COMP2123-Assignment 3

May 23, 2024

Notation Clarification

This section is to clarify the notations used throughout this assignment.

$[a : b]$	The sequence a, $a + 1, \ldots, b - 1$.
`int`	The data type representing integers.
`real`	The data type representing the real numbers.
`void`	Used to show that a function does not return anything.
`null`	The variable representing nothingness.
`bool`	The data type representing a boolean value which is either true or false.

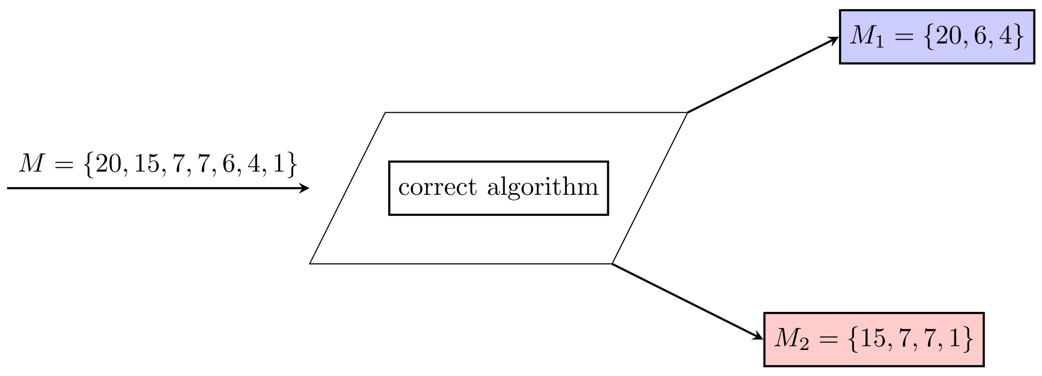
Problem 1

a)

```
function wrongAlgorithm(M)
    $M_1$, $M_2 \leftarrow \emptyset, \emptyset$                              ▷ Create two empty sets
    while $M$ contains some duplicate integer x do
        $M \leftarrow M \setminus \{x, x\}$          ▷ Remove both instances of the duplicate integer x from the set M.
        $M_1 \leftarrow M_1 \cup \{x\}$                ▷ Add one instance of the removed integer x to set M1.
        $M_2 \leftarrow M_2 \cup \{x\}$              ▷ Add the other instance of the removed integer x to set M2.
    while $M \neq \emptyset$ do
        $x \leftarrow$ largest integer in M
        $M \leftarrow M \setminus \{x\}$                               ▷ Remove the largest integer x from set M
        if there exist $M^{'} \subseteq M$ s.t. x = sum of integers in $M^{'}$ then
            $M \leftarrow M \setminus M^{'}$                                  ▷ Remove the subset M' from M.
            $M_1 \leftarrow M_1 \cup \{x\}$                                       ▷ Add the integer x to set M1
            $M_2 \leftarrow M_2 \cup M^{'}$                             ▷ Add all integers in set M' to set M2.
        else
            return False
    return True
```

Algorithm 1: Wrong algorithm

- Let **Correct algorithm** is defined as an algorithm that consistently provides the correct answer for every possible input scenario of problem 1. Specifically, for any given set M, this algorithm will return True if and only if M can be partitioned into two subsets with equal sums, and False otherwise.
- The set M=1,1,7,3,4,2 is provided as input to the Correct Algorithm. The Correct Algorithm returns True. The reasoning is illustrated in the diagram below:



As we can see in the graph, the sum of all elements in set $M_1 = \{20, 6, 4\}$ is $20 + 6 + 4 = 30$.
The sum of all elements in set $M_2 = \{15, 7, 7, 1\}$ is $15 + 7 + 7 + 1 = 30$.
Therefore, both sets M_1 and M_2 have the same sum. **Correct algorithm** have to return True.

1. Testing the **WrongAlgorithm**

- The same set $N = \{20, 15, 7, 7, 6, 4, 1\}$ is also input into a **WrongAlgorithm**
- Expected output: the wrong algorithm should return False.
- Conclusion from expected output: If the expected output is False, **WrongAlgorithm** does not correct in the case $M = \{20, 15, 7, 7, 6, 4, 1\}$. Thus, **WrongAlgorithm** does not always return the correct answer and set M is a counterexample of **WrongAlgorithm**.

2. Tracking **WrongAlgorithm**

Table 1: Description of Note

Note Color	**Meaning**
Red	Duplicates element
Brown	Largest integer
Brown	A set containing the largest integer
Blue	Subset with sum equal to the largest integer

Table 2: Tracking table

Line	**M**	M_1	M_2
2	$\{20, 15, 7, 7, 6, 4, 1\}$	$\{\}$	$\{\}$
3 - 6	$\{20, 15, 6, 4, 1\}$	$\{7\}$	$\{7\}$
7 - 15	$\{6\}$	$\{7, 20\}$	$\{7, 15, 4, 1\}$

- In line 2 of the function, it creates two empty sets, which are M_1 and M_2 so in table 2, we have the following sets.
- From line 3 to line 6 of **WrongAlgorithm**, use to find every duplicate integers then each of them are being seperated in each set. In 2, we can see that set M has dupilicate integer is 7, so 7 is being added one to each of M_1 and M_2. After the first while-loop, $M = \{20, 15, 6, 4, 1\}$, $M_1 = 7$, $M_2 = 7$.
- Because $M \neq \emptyset$ so we run the second loop from line 7 - 15. In that loop, the function find the maximum integer in M, which equals to 20, after that function find the subset $M^{'}$ that have the sum of all elements that equals to maximum integer in M. If we can not find, the function immediately return False. Otherwise, M_1 will add largest elements in M and M_2 equals to the M_2 In 2 we can see that, 20 is the largest integer in M and we can find a subset $M^{'}$ such that sum of all elements in $M^{'}$ equals to 20, which is $M^{'} = \{15, 4, 1\}$. Thus, after the first iteration in the second while loop, $M = \{6\}$, $M_1 = \{7, 20\}$ and $M_2 = \{7, 15, 4, 1\}$
- After the first iteration, we can see that $M = \{6\} \rightarrow M \neq \emptyset$, so the second will loop will continue to run one more time. At the second iteration, the largest integer in M now is 6. However, at this time, there do not have any $M^{'} \subseteq M$ s.t. x = sum of integers in $M^{'}$. Thus, the the conditional statement in line 10 will not run, it will run the else condition. Thus, the function return False and terminate.

3. Conclusion:

- As we can see, with the same input, **Correct Algorithm** return True, while **WrongAgorithm** return False. Thus, as what we say in Testing the **WrongAlgorithm**, if the ouput is False. **Wrong Algorithm** does not correct in the case $M = \{20, 15, 7, 7, 6, 4, 1\}$. Thus, **WrongAlgorithm** does not always return the correct answer and set M is a counterexample of **WrongAlgorithm**

b)

```
1: function BalancingAlgorithm(M)
2:     $M_1$, $M_2 \leftarrow \emptyset, \emptyset$                ▷ Create two empty sets
3:     Sort M in non-increasing order
4:     for each integer x in M do
5:         if sum of integers in $M_1 \leq$ sum of integers in $M_2$ then
6:             $M_1 \leftarrow M_1 \cup \{x\}$
7:         else
8:             $M_2 \leftarrow M_2 \cup \{x\}$
9:     return sum of integers in $M_1$ = sum of integers in $M_2$
```

Algorithm 2: Balancing algorithm

1. Testing **BalancingAlgorithm**
   - The same set $N = \{20, 15, 7, 7, 6, 4, 1\}$ is also input into a **BalancingAlgorithm**
   - Expected output: the wrong algorithm should return False.
   - Conclusion from expected output: If the expected output is False, **BalancingAlgorithm** does not correct in the case $M = \{20, 15, 7, 7, 6, 4, 1\}$. Thus, **BalancingAlgorithm** does not always return the correct answer and set M is a counter-example of **BalancingAlgorithm**.
2. Tracking **BalancingAlgorithm**

Table 3: Tracking balancing algorithm table

Line	**M**	M_1	M_2
3	$\{20, 15, 7, 7, 6, 4, 1\}$	$\{\}$	$\{\}$
4 - 8	$\{20, 15, 7, 7, 6, 4, 1\}$	$\{20\}$	$\{\}$
4 - 8	$\{20, 15, 7, 7, 6, 4, 1\}$	$\{20\}$	$\{15\}$
4 - 8	$\{20, 15, 7, 7, 6, 4, 1\}$	$\{20\}$	$\{15, 7\}$
4 - 8	$\{20, 15, 7, 7, 6, 4, 1\}$	$\{20, 7\}$	$\{15, 7\}$
4 - 8	$\{20, 15, 7, 7, 6, 4, 1\}$	$\{20, 7\}$	$\{15, 7, 6\}$
4 - 8	$\{20, 15, 7, 7, 6, 4, 1\}$	$\{20, 7, 4\}$	$\{15, 7, 6\}$
4 - 8	$\{\}$	$\{20, 7, 4\}$	$\{15, 7, 6, 1\}$

3.
   - In line 2, 3 of the function, it creates two empty sets, which are M_1 and M_2 and set M is sorted in non-increasing order so in table 3, we have the following sets.
   - In the for-loop in line 4 to line 8, the function will iterate each element in M only once. I assume that the for-loop will iterate from left to right, which means from the largest to the smallest.
   - In the first iteration, it will take 20 to set M_1 because it will go through the if condition in line 5, which will calculate the sum of M_1 and M_2, which both equals to 0. Thus, 20 will come to set M_1. Similarity, it will continue to 15. As we can see in table 3, sum of all elements in M_1 is larger than sum of all elements in M_2. Thus, we will add 15 to M_2. As a result, in the second iteration, $M_1 = \{20\}$ and $M_2 = \{15\}$.

- The for-loop will continue to iterate until it reachs every elements in M once time. At the last iteration, as we can see in 3, $M_1 = \{20, 7, 4\}$ have the sum is 31 and $M_2 = \{15, 7, 6\}$ have the sum is 28. Thus, 1 have to be in set M_2 (because sum of all elements in M_1) is larger than sum of all elements in M_2. After the iteration we have $M_1 = \{20, 7, 4\}$ and $M_2 = \{15, 7, 6, 1\}$ so the sum of all elements in M_1 is 31 and sum of all elements in M_2 is 29. So the **Balancing Algorithm** will return False in line 9.

4. Conclusion

   - As we can see, with the same input, **Correct Algorithm** return True, while **Balancing Algorithm** return False. Thus, as what we say in Testing of **Balancing Algorithm**, if the output is False, the **Balancing Algorithm** does not correct in the case $M = \{20, 15, 7, 7, 6, 4, 1\}$. Thus, the **Balancing Algorithm** does not always return the correct answer and set M is a counter example of **Balancing Algorithm**.

Problem 2

a)

This is the following methods of doubly-linked list we have to use.

- `Node first()`: Return the first **Node**(head) of that doubly-linked list. $O(1)$
- `Node last()`: Return the last **Node**(tail) of that doubly-linked list. $O(1)$
- `void remove(Node p)`: Remove the Node p out of that doubly-linked list. This method also update the head and tail of that doubly-linked list after removing Node p. $O(1)$
- `bool is_empty()`: indicates wheather no elements are stored. $O(1)$

To implement doubly-linked list we need to have Node object with some attributes such as **next**, **prev** and **value**.

```
Class Node
    int value      ▷ The value store in the Node
    Node next      ▷ The reference to the next node, null if it is the end node
    Node prev      ▷ The reference to the previous node, null if it is the head of that doubly-linked list
```

Algorithm 3: Class Node

Because **REMOVE-MIN** and **REMOVE-MAX** is on the union of the lists, we have to add more attribute to class Node, which names is **belong**. This attribute will return the doubly-linked list that contains that Node. This attribute will help us know where the Node is in.

Thus, the Node class below is the update version.

```
Class Node
    int value             ▷ The value store in the Node
    Node next             ▷ The reference to the next node, null if it is the end node
    Node prev             ▷ The reference to the previous node, null if it is the head of that doubly-linked list
    Doubly_ll belong      ▷ The reference to the doubly-linked list that the Node is in
```

Algorithm 4: Class Node

- Now, I will build a min-heap and max-heap with size k. To build a min-heap and a max-heap, I implement heap-in-array implementation, I need to have two array of length k, one for max-heap and another one for min-heap.
- With k non-empty sorted doubly-linked list, I assume that all of them is in non-increasing order. Thus, the head of that doubly-linked list will have the largest value in that doubly-linked list, and the tail of that doubly-linked list will be the smallest value of that doubly-linked list.
  - **min_array**: This array uses to build min-heap. Thus, every element in this array will be the last Node(tail) of k sorted doubly-linked list.

  - **max_array**: This array uses to build max-heap. Thus,every element in this array will be the first Node(head) of k sorted doubly-linked list.

- To put k **Node** in array, I will have the following pseudocode. This pseudocode will assume k doubly-linked list will be stored in an array with k elements and each element in that array is doubly-linked list object. I call that array is **collections**.

```
1: function storingNode(Array min_array, Array collections)
2:     index ← 0
3:     for each doubly-linked list object k in collections do
4:         array.add(index, k.last())
5:         index ← index + 1
6:     return min_array
```

Algorithm 5: Add k Node in array

- I always add Node object at the end of an array so add function do not need to shifting forward. Thus, add function always run in $O(1)$, with k doubly-linked list object, **storingNode** will run in $O(k)$ time.

- To make **max_array** store Node, it will be similar to **storingNode** with a few changes in the parameter in line 1, change to max_array and instead of using $k.last()$ in line 4, it has to be $k.first()$

- After having two arrays, now we will use that to implement heap-in-array with some important properties.
  - Special Node
    - root is at 0
    - last node is at n - 1
  - For the node at index i:
    - The left child is at index $2i + 1$
    - The right child is at index $2i + 2$
    - Parent is at index $\lfloor (i - 1)/2 \rfloor$

- This pseudocodes below is how I build max-heap
  - To build max-heap, we need a function calls **heapify**

```
function HEAPIFY(int index, Array max_array)
    length ← length of max_array
    $n \leftarrow \lfloor (length - 1)/2 \rfloor$    ▷ n is the last index that have child
    if index > n then
        return null ▷ Heapify function just take index that have child as a parameter, if not, we will return. This is the base case
    left_child ← $2 \times$ index + 1    ▷ location of left child of index
    right_child_2 ← $2 \times$ index + 2    ▷ location of right child of index
    largest ← index
    if left_child < length and max_array[left_child].value > max_array[index].value then
        largest ← left_child
    if right_child < length and max_array[right_child].value > max_array[largest].value then
        largest ← right_child
    if largest ≠ index then
        node_temporary ← max_array[index]
        max_array[index] ← max_array[largest]
        max_array[largest] ← node_temporary
        heapify(largest, max_array)    ▷ After finish swaping, we continue to heapify at index
```

Algorithm 6: Heapify function

- Then I will use heapify to build a heap with function calls **build_heap**

```
function BUILD_HEAP(Array max_array)
    i ← $\lfloor$(length of max_array − 1)/2$\rfloor$ ▷ We will start heapify from the last index that have child to root index, which is 0
    while i >= 0 do
        heapify(i, max_array)
        i ← i - 1
    return max_array
```

Algorithm 7: Build heap

- Build min-heap is similar to build max-heap with some changes of the function heapify.
  - The parameter is not max_array, it will become min_array
  - Instead of marking **largest** equals to the child index that have the Node contains largest value, We will mark the **largest**(We can change it to **smallest**) equals to the child index that have the Node contains smallest value. (Change > to < in line 9 and line 11)
- After I finish build heap. Now we have some properties in two arrays **min_array** and **max_array**.
  - The first index in **min_array** contains the Node that have the smallest value in **min_array**, which means that Node that have smallest value comparing to every Node in k doubly-linked list. This is mainly because we store every Node that have smallest value in each doubly-linked list in **min_array**.
  - Similar to first index in **max_array** contains the Node that have the largest value in **max_array**, and that Node also have largest value comparing to every Node in k doubly-linked list.

- Before implements **REMOVE-MIN** and **REMOVE-MAX**, I need to create two more functions, which name are **removeTop(Array array)** and **insert(Node node, Array array)**.
  - The reason I need two create two more functions is that, I need to take the root of the heap out when calling **REMOVE-MIN** and **REMOVE-MAX** functions. Then, I will remove that node at that doubly linked list, before insert the tail of that doubly linked list back to the heap.
- `Node removeTop(Array array)`: Return the root of the heap
- `void insert(Node node, Array array)`: insert Node node into a heap.

```
1: function removeTop(Array array)
2:     node ← array[0]                              ▷ the node will be removed
3:     array[0] ← array[length of array - 1]        ▷ Replace the root with the last element
4:     array← array[0 : length of array - 1]
5:     heapify(0, array) ▷ use heapify function at root of the heap, this function can be a version of heapify for
       min-heap and max-heap
6:     return node
```

Algorithm 8: Remove root in heap

```
1: function insert(Node node,Array array)
2:     i ← the length of array                      ▷ first i will equals to array length
3:     array.add(i, node)                           ▷ Assume array will not be full capacity
4:     while array[$\lfloor(i-1)/2\rfloor$].value < array[i].value do   ▷ Fix the heap properties if it is violated
5:         if i = 0 then
6:             break
7:         temporary_node ← array[$\lfloor(i-1)/2\rfloor$]
8:         array[$\lfloor(i-1)/2\rfloor$] ← array[i]
9:         array[i] ← temporary_node
```

Algorithm 9: Insert Node node into heap

- From every properties I have mentioned before and all function I have just built, we can use them to implements **REMOVE-MIN** and **REMOVE-MAX** function.

```
1: function REMOVE_MAX(Array max_array)
2:     node ← removeTop(max_array) ▷ Remove the root of the heap and node will be the node at the root of the
       heap
3:     node.belong.remove(node) ▷ Find that doubly-linked list that contains that node then remove that node using
       remove methods
4:     insert(node.belong.first(), max_array) ▷ After remove that doubly-linked list, then we insert the newest head
       of that doubly-linked list into max-heap
```

Algorithm 10: REMOVE MAX

```
1: function REMOVE_MIN(Array min_array)
2:     node ← removeTop(min_array) ▷ Remove the root of the heap and node will be the node at the root of the heap
3:     node.belong.remove(node) ▷ Find that doubly-linked list that contains that node then remove that node using remove methods
4:     insert(node.belong.last(), min_array) ▷ After remove that doubly-linked list, then we insert the newest head of that doubly-linked list into max-heap
```

Algorithm 11: REMOVE MIN

- Overview data structure.
  - First I will, build a Max-heap and Min-heap with every elements in Max-heap is the head of k doubly-linked list and every elements in Min-heap is the tail of k doubly-linked list.
  - After finishing building heaps, in **REMOVE-MIN** and **REMOVE-MAX**, I calls **removeTop** to remove the node at the root of each heap.
  - Then, I will remove that node at the doubly-linked list that contains that node.
  - Finally, I call **insert** function to insert tail of that-doubly linked list for min heap and head of that doubly-linked list for max heap into each heap

b)

- As I mentioned before, root of Min-heap and Max-heap always holds the node that have maximum value and minimum value of union of thest lists. The reason for that is because all of the doubly-linked list is being sorted so in min_array, we store all node that have smallest value in each doubly-linked list, and in max_array, we store all node that have largest value in each doubly-linked list. At the same time, root of Min-heap is the smallest value in min_array and root of Max-heap is the largest value in max_array. Thus, root of Min-heap is the node that have smallest value and root of Max-heap is the node that have largest value.
- After I remove the root of the heap and that in that doubly-linked list that have that node, then I will insert newest head or tail of that doubly-linked list. Thus, we need to insert the Node that have the second maxima value in that doubly-linked list, so we insert to that heap. Following this rountine ensures the heap's order is consistently.
- **Argue the Correctness (Proving the correctness using loop invariant)**
  - **Invariant**: Because my min_array and max_array after being built by using **BUILD_HEAP** function always contains Nodes with the smallest elements and largest value in each array. All of these elements locate at the head of doubly-linked lists(max) and tail of doubly-linked lists(min), the root of min-heap(first Node in min_array) and max-heap(first Node in max_array) are always be the Nodes with smallest value and largest value in all k doubly-linked lists. This is true because every our doubly-linked lists is being sorted(non-increasing order) and the property of min-heap and max-heap are the root always be the Node that minimum and maximum value.
  - **Initialization**: Before calling **REMOVE-MIN** or **REMOVE-MAX**, the Node with smallest value is at the root of the heap(First element in array). When I call **REMOVE-MIN** and **REMOVE-MAX**, first it will remove the root of the heap by using **REMOVE-TOP** function. This function remove by

swaping root Node and last Node in heap(last element in an array), then using **HEAPIFY** function at the root of the heap to keep the property of the heap always True(The parent always larger than their children(max heap), the parent always smaller than their children(min heap)). Thus, after using removing root Node out of heap, the invariant still holds. After that, I have to insert the second smallest or second largest into heap. Because we have to keep the size of heap constant. Thus, I calls **insert** to insert next Node to heap and it also maintains the heap property because of while loop in insert function to fix the heap properties if it is violated.

– **Termination**: When **REMOVE-MIN** or **REMOVE-MAX** completes, the Node with smallest or largest value is being removed, then it also insert the another Node the prepare for the next call. As a result, the invariant always keep that at every point of the operation, the root of the heap always the Node that we will remove first.

c)

1. Running time of **HEAPIFY** function:
   (a) Running time:
      - The comparision operation at line 9 and 11 take constant time
      - The swap operaton betweeen line 14 and 16 take constant time
      - The recursive call in line 17 in worst case, the element at index may need to be moved down the height of the heap. However, a heap is a complete tree, its height is $O(log(k))$. Thus, in the worst cast the function will move an down from the root to leaf node, making a single recursive call at each level of the tree. Therefore, the running time of the **HEAPIFY** is $O(log(k))$
2. Running time of **REMOVE_TOP** function:
   (a) Running time:
      - Line 2 and 3: Storing the root's value and replacing it with the last element are both constant time operation
      - Adjusting the size of the heap is also constant time
      - Running time of **HEAPIFY** function is $O(log(k))$
      - Therefore the running time of **REMOVE_TOP** function is $O(log(k))$
3. Running time of **INSERT** function:
   (a) Running time:
      - Line 2: Assigning the length of the heap to i is a constant time.
      - Line 3: Adding the new node to the end of the array is also constant time under the assumption that the heap is not at full capacity and does not need to resize. $O(1)$
      - Line 4 - 9: In the worst case, the new node needs to be compared and possibly swapped with each of its ancestors until it reach the correct position in the heap. Since a heap is a completed tree, the height of the heap and therefore the number of ancestors, is $log(k)$
      - Therefore, in the worst case, the new node will be compared and swapped with each of its ancestors, resulting in a running time of $O(logk)$
4. Running time of initialising data structure.

(a) Running time:

- My data strucuture is heap-in-array implementation, so first I need to have two array with size of k, one is min_array and another one is max_array. To optimise the running time of import k Node(head) to max_array and k Node(tail) to min_array, I create two array with size k first. After that, I will create a varable to keep track the place to add in array in O(1), which is the last index of an array. Thus with k time add, the complexity is $O(k)$
- After I have an array, then I will call **BUILD-HEAP** function to make a heap from arrays. This function will scan every Node that have child in heap(from 0 to $\lfloor$(length of array $-$ 1)/2$\rfloor$). At each parents Node, I call **HEAPIFY** at that index. According the lecture, **a heap on n keys can be constructed in $O(n)$ time**. Thus, with k Node in array, I can construct a min-heap and max-heap in $O(k)$

(b) Conclusion: Thus, the total time of initialising is $O(k)$ time

5. Running time of REMOVE-MIN and REMOVE-MAX

6. (a) Running time:
    - Line 2: The **REMOVE_TOP** has running time of $O(log(k))$
    - The removal operation on a doubly-linked list is generally $O(1)$
    - Line 4: The **INSERT** has running time of $O(log(k))$

   (b) Conclusion: Thus, the running time of REMOVE-MIN and REMOVE-MAX $O(log(k))$ time

Problem 3

To support the operations in Problem 3, I will use the following data structure.

- `AVL Binary search tree for main dish`: This binary search tree use to store every main dishes in the restaurant. This data structure will help us to implement **removeMainDish** function and **AddNewMainDish** function. To support functions, this binary search tree have to search by the name of the dish. Thus, this binary search tree will use the name(`String`) as a key to search using(lexicographic order). I call this tree is **MainDishTree**
- `AVL Binary search tree for a side dish`: This binary search tree use to store every side dishes in the restaurant. This data structure will help us to implement **removeSideDish** function and **AddNewSideDish** function. To support functions, this binary search tree have to search by the name of the dish. Thus, this binary search tree will use the name(`String`) as a key to search using(lexicographic order). I call this tree is **SideDishTree**
- `Three arrays with size 16`: The first two arrays use to store the distribution of price of the main dish and side dish, calling **Main_array** and **Side_array**, which means each entry store how many dish with that price. For example, **Main_array[5] = 6** means we have 6 main dish with price 5$. The price will be the index(price is a positive integer) of Main_array and the value stores at Main_array at that index will be the number of Main dish The last array is `prefixDish`, this array will have the size 16, this array use to calculate prefix sum for **(**Side_array[i]), which means **prefixDish[i]** is equals to the sum of **prefixDish[i - 1]** with **Side_array[i]**

The reason we have to build AVL tree instead of normal binary search tree is because, in the worst case height(h) of the binary search tree equals to the number of node in the trees(n) so the running time of insertion and deletion in the worst case is $O(n)$. Thus we need to use AVL tree to storing with the height of AVL tree always $O(log(n))$. This can be proved by using induction(Slide 45 at week binary search tree)

- Let N(h) be the minimum number of keys of an AVL tree of height h
- We easily see that N(1) = 1 and N(2) = 2(base case)
- Clearly $N(h) > N(h-1)$ for any $h \geq 2$
- For $h > 2$, the smallest AVL tree of height h contains the root node, one AVl subtree of height $h-1$ and anotehr of height at least $h-2$

$$N(h) \geq 1 + N(h-1) + N(h-2) \geq 2N(h-2)$$

- By induction we can show that for h even

$$N(h) \geq 2^{\frac{h}{2}}$$

- Taking logarithm: $h < 2log(N(h))$
- Thus the height of an AVL tree is $O(logn)$

In slide 59, week 4, the lecture said that

- The data structure use O(n) space
- Height of the tree is $O(log(n))$

- Searching takes $O(log(n))$
- insertion takes $O(log(n))$
- Removal takes $O(log(n))$

AddNewMainDish(**String** name, **int** price) and AddNewSideDish(**String** name, **int** price) can be implemented as below.

- Using name as a key for AVL tree using lexiographic order. This key(String) is then used as a key to to insert the element into a AVl tree, aloowing for organized and efficient access based on the index. Additionally, the system updated the distribution of prices by incrementing the count in **Main_array[price]** or **Side_array[price]**. If the price is larger than 15$, we will skip the step updating the distribution of prices.
- Remember that I use insert function in AVl tree so it always keep the tree balance and another thing to note that I do not encrypt key String name to integer to compare. I use lexicographic order to compare String with String immediately. Lexicographic order comparison involves sorting elements based on their sequence, similar to how words are arranged in a dictionary. Each element is compared character by character from left to right, with the first differing character determining their order. This method is used not only for words but also for sequences of numbers or other sortable data.

removeMainDish(**String** name) and removeSideDish(**String** name) can be implemented as below

- Because I use name as a key for AVl tree when I insert Node into AVl tree, so when I want to delete some Node out of the tree, I can use **String** name to delete them. Thus, I use delete function of AVl tree with key is name to delete that Node out of the tree. After that, I have to update the distribution of the price by decrementing the value of Main_array[price] or Side_array[price] by 1 with price is the price of the Node we deleted. If the price of the Node being deleted is more than 15$, I will skip the step of updating distribution of price array

countCombination()

- Now, I have two array for main dish and side dish with the index is the price of the dish with the value of that index is the number of main dish or side dish at that price. Now the third array calls **computing** will have size of 16 to do preprocessing.

$$computing[i] = 0 \text{ if i = 0}$$

$$computing[i] = computing[i-1] + Side\_array[i-1] \text{if i larger than 0}$$

- Iterate over every pair $0 \leq i \leq j < 16$
- Use this algorithm, I call i is the index of computing. Thus computing[i] equals to number of dish that have price below (i - 1)$.
- After having this array, I will iterate every element in main_arraym, I have the variable calls **result**, each time I iterate to each element in main_array.

$$result = result + main\_array[i] + computing[16 - i + 1]$$

This one is the pseudocode.

```
1: function countCombination
2:     i ← 0
3:     while i to 15 do
4:         if i = 0 then
5:             computing[i] = 0
6:         else
7:             computing[i] = computing[i - 1] + side_array[i - 1]
8:         i ← i + 1
9:     i ← 0
10:    result ← 0
11:    while i to 15 do
12:        result ← result + main_dish[i] × computing[15 - i + 1]
13:    return result
```

Algorithm 12: countCombination

b) Running time of operation and total space

- The system use name of dish to be the keys and value is price. Thus, then we insert to AVL-tree or delete out of AVL-tree, the running time of them will be $O(log(n))$. At the same time, updating the arrays will take $O(1)$ time in worst case. Thus, the running time of **addMainDish**, **addSideDish**, **removeMainDish** and **removeSideDish** are $O(log(n))$
- About the countCombination(), to have computing array, we need to iterate for loop from 0 to 15(which is the index of computing array)(line 3 to line 8). Because updating value at the specific index in line 5 and line 7 will cost only $O(1)$. Thus, the running time of first while loop is $O(15)$. We have another while-loop from line 7 to update result. This for loop will iterate 15 times, each time just cost $O(1)$ because we just access the specific index to take the value of computing array and main_dish array. Thus the running time of **countCombination** is $O(15) + O(15) = O(1)$.
- About the total space we use two AVL tree with size n and three array with size 16. Thus the total space we use is $2 \times O(n) + 3 \times O(16) = O(n)$

b) Argue the correctness

- addNewMainDish() and addNewSideDish()
  - Utilizing the balanced BST tree, two function ensure that the new dishes are assigned to their perspective trees correctly. With each element insertion of a new node, a verificatrion of price is conducted, and updated to the associdated array are made when needed.
- removeMainDish() and removeSideDish()
  - The procedure of removeMainDish() and removeSideDish() assure the appropiate deletion of nodes from their respective BST. Each node removial is accompained by a price verification and, if required, an update to the array. This process ensure precise tracking of all viable main and side dish combination. Thus operations can be deemed correct with guaranteed accuracy

- countCombination()
  - The operation can be divided by into two part: pre-processing(counting computing) and counting the total number of combination of main dish and side dish that no more than 15$. First I will prove computing[i] = $\sum_{j=0}^{i+1} B[j]$
  - Base case i = 0, computing[i] 0 and i = 1 computing[i] = side_array[0] + B[0] (True)
  - Induction step:
    * Induction step: i = k($0 \leq k \leq 15$), computing[k] true
    * Assume the induction step is true. now I will prove z = k + 1 and computing[z] true
    * computing[z] = computing[k] + side_array[k]. Because computing[k] is true so computing[k] + side_array[k] is the total dish that have price from 0 to k because computing[k] is the total sum of price from 0 to k - 1 so computing[k] + side_array[k] is equal to the number of dishes that have price from 0 to k. Thus this function is true.

  In the second loop, it will iterate every pair with value i and 15 - i. This is like brute-force, so it always true, because we each index at each array, we calculate all possible. Thus this function is true. This function is performed to obtain the final value of total combination. Since the loop run for 16 iteration with i range from 0 to 15, we can ensure that is always true. Because of two steps of the function is true. Thus, the **countCombination** function is true.