CSCI 230 -- Homework 1

Nero Li

Chapter 9

- R-9.1

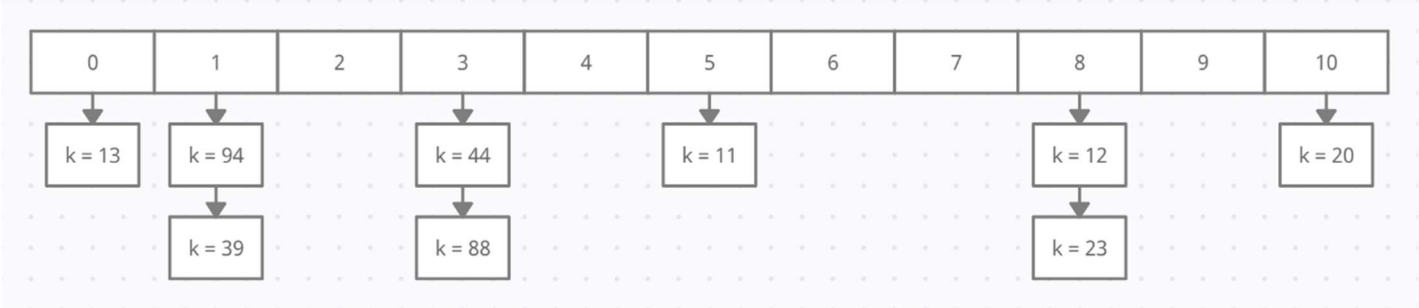
Separate chaining could work with a load factor above 1 and **open addressing** could not.

- R-9.6

Since there are several repeated characters, **cyclic shift hash codes** would work better.

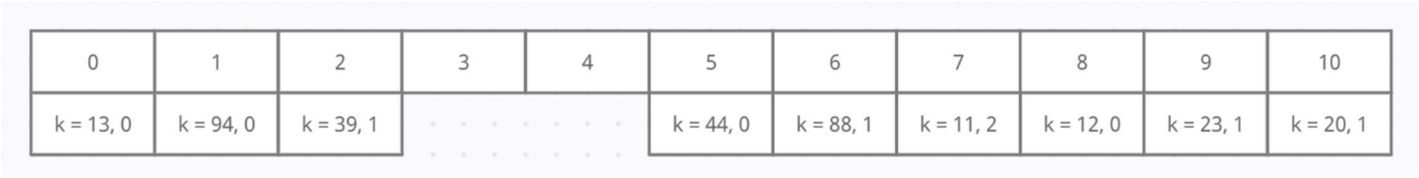
- R-9.7

k	12	44	13	88	23	94	11	39	20
hash(k)	8	5	0	5	8	1	5	1	10



- R-9.8

k	12	44	13	88	23	94	11	39	20
hash(k)	8	5	0	5+1	8+1	1	5+2	1+1	10



- R-9.10

k	h0(k)	h1(k)	h2(k)	h3(k)	h4(k)	h5(k)
12	8					
44	5					
13	0					
88	5	7				
23	8	0	1			
94	1	9				
11	5	7	10			
39	1	6				
20	10	5	10	8	8	3

0	1	2	3	4	5	6	7	8	9	10
k = 13, 1	k = 23, 3		k = 20, 6		k = 44, 1	k = 39, 2	k = 88, 2	k = 12, 1	k = 94, 2	k = 11, 3

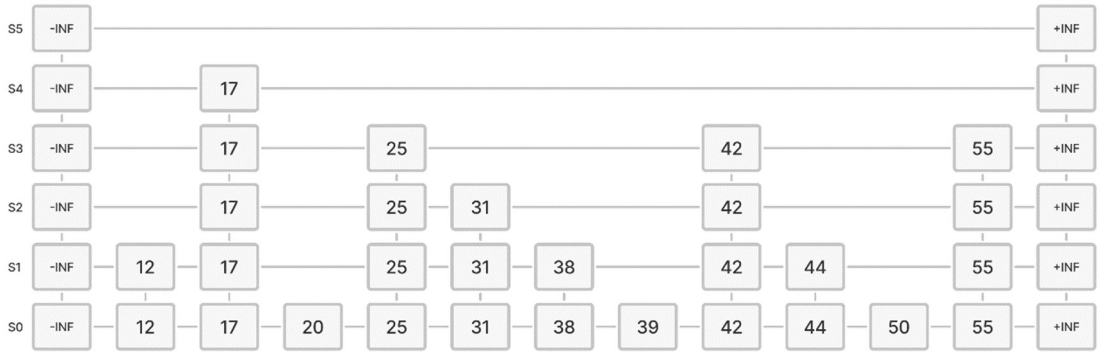
- R-9.14

One of the most important thing for Dictionary ADT is that it has the situation that the same key indicates two different values, and it allows us to use findAll(k) function to see the entries. If we use hash code, those same keys will be hard for hash function to count as collisions or legal entries. Then, it will operate much worse and increase the running time when we call functions.

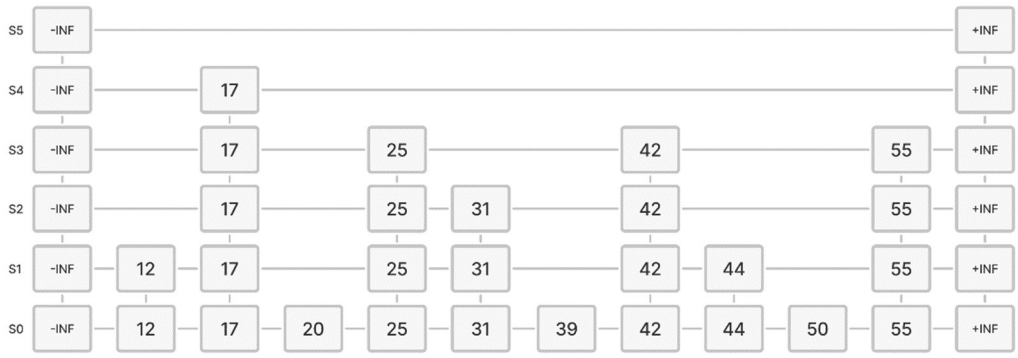
- R-9.15

The **worst case is O(n)** while all the items are in the same bucket; the **best case is O(1)** while all the items are in different buckets.

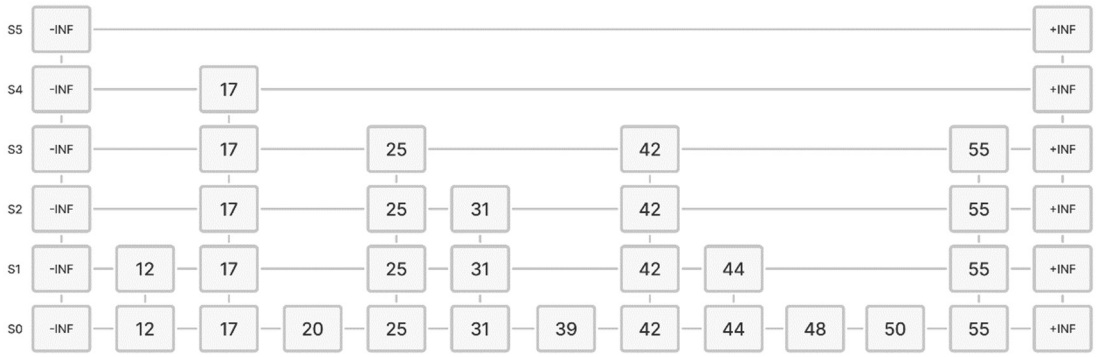
- R-9.16



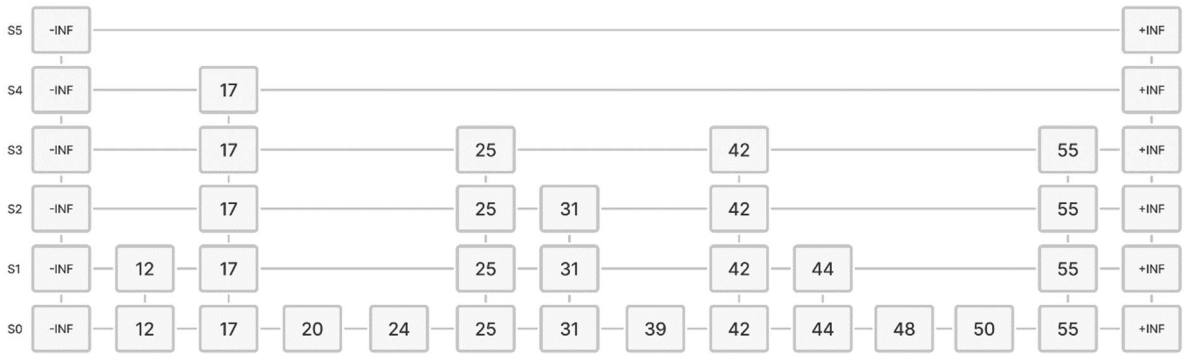
erase(38)



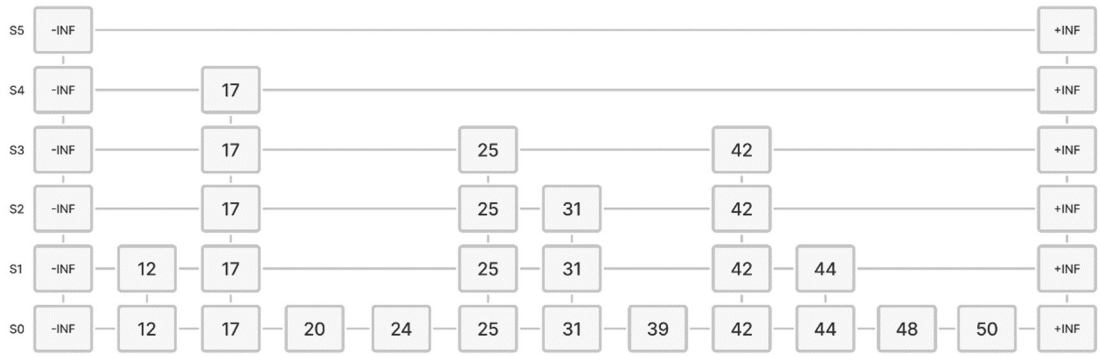
insert(48, x) – Coin flips: 0 heads



insert(24, y) – Coin flips: 1 head



erase(55)



- C-9.3

To find the kth smallest, we first get the ith element in S where i = k/2, search that element in T by binary search, it will terminate index j where the value is lower than the element we have. Then, we get the new index t = i + j + 1 and see if it is equal to k or not. This will take us O(logn) to finish this operation.

If not, we get the element we found in T at index j, and search that element in S at the right section after index i element. We found the new element and mark its index as i, and then do t = i + j + 1, we will definitely get t = k. This will take us O(logn) to finish this operation.

Finally, check the element in S with index i and element in T with index j, find the smallest one and then output the answer.

- C-9.5

First, do binary search in that ordered array to find the kth element that the function want. This will take us O(logn) time to finish this operation where n is the number of elements in the dictionary.

Then, return all the related values that the iterator linked. This will take us O(s) time to finish this operation where s is the size of the iterator returned.

Hence, we will need O(logn + s) to finish the process.

- C-9.14

For bag ADT, we create a vector to store all elements in our bag.

For add(k) function, we just push back an element or put an element at the index that we removed element, and it will take us O(1) to finish this process.

For remove() function, we generate a random number that in the range of the vector size that are not be removed. We will create another data pool to store the index that we didn’t remove, and the random function will get one index from that data pool to indicate that the element at that index has been removed. This will take us O(1) to finish this process.

Chapter 11

- R-11.5

Downward arrow means it is **calling itself for recursion**; upward arrow means **the function ends or returns and go back to the up-level function that called itself before**.

- R-11.8

At first, we create a new vector to store our answer for the union of A and B. Then we create a loop until we go through all the values in both A and B. We will have three different situations inside the loop:

1. If A[i] == B[j], we choose one value, then prepare to push and do ++i and ++j.
2. If A[i] < B[j], we prepare to push A[i] into our answer vector and do ++i.
3. If A[i] > B[j], we prepare to push B[j] into our answer vector and do ++j.

Before we push our value into the vector, check the top value is not the same as the value we prepare to push. If it is the same, do not push but still do add up for i or j. If not, do push and keep rolling.

For the index i and j, if one of them reach the top but the other doesn't, prepare to push all the remaining value until both i and j become the same as A.size() and B.size().

This is the same operation as the merge() function but we add one more operation to check the duplicate elements. Because of this, it will take us O(n) where n is the bigger size for A and B.

- R-11.10

If we directly choose the element at the middle, the running time of this version of quick-sort should be **stabled in O(nlogn)** since we always choose the median value for the whole vector, and that median value will make sure our two sub vectors have almost the same size, and it will become much faster than merge sort.

- R-11.11

If we have **a sequence that the middle of the elements are always the largest or smallest element in our whole array**, our quick sort will run in n Ω(n^2) time since every time we do the recursion, we will get two sub vectors that one with size 0 and another with size n – 1.

- R-11-18

The merge-sort that has shown in Section 11.1 should be **stable**.

Whatever our collection of data is, we will divide them into several different groups with 1 or 0 elements, and then conquer them by compare each sub vector's data. The operation for us to separate our data won't be affected by the looking of our collection, and for our merge part, we at least need to go over all the values in both arrays, so it can be seen as stable.

- R-11-21

Bucket-sort is **not** in-place.

If you want to use bucket-sort, you need to create several additional buckets for you to store the data and sort them in buckets. The number of buckets depends on the code implement, and it is not a fixed number.

- C-11.5

We do the merge sort for this collection A. When we start the merge function, if we saw two values that are the same in two vectors, ignore one of the value to make sure it didn't push into our new vector twice.

With this modification during the merge function, we can do the function in O(nlogn) just like what a general merge sort should cost.

- C-11.23

First of all, we do the merge sort for both sequence A and B, and it will take us O(nlogn) times to get two sorted sequences.

Then, we create index i for A and index j for B. Do A[i] + B[j] and compare the result with m:

1. If sum == m, we found the answer, return true.
2. If sum > m, we cannot get any number that can equal to m, return false.
3. If sum < m, compare A[i + 1] and B[j + 1] and see which one is smaller, do ++ for the index that point to a smaller value, and add A[i] + B[j] to get a new answer.

This work looks pretty the same as the work in merge function. Because of that, we will take O(logn) to finish this part.

Finally, we have O(nlogn + nlogn + logn) and it means we will finish the work in O(nlogn).