Assignment 6

Course: COMPSCI 2C03

Date: 11/14/2025

Author: Sulagna Nandi

MacID: nandis

Student Number: 400575118

Professor: Dr. Vincent Maccio

Problem 1

P1.1 Left-Leaning Red-Black Tree

We insert elements of S in order. After each insertion I show the resulting left-leaning red-black tree (LLRB) state using the textual node format.

After inserting 78:

```
(78, B)
```

After inserting 6 (goes left of 78):

```
        (78, B)
       /
   (6, R)
```

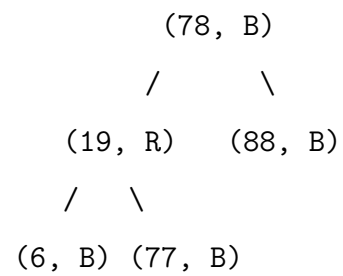
After inserting 88:

```
        (78, B)
       /      \
  (6, B)     (88, B)
```

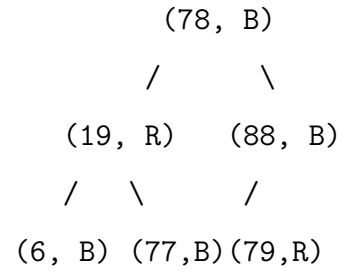
After inserting 19:

```
         (78, B)
        /      \
   (19, B)   (88, B)
   /
 (6, R)
```

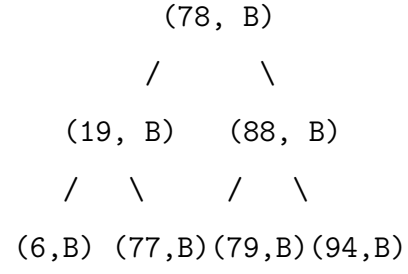
After inserting 77:



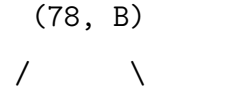
After inserting 79:

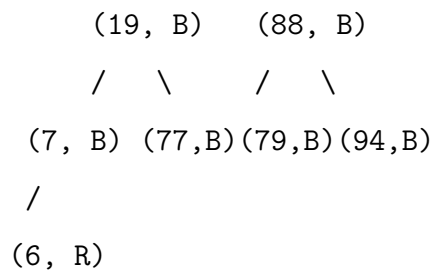


After inserting 94:

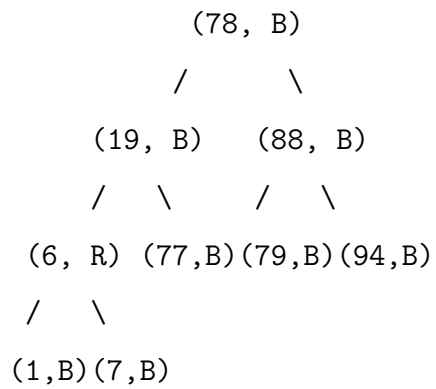


After inserting 7:

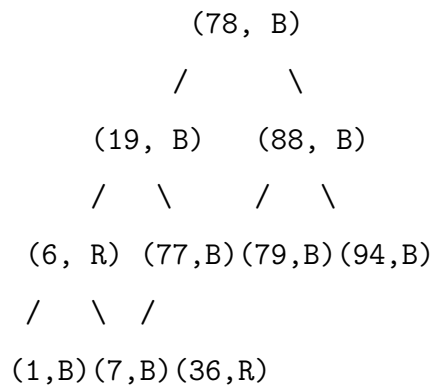




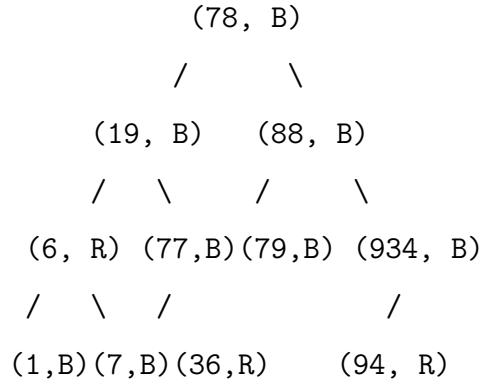
After inserting 1:



After inserting 36:



After inserting 934:


(78, B)
(19, B)
(88, B)
(6, R)
(77,B)
(79,B)
(934, B)
(1,B)
(7,B)
(36,R)
(94, R)

P1.2 Hashing with Separate Chaining

Hash indices used:

$$(x + 5) \bmod 12 :$$

$78 \to 11$, $6 \to 11$, $88 \to 9$, $19 \to 0$, $77 \to 10$, $79 \to 0$, $94 \to 3$, $7 \to 0$, $1 \to 6$, $36 \to 5$, $934 \to 3$.

(Compute each: e.g. $78 + 5 = 83$, $83 \bmod 12 = 11$, etc.)

Initial table, which is empty:

```
Index 0: ->
Index 1: ->
Index 2: ->
Index 3: ->
Index 4: ->
Index 5: ->
Index 6: ->
Index 7: ->
Index 8: ->
Index 9: ->
Index 10: ->
Index 11: ->
```

After inserting 78:

```
Index 11: -> 78
```

After inserting 6:

```
Index 11: -> 78 -> 6
```

After inserting 88:

```
Index 9: -> 88
```

After inserting 19:

```
Index 0: -> 19
```

After inserting 77:

```
Index 10: -> 77
```

After inserting 79:

```
Index 0: -> 19 -> 79
```

After inserting 94:

```
Index 3: -> 94
```

After inserting 7:

```
Index 0: -> 19 -> 79 -> 7
```

After inserting 1:

```
Index 6: -> 1
```

After inserting 36:

```
Index 5: -> 36
```

After inserting 934:

```
Index 3: -> 94 -> 934
```

The final hash table:

```
Index 0:  -> 19 -> 79 -> 7
Index 1:  ->
Index 2:  ->
Index 3:  -> 94 -> 934
Index 4:  ->
Index 5:  -> 36
Index 6:  -> 1
Index 7:  ->
Index 8:  ->
Index 9:  -> 88
Index 10: -> 77
Index 11: -> 78 -> 6
```

P1.3 Hashing with Linear Probing

I used the same hash function. For when we have collisions, it should probe linearly (which means that it checks indices $h, h+1, \ldots$ modulo 12) until it finds the first empty slot. The numbers are inserted in sequence.

Start: all slots empty (we'll use "-" for empty).

After each insertion, the whole table is shown:

1) Insert 78, index 11:

```
Index:  0 1 2 3 4 5 6 7 8 9 10 11
Value:  - - - - - - - - - - -  -  78
```

2) Insert 6, hashed to 11 (occupied), probe to 0 (next free):

```
Value:  6 - - - - - - - - - -  -  78
        ^                        ^
        0                       11
```

3) Insert 88, index 9:

```
Index:  0 1 2 3 4 5 6 7 8 9 10 11
Value:  6 - - - - - - - - 88 -  78
```

4) Insert 19, index 0 (occupied), probe to 1:

```
Value:  6 19 - - - - - - - 88 - 78
```

5) Insert 77, index 10:

```
Value:  6 19 - - - - - - - 88 77 78
```

6) Insert 79, index 0, probe 0,1 (occupied), slot 2 free:

```
Value:  6 19 79 - - - - - - 88 77 78
```

7) Insert 94, index 3:

```
Value:  6 19 79 94 - - - - - 88 77 78
```

8) Insert 7, index 0, probe 0,1,2,3 (occupied), slot 4 free:

```
Value:  6 19 79 94 7 - - - - 88 77 78
```

9) Insert 1, index 6:

```
Value:  6 19 79 94 7 - 1 - - 88 77 78
```

10) Insert 36, index 5:

```
Value:  6 19 79 94 7 36 1 - - 88 77 78
```

11) Insert 934, index 3 (occupied), probe until index 7 found free:

```
Final:
Index:  0  1  2  3  4  5  6  7   8  9  10 11
Value:  6 19 79 94 7 36 1 934  - 88  77 78
```

The final linear-probing table:

```
0: 6
1: 19
2: 79
3: 94
4: 7
5: 36
6: 1
7: 934
8: -
9: 88
10: 77
11: 78
```

Problem 2

P2.1 Algorithm (Aggregation)

To compute aggr(L) efficiently, the most natural idea is to make a single pass over the list L and maintain a hash map that records the running counts for each first-field value that appears. As we read each pair (a, b) from the input, we simply look up a in the hash map and either increment its existing count or create a new entry initialized to 1. Since a hash table can insert and look up keys in expected constant time, this approach ensures that the entire traversal stays within the required expected $O(|L|)$ time bound. Moreover, the only memory we use apart from the loop variables is the space needed for the hash map itself, which ends up holding exactly one entry per distinct value of a that appears in L. Because each such value corresponds to exactly one element in the output, the memory footprint matches the required $O(|\text{result}|)$ bound. This approach also leaves L entirely untouched, since we merely iterate over it.

Algorithm 1 Aggregate(L)

input: list (or iterable) L of pairs (a, b)
output: set (or list) of pairs $(a, \text{count}(a, L))$
$H \leftarrow$ empty hash map from keys to integers
for each (a, b) in L **do**
    if H.contains(a) **then**
        $H[a] \leftarrow H[a] + 1$
    else
        $H[a] \leftarrow 1$
    end if
end for
return $\{(a, H[a]) \mid a \in H\}$

P2.2 Correctness and Complexity

The correctness of the algorithm follows directly from how the hash map is maintained. Every time we encounter a pair (a, b) in L, we update the counter associated with a, ensuring that by the end of the scan, the hash map entry $H[a]$ is exactly the number of times a appears as the first component in L. Keys that never occur are never inserted, so the output precisely matches the definition of $\text{aggr}(L)$.

The expected running time is $O(|L|)$ because the algorithm performs one constant-time hash-table operation per element in L. Constructing the final result from the hash map requires time proportional to the number of distinct keys, which is always at most $|L|$. The memory usage is also optimal: the algorithm only stores one entry per distinct first-field value, so the auxiliary memory footprint is exactly $O(|\text{result}|)$ as required. Even though we do not know the number of distinct keys ahead of time, modern hash tables resize dynamically, and the amortized cost of such resizing still preserves the expected $O(|L|)$ total runtime.

P2.3 Semi-Join Algorithm

To compute the semi-join SemiJoin(F, S, δ) efficiently, the key observation is that we only care about keys a for which (a, δ) appears in S. Therefore, the algorithm first scans S and collects exactly those keys into a hash set T. This set T fully encodes the condition needed for the semi-join. In the second phase, we scan the list F and simply check, for each pair (a, b), whether a is present in T. If it is, we append (a, b) to the result. This method processes every element of F and S only once and uses a hash set whose size is bounded by the number of keys in S, exactly meeting the requirement of at most $O(|S|)$ additional memory beyond the output.

Algorithm 2 SemiJoin(F,S,δ)

input: sets or lists F and S of pairs, value δ
output: $\{(a, b) \in F \mid (a, \delta) \in S\}$
$T \leftarrow$ empty hash set
for each (a, x) in S **do**
  if $x = \delta$ **then**
    insert a into T
  end if
end for
$R \leftarrow$ empty list
for each (a, b) in F **do**
  if $a \in T$ **then**
    append (a, b) to R
  end if
end for
return R

P2.4 Correctness and Complexity

This semi-join algorithm is correct because the hash set T contains exactly the keys a for which the pair (a, δ) exists in S. Therefore, when scanning F, checking whether a is in T is equivalent to checking whether (a, δ) is present in S. The algorithm outputs a pair (a, b) from F if and only if that condition holds, which matches the definition of the semi-join exactly.

The algorithm satisfies the required complexity bounds. The first pass over S does one constant-time hash-table operation per element, giving an expected $O(|S|)$ runtime. The second pass over F again performs a constant-time membership test per element, resulting in an additional $O(|F|)$ expected time. The total expected runtime is therefore $O(|F| + |S|)$. Memory usage is similarly controlled: the hash set T stores at most one entry per distinct key present in S, making its size $O(|S|)$, and no larger auxiliary structures are created. Aside from T and the output itself, the algorithm uses only a constant amount of additional memory, meeting the specification exactly.