자료구조 9주차 실습

Priority queue & Heap

감성인공지능연구실
방윤석 임희수

인하대학교
INHA UNIVERSITY

퀴즈 관련 & 앞으로의 진행

인하대학교 INHA UNIVERSITY

주로 오답 처리된 부분

- 예외 처리를 안 하는 부분
  - None에 접근하는 경우 조심하세요..
- 주어진 양식과 다르게 출력하는 경우
  - 줄 나눔, 띄어쓰기 양식에 맞지 않는 경우
    - 보이지 않는, 예상할 수 없는 띄어쓰기는 예외처리 합니다. (<-이런거)

앞으로의 진행

- 강의 4번과 퀴즈 2번이 남음
- Search tree를 제거할 계획

주차	계획	변경된 계획
7	Priority Queues and Heaps	휴강
8	Mid-term	Mid-term
9	#Quiz2	Priority Queues and Heaps
10	Search Tree	#Quiz2
11	Maps, Dictionaries, and Hashing	Maps, Dictionaries, and Hashing
12	Sorting and Selection	Sorting and Selection
13	#Quiz3	#Quiz3
14	현충일	현충일
15	-	

Tree

Revisit Tree

Tree

- 재귀적으로 연결되는 형태의 자료구조이다.
  - 노드 관점에서, 부모 노드 아래 자식 노드들이 연결되고, 자식 노드 아래 또다른 자식 노드들이 연결됨
  - Tree 관점에서, tree안에 subtree들이 연결되어 있음

이진트리 (Binary Tree)

- 자식 노드가 최대 2개이다.
- 자식 노드가 2개다보니 left, right 로 구분한다

이진 트리에서의 탐색

- 전위 순회(preorder traversal): 자신, 왼쪽 자손, 오른쪽 자손 순서로 방문하는 순회
- 후위 순회(postorder traversal): 왼쪽 자손, 오른쪽 자손, 자신 순서로 방문하는 순회
- 중위 순회(inorder traversal): 왼쪽 자손, 자신, 오른쪽 자손 순서로 방문하는 순회

Revisit Tree

간단한 tree와 InorderTraversal 구현

Tree

- left, right에 child를 넣어둠
- parent는 parent를 가리킴 (optional)
- is_leaf()는 leaf인지 확인

InorderTraversal

- Left, Root, Right 순서로 탐색

```
class Node: # Tree Node class
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
        self.parent = None

    def setLeft(self, left):
        self.left = left
        if left is not None:
            left.parent = self

    def setRight(self, right):
        self.right = right
        if right is not None:
            right.parent = self

    def is_leaf(root):
        return root.left is None and root.right is None

def InorderTraversal(root): # Left, Root, Right
    if root: # Check if the node is not None
        InorderTraversal(root.left)
        print(root.data, end=' ')
        InorderTraversal(root.right)
```

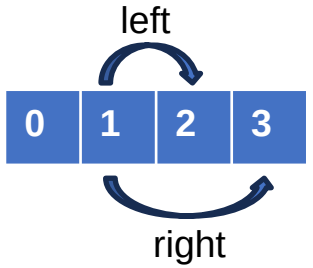
Array based binary tree

Array based binary tree

자식 노드는 2개밖에 없는데, 노드를 하나하나 가리키면서 생성해야 할까?

배열의 index로 접근하면 어떨까?

- index 0는 비우고, index 1을 root로 한다.
- 왼쪽 자식은 index *2 , 오른쪽 자식은 index * 2 + 1의 위치에 저장한다.
- 배열 기반이다보니, 생성할 수 있는 노드의 수가 제한된다.



Index 0을 root로 하면 안되는가?

문제는 없다.

- 왼쪽 자식은 index *2 + 1 , 오른쪽 자식은 index * 2 + 2의 위치에 저장한다.
- Index 0을 비우는 이유는, 나중에 정렬알고리즘 등을 구현할 때 해당 공간을 사용하기도 하기 때문

Array based binary tree

- setLeftChild, setRightChild :
  parent 의 child를 설정

- getValue : 해당 index가 가리키는
  node의 값 return

- getLeftChild, getRightChild, getParent :
  left child, right child, paren의 값 return

```python
class Tree:
    def __init__(self, n, rootValue): # Initialize the tree with the given capacity
        self.capacity = n
        self.array = array.array('h', [0]*self.capacity)
        self.array[1] = rootValue # The first element is the root node

    def setLeftChild(self, parentIndex, childValue):
        # Check if the left child index is within bounds and the parent index is valid
        if parentIndex * 2 >= self.capacity or parentIndex < 1:
            return
        self.array[parentIndex * 2] = childValue # Set the left child

    def setRightChild(self, parentIndex, childValue):
        # Check if the right child index is within bounds and the parent index is valid
        if parentIndex * 2 + 1 >= self.capacity or parentIndex < 1:
            return
        self.array[parentIndex * 2 + 1] = childValue

    def getValue(self, index):
        # Check if the index is within bounds
        if index < 1 or index >= self.capacity:
            return None
        return self.array[index]

    def getLeftChild(self, index):
        # Check if the left child index is within bounds and the parent index is valid
        if index * 2 >= self.capacity or index < 1:
            return None
        return self.array[index * 2]

    def getRightChild(self, index):
        # Check if the right child index is within bounds and the parent index is valid
        if index * 2 + 1 >= self.capacity or index < 1:
            return None
        return self.array[index * 2 + 1]

    def getParent(self, index):
        # Check if the child index is within bounds
        if index <= 1 or index >= self.capacity:
            return None
        return self.array[index // 2] # Return the parent node value
```

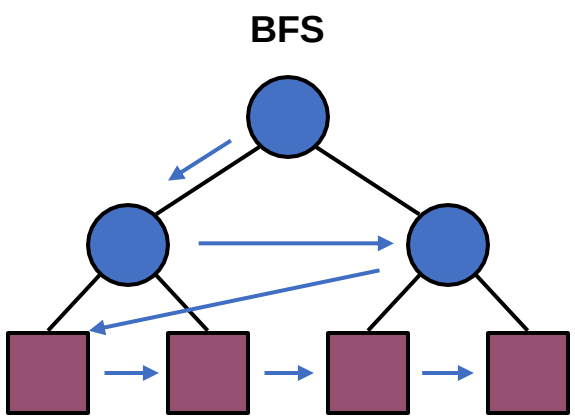
Exercise #1

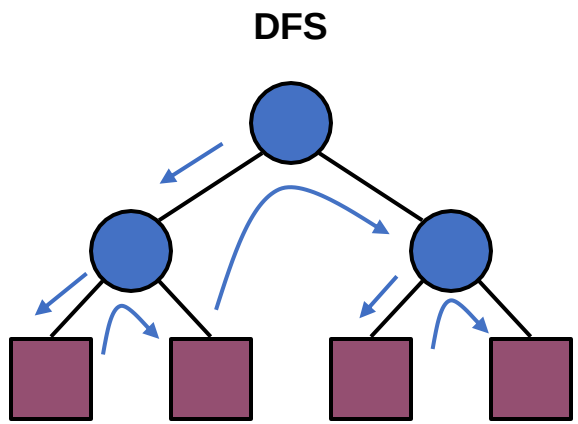
인하대학교 INHA UNIVERSITY

BFS(Breadth First Search)

너비우선탐색(BFS)를 구현하여라. BFS는 루트에서 가까운 노드부터 멀리 있는 노드 순서로 탐색하는 방식

- depth가 동일한 node들을 먼저 출력하면 됨





Problem #1

인하대학교 INHA UNIVERSITY

문제

이진 트리가 주어졌을 때, 트리의 경계를 다음과 같은 규칙으로 순회하여 출력하시오.

- 루트 노드에서 시작하여 트리의 왼쪽 경계를 위에서 아래로 출력한다. 단, 리프 노드는 제외한다.
- 모든 리프 노드를 왼쪽에서 오른쪽으로 출력한다.
- 트리의 오른쪽 경계를 아래에서 위로 출력한다. 단, 리프 노드는 제외한다.

주어진 트리의 경계 순회를 출력하는 class를 작성하시오.

메소드 설명

- boundaryTraversal(root) : 이진 트리의 루트 노드를 입력받아 경계 순회 결과를 리스트 형태로 반환한다.

Problem #1

인하대학교 INHA UNIVERSITY

입력 예시

```
#         20
#        /  \
#       8    22
#      / \     \
#     4   12    25
#        /  \
#       10   14
root = Node(20)
root.leftChild = Node(8)
root.rightChild = Node(22)
root.leftChild.leftChild = Node(4)
root.leftChild.rightChild = Node(12)
root.leftChild.rightChild.leftChild = Node(10)
root.leftChild.rightChild.rightChild = Node(14)
root.rightChild.rightChild = Node(25)

boundaryTraversal(root)
```

출력 예시

```
20 8 4 10 14 25 22
```

Problem #2

문제

이진 트리가 주어졌을 때, 트리 내 두 리프 노드 사이의 경로 중 합이 최대가 되는 경로를 찾아 그 합을 출력하는 method를 작성하시오. 경로의 합은 경로에 포함된 모든 노드의 값을 합산한 값이다.

메소드 설명

- maxPathSum(Node root): 이진 트리의 루트 노드를 입력 받아 두 리프 노드 사이의 경로 중 최대 합을 반환한다.

Problem #2

인하대학교 INHA UNIVERSITY

입력 예시

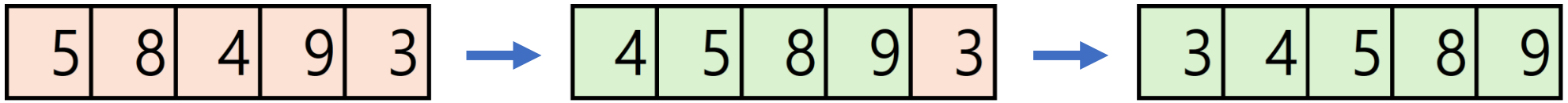
```
#          1
#        /   \
#      -2     3
#      / \   / \
#     8  -1  4  -5


root = Node(1)
root.leftChild = Node(-2)
root.rightChild = Node(3)
root.leftChild.leftChild = Node(8)
root.leftChild.rightChild = Node(-1)
root.rightChild.leftChild = Node(4)
root.rightChild.rightChild = Node(-5)

print(findMaxPathSum(root))
```

출력 예시

Sorting

What is Sorting

- 데이터를 정렬하는 것
- insertion sort와 bubble sort를 사용할 것이며, 이 외에 여러 정렬들이 있음
- insertion sort와 bubble sort은 **정렬된 부분**과 **정렬해야 할 부분**을 구분하는 것이 중요
- 데이터가 1개 이하로 있는 경우는 정렬이 되었다고 보면 된다.

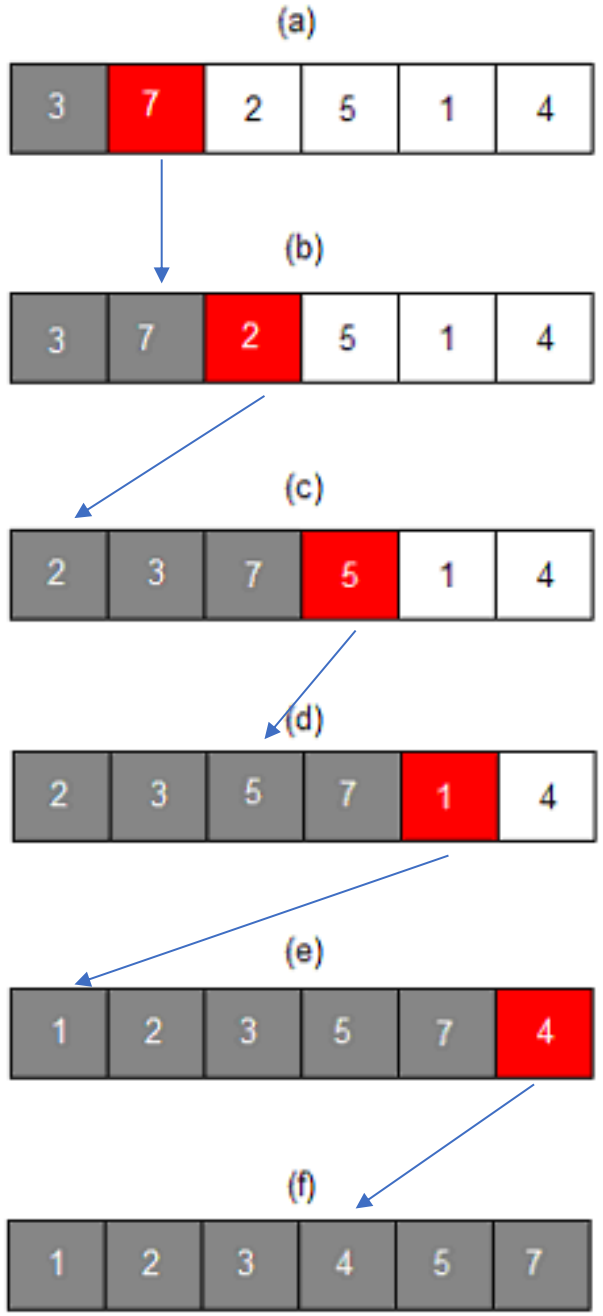


초기 배열 정렬중인 배열 정렬된 배열

Sorting

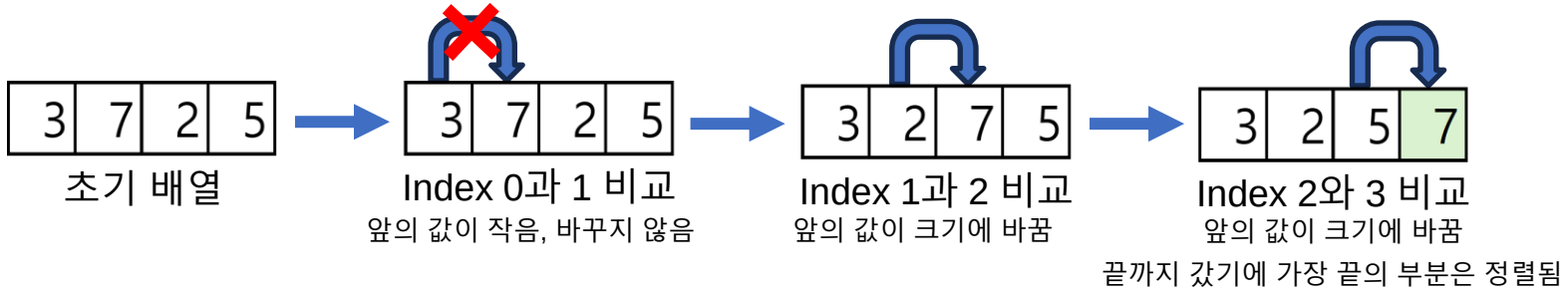
Insertion Sort

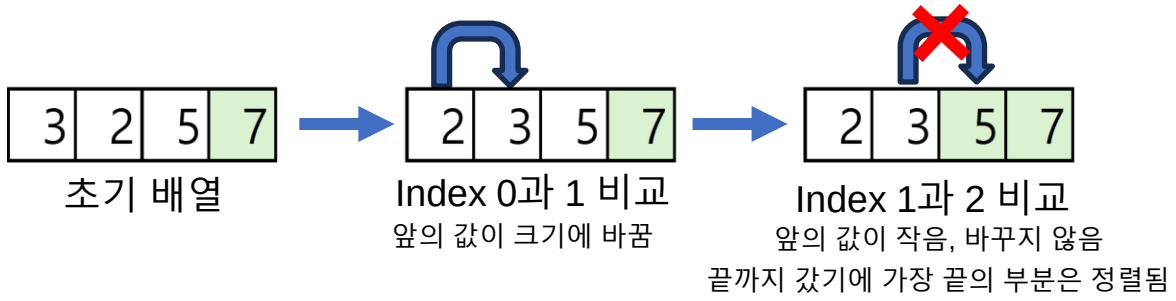
- 올바른 위치로 끼워 넣으며 정렬하기에 insertion sort
- 정렬할 값을 정렬 된 위치에 끼워 넣으면서 정렬하는 방식이다.
- 위의 과정을 반복하며 모든 부분이 정렬되면 된다.



Bubble sort

- 정렬되는 과정이 공기방울들이 있는 것 처럼 있어서 bubble sort
- 앞과 뒤를 비교하고, 만약 앞의 값이 큰 값(혹은 작은 값) 이면 바꾼다.
  이를 앞에서부터 진행하며, 모든 정렬되지 않은 부분에
- 가장 뒤에 들어갈 값부터 확정 지어 넣는다고 생각하면 된다.





Sorting

인하대학교 INHA UNIVERSITY

insertion Sort

- 정렬되지 않은 부분의 가장 앞의 값 (i) 에 대해 정렬된 부분을 거꾸로 진행하며, 넣을 위치를 만들어 가는 방식

Bubble sort

- 배열의 크기만큼 반복을 진행한다.
- 앞의 숫자가 크면 바꾼다.
- i번째 반복은 capacity - i -1 만큼 비교를 반복한다.
  - -1 이 있는 이유 : N개의 숫자가 있으면 앞 뒤 비교는 N-1 번 가능하다.

```
def insertion_sort(self):
    for i in range(1, self.capacity):
        key = self.array[i]
        j = i - 1
        while j >= 0 and key < self.array[j]:
            self.array[j + 1] = self.array[j]
            j -= 1
        self.array[j + 1] = key


def bubble_sort(self):
    for i in range(self.capacity):
        for j in range(self.capacity - i - 1):
            if self.array[j] > self.array[j + 1]:
                temp = self.array[j]
                self.array[j] = self.array[j + 1]
                self.array[j + 1] = temp
```

Priority Queue

Priority Queue

- Queue인데, 우선순위가 더 높은 값이 먼저 나오는 배열이다.
  - 수업에서는 가장 작은 값이 먼저 나오도록 진행된다.
- 즉, 들어있는 값 기준으로 정렬되어서 나온다.

정렬 위치와 방식

- unsorted list vs sorted list
  - Unsorted list : 산출(remove_min) 시 가장 우선순위가 높은 값(작은 값) 을 찾음
  - Sorted list : 삽입(add) 시 우선순위가 높은 값이 앞으로 가도록 정렬한다. 산출 할 때, 가장 앞의 값을 빼기만 하면 된다.
- insertion vs bubble
- 문제 상황에 맞춰서 잘 선택해서 만드시면 됩니다.

Exercise #2

인하대학교 INHA UNIVERSITY

Priority queue

priority queue를 구현하여라

- 실습시에는 bubble sort를 이용해서, sorted list를 만드는 방식으로 진행
- 4가지 방식 중 다른 3가지는 수업 후 파일로 제공할 예정

Problem #3

인하대학교 INHA UNIVERSITY

문제

정렬된 연결 리스트(Linked List)가 스트림처럼 하나씩 주어진다. 각 연결 리스트는 오름차순으로 정렬되어 있다. 매번 새로운 연결 리스트가 추가될 때마다, **지금까지 입력된 모든 리스트를 합쳐서 하나의 정렬된 연결 리스트를 만들어야 한다.**

새롭게 추가된 리스트까지 반영된, 현재까지의 정렬된 결과 리스트의 **head 노드**를 반환하라.

- 각 연결 리스트는 오름차순 정렬되어 있다.
- 연결 리스트를 병합할 때는 항상 **오름차순 정렬된 상태**를 유지해야 한다.
- 병합 과정은 효율적으로 이루어져야 하며, 가능한 한 최적화하라.
- 연결 리스트는 가능한 메모리를 재활용하며, 새 리스트를 복사하지 않고 노드를 직접 이어붙여야 한다.

Problem #3

인하대학교 INHA UNIVERSITY

메소드 설명

- addList(ListNode head) : 새로운 정렬된 연결 리스트의 head가 주어진다.
  이 리스트를 지금까지 입력된 연결 리스트들과 합치고, 병합된 결과 리스트의 head를 반환한다

입력 예시

```
# Create two linked lists
# List 1: 1 -> 3 -> 5
head1 = Node(1)
head1.next = Node(3)
head1.next.next = Node(5)

# List 2: 2 -> 4 -> 6
head2 = Node(2)
head2.next = Node(4)
head2.next.next = Node(6)

# Print the original lists
print("List 1: ", end='')
print_linkedList(head1)
print("List 2: ", end='')
print_linkedList(head2)

# Merge the lists
merged_list = MergeLinkedList()
merged_list.addList(head1)
merged_list.addList(head2)

# Print the merged list
print("Merged List: ", end='')
print_linkedList(merged_list.head)
```

출력 예시

```
List 1: 1 3 5
List 2: 2 4 6
Merged List: 1 2 3 4 5 6
```

Heap

Heap

Heap

- 부모 노드의 값이 자식 노드의 값 보다 항상 큰 (혹은 작은) 자료구조
- Complete Binary tree를 사용한다.
  - Complete Binary tree는 모든 level이 꽉 차 있고, 마지막 level만 예외적으로 왼쪽부터 채워져 있는 트리이다.
  - O(log n)의 복잡도를 유지하기 위해서이다.

배열 기반 이진 트리에서의 구현

- 배열이 연속적인 값을 가지고 있으면 Complete binary tree가 된다.
- 배열의 끝에 값을 추가하며 진행하면 된다. 위치 찾는데 O(1)이 소모

포인터 기반 이진 트리에서의 구현

- 값을 넣을 위치를 찾는 것을 구현해야 한다. 위치 찾는데 O(log n)이 소모
- 수업 후 코드 올려드릴테니 공부하세요..

Heap

Heap

- Root의 index가 1이고,
  left는 2*index, right는 2*index + 1인 tree

- index 0 은 값을 바꾸기 위한 공간
  - Size는 해당 바꾸기 위한 공간을 포함한 크기로 생각하시면 됩니다.
  - 지금은 정수 변수만 사용하지만, 나중에는 여러 구조체를 heap에 넣어서 쓰기 때문에, pointer를 따로 쓰기 보다 해당 방법을 쓰는 것이 효율적입니다.

- Insert시 upHeap을 진행해서, Heap 구조를 유지하도록 한다.
  - parent와 비교해서 parent의 값이 크면 swap

- Remove_min시 downHeap을 진행해서 Heap 구조를 유지시킨다.
  - 자식 중 가장 작은 자식이 parent의 값보다 작으면 swap

```
class Heap:
    def __init__(self, n):
        self.capacity = n
        self.array = array.array('h', [0]*self.capacity)
        self.size = 1

    def insert(self, item):
        if self.size < self.capacity:
            self.array[self.size] = item
            self.upHeap(self.size)
            self.size += 1

    def upHeap(self, index):
        if index <= 1:
            return
        parent = index // 2
        if self.array[index] < self.array[parent]:
            self.array[0] = self.array[index]
            self.array[index] = self.array[parent]
            self.array[parent] = self.array[0]
            self.upHeap(parent)

    def remove_min(self):
        if self.size > 1:
            value = self.array[1]
            self.array[1] = self.array[self.size - 1]
            self.size -= 1
            self.downHeap(1)
            return value
        else:
            return None

    def downHeap(self, index):
        if index >= self.size:
            return
        left = index * 2
        right = index * 2 + 1
        min_index = index
        if left < self.size and self.array[left] < self.array[min_index]:
            min_index = left
        if right < self.size and self.array[right] < self.array[min_index]:
            min_index = right
        if min_index != index:
            self.array[0] = self.array[index]
            self.array[index] = self.array[min_index]
            self.array[min_index] = self.array[0]
            self.downHeap(min_index)
```

Exercise #3

인하대학교 INHA UNIVERSITY

Heap sort 구현

내림차순의 Heap sort 를 구현하여라

- O(n log n)의 시간 복잡도를 가지는 알고리즘이다.

- 입력으로는 배열과 배열의 크기가 들어온다. 해당 배열을 Heap의 알고리즘을 이용해서 구현하여라.

<힌트>

- downHeap만 써서 가능하다.
- 먼저 주어진 배열은 Heap 구조가 아니다. Heap 구조로 만들고 정렬하여라.
- root부터 downHeap을 하려고 하면 subtree중 heap 구조를 만족하지 못하는 부분이 생길 수 있다.