CSCB63 Assignment 1

Zheyuan Wei

February 13, 2023

Q1

Q1-a

Q: *If* $f(n) \in \mathcal{O}(g)$ *and* $g \in \mathcal{O}(h)$ *then* $f \in \mathcal{O}(h)$, *for all* f, g, h *in* $\mathbb{N} \to \mathbb{R}^+$

Proof.

1) From $f \in$ O(g) *we have* :

$$\exists c_1, n_1 \in \mathbb{R} \text{ s.t. } f(n) \leq c_1 \cdot g(n) \forall n \geq n_1$$

2) From $g \in$ O(h) *we have* :

$$\exists c_2, n_2 \in \mathbb{R} \text{ s.t. } g(n) \leq c_2 \cdot h(n) \forall n \geq n_2$$

3) From 1) and 2) we have:

$$f(x) = \begin{cases} f(n) \leq c_1 \cdot g(n) \forall n \geq n_0, \text{for some } c_1 \text{ and } n_1 \\ g(n) \leq c_2 \cdot h(n) \forall n \geq n_1, \text{for some } c_2 \text{ and } n_2 \end{cases} \Rightarrow f(n) \leq c_1(c_2 \cdot h(n)) \forall s.t. \begin{cases} n \geq n_1 \\ n \geq n_2 \end{cases}$$

4) From 3) we have:

$$f(n) \leq c_1 \cdot c_2 \cdot h(n) \ \forall n \geq max(n_1, n_2)$$

$$\Rightarrow f \in \mathcal{O}(h)$$

□

Q1-b

Q: *If* $f \in \Omega(g)$ *and* $g \in \Omega(h)$ *then* $f \in \Omega(h)$, *for all* f, g, h *in* $\mathbb{N} \to \mathbb{R}^+$

Proof.

1) From $f \in \Omega(g)$ we have:

$$\exists c_1, n_1 \in \mathbb{R} \text{ s.t. } f(n) \geq c_1 \cdot g(n) \forall n \geq n_1$$

2) From $g \in \Omega(h)$ we have:

$$\exists c_2, n_2 \in \mathbb{R} \text{ s.t. } g(n) \geq c_2 \cdot h(n) \forall n \geq n_2$$

3) From 1) and 2) we have:

$$f(x) = \begin{cases} f(n) \geq c_1 \cdot g(n) \forall n \geq n_0, \text{for some } c_1 \text{ and } n_1 \\ g(n) \geq c_2 \cdot h(n) \forall n \geq n_1, \text{for some } c_2 \text{ and } n_2 \end{cases} \Rightarrow f(n) \geq c_1(c_2 \cdot h(n)) \forall s.t. \begin{cases} n \geq n_0 \\ n \geq n_1 \end{cases}$$

4) From 3) we have:

$$f(n) \geq c_1 \cdot c_2 \cdot h(n) \ \forall n \geq max(n_0, n_1)$$

$$\Rightarrow f \in \Omega(h)$$

□

Q1-c

Q: $\log_\phi(\sqrt{5}(n+2)) - 2 \in \mathcal{O}(\log_2(n))$ where ϕ is the golden ratio.

Proof. Let $f(n) = \log_\phi(\sqrt{5}(n+2)) - 2, \forall n \geq 0$

Then it follows that:

$$f(n) = \log_\phi(\sqrt{5}(n+2)) - 2 = \frac{\log_2(\sqrt{5}(n+2))}{\log_2(\phi)} - 2$$

$$f(n) = \frac{\log_\phi \sqrt{5}}{\log_2(\phi)} \cdot \log_2(n+2) - 2$$

$$f(n) < \frac{\log_\phi \sqrt{5}}{\log_2(\phi)} \cdot \log_2(n+2) = k \cdot \log_2(n+2) \ \forall n \geq 0$$

where $k = \frac{\log_\phi \sqrt{5}}{\log_2(\phi)} \in \mathbb{R}$

$$\Rightarrow f(n) < k \cdot \log_2(n+2) \leq k \cdot \log_2(n^2) = k \cdot 2 \cdot \log_2(n) \ \forall n \geq 2$$

which is equivalent to:

$$f(n) < 2k \cdot \log_2(n) \ \forall n \geq 2$$

$$\Rightarrow f(n) \in \mathcal{O}(\log_2(n))$$

□

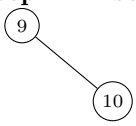
Q2

Q2-a

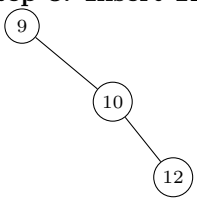
Q: On an initially empty tree, show each step of inserting the keys 9, 10, 12, 14, 3, 34, 19, 37, 20.

Step 1: Insert 9

Step 2: Insert 10



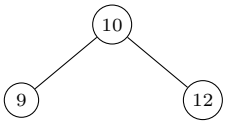
Step 3: Insert 12



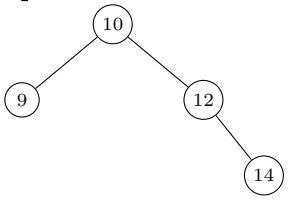
Imbalanced. Right heavy at node 9.

Need a left rotation.

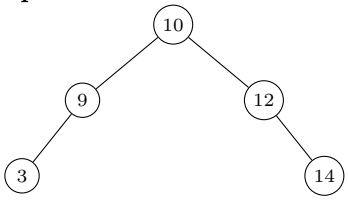
Left rotation:



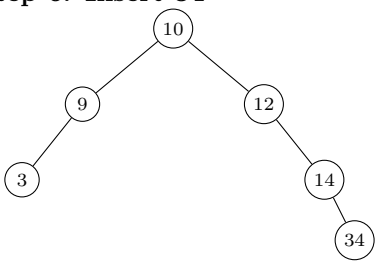
Step 4: Insert 14



Step 5: Insert 3



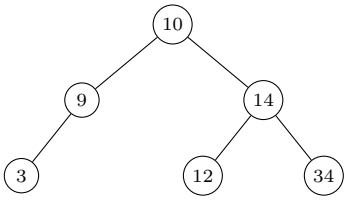
Step 6: Insert 34



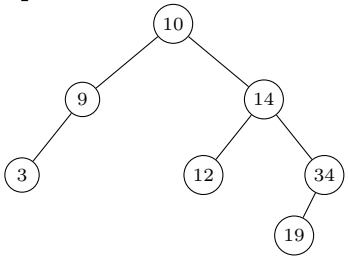
Imbalanced. Right heavy at node 12.

Need a left rotation.

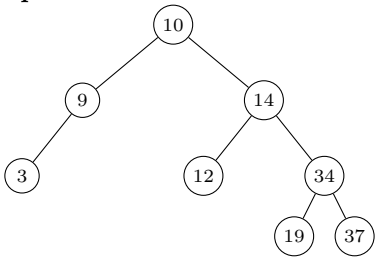
Left rotation:



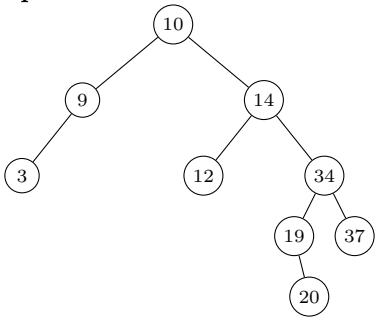
Step 7: Insert 19



Step 8: Insert 37



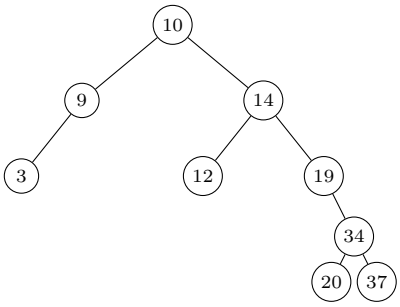
Step 9: Insert 20



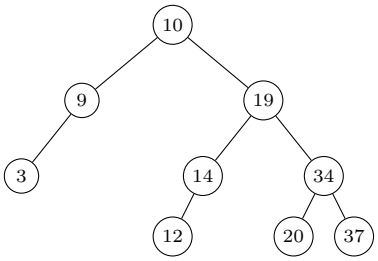
Imbalanced. Right heavy at node 14.

Need a double rotation.

1. Right rotation:



2. Left rotation:

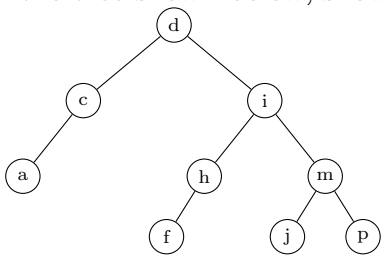


■

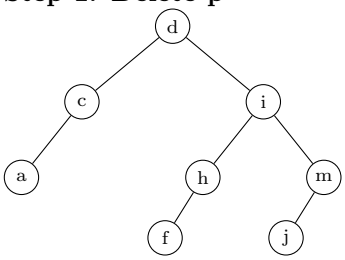
Q2-b-1

Q:

On the tree shown below, show each step of deleting the keys p, d, h.



Step 1: Delete p

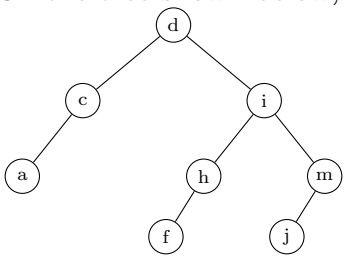


■

Q2-b-2

Q:

On the tree shown below, show each step of deleting the keys p, d, h.

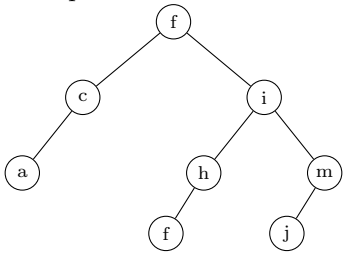


Step 2: Delete d

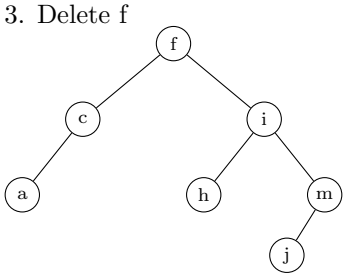
1. Find the successor of d:

   Which is the leftmost node in the right subtree of d, which is f.

2. Replace d with f



3. Delete f

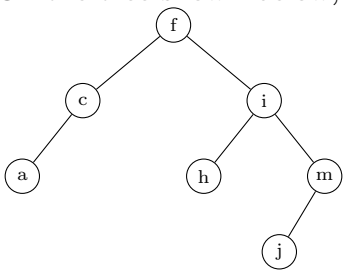


■

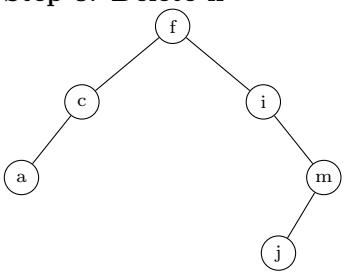
Q2-b-3

Q:

On the tree shown below, show each step of deleting the keys p, d, h.



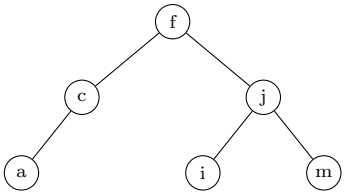
Step 3: Delete h



Imbalanced. Right heavy at node i.

Need a left rotation.

Left rotation:

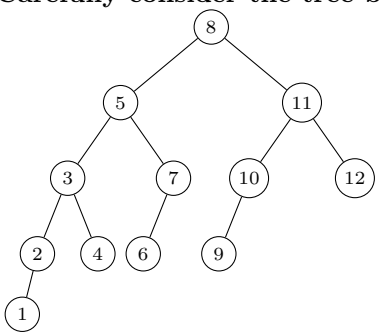


■

Q2-c

Q:

Carefully consider the tree below.



- If we start with an empty AVL tree, what sequence of insertions would result in this tree?

- Show each step of deleting key 11.

1. Insertion Sequence: (one possible sequence)

- Insert 8
- Insert 5
- Insert 11
- Insert 3
- Insert 7
- Insert 10
- Insert 12
- Insert 2
- Insert 4
- Insert 6
- Insert 9
- Insert 1

The only constraint is the key 1 must be inserted last. ■

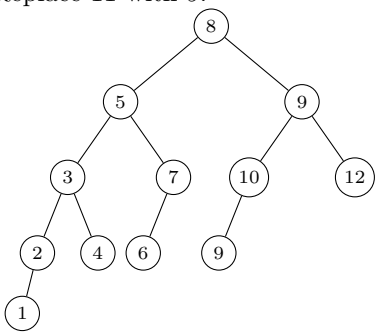
2. Delete key 11:

Step 1: Find the seccessor of 11.

The leftmost node in the right subtree of root, i.e. 9.

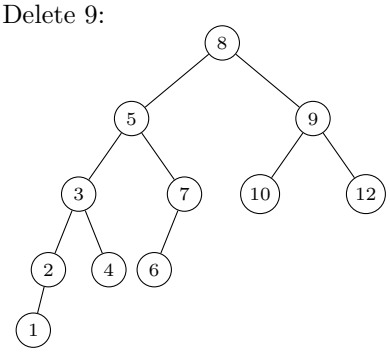
Step 2: Replace 11 with 9.

Replace 11 with 9:



Step 3: Delete 9.

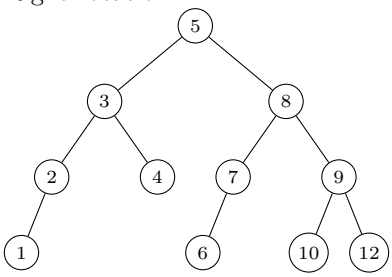
Delete 9:



Imbalanced. Left heavy at node 8.

Need a right rotation.

Right rotation:



■

Q4

Q: Consider an ADT consisting of a set S of distinct integers and the following operations:

- search(S, x): Return true if x is in S and false otherwise.
- insert(S, x): Insert the element x into the set S. This operation has no effect if x is already in S.
- delete(S, x): Delete the element x from the set S. This operation has no effect if x is not in S.
- min_difference(S): Given a set S with size of at least 2, return a pair of distinct integers (x, y), with x $\in$ S, y $\in$ S, with the minimum absolute difference, i.e.
  $\forall x', \forall y', (x' \neq y' \rightarrow |x - y| \leq |x' - y'|)$

Your task is to design a data structure to implement this ADT,
such that all operations are performed in $\mathcal{O}(\log\ n)$ time,
where $n = |S|$. You will do so by augmenting our familiar AVL tree.

1. Describe all information that will be stored in the nodes.
2. Provide pseudo-code for each required operation.
3. Justify why your algorithms are correct and why they achieve the required time bound.

Augmented AVL Tree

1. Describe all information that will be stored in the nodes.

- int key;
- void *data;
- int height;
- node *left;
- node *right;
- int relative_min_diff; // augmented information
- tuple (x, y) min_pair; // augmented information

*Note that here 2 augmented information are introduced:

- relative_min_diff: the min difference in the subtree at the node.
- min_pair: the pair of integers with the minimum difference in the subtree at the node.

2. Provide pseudo-code for each required operation.

- search(S, x): Return true if x is in S and false otherwise.

```
return BST_search(S, x);
// Where BST_search is the search function of a BST
// See below:
def BST_search(S, x):
    if S == NULL:
        return False
    if x == S.key:
        return True
    else:
        if x < S.key:
            return BST_search(S.left, x)
        else:
            return BST_search(S.right, x)
```

- insert(S, x): Insert the element x into the set S. This operation has no effect if x is already in S.

```
def insert(S, x):
    if S == NULL:
        S = new node
        S.key = x
        S.left = NULL
        S.right = NULL
        return S
    if x == S.key:
        return S
    else:
        if x < S.key:
            S.left = insert(S.left, x)
        else:
            S.right = insert(S.right, x)

    update_height(S) // The same as AVL tree
    S = balance(S) // The same as AVL tree
    update_min_diff(S) // See helper functions below
    return S
```

- delete(S, x): Delete the element x from the set S. This operation has no effect if x is not in S.

```
def delete(S, x):
    if S == NULL:
        return S
    if x == S.key:
        if S.left == NULL and S.right == NULL:
            return NULL
        if S.left == NULL:
            return S.right
        if S.right == NULL:
            return S.left
        else:
            S.key = find_min(S.right)
            S.right = delete(S.right, S.key)
    else:
        if x < S.key:
            S.left = delete(S.left, x)
        else:
            S.right = delete(S.right, x)

    update_height(S) // The same as AVL tree
    S = balance(S) // The same as AVL tree
    update_min_diff(S) // See helper functions below
    return S
```

- min_difference(S): Given a set S with size of at least 2, return a pair of distinct integers (x, y), with x ∈ S, y ∈ S, with the minimum absolute difference, i.e.
  $\forall x', \forall y', (x' \neq y' \rightarrow |x - y| \leq |x' - y'|)$

```
def min_difference(S):
    return S.min_pair
```

Helper functions

- update_height(S): Update the height of the node S. $\mathcal{O}(1)$ time complexity.

```
def height(S):
    if S == NULL:
        return -1
    else:
        return S.height


def update_height(S):
    S.height = 1 + max(height(S.left), height(S.right))
```

- balance(S): Balance the node S. $\mathcal{O}(1)$ time complexity.

```
def balance(S):
    if S == NULL:
        return S
    if height(S.left) - height(S.right) > 1:
        if height(S.left.left) >= height(S.left.right):
            S = rotate_right(S)
        else:
            S.left = rotate_left(S.left)
            S = rotate_right(S)
    else if height(S.right) - height(S.left) > 1:
        if height(S.right.right) >= height(S.right.left):
            S = rotate_left(S)
        else:
            S.right = rotate_right(S.right)
            S = rotate_left(S)
    return S
```

- update_min_diff(S): Update the augmented information of the node S.

  $\mathcal{O}(1)$ time complexity.

```
def update_min_diff(S):
    if S == NULL:
        return
    if S.left == NULL and S.right == NULL:
        S.relative_min_diff = MAX_INT
// It it more understandable to use 0 since the difference is always positive.
// But we need to use MAX_INT for the ease of implementation.
// Actually, 'if S.left.relative_min_diff <= S.right.relative_min_diff:'
        S.min_pair = (NULL, NULL)
        return
    if S.left == NULL:
        S.relative_min_diff = S.right.relative_min_diff
        S.min_pair = S.right.min_pair
        return
    if S.right == NULL:
        S.relative_min_diff = S.left.relative_min_diff
        S.min_pair = S.left.min_pair
        return

    if S.left.relative_min_diff <= S.right.relative_min_diff:
        S.relative_min_diff = S.left.relative_min_diff
        S.min_pair = S.left.min_pair
    else:
        S.relative_min_diff = S.right.relative_min_diff
        S.min_pair = S.right.min_pair

    return S
```

■

3. Justify why your algorithms are correct and why they achieve the required time bound.

Proof of correctness

- search(S, x): The search function of a AVL is correct, and the introduced helper function 'update_min_diff()' does not change the correctness of the search function.
- insert(S, x): The insert function of a AVL is correct, and the introduced helper function 'update_min_diff()' are correct.
- delete(S, x): The delete function of a AVL is correct, and the introduced helper function 'update_min_diff()' are correct.
- min_difference(S): It basically returns a value stored in the input node. To prove the correctness of this function, we need to prove that the value stored in the input node is correct, which will be proved in the proof of the 'update_min_diff()' function.
- update_min_diff(S): The correctness of this function is based on the following two facts:

  1. [*Base Case*] If the node S is a leaf node, then the value stored in S is correct.
     - The value stored in S is the minimum absolute difference of the set S, where the set S is a set with only one element, so the minimum absolute difference of the set S is a special case of the minimum absolute difference of a set, which is defined as MAX_INT (because the real minimum absolute difference is always greater than MAX_INT in a set of distinct integers).
     - Therefore, the value stored in S is correct.
  2. [$I.H.$] The value stored in the node S is correct.
  3. [$I.S.$] Prove: the value stored in the parent node of S is correct.
     Let S' be the parent node of S, and let S_L and S_R be the left child and right child of S' respectively.
     - The value of S_L.relative_min_diff is the minimum absolute difference of the set S_L.
     - The value of S_R.relative_min_diff is the minimum absolute difference of the set S_R.
     - The value of S'.relative_min_diff is the minimum of S_L.relative_min_diff and S_R.relative_min_diff.
     - $\Rightarrow$ The value of S'.relative_min_diff is the minimum absolute difference of both subset S_L and S_R.
     - $\Rightarrow$ The value of S'.relative_min_diff is the minimum absolute difference of the set S'.

- $\Rightarrow$ The value of S'.relative_min_diff is correct.

■

Proof of time complexity

- search(S, x): O(log n) because the search function of a AVL is O(log n), and the introduced helper functions are O(1).
- insert(S, x): O(log n) because the insert function of a AVL is O(log n), and the introduced helper functions are O(1).
- delete(S, x): O(log n) because the delete function of a AVL is O(log n), and the introduced helper functions are O(1).
- min_difference(S): O(1) because it basically returns a value stored in the input node.

Above are the time complexities of the four functions, which are all within O(log n).

■