ADS 2021: Week 6 Solutions

Solutions for week 6 of Algorithms and Data Structures.

3.1.10 - Green

```
key  value  first
 E     0    E0                                  0 compares
 A     1    A1 E0                               1 compare
 S     2    S2 A1 E0                            2 compares
 Y     3    Y3 S2 A1 E0                         3 compares
 Q     4    Q4 Y3 S2 A1 E0                      4 compares
 U     5    U5 Q4 Y3 S2 A1 E0                   5 compares
 E     6    U5 Q4 Y3 S2 A1 E6                   6 compares
 S     7    U5 Q4 Y3 S7 A1 E6                   4 compares
 T     8    T8 U5 Q4 Y3 S7 A1 E6                6 compares
 I     9    I9 T8 U5 Q4 Y3 S7 A1 E6             7 compares
 O    10    O10 I9 T8 U5 Q4 Y3 S7 A1 E6         8 compares
 N    11    N11 O10 I9 T8 U5 Q4 Y3 S7 A1 E6     9 compares

Total: 55 compares
```

3.1.11 - Green

```
                keys[]                                  vals[]
key  value      0  1  2  3  4  5  6  7  8  9    N   0  1  2  3  4  5  6  7  8  9
 E     0        E                                1   0                             0 cmp
 A     1        A  E                             2   1  0                          2 cmp
 S     2        A  E  S                          3   1  0  2                       2 cmp
 Y     3        A  E  S  Y                       4   1  0  2  3                    2 cmp
 Q     4        A  E  Q  S  Y                    5   1  0  4  2  3                 3 cmp
 U     5        A  E  Q  S  U  Y                 6   1  0  4  2  5  3              4 cmp
 E     6        A  E  Q  S  U  Y                 6   1  6  4  2  5  3              4 cmp
 S     7        A  E  Q  S  U  Y                 6   1  6  4  7  5  3              4 cmp
 T     8        A  E  Q  S  T  U  Y              7   1  6  4  7  8  5  3           4 cmp
 I     9        A  E  I  Q  S  T  U  Y           8   1  6  9  4  7  8  5  3        4 cmp
 O    10        A  E  I  O  Q  S  T  U  Y        9   1  6  9  10 4  7  8  5  3     4 cmp
 N    11        A  E  I  N  O  Q  S  T  U  Y     10  1  6  9  11 10 4  7  8  5  3  5 cmp
                A  E  I  N  O  Q  S  T  U  Y         1  6  9  11 10 4  7  8  5  3

Total: 38 compares
```

3.1.13 - Green

- Sequential search ST:
  - The 10^3 *put*() operations are all constant time and take 10^3 object allocations.
  - As there are 1000 *get*() operations for each *put*(), the expected number of elements in the ST at the first 1000 call to *get*() is 0.5, then 1.5 for the second, 2.5 for the third etc. Assuming only search hits, we expect to find the element on average at the

halfway point of the sequential search. *Sum[j*0.5, j, 0.5, 999.5, 1]* $*10^3 = 2.5 * 10^8$ compares.

- Binary search ST:
  - *put*() uses *sum log2(j), j=1 to* $10^3 = 8.5 * 10^3$ compares to put the elements, and allocates arrays with a total size of $2 * 2 * 2 * 10^3$ after all array resizings. (The first 2 is due to the binary search ST using two arrays, one for keys and one for values, the second because the sum is 2N, the third because we may have just doubled the size at the last insert and the array may therefore be twice the size of N).
  - For the 10^6 *get*() operations, the Binary search ST uses *Sum[log2(j*1.0), j, 0.5, 999.5, 1]* $*10^3 = 8.5 * 10^6$ compares.
- The binary search ST uses considerably fewer compares than the sequential search ST. It is also reasonable to consider the total memory allocation cost to be lower as the sequential search ST requires a minimum of 16 bytes for creating each Node object plus 2N references to keys and values and N *next* references.

3.1.14 - Green

- Sequential search ST:
  - The 10^6 *put*() operations are all constant time and take 10^6 object allocations.
  - As there are 1000 *put*() operations for each *get*(), the expected number of elements in the ST at the first call to *get*() is 500, then 1500 for the second, 2500 for the third etc. Assuming only search hits, we expect to find the element on average at the halfway point of the sequential search. *Sum[j*0.5, j, 500, 999500, 1000]* $= 2.5 * 10^8$ compares.
- Binary search ST:
  - *put*() uses *sum log2(j), j=1 to* $10^6 = 1.8 * 10^7$ compares to put the elements, and allocates arrays with a total size of $2 * 2 * 2 * 10^6$ after all array resizings. (The first 2 is due to the binary search ST using two arrays, one for keys and one for values, the second because the sum is 2N, the third because we may have just doubled the size at the last insert and the array may therefore be twice the size of N).
  - For the 10^3 *get*() operations, the Binary search ST uses *Sum[log2(j*1.0), j, 500, 999500, 1000]* $= 1.8 * 10^4$ compares.
- The binary search ST uses 10 times fewer compares than the sequential search ST. It is also reasonable to consider the total memory allocation cost to be lower as the sequential search ST requires a minimum of 16 bytes for creating each Node object plus 2N references to keys and values and N *next* references. But it is an interesting discussion.

3.4.1 - Green

```
key hash value
  E   0    0

0 E0
1 null
2 null
```

```
3 null
4 null

key hash value
  A   1    1

0 E0
1 A1
2 null
3 null
4 null

key hash value
  S   4    2

0 E0
1 A1
2 null
3 null
4 S2

key hash value
  Y   0    3

0 Y3 -> E0
1 A1
2 null
3 null
4 S2

key hash value
  Q   2    4

0 Y3 -> E0
1 A1
2 Q4
3 null
4 S2




key hash value
  U   1    5

0 E0 -> Y3
1 U5 -> A1
2 Q4
3 null
```

```
4 S2

key hash value
  T   0    6

0 T6 -> E0 -> Y3
1 U5 -> A1
2 Q4
3 null
4 S2

key hash value
  I   4    7

0 T6 -> E0 -> Y3
1 U5 -> A1
2 Q4
3 null
4 I7 -> S2

key hash value
  O   0    8

0 O8 -> T6 -> E0 -> Y3
1 U5 -> A1
2 Q4
3 null
4 I7 -> S2

key hash value
  N   4    9

0 O8 -> T6 -> E0 -> Y3
1 U5 -> A1
2 Q4
3 null
4 N4 -> I7 -> S2
```

3.4.10 - Green

```
M = 16

key hash value
  E   7    0

0 null      8 null
1 null      9 null
2 null     10 null
3 null     11 null
4 null     12 null
```

```
5 null     13 null
6 null     14 null
7 E0       15 null

key hash value
  A   11   1

0 null      8 null
1 null      9 null
2 null     10 null
3 null     11 A1
4 null     12 null
5 null     13 null
6 null     14 null
7 E0       15 null

key hash value
  S    1   2

0 null      8 null
1 S2        9 null
2 null     10 null
3 null     11 A1
4 null     12 null
5 null     13 null
6 null     14 null
7 E0       15 null

key hash value
  Y    3   3

0 null      8 null
1 S2        9 null
2 null     10 null
3 Y3       11 A1
4 null     12 null
5 null     13 null
6 null     14 null
7 E0       15 null

key hash value
  Q   11   4

0 null      8 null
1 S2        9 null
2 null     10 null
3 Y3       11 A1
4 null     12 Q4
5 null     13 null
```

```
6 null     14 null
7 E0       15 null


key hash value
  U    7   5

0 null      8 U5
1 S2        9 null
2 null     10 null
3 Y3       11 A1
4 null     12 Q4
5 null     13 null
6 null     14 null
7 E0       15 null

key hash value
  T   12   6

0 null      8 U5
1 S2        9 null
2 null     10 null
3 Y3       11 A1
4 null     12 Q4
5 null     13 T6
6 null     14 null
7 E0       15 null

key hash value
  I    3   7

0 null      8 U5
1 S2        9 null
2 null     10 null
3 Y3       11 A1
4 I7       12 Q4
5 null     13 T6
6 null     14 null
7 E0       15 null

key hash value
  O   5    8

0 null      8 U5
1 S2        9 null
2 null     10 null
3 Y3       11 A1
4 I7       12 Q4
5 O8       13 T6
```

```
6 null     14 null
7 E0       15 null


key hash value
  N   10   9

0 null      8 U5
1 S2        9 null
2 null     10 N9
3 Y3       11 A1
4 I7       12 Q4
5 O8       13 T6
6 null     14 null
7 E0       15 null

##### M = 10 #####

key hash value
  E   5    0

0 null      5 E0
1 null      6 null
2 null      7 null
3 null      8 null
4 null      9 null

key hash value
  A   1    1

0 null      5 E0
1 A1        6 null
2 null      7 null
3 null      8 null
4 null      9 null

key hash value
  S   9    2

0 null      5 E0
1 A1        6 null
2 null      7 null
3 null      8 null
4 null      9 S2

key hash value
  Y   5    3

0 null      5 E0
```

```
1 A1        6 Y3
2 null      7 null
3 null      8 null
4 null      9 S2

key hash value
  Q   7    4

0 null      5 E0
1 A1        6 Y3
2 null      7 Q4
3 null      8 null
4 null      9 S2

key hash value
  U   1    5

0 null      5 E0
1 A1        6 Y3
2 U5        7 Q4
3 null      8 null
4 null      9 S2

key hash value
  T   0    6

0 T6        5 E0
1 A1        6 Y3
2 U5        7 Q4
3 null      8 null
4 null      9 S2

key hash value
  I   9    7

0 T6        5 E0
1 A1        6 Y3
2 U5        7 Q4
3 I7        8 null
4 null      9 S2

key hash value
  O   5    8

0 T6        5 E0
1 A1        6 Y3
2 U5        7 Q4
3 I7        8 O8
4 null      9 S2
```

```
key hash value
  N   4    9

0 T6        5 E0
1 A1        6 Y3
2 U5        7 Q4
3 I7        8 O8
4 N9        9 S2
```

3.4.4 - Yellow Given as Python code below. Note that (xM+y)k % M = xMk % M + yk % M = 0 + yk % M = yk % M when y < M, and it is therefore only necessary to check for a in the range 1 to M.

```
def perfect_hash(keys):
    M = len(keys) # M must at least be large enough to fit all the keys
    while (True):
        for a in range(1, M+1): # See above
            hash_set = set()
            for key in keys:
                hash_set.add(akm_hash(a, key, M))

            if len(hash_set) == len(keys): # No collisions
                return (a, M)
        M += 1

def akm_hash(a, k, m):
    return a * k % m

if __name__ == "__main__":
    test_keys = {19, 5, 1, 18, 3, 8, 24, 13, 16, 12} # char values
    print(perfect_hash(test_keys))
```

3.4.15 - Yellow

```
In the worst case all keys hash to the same index.

The number of compares per insert is:
1 for the first insert, 2 for the second insert, 3 for the third insert
and so on, until the (N/2)th insert. When the table is half full it is
resized to 2N and the keys are reinserted, with 1, 2, 3, ..., N/2 compares.

Then, for the insert of the other keys there are N/2 + 1, N/2 + 2, ..., N
compares per insert.

This is equal to:
Number of compares = (1 + 2 + 3 + ... + N/2) + 1 + 2 + 3 + ...
                      + N/2 + (N/2 + 1) + (N/2 + 2) + ... + N
Number of compares = (N/2 + 1) * N / 2 / 2 + (N + 1) * N / 2
```

```
Number of compares = (N^2/2 + N) / 4 + (N^2 + N) / 2
Number of compares = (N^2/2 + N) / 4 + (2N^2 + 2N) / 4
Number of compares = (N^2/2 + (2N / 2)) / 4 + (2N^2 + 2N) / 4
Number of compares = (N^2 + 2N) / 8 + (4N^2 + 4N) / 8
Number of compares = (5N^2 + 6N) / 8

In the worst case, to insert N keys into an initially empty table, using
linear probing with array resizing it would take (5N^2 + 6N) / 8 compares.
```

3.4.26 - Yellow The following changes to the LinearProbingHashST are required:

- Class: add *int deletedKeys*
- put(): update the resize check to check for (size + deletedKeys). If the key to be inserted is in the table already with the value null, decrement deletedKeys.
- resize(): make sure to only copy over keys that have a non-null value and reset deletedKeys to 0.
- lazyDelete(): Find the key as usual, then set its value to null, increment deletedKeys and decrement size.
- get(): No change required, it returns null when not found, and null as the value when the key has been deleted.

3.4.6 - Red Consider the hash function k % M where k is a binary integer and M is a prime number larger than 2.

Every time that we modify exactly one bit in an integer, we either add (when modifying from 0 to 1) or subtract (when modifying from 1 to 0) a value that is a power of 2 (0, 1, 2, 4, etc). Since we are never adding or subtracting a prime number (other than 2) or any prime number multiples, the modular hash function with a prime M (other than 2) will yield a different result for both numbers.

3.4.16 - Red In a linear-probing hash table, later indices have a slightly higher chance of being filled than earlier indices. But as the set of indices divisible by 100 is evenly spread through the n indices, this is likely to be a small error. And note that the question asks for an estimate, not an exact number.

Therefore, with half the indices randomly filled, there is a $(\frac{1}{2})^{10^4}$ chance that all 10^4 indices divisible by 100 is filled.