자료구조 과제#2

Heap 을 사용한 LFU 시뮬레이터 구현하기

20231765 박종승

1) LFU 시뮬레이터

소스 코드:

```
jaryogujo > 과제#2 heap > lfuSim.py > ...
1   from MinHeap import MinHeap
2
3   def lfu_sim(cache_slots):
4       cache_hit = 0
5       tot_cnt = 0
6       acc = {}
7       heap = MinHeap()
8       data_file = open("C:\\Users\\parkj\\OneDrive\\바탕 화면\\숭실대\\2-1\\자료구조\\jaryogujo\\과제#2 heap\\linkbench.trc")
9
10      for line in data_file.readlines():
11          lpn = line.split()[0]
12          acc[lpn] = acc.get(lpn, 0) + 1
13
14          if heap.count_access(lpn) != 0:
15              cache_hit += 1
16              heap.update_frequency(lpn)
17          else:
18              if heap.size() >= cache_slots:
19                  heap.deleteMin()
20              heap.insert(lpn, acc[lpn])
21          tot_cnt += 1
22
23      print("cache_slot =", cache_slots, "cache_hit =", cache_hit, "hit ratio =", cache_hit / tot_cnt)
24
25  if __name__ == "__main__":
26      for cache_slots in range(100, 1000, 100):
27          lfu_sim(cache_slots)
28
```

실행 결과:

```
gpy\adapter/../../debugpy\launcher   62496   --   C:\Users\pa
cache_slot = 100 cache_hit = 22610 hit ratio = 0.2261
cache_slot = 200 cache_hit = 29375 hit ratio = 0.29375
cache_slot = 300 cache_hit = 33052 hit ratio = 0.33052
cache_slot = 400 cache_hit = 33290 hit ratio = 0.3329
cache_slot = 500 cache_hit = 33440 hit ratio = 0.3344
cache_slot = 600 cache_hit = 33513 hit ratio = 0.33513
cache_slot = 700 cache_hit = 33660 hit ratio = 0.3366
cache_slot = 800 cache_hit = 33820 hit ratio = 0.3382
cache_slot = 900 cache_hit = 33985 hit ratio = 0.33985
PS C:\Users\parkj\OneDrive\바탕 화면\숭실대\2-1\자료구조>
```

우선 제공된 코드에서 페이지 번호의 액세스를 추적하기 위한 딕셔너리인 "acc"를 지정해주고 작성한 최소힙을 불러왔다.

파일에서 각 줄을 읽어오며 lpn을 추출하고 acc에 횟수를 업데이트 해준다. 이때 lpn에 이미 캐시가 존재한다면 캐시히트 값을 증가 후 우선순위를 부여해준다. 캐시가 존재하지 않다면 캐시에 lpn을 추가하고 캐시의 크기를 비교해 가며 전체 액세스 수를 기록하고 최종적으로 캐시 슬롯 수와 캐시 히트 횟수, 히트 비율을 출력해 주었다.

그 결과 출력 값이 답과 동일함을 확인 할 수 있다.

2) 최소힙 (MinHeap)

기존에 강의 자료를 통해 제공받은 코드는 최대힙 코드였기에 최소힙 코드로 코드를 수정 및 필요한 기능을 추가해 주었다.

소스 코드:

```
class MinHeap:
    def __init__(self, *args):
        self.heap = list(args[0]) if args else []

    def insert(self, lpn, priority):
        self.heap.append([lpn, priority])
        self._percolate_up(len(self.heap) - 1)

    def _percolate_up(self, index: int):
        parent = (index - 1) // 2
        if index > 0 and self.heap[index][1] < self.heap[parent][1]:
            self.heap[index], self.heap[parent] = self.heap[parent],
self.heap[index]
            self._percolate_up(parent)

    def deleteMin(self):
        if not self.isEmpty():
            min_val = self.heap[0]
            last_element = self.heap.pop()
            if not self.isEmpty():
                self.heap[0] = last_element
                self._percolate_down(0)
            return min_val
        else:
            return None

    def _percolate_down(self, index: int):
        # 자식과 우선순위를 비교하여 힙 속성을 유지하는 재귀 함수
        left = 2 * index + 1
        right = 2 * index + 2
        smallest = index
        if left < len(self.heap) and self.heap[left][1] <
self.heap[smallest][1]:
            smallest = left
        if right < len(self.heap) and self.heap[right][1] <
self.heap[smallest][1]:
            smallest = right
        if smallest != index:
            self.heap[index], self.heap[smallest] = self.heap[smallest],
self.heap[index]
            self._percolate_down(smallest)
```

```
    def min(self):
        return self.heap[0] if self.heap else None

    def isEmpty(self) -> bool:
        return not self.heap

    def clear(self):
        self.heap.clear()

    def size(self) -> int:
        return len(self.heap)

    def reorder(self, index):
        # 힙 속성을 다시 유지하기 위해 요소 재배열
        parent = (index - 1) // 2
        if index > 0 and self.heap[index][1] < self.heap[parent][1]:
            self._percolate_up(index)
        else:
            self._percolate_down(index)

    def update_frequency(self, lpn):
        # 주어진 lpn 에 대한 우선순위를 업데이트하고 힙 속성을 유지하기 위해 재배열
        for index, item in enumerate(self.heap):
            if item[0] == lpn:
                item[1] += 1
                self.reorder(index)
                return True
        return False

    def count_access(self, lpn) -> int:
        # 주어진 lpn 의 액세스 횟수 반환
        return sum(1 for item in self.heap if item[0] == lpn)
```

Def _percolate_down : 기존의 최대힙과 상반되는 함수로 주어진 인덱스의 노드와 그의 자식 노드를 비교해 자리를 바꾸는 과정을 재귀적으로 수행해 힙의 속성을 유지한다.

주어진 인덱스의 왼쪽과 오른쪽 자식 노드를 계산한다. 더 작은 자식노드를 찾고 그 값이 현재 노드값보다 작다면 두 노드를 교환한다. 이 과정을 재귀적으로 반복해 준다.

Def reorder : 주어진 인덱스 노드를 기준으로 부모 노드보다 작은 값을 가지는지 확인하고 그렇지 않은 경우 교환해주며 힙의 속성을 만족시킨다.

Def update_frequency : 주어진 lpn에 대해서 우선순위를 갱신하는 역할을 한다. 주어진 lpn에 힙이 이미 존재하나 확인하고 존재할시 우선순위를 증가시킨다. 그후 위치를

재배열하여 힙의 속성을 유지한다. 힙이 존재하지 않을시, lpn과 우선순위를 새로 추가하여준다.

Def count_access : 주어진 lpn의 액세스 횟수를 반환하는 역할을 한다. 주어진 lpn에 힙이 존재하나 확인을 하고 힙이 존재할시 lpn의 횟수를 카운트 해준다. 그 후 카운트한 횟수를 반환한다.

2.1) MinHeap이 정확히 동작하는지 검증하기 위해 간단한 검증과정을 거쳤다.

jaryogujo > 과제#2 heap > MinHeap_test.py > ...

```python
from MinHeap import MinHeap

minheap = MinHeap()

# 요소 삽입
minheap.insert('a', 10)
minheap.insert('b', 5)
minheap.insert('c', 8)
minheap.insert('d', 18)


print("최소값", minheap.min())  # 예상 출력: ('b', 5)
print("요소 삭제:", minheap.deleteMin())  # 예상 출력: ('b', 5)
print("삭제후 최소값 확인:", minheap.min())  # 예상 출력: ('c', 8)

```

요소를 삽입하고 삭제하며 그 과정속에서 최솟값을 찾아서 출력하였다.

```
gpy\adapter/../..\debugpy\launcher' '57842' '--' 'c:\Users\parkj\One
'
최소값 ['b', 5]
요소 삭제: ['b', 5]
삭제후 최소값 확인: ['c', 8]
PS C:\Users\parkj\OneDrive\바탕 화면\숭실대\2-1\자료구조>
```

그 결과 예상 출력값과 최소힙을 사용해 구한 값과 동일함을 알 수 있다.