**Data Structure – Midterm(Honor Code제출하였습니다.)**

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: #LList에 원소가 하나 또는 없을 때 reverse해도 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 head를 할당해준 값을 마지막 loop에서 활용하기 위해 argument에 head값을 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 Sequence라 가정한다.(Expression이라 부른다고 하자.)

2) 연산자가 나올 때까지 Expression을 pop()하여 S에 push한다.

3) Expression에서 pop하여 연산자가 나온다면 S에서pop()을 두번 한다. 이때 먼저 pop()된 원소를 opearand2으로 할당하고 두번째로 pop()된 원소를 operand1으로 순서를 바꿔서 할당한다. 이후 operand1 operator operand 2 순으로 연산하고 연산된 결과값을 다시 S에 push한다.

4) 1),2),3)과정을 반복하고 S에 아무런 값도 없으면 연산이 완료되었다는 뜻이므로 마지막에 S에서 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): # k값보다 작은값 중에 최대의 key값을 가지고 있는 node들을 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 del하는 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 del하고 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.

\*skiplist의 높이는 O(logN)만큼의 저장공간을 가진다. Skiplist의 높이를 h라 할 때 0층의 n개의 item들이 h층까지 올라올 확률은 이다.(0층부터 시작했다고 하자.) 즉 h에 있는 item의 개수는 이다. 이때 확률적으로 h의 최대값이 klogn일 확률은 1 - 이므로 매우 크다. 따라서 높이는 O(logN)만큼의 저장공간을 가진다고 할 수 있다.

\*Prev\_nodes: getitem이 해당 key값을 가지는 item의 value를 반환한다. 또 SkipList 각 층이 Singly Linked-List 형태로 구현되어 있어서 setitem이나 delitem을 수행할 때 해당 key값을 넣거나 제거하기 위해서는 각 층마다 그 이전 노드들이 무엇이었는지 찾아줄 필요가 있다. 따라서 Prev\_nodes method는 key값이 k인 노드에 대해 각 층마다 k값보다 작은 노드들을 찾아서 List형태로 반환한다. SkipList의 item개수가 n개라 할 때 SkipList의 높이는 확률상 O(logN)이다. 이때 Prev\_nodes는 SkipList의 LeftMostTower, 즉 Head부터 시작하여 Node\_next[i].\_key값이 k보다 작은 노드들을 하나씩 찾으므로 실행시간은 SkipList의 높이와 비례한다. 따라서 Prev.\_nodes의 실행시간은 O(logn)이다.

**1)\_\_getitem\_\_**: n개의 item이 skiplist에 있다고 할 때 getitem의 worst case는 node의 tower\_height가 1이고node.\_next[0] == self.tail인 node를 찾는 것이다. getitem의 실행방식은 위에서 언급한 Prev.\_node와 유사하지만 getitem은 각 층마다 k보다 작은 key값을 가지는 node를 찾아 list의 형태로 저장하지 않는다는 차이가 있다. 따라서 worst case의 경우 skiplist의 높이 만큼 탐색을 해야하므로 마찬가지로 실행시간은 skiplist의 높이와 비례한다. 따라서 getitem의 실행시간도 O(logn)이다.

**2)\_\_setitem\_\_**: 우선 새로 넣어줄 노드의 tower.\_height = t라 하자. setitem에서 worst case는 t값이 기존의 self.\_height보다 클 때이다. setitem의 실행시간을 계산하기 위하여 몇가지 단계로 쪼개어 설명해보면 다음과 같다.

-1. 50% 확률로 tower.\_height 증가시키기: 이 경우 실행시간은 t에 비례한다. t는 전체 실행시간에 영향을 미치지 못한다.

-2. (t + 1) – self.\_height 만큼 각 노드의 \_next 리스트에 None값을 추가하기: 이 경우 기존의 skiplist내에 모든 원소에 대하여 실행되어야 함으로 O(N)의 시간복잡도를 가지지만 t + 1이 self.\_height보다 높아질 확률이 (h = klogn의 꼴을 가짐)이므로 전체적인 setitem의 실행시간을 계산할 때는 무시해도 된다.

-3. Prev.\_node method를 진행시킨다.: O(logN)만큼 실행된다.

-4. 새로운 key값을 가지는 노드를 기존의 노드들과 연결시켜준다.: 새로 만들어진 노드의 tower.\_height와 실행시간이 비례하지만 이때 tower.\_height는 상수 취급되어 setitem 실행시간 분석할 때 무시해도 된다.

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

**3)\_\_delitem\_\_**: 기존의 찾으려는 노드의 tower.\_height를 t라 하자. 다음의 단계로 시간복잡도를 분석할 수 있다.

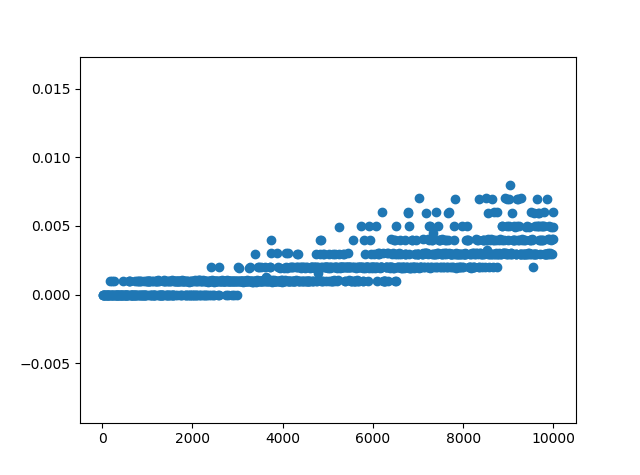
-1. Prev.\_node method를 진행시킨다.: O(logN)만큼 실행된다.

-2. t만큼 찾으려는 노드를 삭제시키기 위해 node의 연결을 수정한다.: t값과 실행시간이 비례하지만 t는 마찬가지로 상수이므로 전체 delitem의 실행시간을 결정할 때에는 무시가능하다.

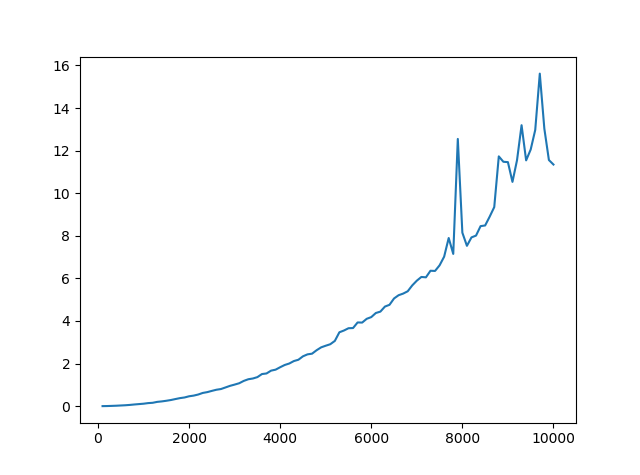
-3. 만약 제거하려는 노드가 skiplist내에서 t값이 가장 큰 node라면 node삭제이후 바뀐 skiplist의 높이만큼 원소가 없는 층을 없앤다. : wosrt case는 가장 높이가 높은 node를 지우고 난 후 노드가 없거나 남은 노드들 중에서 가장 높은 node의 높이가 1인 상태이다. 이때 점점 item개수가 추가되어 높이가 높아진 상태에서 delitem을 수행할 때 (높이를 h = klogN라고 함) h – 1층에 원소가 있을 확률은 에 근사하므로 매우 작다. 따라서 전체적인 실행시간을 분석할 때 무시할 수 있다.

1,2,3 단계를 모두 고려한다면 delitem의 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 skiplist에 100번째 200번째 …10000번째 원소를 추가하였을 때 실행시간을 나타난 그림이다. 그림을 통해 setitem의 실행시간이 O(logN)임을 알 수 있다.



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