자료구조 11주차 실습

Hash table

감성인공지능연구실
방윤석 임희수

인하대학교
INHA UNIVERSITY

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. 값을 추가할 위치를 찾는데 O(1)이 소모

포인터 기반 이진 트리에서의 구현

- 값을 넣을 위치를 찾는 것을 구현해야 한다. 값을 추가할 위치를 찾는데 O(log n)이 소모

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)
```

Heap

```
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 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
```

Heap

```
1  def upHeap(self, index):
2      if index <= 1:
3          return
4      parent = index // 2
5      if self.array[index] < self.array[parent]:
6          self.array[0] = self.array[index]
7          self.array[index] = self.array[parent]
8          self.array[parent] = self.array[0]
9          self.upHeap(parent)
10
11 def downHeap(self, index):
12     if index >= self.size:
13         return
14     left = index * 2
15     right = index * 2 + 1
16     min_index = index
17     if left < self.size and self.array[left] < self.array[min_index]:
18         min_index = left
19     if right < self.size and self.array[right] < self.array[min_index]:
20         min_index = right
21     if min_index != index:
22         self.array[0] = self.array[index]
23         self.array[index] = self.array[min_index]
24         self.array[min_index] = self.array[0]
25         self.downHeap(min_index)
```

Exercise #1

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.

Exercise #1

```
class HeapSort:
    def __init__(self, arr : array.array, n):
        self.capacity = n
        self.array = arr
        self.size = n

    def sort(self):
        self.build_heap()

        for i in range(self.size-1, 0, -1):
            temp = self.array[i]
            self.array[i] = self.array[0]
            self.array[0] = temp
            self.size -= 1
            self.downHeap(0)

    def build_heap(self):
        for i in range(self.size//2-1, -1, -1):
            self.downHeap(i)
```

Exercise #1

```
1   def downHeap(self, index):
2       if index >= self.size:
3           return
4       left = index * 2 + 1
5       right = index * 2 + 2
6       max_index = index
7       if left < self.size and self.array[left] > self.array[max_index]:
8           max_index = left
9       if right < self.size and self.array[right] > self.array[max_index]:
10          max_index = right
11      if max_index != index:
12          temp = self.array[index]
13          self.array[index] = self.array[max_index]
14          self.array[max_index] = temp
15          self.downHeap(max_index)
16
17  def print_array(self):
18      for i in range(self.capacity):
19          print(self.array[i], end=' ')
20      print()
```

Problem #1

인하대학교 INHA UNIVERSITY

문제

데이터 스트림이 주어진다. 스트림이 진행되는 동안 원소가 하나씩 추가된다. 항상 지금까지 추가된 원소들 중 **k번째로 큰 값**을 반환하는 클래스를 작성하라.

초기에는 정수 배열 nums가 주어지며, 이 배열은 스트림의 초기 상태를 나타낸다. 이후 새로운 값이 추가될 때마다 현재 스트림에서 **k번째로 큰 값**을 반환해야 한다.

add 메소드에 대해 n개의 값이 배열에 있으면, O(**log n**) 의 시간복잡도를 만족하여라.

데이터는 총 1,000 개 이하의 값이 입력된다.

메소드 설명

- KthLargest(int k, int m, array.array nums) : 정수 k와 크기가 m인, 정수 배열 nums를 받아 객체를 초기화 한다. 입력받은 데이터는 내림차순으로 출력한다.
- int add(val) : 새로운 정수 val을 스트림에 추가하고, 지금까지의 원소들 중 **k번째로 큰 값을 반환**한다. 지금까지의 원소 수가 k보다 적으면 가장 작은 값을 반환한다.

Problem #1

인하대학교 INHA UNIVERSITY

입력 예시

```
print("1st test")
kth_largest = Kth_largest()
nums = array.array('h', [4,5,8,2])
k = 3
m = 4
kth_largest.findKthLargest(k, m, nums)
print(kth_largest.add(3)) # 4
print(kth_largest.add(5)) # 5
print(kth_largest.add(10)) # 5
print(kth_largest.add(9)) # 8
print(kth_largest.add(4)) # 8

print("2nd test")
kth_largest2 = Kth_largest()
nums2 = array.array('h', [7, 7, 7, 7, 8, 3])
k2 = 4
m2 = 6
kth_largest2.findKthLargest(k2, m2, nums2)
print(kth_largest2.add(2)) # 7
print(kth_largest2.add(10)) # 7
print(kth_largest2.add(9)) # 7
print(kth_largest2.add(9)) # 8
```

출력 예시

```
1st test
4
5
5
8
8
2nd test
7
7
7
8
```

Hash Table

Tuple vs Structure

인하대학교 INHA UNIVERSITY

Key, Value 를 담을 공간 만들기

- tuple로 만들 시
  - index로 값 접근이 가능
  - 값을 수정할 수 없기에 새로 만들고 대체하는 방식으로 해야 함
- structure(class)로 만들 시
  - 변수명으로 값 접근이 가능
  - 값을 수정할 수 있어서, 수정이 필요하면 간단하게 대처 가능

```
data_tuple = (1, "A")

print(data_tuple[0])  # Output: 1
print(data_tuple[1])  # Output: A
#data_tuple[1] = "B"
print(data_tuple[1])  # Output: A
```

```
class Item:
    def __init__(self, key, value):
        self.key = key
        self.value = value

data_structure = Item(1,"A")

print(data_structure.key)  # Output: 1
print(data_structure.value)  # Output: A
data_structure.value = "B"
print(data_structure.value)  # Output: B
```

Numpy

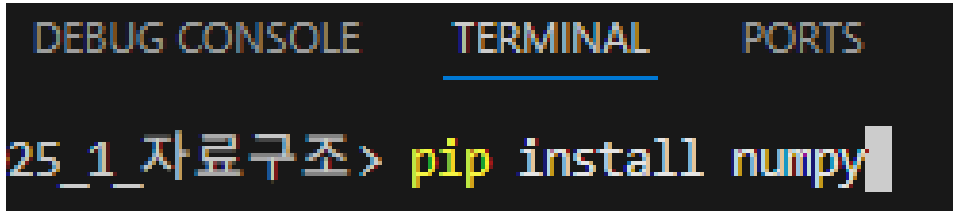
인하대학교 INHA UNIVERSITY

Numpy

- NumPy는 고성능 수치 계산을 위한 파이썬 라이브러리로, 다차원 배열 객체와 벡터화된 연산을 제공

설치

- 명령프롬포트, 혹은 vscode 아래 termial에서 pip install numpy 명령어를 입력해 설치 가능



Numpy 사용

- np.empty(size) 로 사용 가능
- Index 접근은 기존 array와 동일

```
self.table = np.empty(self.size, dtype=object)
for i in range(self.size):
    self.table[i] = None
```

Hash Table

- (key, value)로 이루어진 값들을 저장하는 자료구조
  - key는 unique하다.
- Hash function을 이용해서 효율적으로 값을 관리하는 방식

Hash function

- Hash table에 있는 값을 가리킬 수 있게 만드는 함수
- 보통 나머지 연산을 활용한다.
  - 나머지 연산 시, 소수 혹은 배열의 크기를 사용
  - $h(x) = x \bmod N$

Hash Table

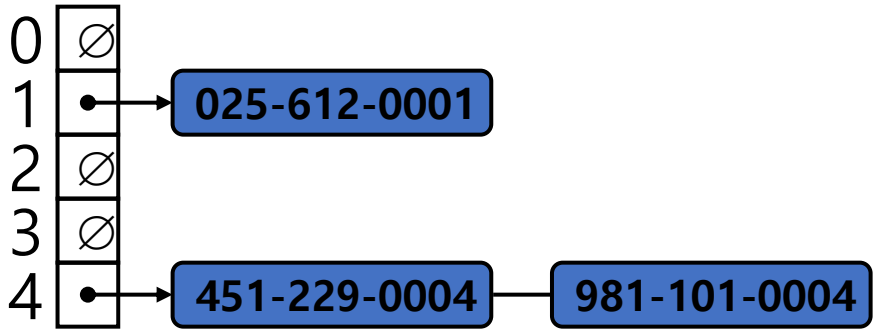
인하대학교 INHA UNIVERSITY

Collisions occur

Hash function으로 나오는 주소로 접근하지만, hash function이 unique한 값을 줄 것이라는 것이 보장되지 않음

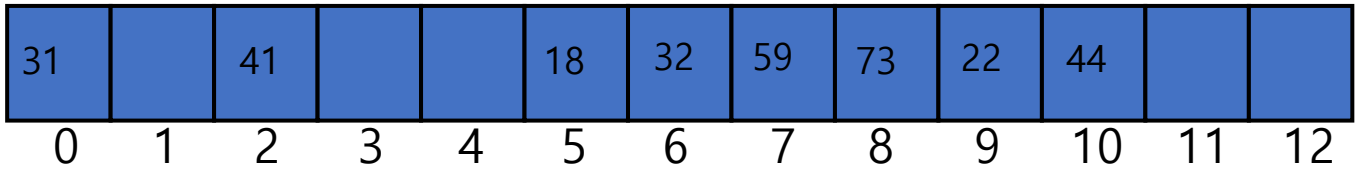
Separate Chaining

동일한 hash 값이 나오면, linked list처럼 연결하며 값을 저장



Linear Probing

동일한 값이 나오면, 그 다음 값으로 이동하면서 값을 저장



Hash Table - chaining

인하대학교
INHA UNIVERSITY

```
class Item:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.next = None

class HashMap:
    def __init__(self):
        self.size = 1000
        self.PrimeNumber = 1000
        self.table = np.empty(self.size, dtype=object)
        for i in range(self.size):
            self.table[i] = None

    def hash(self, key):
        return key % self.PrimeNumber

    def setitem(self, key, value):
        index = self.hash(key)
        current = self.table[index]
        while current is not None:
            if current.key == key:
                current.value = value
                return
            current = current.next
        new_item = Item(key, value)
        new_item.next = self.table[index]
        self.table[index] = new_item
```

Hash Table - chaining

```
def getitem(self, key):
    index = self.hash(key)
    current = self.table[index]

    while current is not None:
        if current.key == key:
            return current.value
        current = current.next
    return None

def delitem(self, key):
    index = self.hash(key)
    current = self.table[index]
    prev = None

    while current is not None:
        if current.key == key:
            if prev is None:
                self.table[index] = current.next
            else:
                prev.next = current.next
            return True
        prev = current
        current = current.next
    return False
```

Hash Table - Probing

```
class Exists:
    NOT_EXISTS = 0
    EXISTS = 1
    AVAILABLE = 2

class Item:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.exists = Exists.NOT_EXISTS

class HashMap:
    def __init__(self):
        self.size = 1000
        self.PrimeNumber = 101
        self.table = np.empty(self.size, dtype=object)
        for i in range(self.size):
            self.table[i] = Item(0,None)


    def hash(self, key):
        return key % self.PrimeNumber
```

Hash Table - Probing


```
def setitem(self, key, value):
    index = self.hash(key)
    new_item = Item(key, value)

    while self.table[index].exists == Exists.EXISTS:
        if self.table[index].key == key:
            self.table[index].value = value
            return
        index = (index + 1) % self.size
    self.table[index] = new_item
    self.table[index].exists = Exists.EXISTS

def getitem(self, key):
    index = self.hash(key)

    while self.table[index].exists != Exists.NOT_EXISTS:
        if self.table[index].key == key:
            if self.table[index].exists == Exists.AVAILABLE:
                return None
            else:
                return self.table[index].value
        index = (index + 1) % self.size
    return None

def delitem(self, key):
    index = self.hash(key)

    while self.table[index].exists != Exists.NOT_EXISTS:
        if self.table[index].key == key:
            self.table[index].exists = Exists.AVAILABLE
            return True
        index = (index + 1) % self.size
    return False
```

Exercise #2

인하대학교 INHA UNIVERSITY

교집합 구하기

크기가 n인 배열과 크기가 m인 배열이 주어진다. 두 배열 사이에 동일한 값이 있는지 판단하는 매소드를 만들어라

시간 복잡도는 O(n + m)을 만족하여라

<힌트>

- Hash table 방식을 쓰면 수월하다

Exercise #2

```python
import numpy as np
import array

class Exists:
    NOT_EXISTS = 0
    EXISTS = 1
    AVAILABLE = 2

class Item:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.exists = Exists.NOT_EXISTS

class HashMap:
    def __init__(self):
        self.size = 1000
        self.PrimeNumber = 101
        self.table = np.empty(self.size, dtype=object)
        for i in range(self.size):
            self.table[i] = Item(0,None)


    def hash(self, key):
        return key % self.PrimeNumber

    def setitem(self, key, value):
        index = self.hash(key)
        new_item = Item(key, value)

        while self.table[index].exists == Exists.EXISTS:
            if self.table[index].key == key:
                self.table[index].value = value
                return
            index = (index + 1) % self.size
        self.table[index] = new_item
        self.table[index].exists = Exists.EXISTS

    def getitem(self, key):
        index = self.hash(key)

        while self.table[index].exists != Exists.NOT_EXISTS:
            if self.table[index].key == key:
                return self.table[index].value
            index = (index + 1) % self.size
        return None
```

인하대학교 INHA UNIVERSITY

Exercise #2


```
def disjoint(array1 : array.array, n1, array2 : array.array, n2):
    hashmap = HashMap()

    for i in range(n1):
        hashmap.setitem(array1[i], array1[i])
    for i in range(n2):
        if hashmap.getitem(array2[i]) is not None:
            return False
    return True
```

Problem #2

인하대학교 INHA UNIVERSITY

문제

길이가 짝수인 정수 배열 arr, 정수 배열의 크기 n 과 정수 k 가 주어졌을 때, **모든 원소를 두 개씩 짝지어 각 쌍의 합이 k로 나누어떨어지도록** 배열을 나눌 수 있는 지를 확인하는 함수를 작성하시오.

- 배열 내 모든 원소는 정확히 한 번만 사용되어야 함
- 각 쌍 (a, b) 에 대해 (a + b) % k == 0을 만족해야 함
- 시간 복잡도는 O(n) 이다.

메소드 설명

- canArray(arr, n, k) : 정수로 구성된 배열 arr 과 배열의 크기 n과 정수 k가 주어진다.
  문제의 조건을 만족하는 쌍으로 배열을 나눌 수 있으면 True를, 아니면 False를 반환한다.

Problem #2

인하대학교 INHA UNIVERSITY

입력 예시

```
c = checker()

n = 4
k = 6
arr = array.array('h', [9, 7, 5, 3])
if c.canArray(arr, n, k):
    print("YES")
else:
    print("NO")
```

출력 예시