Data Structure Assignment 2

201911013 곽현우

A4 - Linked List (5+1 points)

All the required python files are in 'Ch7' directory. Find `PositionalList` class in `positional_list.py`.

A4-1 (1 point)

Describe in detail how to swap two nodes *x* and *y* (and not just their contents) in a singly linked list ***L*** given references only to *x* and *y* (You may use `L.head` for the reference to the head node of the ***L***). Repeat this exercise for the case when L is a doubly linked list. Which algorithm takes more time?

1) Singly Linked List) Singly Linked List 의 경우 x 와 y 의 노드를 교환하기 위해서는 x 와 y 각각의 노드에 대하여 이전의 노드가 reference 되어있지 않으므로 singly linked list 의 Head 노드(L.head)로부터 이후 노드가 x 혹은 y 일 때까지 전체 노드 수만큼의 탐색이 필요하여 O(n)만큼의 시간이 요구된다.

2) Doubled Linked List) 그러나 Doubled Linked List 의 경우 노드 x 와 y 는 각각의 노드 앞 위의 노드에 대하여 reference 가 되어있으므로 굳이 전체 노드 수를 탐색하는 불필요한 작동을 하지 않게 된다.

따라서 두 노드 x 와 y 를 교환하는 Operation 을 작동할 때 Singly linked list 가 더 오래걸린다.

A4-2 (1 point)

Implement `__reversed__` method for the `PositionalList` class. This method is similar to the given `__iter__`, but it iterates the elements in *reversed* order.

```
def __reverse__(self):
    """Generate a forward iteration of the elements of the list."""
    cursor = self.last()
    while cursor is not None:
      yield cursor.element()
      cursor = self.before(cursor)
```

(주어진 __iter__method 에 대하여 first 는 last 로 after 는 before 로 바꾼다.)

A4-3 (1 point)

Modify `add_last` and `add_before` methods, only using methods in the set `{is_empty, first, last, before, after, add_after, add_first}`.

```
def add_first(self, e):
"""Insert element e at the front of the list and return new Position."""
    if self.is_empty():
        return self._insert_between(e, self._header, self._trailer)
    else:
        return self.add_before(self.first(), e)

def add_last(self, e):
 """Insert element e at the back of the list and return new Position."""
    if self.is_empty():
        return self._insert_between(e, self._header, self._trailer)
    else:
        return self.add_after(self.last(), e)
```

A4-4 (1 point)

Update the `PositionalList` to support an additional method `max()`, that returns the maximum element from a `PositionalList`. For example. for instance ***L*** containing comparable elements, you execute the function by calling`L.max()`.

```
def max(self):
    maximum = self.delete(self.first())
    for element in self:
        if maximum < element:
            maximum = element
        else:
            pass
    return maximum
```

A4-5 (1 point)

Update the `PositionalList` to support an additional method `find(e)`, which returns the position of the (first occurance of) element ***e*** in the list (or **None** if not found).

```
def find(self, e):
    cursor = self.first()
    for element in range(len(self)):
        if cursor.element() == e:
            return element
        else:
            cursor = self.after(cursor)
    return None # 찾으려는 e가 없을 때 None 반환
```

A4-Bonus (1 point)

Update the previous `find(e)` using recursion. Your method should not contain any loops. How much space does your method use in addition to the space used for ***L***?

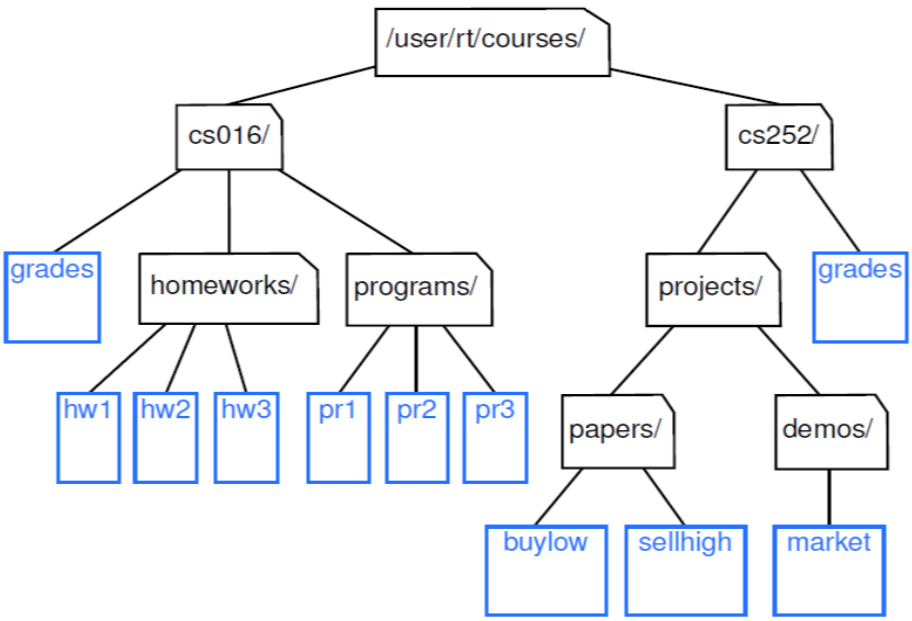
```
def find(self, e):
    start = self.first()
    def walk(cursor, e): #새로운 함수를 내부에 만들어서 recursion 시킨다.
        if cursor == None:
            return None
        if cursor.element() == e:
            return 0
        recur_val = walk(self.after(cursor), e)
        if recur_val == None:
            return None
        return recur_val + 1
    result = walk(start, e)
    return result
```

find method 자체적으로는 self argument 때문에 recursion 하지 못하므로 내부에 새로운 함수를 정의하여 recursion 하였다. 이때 차지하는 공간은 처음의 위치와 찾고자하는 e 의 위치의 차이만큼 recursion 하게 되고 그 차이를 n 이라 할 때 recursion 시킨 find method 는 본래 차지하던 공간의 n 배만큼의 공간이 필요하다.

A5 - Trees (7 points)

A5-1 (1 point)

Answer the following questions. You have to answer all of them correctly to get the point.



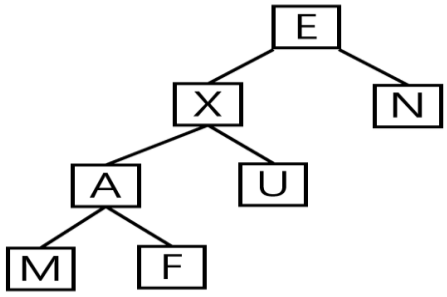
1. Which node is the root?
   Node `/user/rt/courses/`
2. What are the internal nodes?
   Node `/user/rt/co`, Node `cs016/`, Node `homeworks/` ,Node `programs/`, Node `cs252/`, Node `projects/`, Node `papers/`, Node `demos/`
3. How many descendants does node `cs016/` have?
   The number of descendants of node `cs016/` is 9
4. How many ancestors does node `cs016/` have?
   The number of ancestors of node `cs016/` is 1(root)
5. What are the siblings of node `homeworks/`?
   The siblings of node `homeworks/` are Node `programs/` and Node `grades`
6. Which nodes are in the subtree rooted at node `projects/`?
   The Nodes of the subtree rooted at node `projects/` are node `projects/` , node `papers/` , node `demos/` , node `buylow` , node `sellhigh` , node `market`
7. What is the depth of node `papers/`?
   The depth of the node `papers/` is same with the number of ancestors of node `papers/`
   So It is 3

8. What is the height of the tree?
   It is same with the maximum depth of any node in this Tree. So it is 4

A5-2 (1 point)

Draw a binary tree ***T*** that simultaneously satisfies the followings

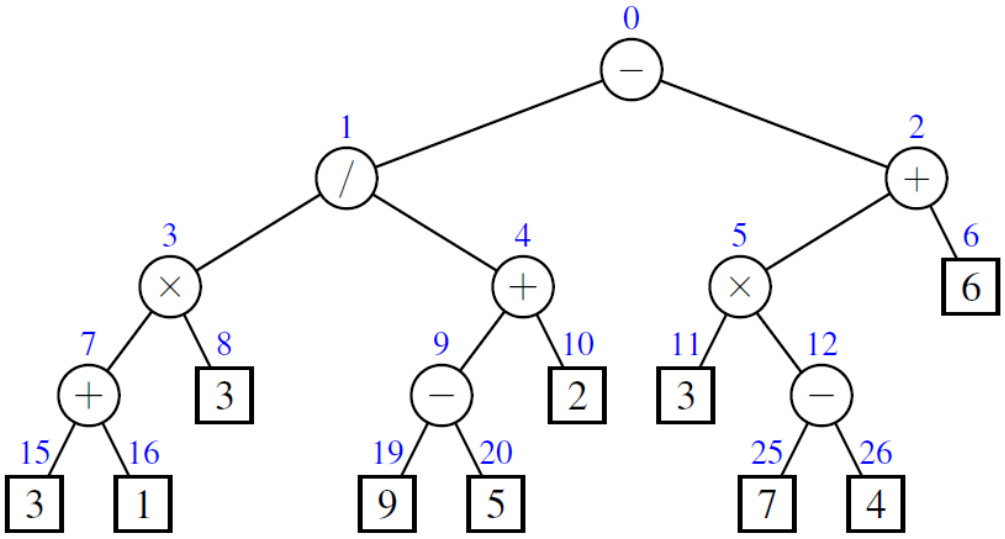
- Each internal node ***T*** stores a single character
- A *preorder* traversal of ***T*** yields **EXAMFUN**
- A *inorder* traversal of ***T*** yields **MAFXUEN**



Preorder traversal 에 따라 갈 때 맨 처음 방문하는 Node 는 Root 이고 그 다음 방문하는 Node 는 무조건 Root 의 Left child 가 된다. 따라서 이 Tree 의 Root 는 E 이고 Left child of Root 는 X 이다. 이때 Inorder traversal 에 따라 MAFXU 는 Root 를 X 로 가지는 Subtree 의 Node 라는 것과 Right child of Root 가 N 이라는 것을 알 수 있다. Subtree rooted at Node X 는 Inorder Traversal 에서 X 이후에 방문한 Node U 는 Right child of X 임을 알 수 있고 Preorder Traversal 을 통해 Left Child of X 가 A 라는 것과 A 의 Left, Right Child 가 각각 M 과 A 임을 알 수 있다.

A5-3 (2 points)

Let ***T*** be a binary tree with *n* positions that is realized with an array representation ***A***, and let *f()* be the level numbering function of the positions of ***T***.



A *level numbering* function is a function that numbers the position on each level of ***T*** in increasing order from left to right. For example, check this figure.

Give pseudo-code descriptions of each of the methods `root()`, `parent(p)`, `left(p)`, `right(p)`, `is_leaf(p)`, `is_root(p)`, when *p* is given as the current position in the array. Assume the numbering starts from 0.

위의 Tree 가 Array 로 표현되었을 때 각 Node 에 대한 위치에 대하여 다음과 같은 성질을 만족한다.
① f(root) = 1
② If node is the left child of the parent(node), f(node) = 2 * f(parent(node)) + 1
③ If node is the right child of the parent(node), f(node) = 2 * f(parent(node)) + 2

1)
```
def root():
    Return A[0]
```
2)
```
def parent(p):
    Index = f(p)
    if Index != 0:
        if p%2 == 0:
            return A[(Index-2]/2]
        else:
            return A[(Index-1)/2]
    else:
        return None
```

3)
```
def left(p):
        Index = f(p)
        if A[((Index*2 + 1] is not None:
                return A[ Index*2 + 1]
        else:
                return None
```
4)
```
def right(p):
        Index = f(p)
        if A[Index*2 + 2] is not None:
                return A[Index*2 + 2]
        else:
                return None
```
5)
```
def is_leaf(p):
        Index = f(p)
        return A[2*Index+1] is None and A[2*Index+2] is None:
```
6)
```
def is_root(p):
        Index = f(p)
        return Index == 0
```

A5-4 (3 points)

All the required python files are in 'Ch8' directory. Find `LinkedBinaryTree` class in `linked_binary_tree.py`. Implement a new method, `_delete_subtree(p)`, that removes the entire subtree rooted at position *p*, making sure to maintain the count on the size of the tree. What is the running time of your implementation?

```
def _delete_subtree(self, p):
    if self.num_children(p) == 0:
        node = self._validate(p)
        node._parent = node
        node._element = None
        self._size -= 1
    else:
        for sub_left in self._subtree_postorder(self.left(p)):
            node = self._validate(sub_left)
            node._parent = node
            node._element = None
            self._size -= 1
        for sub_right in self._subtree_postorder(self.right(p)):
            node = self._validate(sub_right)
            node._parent = node
            node._element = None
            self._size -= 1

        root_node = self._validate(self.root())
        if p == self.root():
            root_node = None
            self._size -= 1
        elif p == self.left(self.parent(p)):
            parent_node = self._validate(self.parent(p))
            parent_node._left = None
            self._size -= 1
        else:
            parent_node = self._validate(self.parent(p))
            parent_node._right = None
            self._size -= 1
```

A6 - Priority Queues, Heaps (8+2 points)

A6-1 (1 point)

How long would it take to remove the $\lceil \log n \rceil$ smallest elements from a heap that contains *n* entries, using the `remove_min` operation?

`remove_min` operation 을 사용하면 크게 2 가지 단계로 진행된다. 첫번째로 Heap 의 특징에 따라 Heap 의 root 의 값이 가장 작기 때문에 root 를 제거하고 반환한다. 두번째로 Heap 의 last element 를 root 로 만들고 Heap – Order 에 맞게 다시 정렬한다. 이때 첫번째 단계에서는 O(1) 만큼의 시간이 걸리고 두번째 단계에서는 Worst – Case 를 생각할 때 그 Heap 의 Height 만큼 시간이 걸리므로 O(log n)만큼의 시간이 걸린다. 따라서 `remove_min` operation 총 running time 은 O(log n)이다. 이때 가장 작은 원소 $\lceil \log n \rceil$ 개 만큼 제거해야 함으로 문제에서 주어진 상황의 총 Running Time 은 $O((logn)^2)$이다.

A6-2 (1 point)

What does each `remove_min` call return within the following sequence of priority queue ADT methods: `add(5,A), add(4,B), add(7,F), add(1,D), remove_min(), add(3,J), add(6,L), remove_min(), remove_min(), add(8,G), remove_min(), add(2,H), remove_min(), remove_min()`?

Note: Use only a pen and paper. Don't cheat by executing the code in a computer.

`add(5,A), add(4,B), add(7,F), add(1,D)` -> [(5, A), (4, B), (7, F), (1, D)]
∴ `remove_min()` = (1, D)
`add(3,J), add(6,L)` -> [(5, A), (4, B), (7, F), (3, J), (6, L)]
∴ `remove_min()` = (3, J) and next `remove_min()` = (4, B)
`add(8,G)` -> [(5, A), (7, F), (6, L), (8, G)] ∴ `remove_min()` = (5, A)
`add(2,H)` -> [(7, F), (6, L), (8, G), (2, H)] ∴ `remove_min()` = (2, H)
and next `remove_min()` = (6, L)

A6-3 (1 point)

An airport is developing a computer simulation of air-traffic control that handles events such as landings and takeoffs. Each event has a time stamp that denotes the time when the event will occur. The simulation program needs to efficiently perform the following two fundamental operations:

- Insert an event with a given time stamp (that is, add a future event).
- Extract the event with smallest time stamp (that is, determine the next event to process).

Which data structure should be used for the above operations? Why?

이러한 프로그램을 만들기 위해서는 우선 Time Stamp 를 Key 로 가지고 해당하는 key 값에 일어나는 사건을 그 key 의 value 값으로 가지는 자료구조여야 한다. 또한 위의 2 가지 작동인 key 에 대한 value 를 삽입 operation 이 가능해야하고 가장 key 값이 작은 사건들을 꺼낼 수 있어야 하므로 Priority Queue 자료구조가 적합할 것이다.

A6-4 (1 point)

Illustrate the execution of the selection-sort algorithm on the following input sequence: (22, 15, 36, 44, 10, 3, 9, 13, 29, 25). ***Note:*** Display the internal data in the priority queue for each step.

selection – sort Algorithm 은 주어진 Sequence 의 n 개의 원소를 Sorting 하지 않고 순서대로 Priority Queue 에 삽입한다. 이때 Priority Queue 는 Unsorted 되어있다.(Phase 1) 이후 Priority Queue 에 remove_min operation 을 통해 가장 작은 원소들 순서대로 다시 Sequence 로 옮겨준다.(Phase 2)

	Sequence	Priority Queue
Input	(22, 15, 36, 44, 10, 3, 9, 13, 29, 25)	()
Phase 1		
	(15, 36, 44, 10, 3, 9, 13, 29, 25)	(22)
	(36, 44, 10, 3, 9, 13, 29, 25)	(22, 15)
	(44, 10, 3, 9, 13, 29, 25)	(22, 15, 36)
	(10, 3, 9, 13, 29, 25)	(22, 15, 36, 44)
	(3, 9, 13, 29, 25)	(22, 15, 36, 44, 10)
	(9, 13, 29, 25)	(22, 15, 36, 44, 10, 3)
	(13, 29, 25)	(22, 15, 36, 44, 10, 3, 9)
	(29, 25)	(22, 15, 36, 44, 10, 3, 9, 13)
	(25)	(22, 15, 36, 44, 10, 3, 9, 13, 29)
	()	(22, 15, 36, 44, 10, 3, 9, 13, 29, 25)
Phase 2		
	(3)	(22, 15, 36, 44, 10, 9, 13, 29, 25)
	(3, 9)	(22, 15, 36, 44, 10, 13, 29, 25)
	(3, 9, 10)	(22, 15, 36, 44, 13, 29, 25)
	(3, 9, 10, 13)	(22, 15, 36, 44, 29, 25)
	(3, 9, 10, 13, 15)	(22, 36, 44, 29, 25)
	(3, 9, 10, 13, 15, 22)	(36, 44, 29, 25)
	(3, 9, 10, 13, 15, 22, 25)	(36, 44, 29)

	(3, 9, 10, 13, 15, 22, 25, 29)	(36, 44)
	(3, 9, 10, 13, 15, 22, 25, 29, 36)	(44)
	(3, 9, 10, 13, 15, 22, 25, 29, 36, 44)	()

A6-5 (1 point)

Repeat A6-4 with the insertion-sort algorithm.

Insertion – sort Algorithm 은 주어진 Sequence 의 n 개의 원소를 순서대로 Sorting 하여 Priority Queue 에 삽입한다.(Phase 1) 이후 Priority Queue 에 Sorting 된 원소들을 처음부터 순서대로 Sequence 로 옮겨준다.(Phase 2)

	Sequence	Priority Queue
Input	(22, 15, 36, 44, 10, 3, 9, 13, 29, 25)	()
Phase 1		
	(22, 15, 36, 44, 10, 3, 9, 13, 29, 25)	(22)
	(36, 44, 10, 3, 9, 13, 29, 25)	(15, 22)
	(44, 10, 3, 9, 13, 29, 25)	(15, 22, 36)
	(10, 3, 9, 13, 29, 25)	(15, 22, 36, 44)
	(3, 9, 13, 29, 25)	(10, 15, 22, 36, 44)
	(9, 13, 29, 25)	(3, 10, 15, 22, 36, 44)
	(13, 29, 25)	(3, 9, 10, 15, 22, 36, 44)
	(29, 25)	(3, 9, 10, 13, 15, 22, 36, 44)
	(25)	(3, 9, 10, 13, 15, 22, 29, 36, 44)
	()	(3, 9, 10, 13, 15, 22, 25, 29, 36, 44)
Phase 2		
	(3)	(9, 10, 13, 15, 22, 25, 29, 36, 44)
	(3, 9)	(10, 13, 15, 22, 25, 29, 36, 44)
	(3, 9, 10)	(13, 15, 22, 25, 29, 36, 44)
	(3, 9, 10, 13)	(15, 22, 25, 29, 36, 44)
	(3, 9, 10, 13, 15)	(22, 25, 29, 36, 44)
	(3, 9, 10, 13, 15, 22)	(25, 29, 36, 44)

	(3, 9, 10, 13, 15, 22, 25)	(29, 36, 44)
	(3, 9, 10, 13, 15, 22, 25, 29)	(36, 44)
	(3, 9, 10, 13, 15, 22, 25, 29, 36)	(44)
	(3, 9, 10, 13, 15, 22, 25, 29, 36, 44)	()

A6-6 (3 points)

In 'Ch9' directory, find `HeapPriorityQueue` class in `heap_priority_queue.py`. Implement `heappushpop` method and `heapreplace` methods, with semantics akin to the described methods for the `heapq` Python module as follow:

- `heappushpop(e)`: Push element e on and then pop and return the smallest item. The time is O(logn), but it is slightly more efficient than separate calls to push and pop because the size of the list never changes. If the newly pushed element becomes the smallest, it is immediately returned. Otherwise, the new element takes the place of the popped element at the root and a down-heap is performed.
- `heapreplace(e)`: Similar to heappushpop, but equivalent to the pop being performed before the push (in other words, the new element cannot be returned as the smallest). Again, the time is O(logn), but it is more efficient than two separate operations.

```
def heappushpop(self, key, value):
    if self.is_empty() or key <= self._data[0]._key:
        return (key, value)
    else:
        new = self._Item(key, value)
        old = self._data[0]
        self._data[0] = new
        self._downheap(0)
        return (old._key, old._value)

def heapreplace(self, key, value):
    if self.is_empty():
        return None
    new = self._Item(key, value)
    old = self._data[0]
    self._data[0] = new
    self._downheap(0)
    return (old._key, old._value)
```

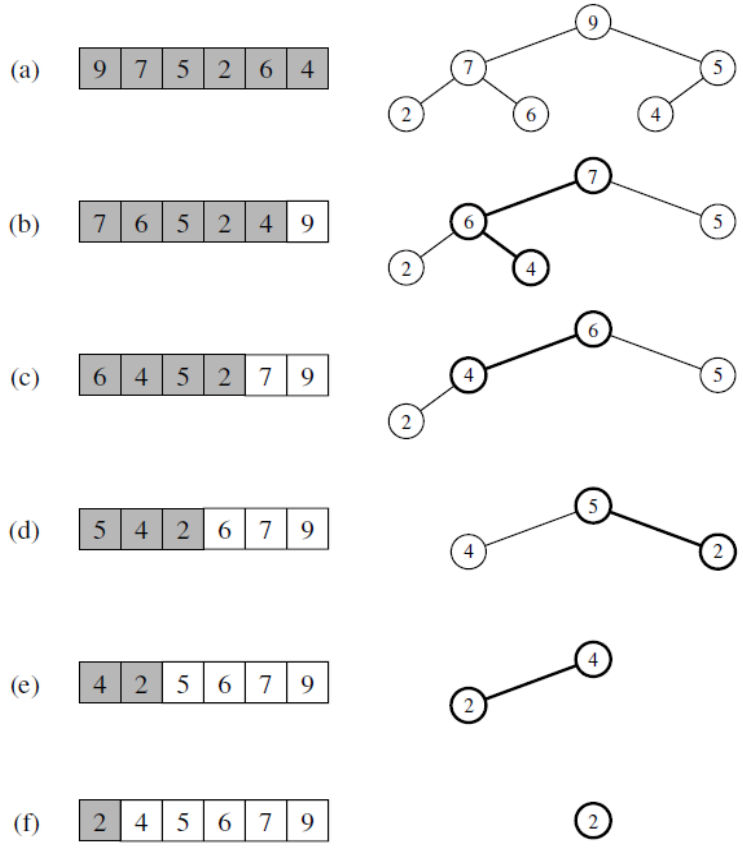
A6-Bonus (2 points)

Implement an in-place heap sort algorithm, `heapsort(a)`, where **a** is a list of unsorted numbers before running the method, and **a** becomes a sorted list after the method. Follow these steps for implementing it (textbook p389).

If the collection C to be sorted is implemented by means of an array-based sequence, most notably as a Python list, we can speed up heap-sort and reduce its space requirement by a constant factor using a portion of the list itself to store the heap, thus avoiding the use of an auxiliary heap data structure. This is accomplished by modifying the algorithm as follows:

1. We redefine the heap operations to be a maximum-oriented heap, with each position's key being at least as large as its children. This can be done by recoding the algorithm, or by adjusting the notion of keys to be negatively oriented. At any time during the execution of the algorithm, we use the left portion of C, up to a certain index i−1, to store the entries of the heap, and the right portion of C, from index i to n−1, to store the elements of the sequence. Thus, the first i elements of C (at indices 0, . . . , i−1) provide the array-list representation of the heap.
2. In the first phase of the algorithm, we start with an empty heap and move the boundary between the heap and the sequence from left to right, one step at a time. In step i, for i = 1, . . . ,n, we expand the heap by adding the element at index i−1.
3. In the second phase of the algorithm, we start with an empty sequence and move the boundary between the heap and the sequence from right to left, one step at a time. At step i, for i = 1, . . . ,n, we remove a maximum element from the heap and store it at index n−i.

In general, we say that a sorting algorithm is ***in-place*** if it uses only a small amount of memory in addition to the sequence storing the objects to be sorted. The variation of heap-sort above qualifies as in-place; instead of transferring elementsout of the sequence and then back in, we simply rearrange them. We illustrate the second phase of in-place heap-sort in the following figure.



```
def heapsort(a):
    n = len(a)
    for index1 in range(n):
        if index1 == 0:
            continue
        parent = []
        index1_1 = index1
        while True:
            parent_index = (index1_1 - 1) // 2
            parent.append(parent_index)
            index1_1 = parent_index
            if index1_1 <= 0:
                if parent[-1] < 0:
                    parent.remove(parent[-1])
                break

        for in_index1 in parent:
            if a[index1] > a[in_index1]:
                a[index1], a[in_index1] = a[in_index1], a[index1]
            for i in range(len(parent)):
                if (i + 1) < len(parent) and a[parent[i]] > a[parent[i + 1]] :
                    a[parent[i]] , a[parent[i + 1]] = a[parent[i + 1]],a[parent[i]]
                else:
                    break
            else:
                break
    len_heap = len(a)
    for index2 in range(len_heap):
        a[0], a[(len_heap - 1)] = a[(len_heap - 1) ], a[0]
        len_heap -= 1
        if len_heap <= 0:
            break
        parent1 = []
        index2_1 = index2
        while True:
            parent1_index = (index2_1 - 1) // 2
            parent1.append(parent1_index)
            index2_1 = parent1_index
            if index2_1 < 0:
                if parent1[-1] < 0:
                    parent1.remove(parent1[-1])
                break
        for in_index2 in parent1:
            if a[index2] > a[in_index2]:
                a[index2], a[in_index2] = a[in_index2], a[index2]
            for i in range(len(parent1)):
                if (i + 1) < len(parent1) and a[parent1[i]] > a[parent1[i + 1]] :
                    a[parent1[i]], a[parent1[i + 1]] = a[parent1[i + 1]],
a[parent1[i]]
                else:
                    break
            else:
                break
    return a
```

Upheap 까지는 구현을 했는데 밑에 Downheap 의 구현을 완벽히 하지 못했습니다.