Data Structure – Midterm(Honor Code 제출하였습니다.)

201911013 곽현우

1. Give a recursive algorithm for reversing a singly linked list (present a pseudo-code)

```
def reverse(self):
    bool = True
    if len(self) <= 1: #LList 에 원소가 하나 또는 없을 때 reverse 해도 같으므로 그대로 반환
        return self
    def switch(LList, bool, head=None):
        prev = None
        current = LList._head
        while current != LList._tail:  # 이전의 노드가 무엇인지 찾음
      prev = current
            current = current._next
        current._next = prev
        if bool:
            head = LList._tail
            bool = False
        prev._next = None
        LList._tail = prev
        if LList._head == LList._tail:  # 종료조건
      LList._head = head
        else:
            switch(LList, bool, head)#처음 head 를 할당해준 값을 마지막 loop 에서
활용하기 위해 argument 에 head 값을 새로 넣음
        return LList
    result = switch(self, bool)
    return result
```

2. *Postfix notation* is an unambiguous way of writing an arithmetic expression without parentheses. It is defined so that if ***(exp1) op (exp2)*** is a normal, fully parenthesized expression whose operation is op, the postfix version of this is ***exp1 exp2 op***.

For example, the *postfix* version of **((5+2) ∗ (8−3))/4** is **5 2 + 8 3 − ∗ 4 /**.

Describe a nonrecursive way of evaluating an expression in *postfix* notation, using a stack S.

1) 연산이 표현된 것을 Sequence 라 가정한다.(Expression 이라 부른다고 하자.)

2) 연산자가 나올 때까지 Expression 을 pop()하여 S 에 push 한다.

3) Expression 에서 pop 하여 연산자가 나온다면 S 에서 pop()을 두번 한다. 이때 먼저 pop()된 원소를 opearand2 으로 할당하고 두번째로 pop()된 원소를 operand1 으로 순서를 바꿔서 할당한다. 이후 operand1 operator operand 2 순으로 연산하고 연산된 결과값을 다시 S 에 push 한다.

4) 1),2),3)과정을 반복하고 S 에 아무런 값도 없으면 연산이 완료되었다는 뜻이므로 마지막에 S 에서 pop()된 원소를 반환한다.

문제에서 주어진 예시를 통해 설명하면 다음과 같은 과정을 거친다.

- S = [2, 5], Expression.pop() = +
- S = [3, 8, 7] Expression.pop() = -
- S = [5, 7] Expression.pop() = *
- S = [4, 35] Expression.pop() = /
- S = [8.75]
- 8.75 is returned

3. Using an array-based heap representation, fill in the items in the array after the following priority-queue operations, one-by-one.

1. `add(4) -> [ 4 ]`
2. `add(9) -> [ 4 , 9 ]`
3. `add(1) -> [ 1 , 9 , 4 ]`
4. `add(5) -> [ 1 , 5 , 4 , 9 ]`
5. `remove_min -> [ 4 , 5 , 9 ] / ( 1 ) is returned.`
6. `add(6) -> [ 4 , 5 , 9 , 6 ]`
7. `add(3) -> [ 3 , 4 , 9 , 6 , 5 ]`
8. `add(7) -> [ 3 , 4 , 7 , 6 , 5 , 9 ]`
9. `remove_min -> [ 4 , 5 , 7 , 6 , 9 ] / ( 3 ) is returned.`
10. `remove_min -> [ 5 , 6 , 7 , 9 ] / ( 4 ) is returned.`
11. `remove_min -> [ 6 , 9 , 7 ] / ( 5 ) is returned.`

4. Skip List (20 points)

On `skiplist.py`, fully implement a skip list that completes `MutableMapping` ADT. Specifically, fill in the `__getitem__`, `__setitem__`, and `__delitem__`. Feel free to add supplimentatry methods if needed.

With your implementation, report the following questions.

```
def __getitem__(self, k, update=None):
    """Return value associated with key k (raise KeyError if not found).--
Search"""
    if k == -math.inf:
        return self._head._value
    find = self._head
    i = self._height
    while i != 0:
        if find._next[i - 1]._key == k:
            break
        while find._next[i - 1]._key < k:
            find = find._next[i - 1]
        i -= 1
    if i == 0:
        if find._next[i]._key != k:
            if update == None:
                raise KeyError("There is no item with key k in this SkipList")
            else:
                return update
```

```
        else:
            return find._next[i]._value
    return find._next[i - 1]._value
def Prev_nodes(self, k):  # k 값보다 작은값 중에 최대의 key 값을 가지고 있는 node 들을 list 형태로 반환
  result = [None] * self._height
    find = self._head
    i = self._height
    while i != 0:
        while find._next[i - 1]._key < k:
            find = find._next[i - 1]
        result[i - 1] = find
        i -= 1
    return result


def __setitem__(self, k, v):
    """Assign value v to key k, overwriting existing value if present.--Insert"""
    tower_height = 1
    while random.randint(1, 2) != 2:
        tower_height += 1
    node = self._head
    while node != None:
        t = tower_height + 1
        while t > self._height:
            node._next.append(None)
            t -= 1
        node = node._next[0]

    h = max(self._height, tower_height + 1)
    new_node = self._Node(k, v, h)
    position = self.Prev_nodes(k)  # type: List
    if position[0]._next[0]._key == k:
        position[0]._next[0]._value = v
    else:
        if tower_height >= self._height:
            for i in range(self._height):
                new_node._next[i] = position[i]._next[i]
                position[i]._next[i] = new_node
            for i in range(self._height - 1, tower_height):
                self._head._next[i] = new_node
                new_node._next[i] = self._tail
            self._head._next[tower_height] = self._tail
            self._height = h
        else:
            for i in range(tower_height):
                new_node._next[i] = position[i]._next[i]
                position[i]._next[i] = new_node
        self._n += 1
```

```
def __delitem__(self, k):
    """Remove item associated with key k (raise KeyError if not found).--
Delete"""
    position = self.Prev_nodes(k)
    if position[0]._next[0]._key != k:  # raise Error
        raise KeyError("There is no item with key k in this SkipList")
    else:
        for i in range(len(position)):  # item을 del 하는 과정
      if position[i]._next[i]._key == k:
                del_node = position[i]._next[i]
                position[i]._next[i] = del_node._next[i]
        while self._height != 1 and self._head._next[self._height - 2] ==
self._tail:  # 원소를 del 하고 높이를 하나씩 줄여가는 과정
      node = self._head
            while node != None:
                node._next.remove(node._next[self._height - 1])
                node = node._next[0]
            self._height -= 1
        self._n -= 1
```

1. Analyze your logic in terms of time complexity. get, set, del functions should be run in O(logn) time.

*skiplist 의 높이는 O(logN)만큼의 저장공간을 가진다. Skiplist 의 높이를 h 라 할 때 0 층의 n 개의 item 들이 h 층까지 올라올 확률은 $\frac{1}{2^h}$이다.(0 층부터 시작했다고 하자.) 즉 h 에 있는 item 의 개수는 $\frac{n}{2^h}$이다. 이때 확률적으로 h 의 최대값이 klogn 일 확률은 1 - $\frac{1}{n^{k-1}}$이므로 매우 크다. 따라서 높이는 O(logN)만큼의 저장공간을 가진다고 할 수 있다.

*Prev_nodes: getitem 이 해당 key 값을 가지는 item 의 value 를 반환한다. 또 SkipList 각 층이 Singly Linked-List 형태로 구현되어 있어서 setitem 이나 delitem 을 수행할 때 해당 key 값을 넣거나 제거하기 위해서는 각 층마다 그 이전 노드들이 무엇이었는지 찾아줄 필요가 있다. 따라서 Prev_nodes method 는 key 값이 k 인 노드에 대해 각 층마다 k 값보다 작은 노드들을 찾아서 List 형태로 반환한다. SkipList 의 item 개수가 n 개라 할 때 SkipList 의 높이는 확률상 O(logN)이다. 이때 Prev_nodes 는 SkipList 의 LeftMostTower, 즉 Head 부터 시작하여 Node_next[i]._key 값이 k 보다 작은 노드들을 하나씩 찾으므로 실행시간은 SkipList 의 높이와 비례한다. 따라서 Prev._nodes 의 실행시간은 O(logn)이다.

1)__getitem__: n 개의 item 이 skiplist 에 있다고 할 때 getitem 의 worst case 는 node 의 tower_height 가 1 이고 node._next[0] == self.tail 인 node 를 찾는 것이다. getitem 의 실행방식은 위에서 언급한 Prev._node 와 유사하지만 getitem 은 각 층마다 k 보다 작은 key 값을 가지는 node 를 찾아 list 의 형태로 저장하지 않는다는 차이가 있다. 따라서 worst case 의 경우 skiplist 의 높이만큼 탐색을 해야하므로 마찬가지로 실행시간은 skiplist 의 높이와 비례한다. 따라서 getitem 의 실행시간도 O(logn)이다.

2)__setitem__: 우선 새로 넣어줄 노드의 tower._height = t 라 하자. setitem 에서 worst case 는 t 값이 기존의 self._height 보다 클 때이다. setitem 의 실행시간을 계산하기 위하여 몇가지 단계로 쪼개어 설명해보면 다음과 같다.

-1. 50% 확률로 tower._height 증가시키기: 이 경우 실행시간은 t 에 비례한다. t 는 전체 실행시간에 영향을 미치지 못한다.

-2. (t + 1) – self._height 만큼 각 노드의 _next 리스트에 None 값을 추가하기: 이 경우 기존의 skiplist 내에 모든 원소에 대하여 실행되어야 함으로 O(N)의 시간복잡도를 가지지만 t + 1 이 self._height 보다 높아질 확률이 $\frac{1}{n^{k-1}}$(h = klogn 의 꼴을 가짐)이므로 전체적인 setitem 의 실행시간을 계산할 때는 무시해도 된다.

-3. Prev._node method 를 진행시킨다.: O(logN)만큼 실행된다.

-4. 새로운 key 값을 가지는 노드를 기존의 노드들과 연결시켜준다.: 새로 만들어진 노드의 tower._height 와 실행시간이 비례하지만 이때 tower._height 는 상수 취급되어 setitem 실행시간 분석할 때 무시해도 된다.

1,2,3,4 단계를 모두 고려한다면 setitem 의 최종적인 시간복잡도는 O(logN)이다.

3)__delitem__: 기존의 찾으려는 노드의 tower._height 를 t 라 하자. 다음의 단계로 시간복잡도를 분석할 수 있다.

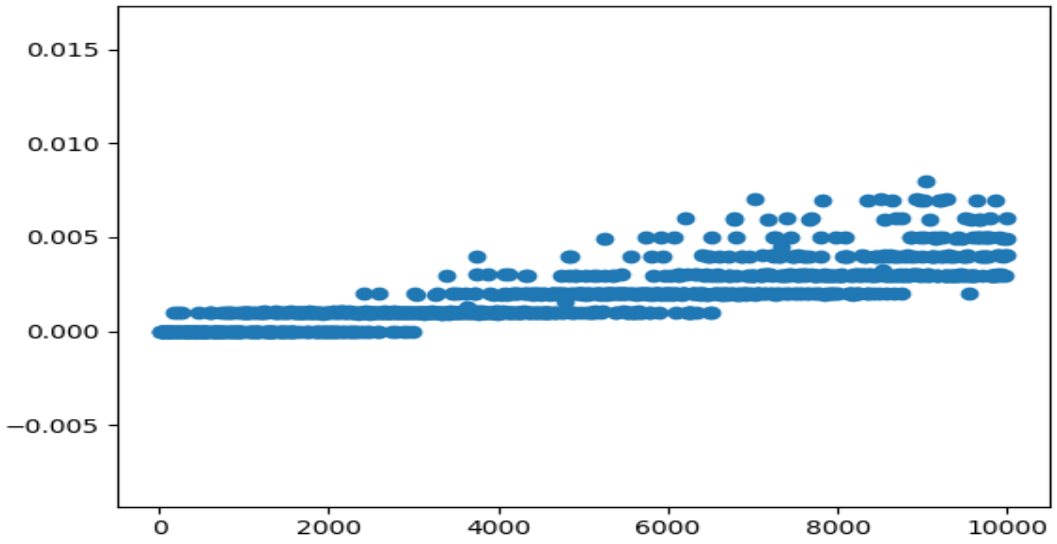
-1. Prev._node method 를 진행시킨다.: O(logN)만큼 실행된다.

-2. t 만큼 찾으려는 노드를 삭제시키기 위해 node 의 연결을 수정한다.: t 값과 실행시간이 비례하지만 t 는 마찬가지로 상수이므로 전체 delitem 의 실행시간을 결정할 때에는 무시가능하다.

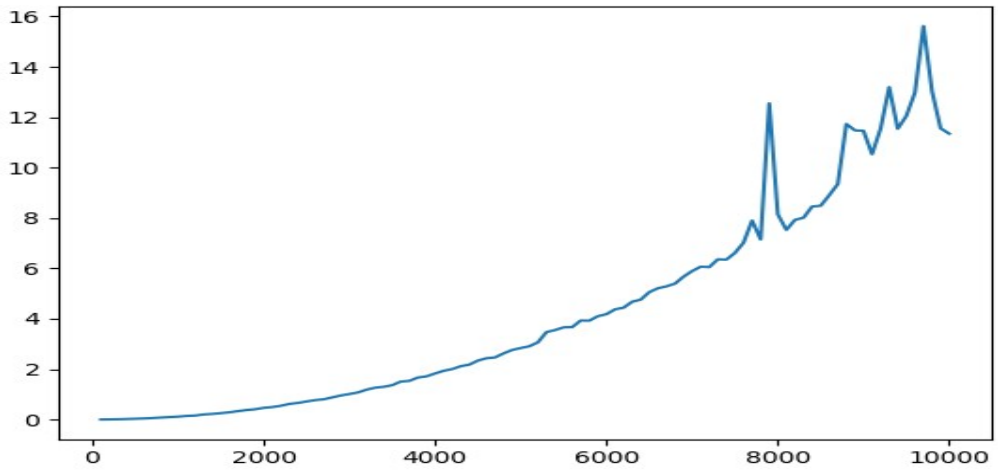
-3. 만약 제거하려는 노드가 skiplist 내에서 t 값이 가장 큰 node 라면 node 삭제이후 바뀐 skiplist 의 높이만큼 원소가 없는 층을 없앤다. : wosrt case 는 가장 높이가 높은 node 를 지우고 난 후 노드가 없거나 남은 노드들 중에서 가장 높은 node 의 높이가 1 인 상태이다. 이때 점점 item 개수가 추가되어 높이가 높아진 상태에서 delitem 을 수행할 때 (높이를 h = klogN 라고 함) h – 1 층에 원소가 있을 확률은 $\frac{1}{n^{k-1}}$에 근사하므로 매우 작다. 따라서 전체적인 실행시간을 분석할 때 무시할 수 있다.

1,2,3 단계를 모두 고려한다면 delitem 의 최종적인 시간복잡도는 O(logN)이다.

2. Experimentally show your `__setitem__` runs in O(logn)time. For example, measure the elapsed time of adding 100, 200, 300, ... , 10000 random items to your skiplist. The tested numbers may vary depend on your computer's performance. Plot them, with x-axis set to the problem size and y-axis set to the elapsed time. The trend should follow logn shape.



위 그림은 skiplist 에 100 번째 200 번째 ...10000 번째 원소를 추가하였을 때 실행시간을 나타난 그림이다. 그림을 통해 setitem 의 실행시간이 O(logN)임을 알 수 있다.



위 그림은 skiplist 에 100 개 200 개 ...10000 개 원소를 새로운 skiplist 에 대해 추가하였을 때 실행시간을 나타난 그림이다. 이 경우 그래프 개형이 O(NlogN)임을 알 수 있다.