CSCB58-Assignment1

Yiyun He

June 7 2024

1 Question 1

1.1 part a - False

Counterexample:

Let $f(n) = n, g(n) = n$ and $h(n) = n$.

$f(n) \in O(h)$ and $g(n) \in O(h)$. However, $f(n) * g(n) = n^2 \notin O(h)$.

Therefore, the statement is false.

1.2 part b - True

Proof:

$f \in \theta(g)$ indicates that there exists constants c_1, c_2 and n_0 such that

$$0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n) \tag{1}$$

for all $n \geq n_0$.

The definition also indicates that:

$$g(n) \leq \frac{1}{c_1} * f(n) \tag{2}$$

$$\frac{1}{c_2} * f(n) \leq g(n) \tag{3}$$

The in-equations (1) and (2) shows that there exists constants $\frac{1}{c_1}, \frac{1}{c_2}$ and n_0 such that

$$\frac{1}{c_2} * f(n) \leq g(n) \leq \frac{1}{c_1} * f(n) \tag{4}$$

for all $n \geq n_0$, which is the definition of $g \in \theta(f)$.

1.3 part c - False

Counterexample:

Let $f(n) = sin(n) + 1, g(n) = cos(n) + 1$.

As n increases, the values of sin(n) + 1 and cos(n) + 1 oscillate continuously between 0 and 2.

Therefore, $f(n) \notin O(g)$ and $g(n) \notin O(f)$.

Therefore, the statement is false.

1.4 part d - True

Proof:

$f \in O(h)$ indicates that there exists constants c_1 and n_0 such that

$$0 \leq f(n) \leq c_1 * h(n) \tag{5}$$

for all $n \geq n_0$.

$g \in O(k)$ indicates that there exists constants c_2 and n_1 such that

$$0 \leq g(n) \leq c_2 * k(n) \tag{6}$$

for all $n \geq n_1$.

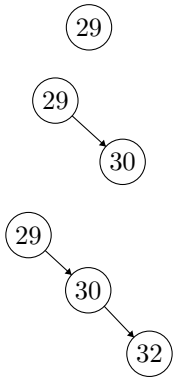
The addition of (5) and (6) shows that there exists constants c_1, c_2 and n_3 such that:

$$f(n) + g(n) \leq c_1 * h(n) + c_2 * k(n) \leq (c_1 + c_2) * max(h(n), k(n)) \tag{7}$$

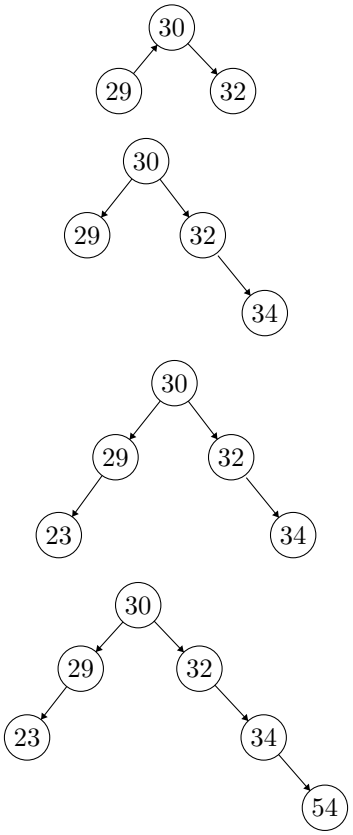
for all $n \geq n_3$, which is the definition of $f(n) + g(n) \in O(max(h(n), g(n)))$.

2 Question 2

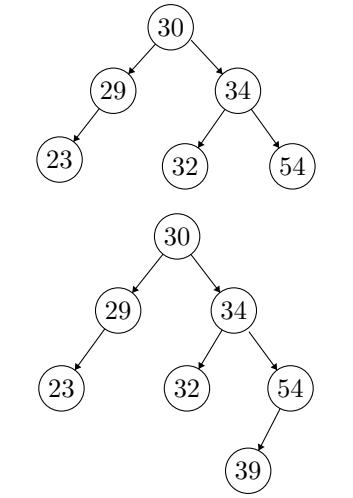
2.1 Insertion

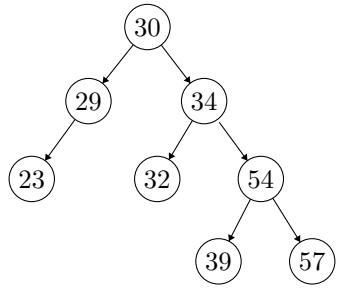


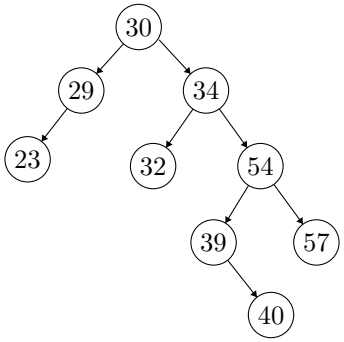
2.1.1 Left Rotation - Applys to 29



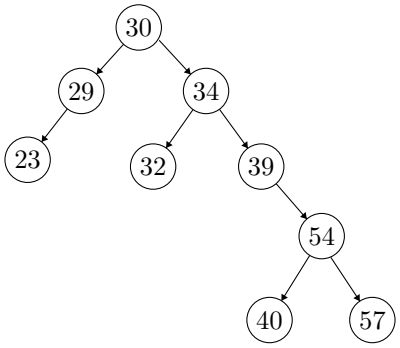
2.1.2 Left Rotation - Applys to 32

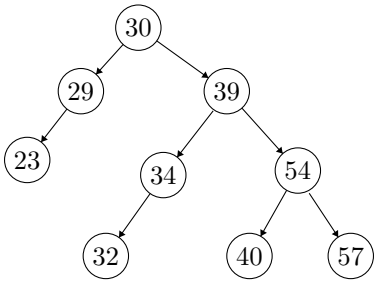




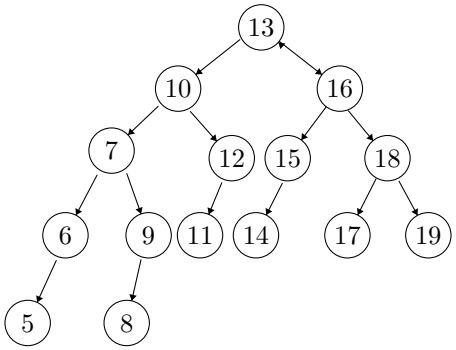


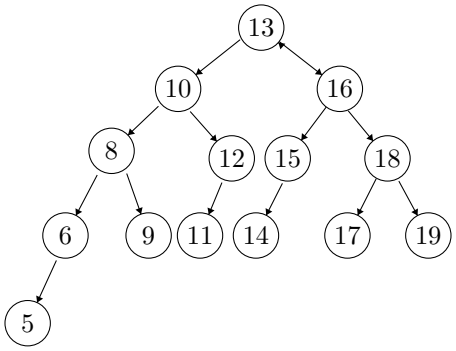
2.1.3 Right-Left Rotation - Applys to 34

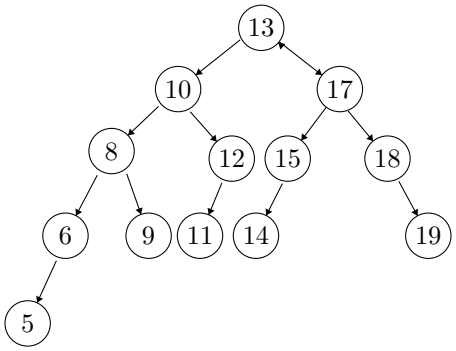


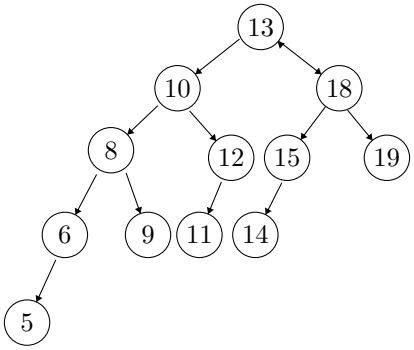


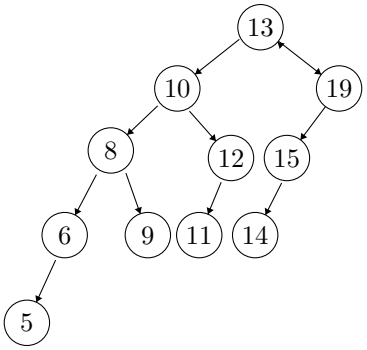
2.2 Deletion


13
10
16
7
12
15
18
6
9
11
14
17
19
5
8

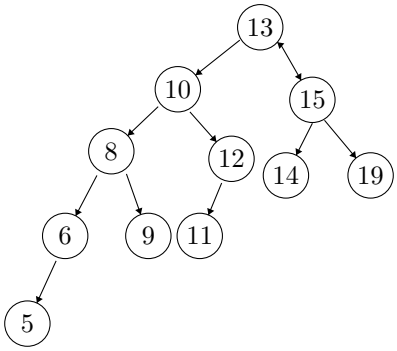

13
10
16
8
12
15
18
6
9
11
14
17
19
5


13
10
17
8
12
15
18
6
9
11
14
19
5

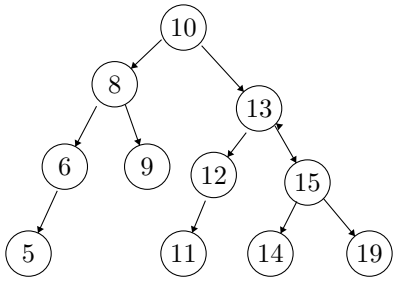




2.2.1 Right Rotation - Applys to 19



2.2.2 Right Rotation - Applys to 13



3 Question 4

3.1 Data Structure

```
typedef struct tree_linked_list
{
  struct linked_list* first;  // the first node of the linked list
  struct linked_list* last;   // the last node of the linked list
  int length;                 // the length of the linked list, 0 <= length <= k.
  AVL_tree* root;             // the root of the tree
} tree_linked_list;

typedef struct linked_list
{
  int value;              // value stored in this node
  linked_list* prev;      // pointer to the previous node
  linked_list* next;      // pointer to the next node
} linked_list;

typedef struct AVL_tree
{
  int key;                    // key stored in this node
  struct linked_list* node;  // the linked list node associated with this tree node
  int height;                 // height of tree rooted at this node
  struct AVL_tree* left;     // this node's left child
  struct AVL_tree* right;    // this node's right child
} AVL_tree;
```

Explain:

The data structure, called the "tree-linked list," consists of a "linked list" and an "AVL tree."

The linked list records the sequence of students who have requested help, with each node's value attribute storing a student's number.

The AVL tree stores all the students who have recently asked for help. In this tree, each node's key represents a student's number. Other students with

smaller student numbers are stored in the left subtree, and those with larger numbers are stored in the right subtree.

3.2 Operations

```
bool needsMoreHelp(tree_linked_list* H, int x)
{
  return needsMoreHelpHelper(H->root, x);
}

bool needsMoreHelpHelper(AVL_tree* T, int x)
{
  if T is empty tree:
    return false;
  else if T is a leaf:
    return if T->key is x;
  else:
    if T->key is bigger than x:
      return needsMoreHelpHelper(T->left, x);
    else if T->key is smaller than x
      return needsMoreHelpHelper(T->right, x);
    else:
      return true;
}

void drop(tree_linked_list* H, int x)
{
  if the student x is in the list:
    create a new linked list called "deleted_node_linked_list";
    calls dropHelper(H->root, x) and update the root and "deleted_node_linked_list";
    update length of H;
    remove "deleted_node_linked_list" from H's linked list;
    free the memory of "deleted_node_linked_list";
}

(AVL_tree*, linked_list*) dropHelper(AVL_tree* T, int x)
{
  create a temporary tree recording the result of the function called "result";
  create a new linked list called "deleted_node_linked_list";

  if T->key is bigger than x:
    T->left, deleted_node_linkeded_list = dropHelper(T->left, x);
  else if T->key is smaller than x:
    T->right, deleted_node_linkeded_list = dropHelper(T->right, x);
  else:
    set the deleted_node_linked_list as T->node;
```

```
    if T is a leaf:
      set the result as NULL;
      free the memory of T;
    else if T has no left subtree:
      set the result as T->right;
      free the memory of T;
    else if T has no right subtree:
      set the result as T->left;
      free the memory of T;
    else:
      set the result as T;
      find the successor of T;
      replace T's pair of key and linked list node with the successor's pair;
      T->right, deleted_node_linkeded_list = dropHelper(T->right, successor->key);

  update the values of result;
  rebalance result if necessary;
  return result, deleted_node_linked_list;
}

void help(tree_linked_list* H, int x)
{
  if the student x is not in the list:
      if the length is k:
        drop(H, H->first->value);
      add x to the end of the linked_list
      update length;
      insert x into T->root by calling helpHelper(H->root, x);
}

AVL_tree* helpHelper(AVL_tree* T, int x)
{
  if T is empty tree:
    create a new node with key x and its corresponding linked list node;
    return the pointer of the new node;

  if T->key is bigger than x:
    T->left = helpHelper(T->left, x);
  else if T->key is small than x:
    T->right = helpHelper(T->right, x);

  update the values of T;
  rebalance T if necessary;
  return T;
}
```

3.3 Correctness and Time Complexity

3.3.1 For the function "needsMoreHelp" ("needsMoreHelpHelper"):

Proof:

For the function "needsMoreHelp", let an integer $n \geq 0$ as the size of the input H.

The function "needsMoreHelp" calls "needsMoreHelpHelper" with n as the size of the input T, which has a time complexity of O(log(n)) (proves later).

Therefore, the function "needsMoreHelp" has a time complexity of O(log(n)) too.

For the function "needsMoreHelpHelper", given an integer $n \geq 0$ as the size of the input T, then there are two paths:

Path 1 (base cases):

If T is empty, the function will return false.

If T has only one node, the function will return the result of the comparison of the root value and the target value.

In these cases, the operations take constant time and the time complexity is O(1).

Path 2 (recursive case):

If $n > 1$, then the algorithm computes needsMoreHelpHelper(left, x) or needsMoreHelpHelper(right, x).

Demonstrating Recursive Call on Smaller Values:

In this case, the recursive calls will access the right subtrees or the left subtrees.

Note that left and right each contains almost half of the input T. So the size of inputs for the recursive calls is decreasing.

Since each recursive calls only accesses half of the tree, the time complexity is $O(log_2(n))$ = O(log(n)).

Proving Precondition:

Note that the subtrees are either empty trees (NULL) or smaller AVL trees, which satisfies the precondition of the input T's data type.

The inductive hypothesis assumes that the outputs of needsMoreHelpHelper(left, x) or needsMoreHelpHelper(right, x) satisfy the postcondition.

Proving Termination:

At each call the size of the input n halves.

So the sequence n is an integer strictly decreasing sequence, therefore it must become less than 2 at some point.

This implies Path 2 will ultimately terminate to the base case (Path 1).

In conclusion, the function "needsMoreHelpHelper" is correct and has a time complexity of O(log(n)).

3.3.2 For the function drop (dropHelper):

Proof:

For the function "drop", let an integer $n \geq 0$ as the size of the input H.

The function "drop" calls the function "dropHelper" with n as the size of the input T, which has a time complexity of O(log(n)) (proves later).

Then "drop" updates the length, removes the last linked list node and frees the memory of the removed linked list node, and these operations have a constant time complexity of O(1).

Note that the time complexity of the removal of a node from the linked list is O(1) since the algorithm has direct access to the end of the linked list.

For the function "dropHelper", given an integer $n \geq 0$ as the size of the input T, then there are three paths:

Path 1 (base case):

If the target key matches the node's key and the node has less than two subtrees, the algorithm sets the result value and frees the memory of the node. The time complexity of these operations is O(1).

In these cases, the operations take constant time and the time complexity is O(1).

Path 2 (recursive case):

If the target key matches the node's key and the node has two subtrees, then the algorithm computes dropHelper(right, successor) to delete the leftmost node of the right subtree.

Demonstrating Recursive Call on Smaller Values:

In this case, the first recursive call will access the right subtree, and the following recursive calls will access the left subtrees.

Note that left and right each contains almost half of the input T. So the size of inputs for the recursive calls is decreasing.

Since each recursive calls only accesses half of the tree, the time complexity is $O(log_2(n)) = O(log(n))$.

Proving Precondition:

Note that the subtrees are either empty trees (NULL) or smaller AVL trees, which satisfies the precondition of the input T's data type.

The inductive hypothesis assumes that the outputs of dropHelper(right, successor) satisfy the postcondition.

Proving Termination:

After each calls, the node's key must get closer to the target key.

Since the difference between the node's key and the target key is decreasing, the keys must match at some point.

This implies Path 2 will ultimately terminate to the base case (Path 1).

Path 3 (recursive case):

If the target key does not match the node's key, then the algorithm computes dropHelper(left, x) or dropHelper(right, x).

Demonstrating Recursive Call on Smaller Values:

In this case, the recursive calls will access the right subtrees or the left subtrees.

Note that left and right each contain almost half of the input T. So the size of inputs for the recursive calls is decreasing.

Since each recursive calls only accesses half of the tree, the time complexity is $O(log_2(n))$.

Proving Precondition:

Note that the subtrees are either empty trees (NULL) or smaller AVL trees, which satisfies the precondition of the input T's data type.

The inductive hypothesis assumes that the outputs of dropHelper(left, x) and dropHelper(right, x) satisfy the postcondition.

Proving Termination:

At each call, the node's key must get closer to the target key.

Since the difference between the node's key and the target key is decreasing, the keys must match at some point.

This implies Path 3 will ultimately terminate to the base case (Path 1).

In conclusion, the function "dropHelper" is correct and has a time complexity of O(log(n)).

3.3.3 For the function help (helpHelper):

Proof:

For the function "help", let an integer $n \geq 0$ as the size of the input H.

If the length of the list is filled (k), the function "help" calls the function "drop", which has a time complexity of O(log(n)), to remove one student.

The function drop also updates the length and adds a new node to the linked list, and these operations have a time complexity of O(1).

Note that the time complexity of the addition to the linked list is O(1) since the algorithm has direct access to the end of the linked list.

Finally, the function "help" calls the function "helpHelper", which has a time complexity of O(log(n)) (proves later).

To summarize, the function "help" has an overall time complexity of $\theta(log(n))$.

For the function "helpHelper", given an integer $n \geq 0$ as the size of the input T, then there are two paths:

Path 1 (base case):

If T is an empty tree, the algorithm creates a new node with key x and its corresponding linked list node and returns its pointer.

In this case, the function will return the correct results and the time complexity is O(1).

Path 2 (recursive case):

If T is not an empty tree, then the algorithm computes helpHelper(left, x) or helpHelper(right, x) to find an empty spot for the value x.

Demonstrating Recursive Call on Smaller Values:

In this case, the recursive calls will access the right subtrees or the left subtrees.

Note that left and right each contain almost half of the input T. So the size of inputs for the recursive calls is decreasing.

Since each recursive calls only accesses half of the tree, the time complexity is $O(log_2(n))$ = O(log(n)).

Proving Precondition:

Note that the subtrees are either empty trees (NULL) or smaller AVL trees, which satisfies the precondition of the input T's data type.

The inductive hypothesis assumes that the outputs of helperHelper(left, x) and helperHelper(right, x) satisfy the postcondition.

Proving Termination:

After each calls, x is getting closer to its right spot in the tree T.

Since the distance between x and its right spot is decreasing, x must find an empty spot at some point.

This implies Path 2 will ultimately terminate to the base case (Path 1).

In conclusion, the function "helpHelper" is correct and has a time complexity of O(log(n))

3.4 Space Complexity

Explain:

The data structure "tree-linked list" needs two pointers to linked list nodes, an integer and a pointer to an AVL tree node.

Each node of linked list needs an integer and two pointers to linked list nodes.

Each node of AVL tree needs two integers, a pointer to linked list node, and two pointers to AVL trees.