;
        }
        int peek() {
                if (isEmpty()) return ERROR;
                return front->data;
        }
        int isEmpty() {
                return size == 0;
        }
        int size() {
                return size;
        }
}
```

top을 맨 앞에 있도록 항상 유지
그래야 linked list상 stack처럼 사용하기 위해
LIFO을 하기 위해.

맨 마지막에 넣었다면, 비효율적 구현임.(매번 뒤까지 이동을 해야하니까)

Multiple Levels of Design Decisions

- Choice of ADT
  - Which of the ADT's that you've seen is the best fit for your problem?
- Data structure choice
  - Do we use arrays or linked lists?
- Implementation details
  - Do we ovewrite old values with `null` or do we leave the garbage value?
  - Do we validate input and throw exceptions or just wait for the code to fail?

문제 정의에서 사용되는 최적의 data structure은 어떤 형태? ADT :
bank transection 관리 : queue(FIFO)

그런 data structure을 구현해 내기 위해 사용할 data structure 선택

Priority Queues

Review: Queues

- Retreves elements in the order they were added
  - First-In, First-Out (“FIFO”)
  - Elements are stored in order of insertion, but don’t have indices
  - Users can only add to the end of the queue and only examine/remove the front of the queue

Queue ADT
state Set of ordered items Number of items **behavior** add(item) add item to back remove() remove and return item at front peek() return item at front size() count of items isEmpty() count of items is 0?

supported operations:

- **add(item):** aka “enqueue” add an element to the back.
- **remove():** aka “dequeue” Remove the front element and return.
- **peek()**: Examine the front element without removing it.
- **size():** how many items are stored in the queue?
- **isEmpty():** if 1 or more items in the queue returns true, false otherwise

Priority Queue ADT

A **priority queue** holds *comparable data*

- Given *x* and *y*, is *x* less than, equal to, or greater than *y*
- Meaning of the ordering can depend on your data
- Typically, elements are comparable types, or have two fields: the *priority* and the *data*
- Like regular queues, you cannot index into them to get an item at a particular position
- Useful for maintaining data sorted based on priorities
  - Filtering data to get the top X results
  - Emergency room waiting rooms

넣고 뺄 때, priority 를 기준으로 in and out

Operations

- **`enqueue (priority, elem):`** insert elem with given priority
- **`dequeue():`** removes the element with the highest priority from the queue
- **`peek():`** returns the element with the highest priority in the queue without removing it
- Uses priorities instead of FIFO

Operations

- **`size():`** returns the number of elements in the queue
- **`isEmpty():`** returns true if there are no elements in the queue, false otherwise
- **`clear():`** empties the queue

Priority Queue Example

Given the following, what values are **a**, **b**, **c** and **d**?

enqueue *element1* with priority *5*

enqueue *element2* with priority *3*

enqueue *element3* with priority *4*

a = **dequeue** **// a = ?** element 2

b = **dequeue** **// b = ?** element 3

enqueue *element4* with priority *2*

enqueue *element5* with priority *6*

c = **dequeue** **// c = ?** element 4

d = **dequeue** **// d = ?** element 1

Priority Queue Example

Given the following, what values are **a**, **b**, **c** and **d**?

enqueue *element1* with priority *5*

enqueue *element2* with priority *3*

enqueue *element3* with priority *4*

a = **dequeue** `// a = element2`

b = **dequeue** `// b = element3`

enqueue *element4* with priority *2*

enqueue *element5* with priority *6*

c = **dequeue** `// c = element4`

d = **dequeue** `// d = element1`

Applications

- Run multiple programs in the operating system
  - ~~"critical" bef~~ore "interactive" before "compute-intensive", or let users set priority level
- Select print jobs in order of decreasing length
- "Greedy" algorithms (we'll revisit this idea)
- Forward network packets in order of urgency
- Sorting (first **`enqueue`** all, then repeatedly **`dequeue`**)

Possible Implementations

- Unsorted Array 이걸로 priority를 구현하기.
  - **`enqueue`** by inserting at the end 계속 끝에 넣기.
  - **`dequeue`** by linear search priorty를 다 검색 및 비교하면서 해야함.

- Sorted Circular Array
  - **`enqueue`** by binary search, shift elements over
  - **`dequeue`** by moving “front”

빼고, 움직여서 다시 채워넣어줘야함.

Possible Implementations

- Unsorted Linked List
  - **`enqueue`** by inserting at the front
  - **`dequeue`** by linear search

- Sorted Linked List
  - **`enqueue`** linear search
  - **`dequeue`** remove at front

- Binary Search Tree
  - **`enqueue`** by search traversal
  - **`dequeue`** by find min traversal

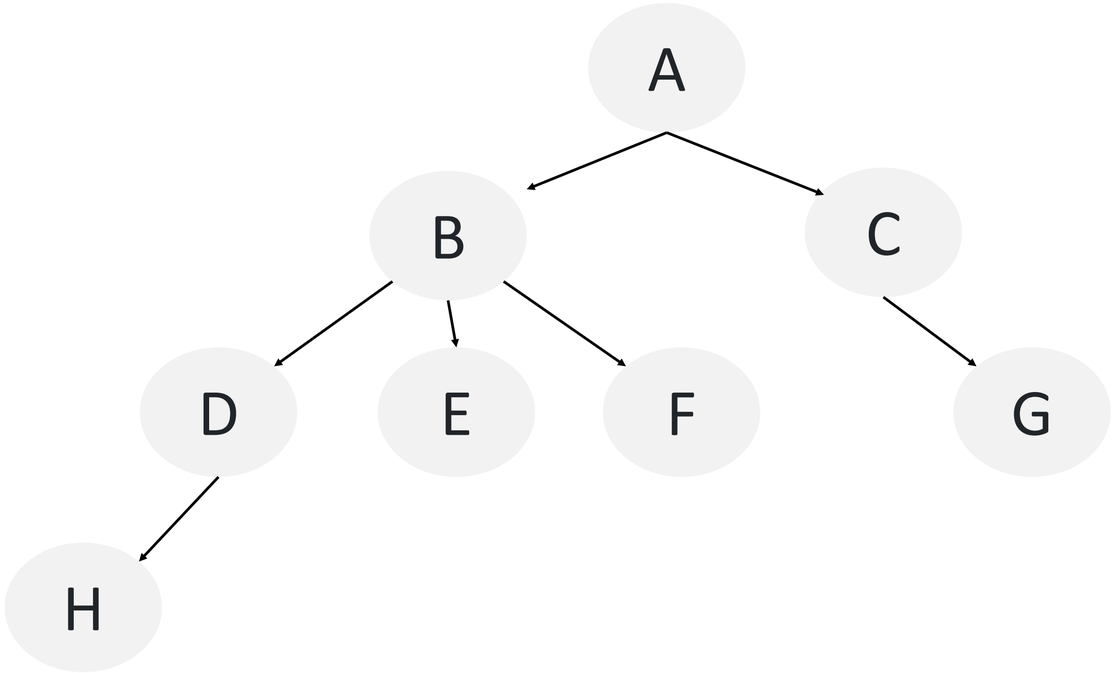
Heaps

priority. queue를 구현하기 위한 가장 적절한 data structure

One Implementation: Heap

- A ***binary min-heap (or just binary-heap or heap)*** is a tree-based structure with the following properties
  - Structure property: A *complete* binary tree (child는 모두 2개 (leaf 제외))
  - Heap property: The priority of every (non-root) node is greater than the priority of its parent (parent의 priority가 더 먼저 , 즉 값이 더 작음. (Min-heap인 경우))
  - ***Not*** a binary search tree
    parent보다 왼쪽이 더 작은 값, 오른쪽이 더 큰 값.
    binary min-heap은 parent가 항상 child보다 작은 값
- Two types (depends on what we define as a “higher” priority)
  - Min-heap: smaller numbers = higher priority
  - Max-heap: larger numbers = higher priority

Tree Review

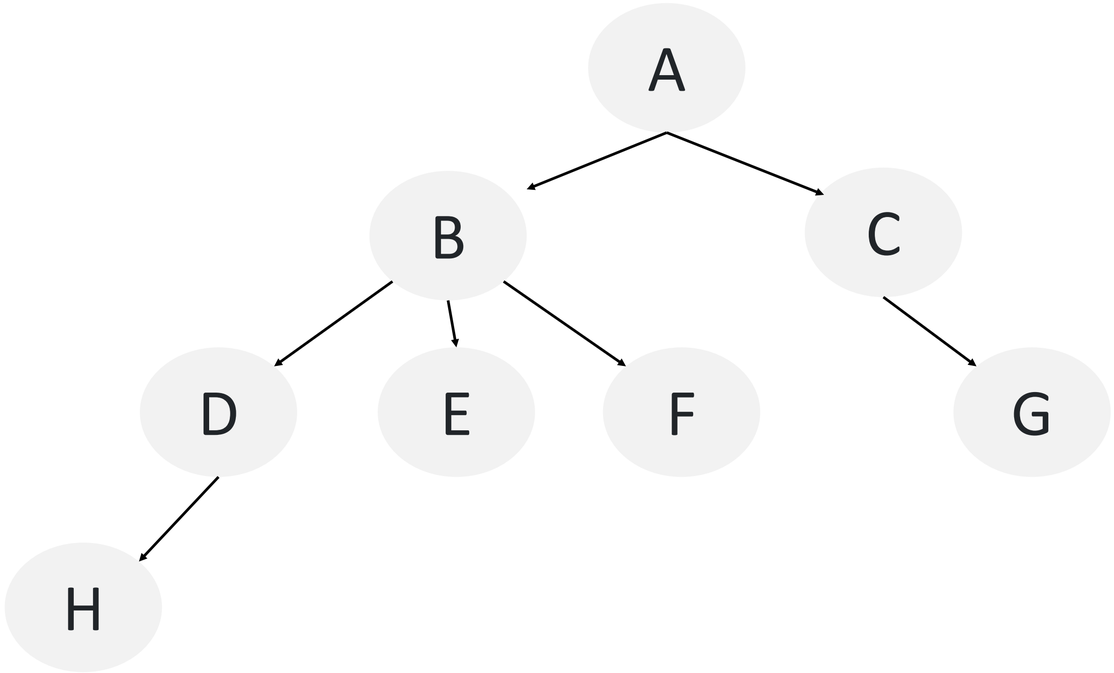


- root of tree: A
- leaves of tree: H ,E, F, G
- children of B: D, E, F
- parent of C: A
- subtree of C: C,G
- height of tree: 3 -> root에서 leaf까지의 거리(edge의 개수)
- height of E: 0(E자체가 leaf여서)
- depth of E: 2 (root까지의 edge 개수)
- degree of B: 3 (children의 개수)

- perfect tree:
- complete tree:

Tree Review

- root of tree: A
- leaves of tree: H,E,F,G
- children of B: D, E, F
- parent of C: A
- subtree C: C, G
- height of tree: 3
- height of E: 0
- depth of E: 2
- degree of B: 3

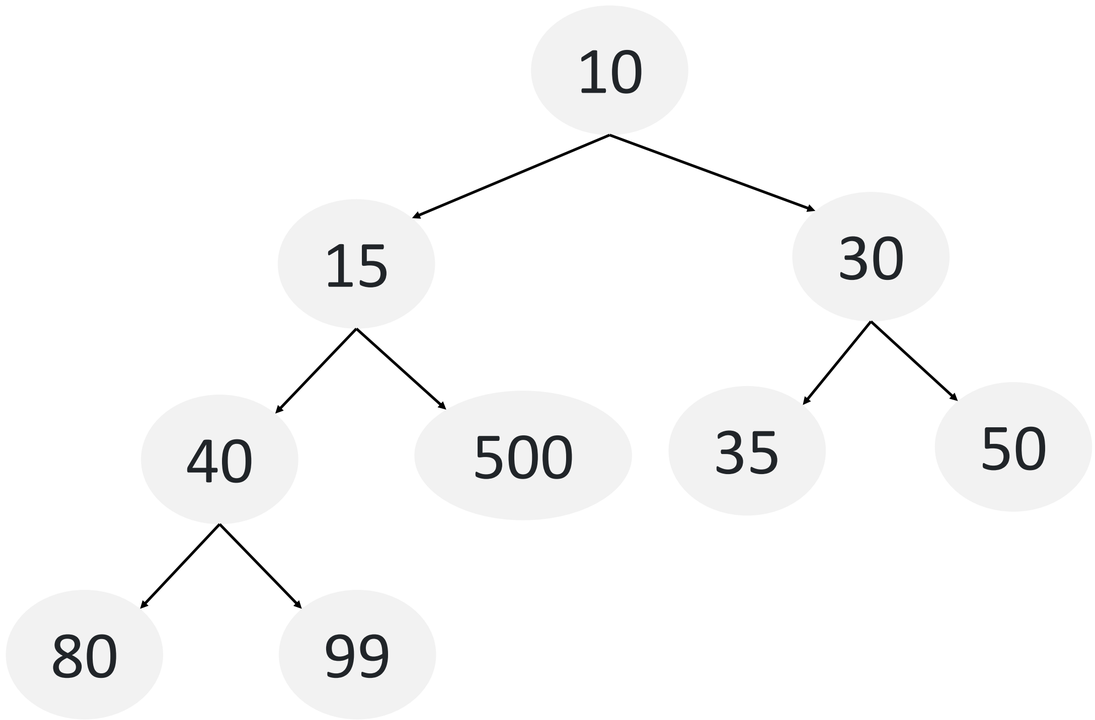


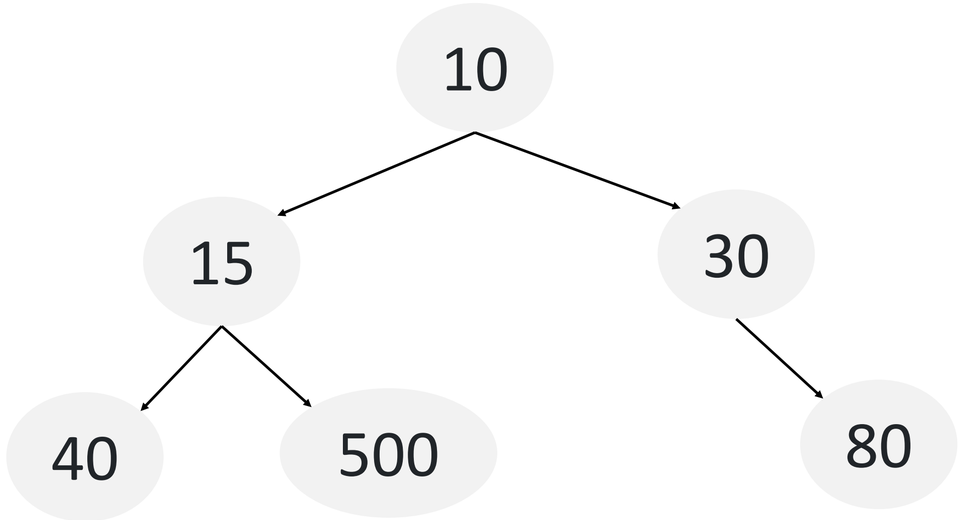
- perfect tree: 모든 레벨에 2개 이상 있는 상태.
  - every level is completely full
- complete tree:
  - all levels full, with a possible exception being the bottom level, which is filled left to right

왼쪽부터 채워져야 하는데, 마지막 level은 꼭 굳이 다 안차도 된다.

Structured Property: Completeness

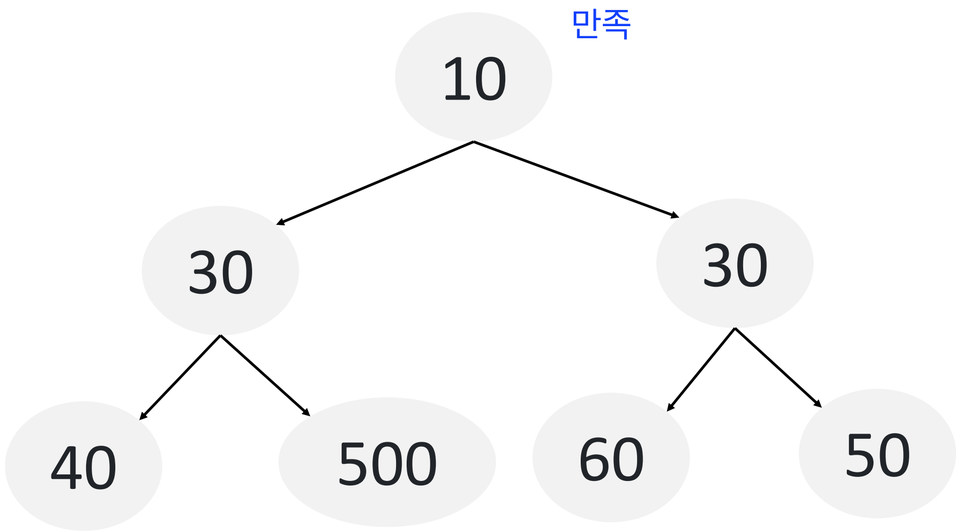
- A binary heap is a **complete** binary tree:
  - A binary tree with all levels full, with a possible exception being the bottom level, which is filled left to right

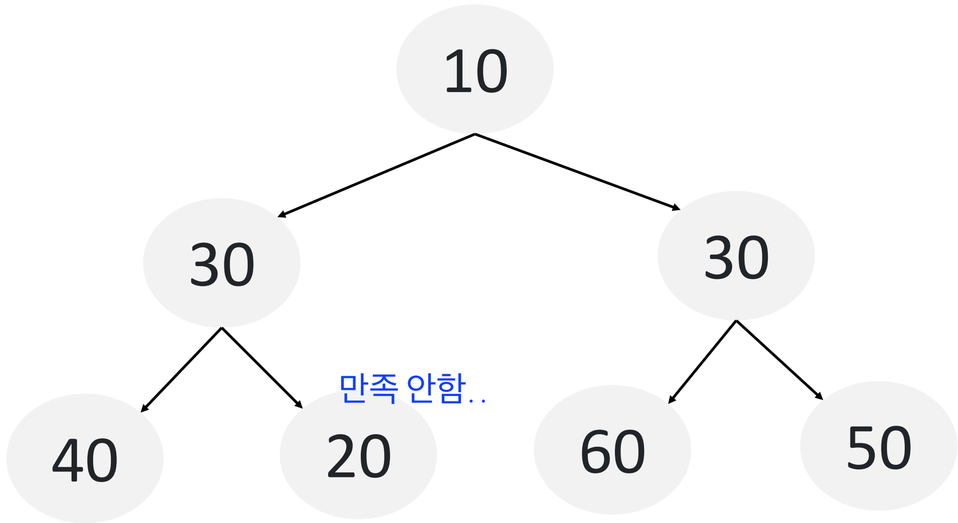




Heap Property

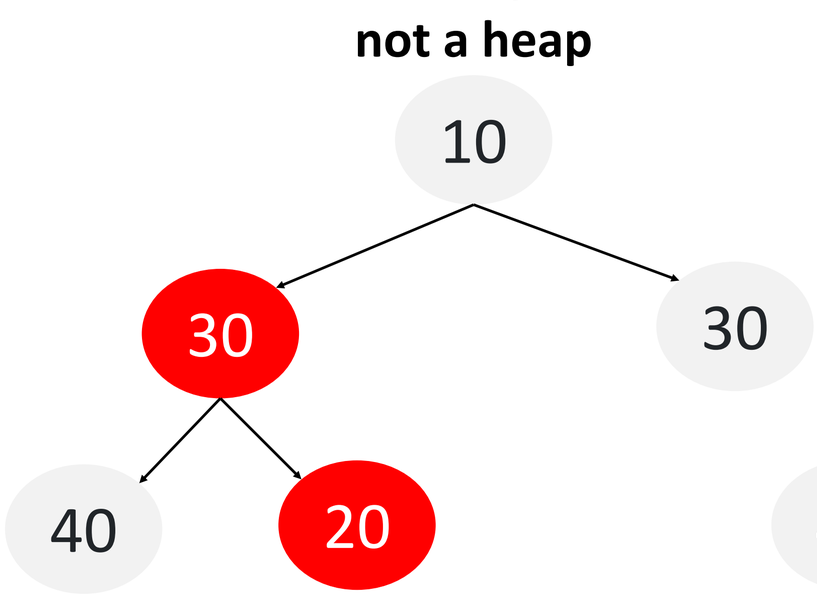
- The priority of every (non-root) node is greater than (or equal to) that of its parent
  - $priority(v) \geq priority(parent(v))$

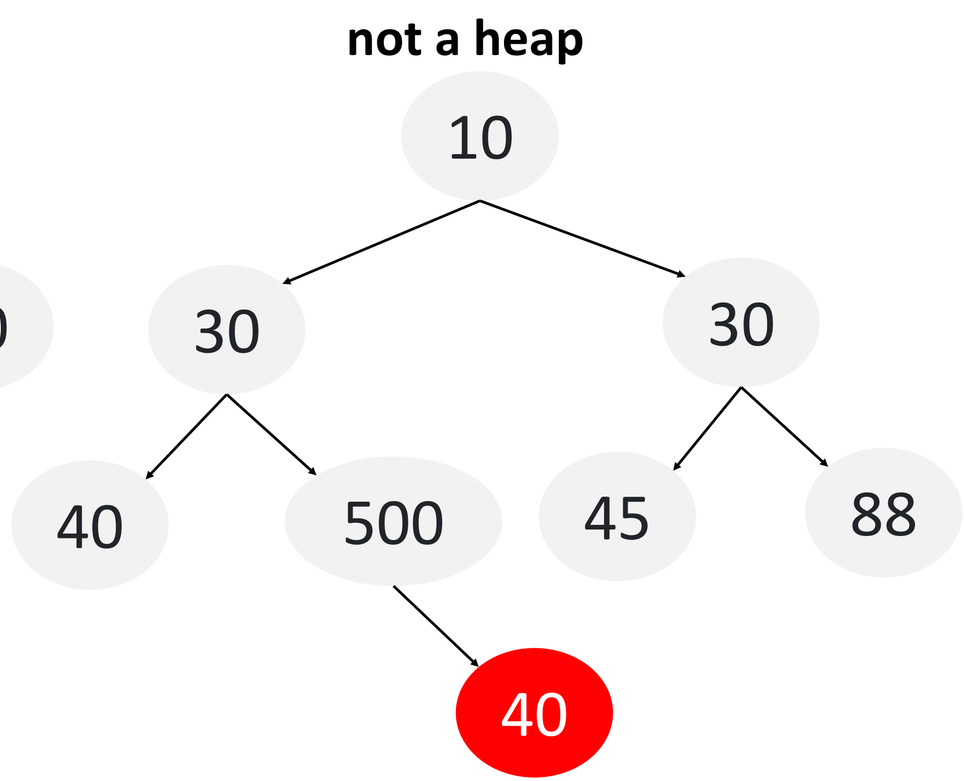


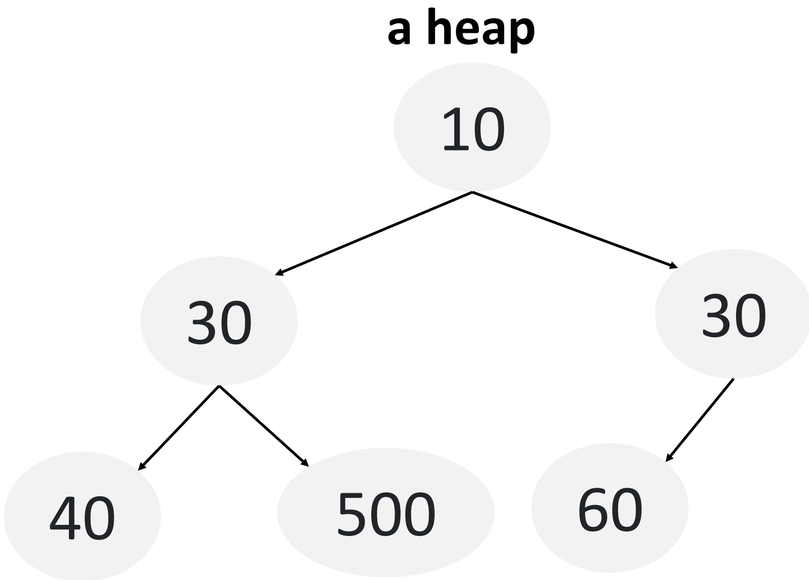


Heaps

- A ***binary min-heap (or just heap)*** is a tree-based structure with the following properties
  - Structure property: A *complete* binary tree
  - Heap property: The priority of every (non-root) node is greater than the priority of its parent
  - ***Not*** a binary search tree

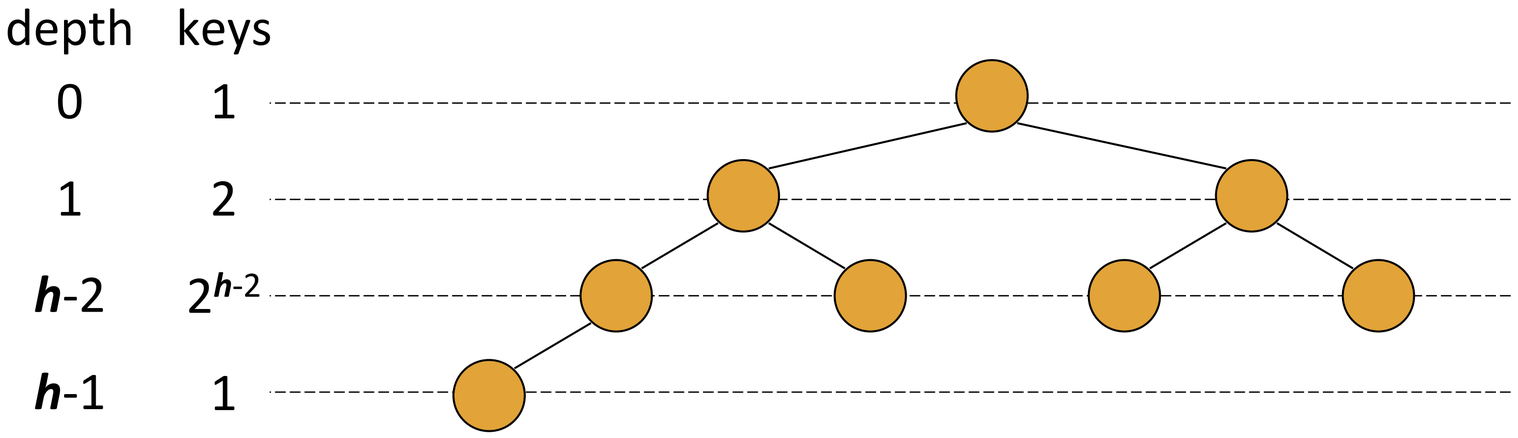






Height of a Heap

- **Theorem:** A heap storing n keys has height $O(\log n)$
- **Proof:** (we apply the complete binary tree property)
  - Let h be the height of a heap storing n keys
  - Since there are 2^i keys at depth $i = 0, \ldots, h-2$ and at least one key at depth $h-1$, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-2} + 1$
  - Thus, $n \geq 2^{h-1}$, i.e., $h \leq \log n + 1$



Heaps and Priority Queues

- We can use a heap to implement a priority queue
- Store a (data, priority) item at each internal node
- We keep track of the position of the last node
- **Overall starategy**
  - *Preserve structure property*
  - *Break and restore heap property*

최대한 binary complete tree 형태를 유지하려고.

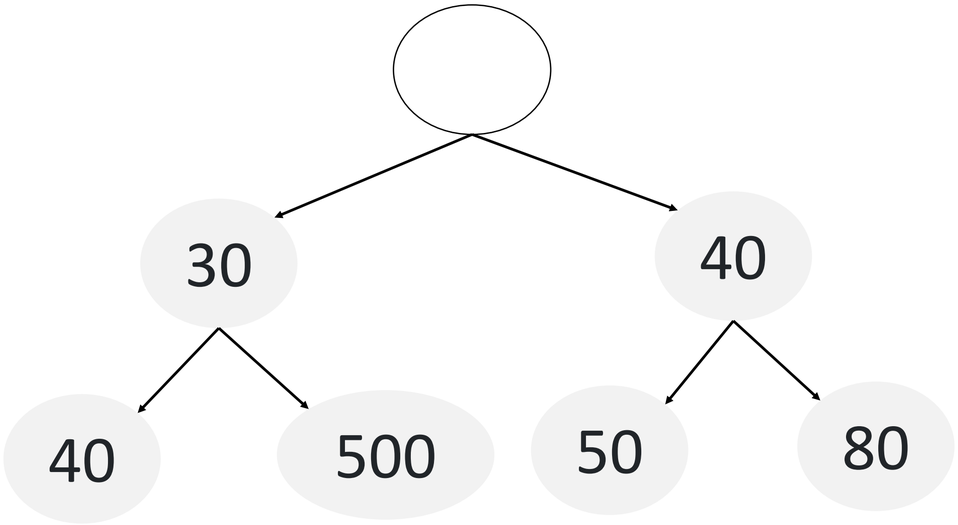
일단 넣고, 다시 복원하는 느낌으로.
complete랑, heap property를 복원

Operations

- **dequeue**
  - **answer = root.data**
  - Move right-most node in last row to root to restore structure property
  - "Percolate down" to restore heap property
- **enqueue**
  - Put new node in next position on bottom row to restore structure property
  - "Percolate up" to restore heap property

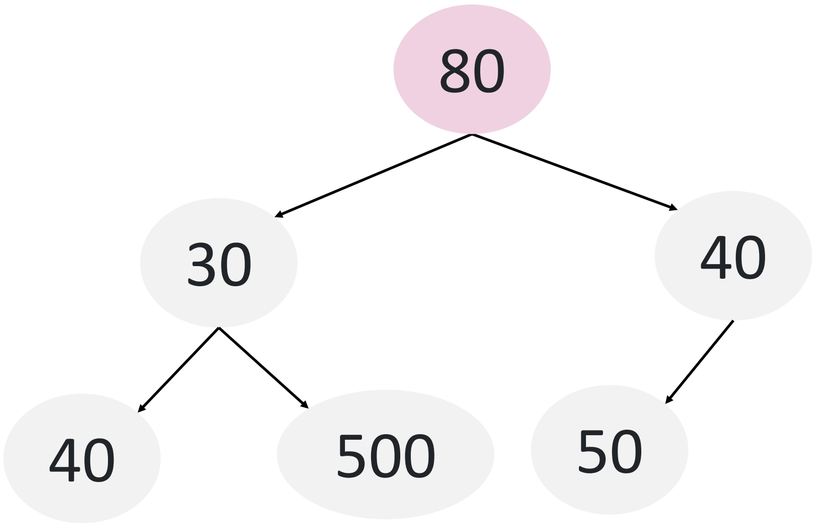
Dequeue

1. Delete (and later return) value at root node



Dequeue

1. Delete (and later return) value at root node
2. Replace the root node with the last node structure property를 복원하기 위해.
   - The tree will have one less node and must still be complete

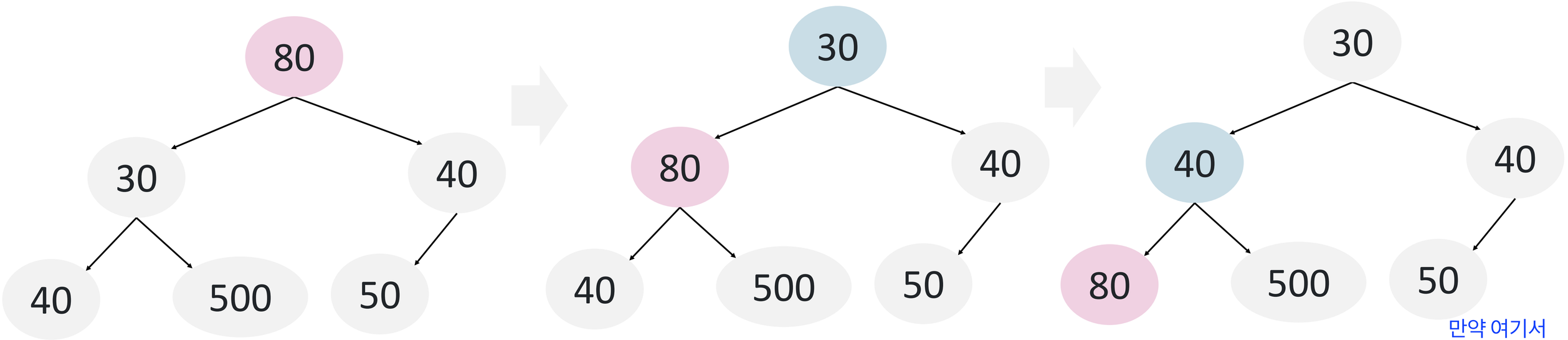


Dequeue

3. Restore the heap property (Downheap or ”percolate down”)
   - Swap the node with its child with the smallest priority along a downward path from the root
   - Downheap terminates when the node reaches a leaf or both children are ≥ item

큰것으로 하면 다시 heap property를 맞춰주기 위해, 다시 해야함. 아무런 도움이 안됨.

- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time

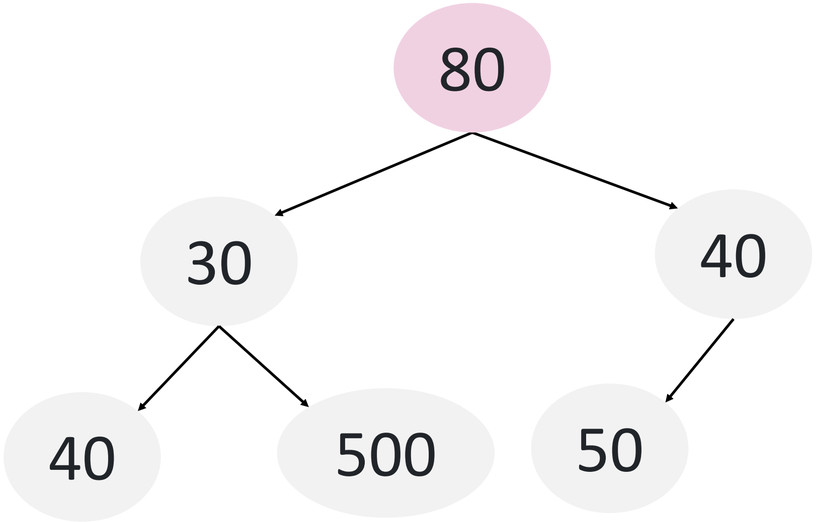


만약 여기서
30 dequeue하고는
50은 두 40 중 어느 쪽이든 상관이 없음

Dequeue

3. Restore the heap property (Downheap or ”percolate down”)
   - Swap the node with its child with the smallest priority along a downward path from the root
   - Downheap terminates when the node reaches a leaf or both children are ≥ item

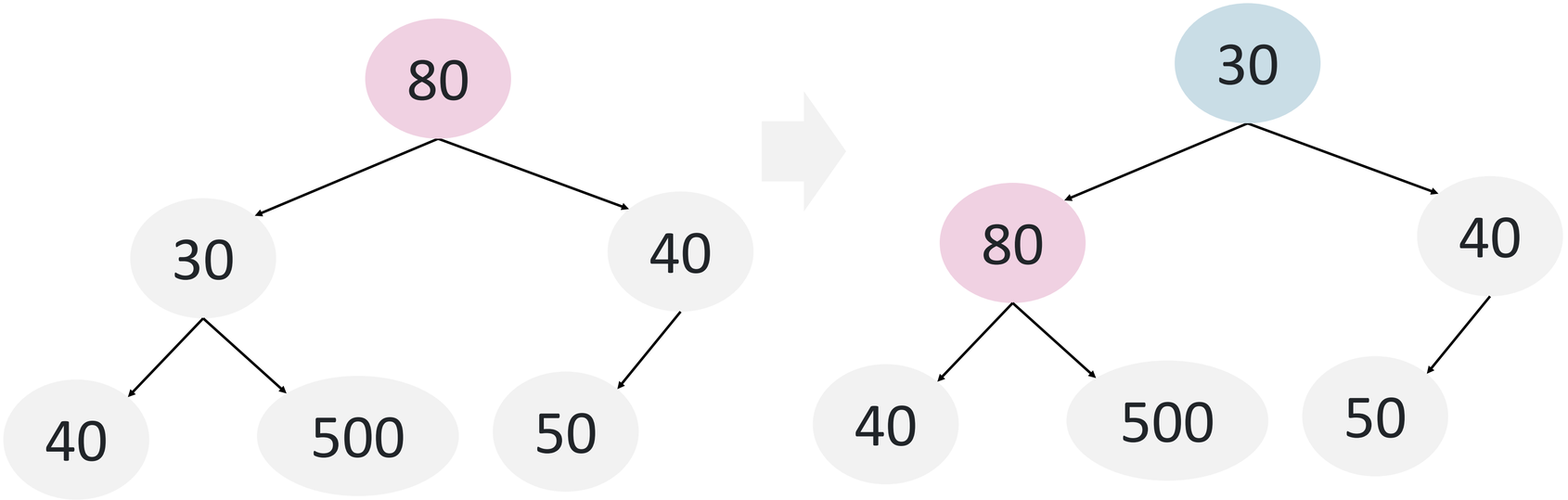
- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time



Dequeue

3. Restore the heap property (Downheap or ”percolate down”)
   - Swap the node with its child with the smallest priority along a downward path from the root
   - Downheap terminates when the node reaches a leaf or both children are ≥ item

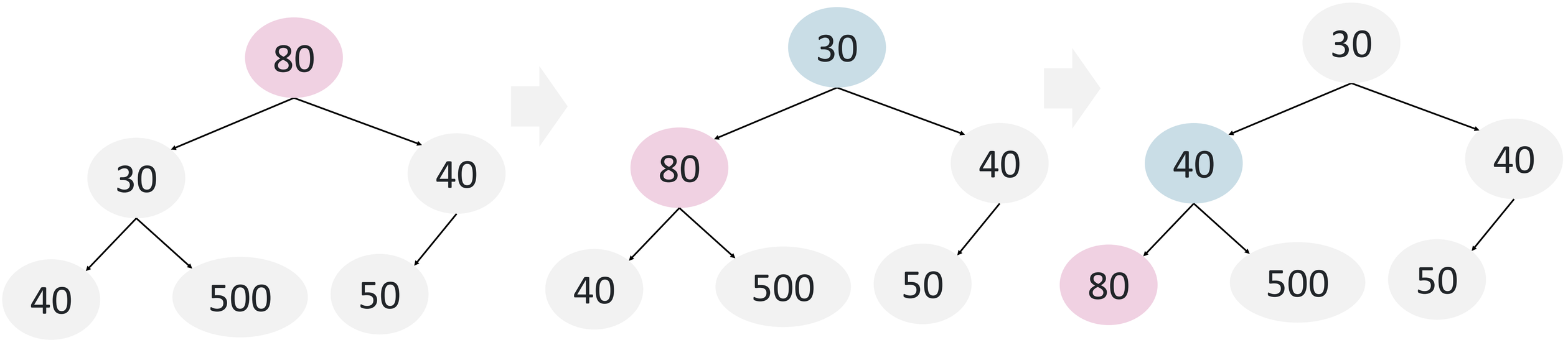
- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time



Dequeue

3. Restore the heap property (Downheap or ”percolate down”)
   - Swap the node with its child with the smallest priority along a downward path from the root
   - Downheap terminates when the node reaches a leaf or both children are ≥ item

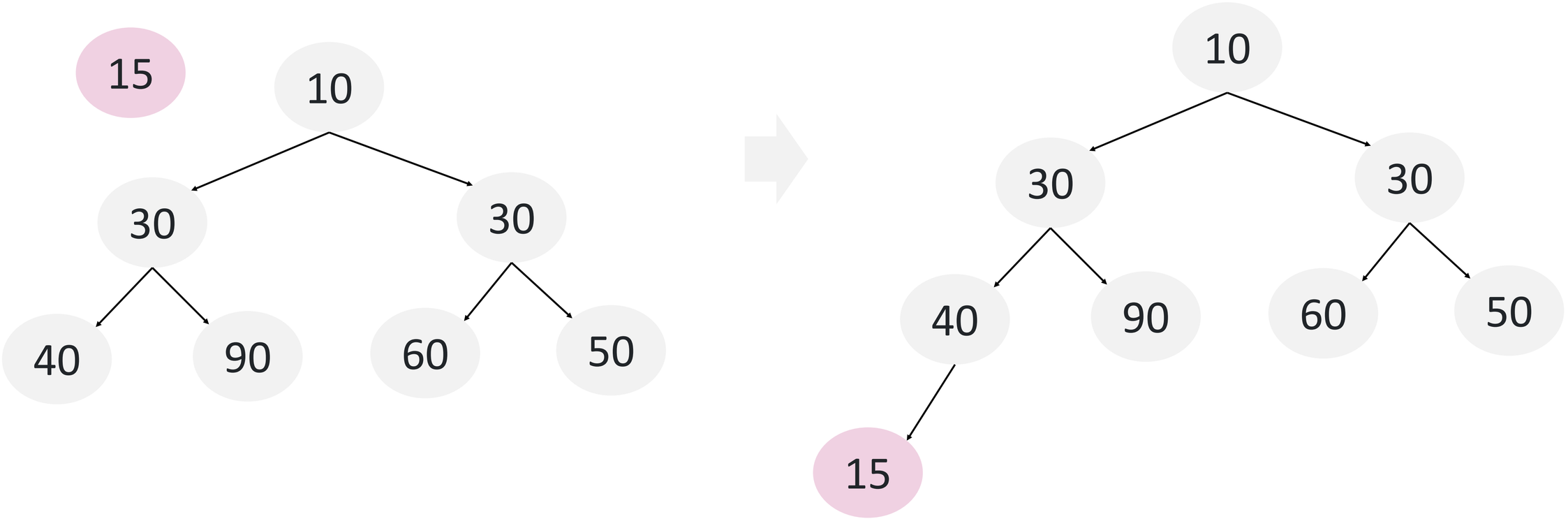
- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time



Enqueue

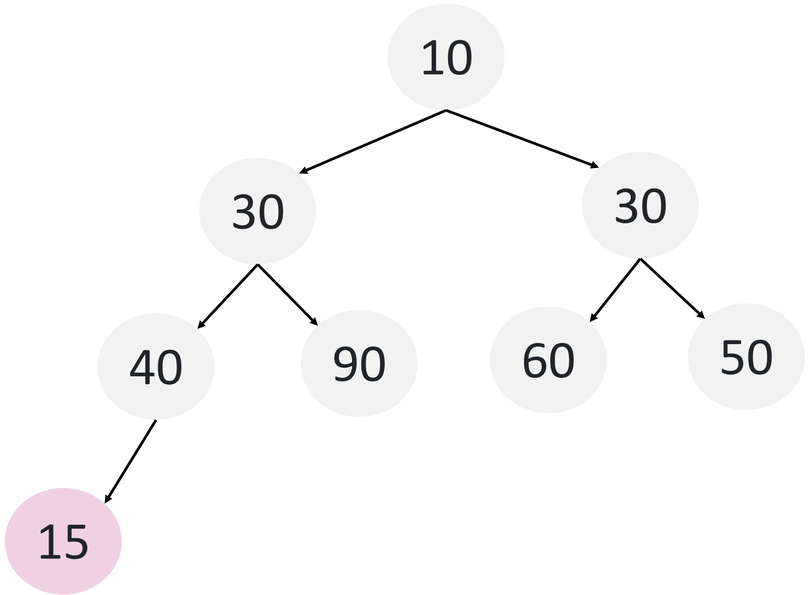
1. Insert a node as a new last node
   - There is only one valid tree shape after we add one more node

그냥 마지막 레벨의 왼쪽부터.(가장 last node)



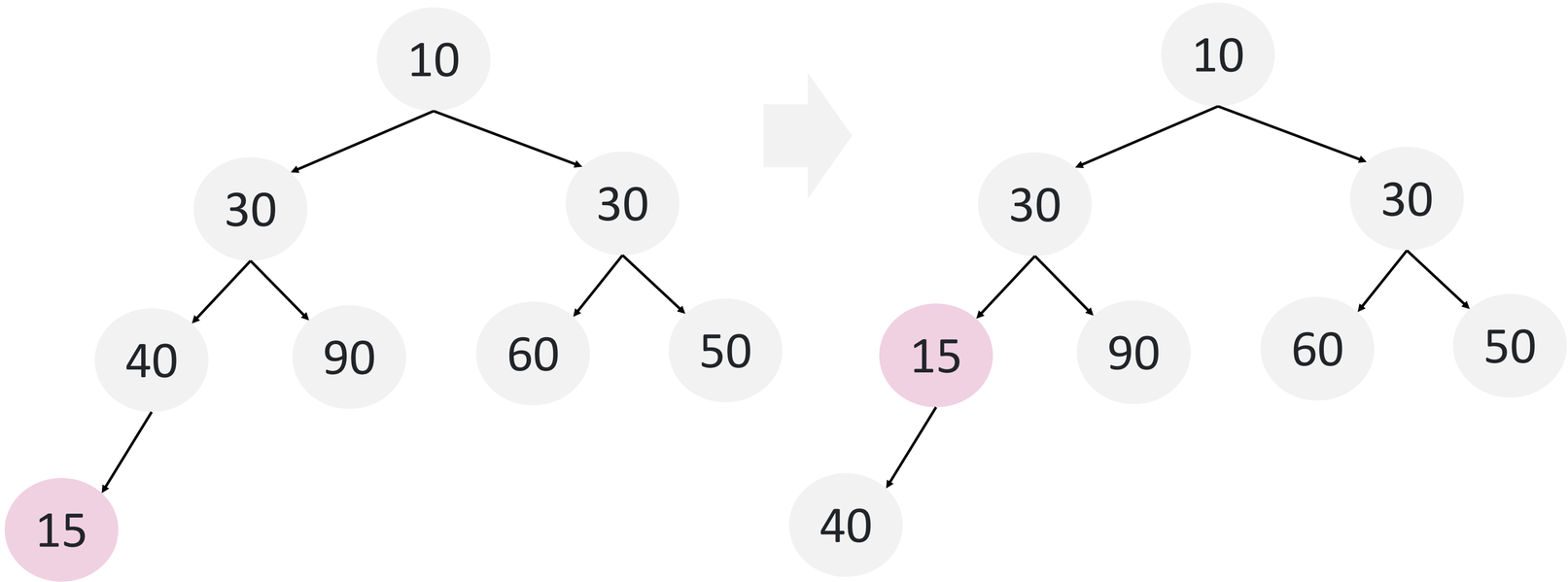
Enqueue

2. Restore the heap property (Upheap of “percolate up”)
   - Swap the node along an upward path from the insertion node
   - Upheap terminates when the node reaches the root or if parent ≤ item



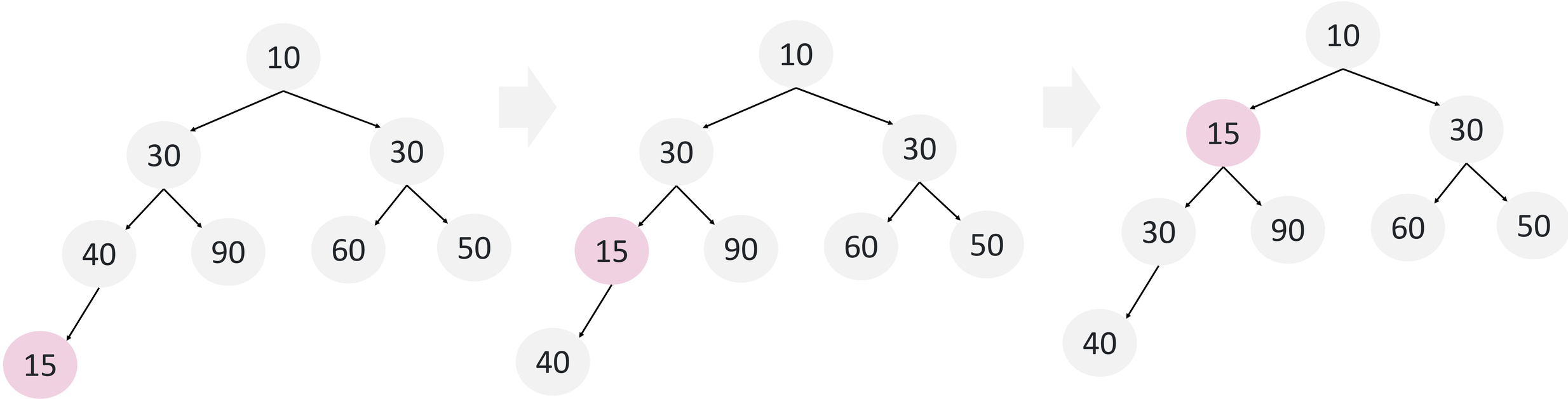
Enqueue

2. Restore the heap property (Upheap of “percolate up”)
   - Swap the node along an upward path from the insertion node
   - Upheap terminates when the node reaches the root or if parent $\leq$ item



Enqueue

2. Restore the heap property (Upheap of “percolate up”)
   - Swap the node along an upward path from the insertion node
   - Upheap terminates when the node reaches the root or if parent $\leq$ item

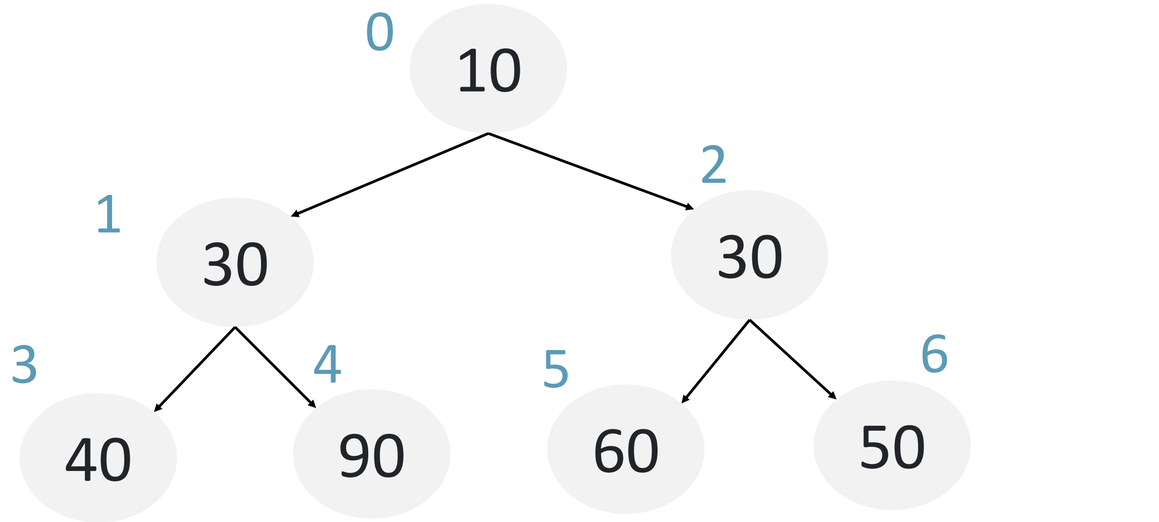


Array Implementation

- Binary heaps are a way to implement a priority queue and also an abstraction on top of arrays
- A simple array is the best solution for heaps because the complete nature of the structure, with all levels filled from left to right

Array Implementation

- Binary heaps are a way to implement a priority queue and also an abstraction on top of arrays
- A simple array is the best solution for heaps because the complete nature of the structure, with all levels filled from left to right



10	30	30	40	90	60	50
0	1	2	3	4	5	6

Array Implementation

priority queue를 heap으로 구현하기로 했는데, 실제로 어떻게 구현할까. 에 대한 내용

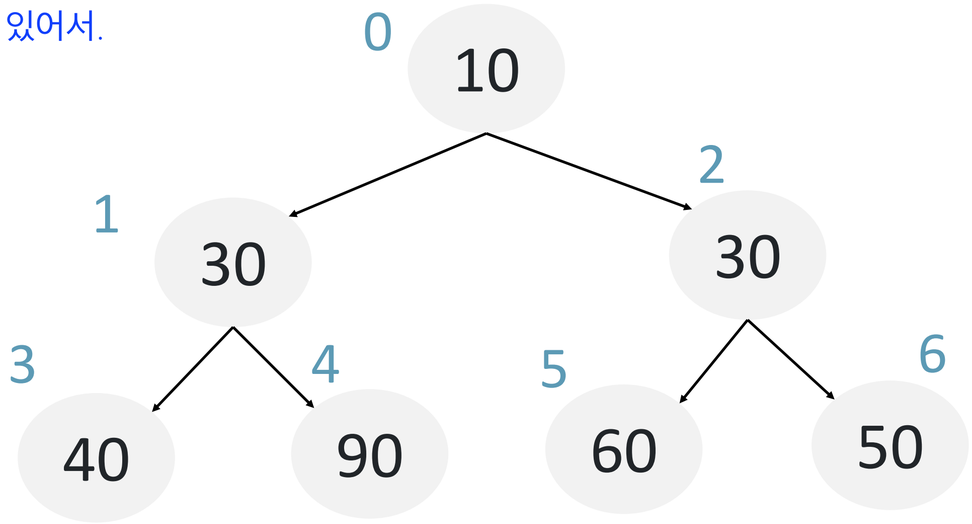
Binary heap for priority queue구현을 위해

- Binary heaps are a way to implement a priority queue and also an abstraction on top of arrays

array로 heap을 구현하고, 이를 통해서 priority queue를 구현

- A simple array is the best solution for heaps because the complete nature of the structure, with all levels filled from left to right

dept별로 왼쪽에서 오른쪽으로 채워 넣을 수 있어서.
binary heap의 complete property 를
잘 표현해 낼 수 있다.



From node `i`:

left child: `i*2+1`
right child: `i*2+2`
parent: `(i-1)/2`

10	30	30	40	90	60	50
0	1	2	3	4	5	6

Array Implementation

- Pros
  - Less "wasted" space
    - Just unused space on right
    - Array would waste more space if tree were not complete
  - Multiplying and dividing by 2 is very fast (shift operations in hardware)
  - Last used position is just index **`size`**

특히 더 shift operation으로 2를 곱하고 나누는 것이 더 빠름.

- Cons

array라는 단점. fixed size

  - Same might-be-empty or might-get-full problems we saw with stacks and queues (resize by doubling as necessary)

Heaps: Implementation

```
// A class for Heap
class Heap
{
    int *harr; // pointer to array of elements in heap
    int capacity; // maximum possible size of heap
    int heap_size; // Current number of elements in heap
public:
    // Constructor
    Heap(int capacity);

    // Inserts a new key 'k'
    void insertKey(int k);

    // to get index of parent node at index i
    int parent(int i) { return (i-1)/2; }

    // to get index of left child of node at index i
    int left(int i) { return (2*i + 1); }

    // to get index of right child of node at index i
    int right(int i) { return (2*i + 2); }
}
```

```
// Inserts a new key 'k'
void Heap::insertKey(int k)
{
    if (heap_size == capacity)
    {
        cout << "\nOverflow: Could not insertKey\n";
        return;
    }

    // First insert the new key at the end
    heap_size++;
    int i = heap_size - 1;
    harr[i] = k;
}
```

Summary

- Priority queues are queues ordered by priority of their elements, where the highest priority elements get dequeued first
- Binary heaps are a good way of organizing data when creating a priority queue
- There can be multiple ways to implement the same abstraction

꼭 굳이 binary heap이 아니더라도 priority queue라는 ADT를 구현 하는 방법은 많음.

References

- Previous Computing 1 lecture slides, Stanford CS106B, U of W CSE373, the textbooks

Next class
Minimum Spanning Trees