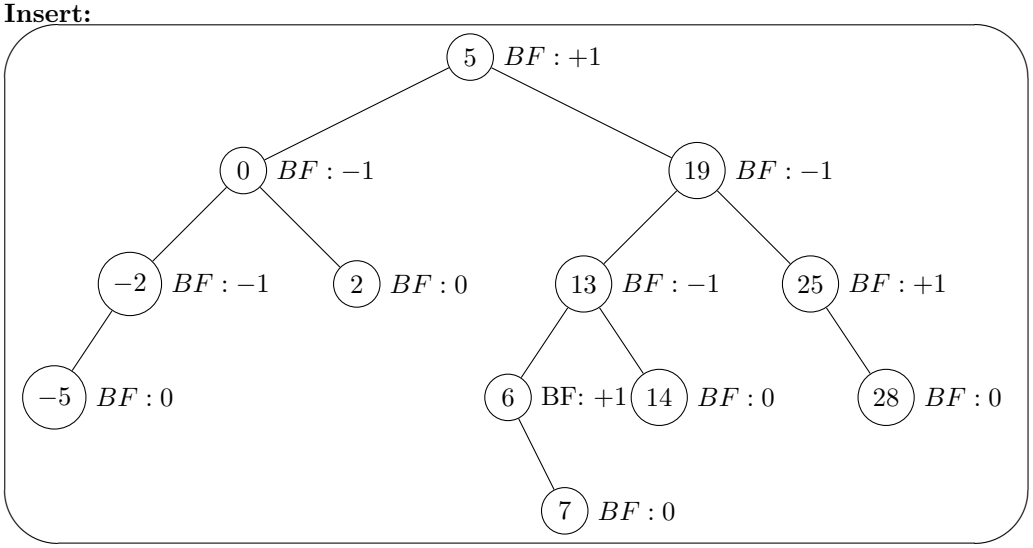
CSC263H1 Assignment 3

Jiatao Xiang, Xu Wang, Huakun Shen

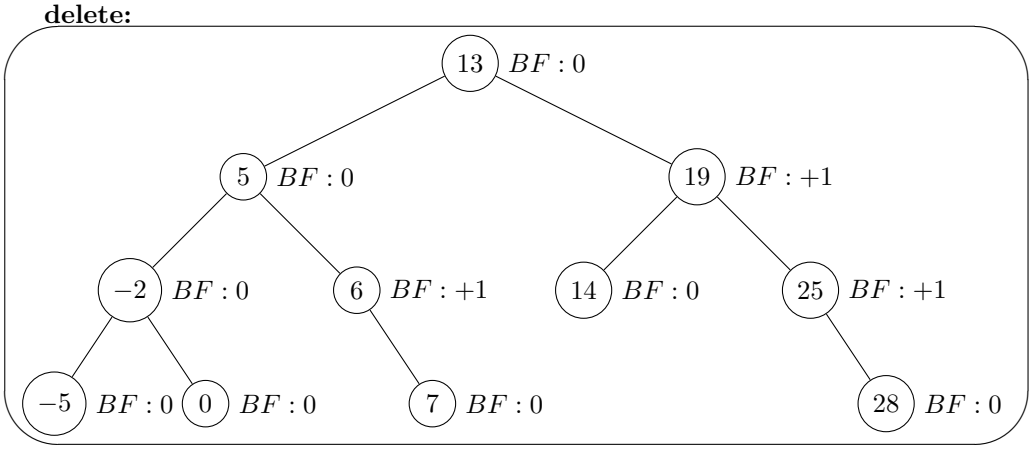
February 14th, 2019

Question 1

Insert:



delete:



Question 2

a.

b. Description:

We add one more AVL tree such that each of its node contains price (which is also the key), the pointer to points its left child, the pointer points to its right child, and the max_rating that stores the maximum rating among all nodes in the subtree rooted at the current node.

We use recursive algorithm to achieve the operation. The base case happens when D is leaf, thus, it has only two conditions to discuss. First Case: when D's price is less than or equal to input p, the max_rating is the rating of D, thus, D.rating is returned. Second Case: when D's price is greater than input p, -1 is returned.

We start with the root of D, there exist three conditions. First, when D has only right children: we check if $D.price \leq p$, we need to check the max_rating of right subtree and D's rating, and return the greater one. If $D.price > p$, -1 is returned, because there is no node satisfies the requirement.

```
def BestBookRating(D, p):
  if D is leaf:
    if D.price <= p:
      return D.rating
    return -1
  elif D only has right child:
    if D.price <= p:
      return max(BestBookRating(D.right, p), D.rating)
    return -1
  elif D only has left child:
    if D.price <= p:
      return max(D.left.max_rating, D.rating)
    return BestBookRating(D.left, p)
  elif D has both left and right child:
    if D.price > p:
      return BestBookRating(D.left, p)
    return max(BestBookRating(D.left, p), BestBookRating(D.right, p), D.rating)
```

c.

d.

e.

Question 3

a. Our data structure is based on **hash table**.

Idea: Put every element of set **B** into a hash table. Then hash every element of set **A** to a slot in the hash table, and check whether the element of **A** is in this slot. If it is not in the slot, then it means that the element of set **A** is not in set **B**.

Pseudo Code:

Suppose $\alpha = \mathbf{10}$, that is, each slot contains a linked list with size of at most approximately 10 elements. Suppose we have a hash table $\boldsymbol{T}$ with a size of $\frac{n}{\alpha}$.

Suppose we have a hashing function ***h(x)*** that would return the index of one of the slots of **T** given $\boldsymbol{x}$ as a input. Also assume **SUHA**.

```
def h(x, num_slot):
  return x % num_slot
```

```
for element in B:
    linked_list = T[h(element, len(T))]
    linked_list.append(element)

result = []
for element in A:
    linked_list = T[h(element, len(T))]
    for item in linked_list:
        if element == item:
            break
    result.append(element)
```

b. Assumptions:

- The linked list in each slot of hash table has approximately a length of $\alpha = 10$
- Hash table **T** has a length of $\frac{len(A)}{\alpha}$
- SUHA for hash function ***h(x, num_slot)***

Explanation: Part I: put **B** into **T**

1. Hashing function costs constant time

2. There are n elements in **B**, performing hashing function $\boldsymbol{h}$ for each element of **B** costs $\mathcal{O}(n \times 1)$ of time.

Part II: match element of **A** to **T**

1. Each linked list in each slot of **T** has a size of at most α (by **SUHA**), which is constant. In the worst case, we have to traverse through every linked list, which costs $\mathcal{O}(\alpha) = \mathcal{O}(1)$ of time.
2. There are n elements in **A**, performing step 1 for each of them costs $\mathcal{O}(\alpha n) = \mathcal{O}(n)$ of time.

Part I and Part II altogether cost $\mathcal{O}(n)$ of time.

c. In The worst case scenario, **SUHA** may not hold, and every element of B may be hashed into one single slot of **T**. Then when element of set **A** tries to find a matching element in this slot, it needs to traverse through the entire linked list and takes $\mathcal{O}(n)$ of time.
While there are n elements in set **A**. If for each of the element in **A**, the program traverses through the linked list of size n and do not exit early, it will take $\mathcal{O}(n \times n) = \mathcal{O}(n^2)$ of time.