FNU ANIRUDH

Assignment-5

① $A = \langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$

A:	6	0	2	0	1	3	4	6	1	3	2

We build an array of counts:

	0	1	2	3	4	5	6
C:	2	2	2	2	1	0	2

The number of elements before each

	0	1	2	3	4	5	6
C:	2	4	6	8	9	9	11

Then we start iterating (11)

	1	2	3	4	5	6	7	8	9	10	11
A:						2					

C:	0	1	2	3	4	5	6
	2	4	5	8	9	9	11

A:						2		3			

C:	2	4	5	7	9	9	11

A:				1		2		3			

C:	2	3	5	7	9	9	11

A:				1		2		3			6

C:	2	3	5	7	9	9	10

A:				1		2		3	4		6

C:	2	3	5	7	8	9	10

A:				1		2	3	3	4		6

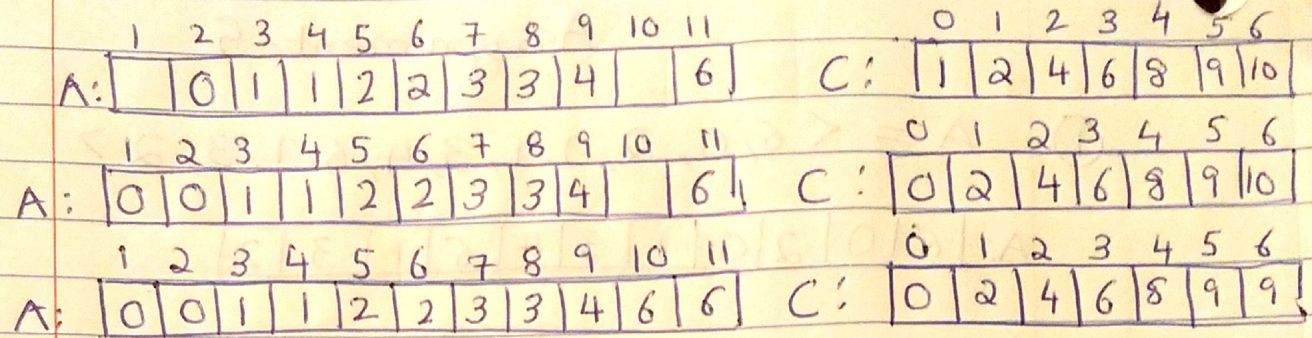
C:	2	3	5	6	8	9	10

A:			1	1		2	3	3	4		6

C:	2	2	5	6	8	9	10

A:		0	1	1		2	3	3	4		6

C:	1	2	5	6	8	9	10



(2) The algorithm is correct no matter what order is used but modified algorithm is not stable. As before, in the final for loop an element equal to one taken from A earlier is placed before the earlier one (i.e. at lower index position) in the output array B.

The original algorithm was stable because an element taken from A later started out with a lower index than one taken earlier. But in the modified algorithm, an element taken from A later started out with a higher index than one taken earlier.

In particular, the algorithm still places the elements with value K in positions $C[K-1]+1$ through $C[K]$ but in the reverse order of their appearance in A.

(3) Treat the numbers as 2-digit numbers in radix n. Each digit ranges from 0 to $n-1$.
Sorting these 2-digit numbers with radix sort.

There are 2 calls to counting sort, each taking $\Theta(n+n) = \Theta(n)$ time, so that the total time is $\Theta(n)$.

④ We split the elements into pairs, compare each pair and then proceed to compare the winners in the same fashion. We need to keep track of each "match" the potential winners have participated in. (Tournament Style)

We select a winner in n-1 matches. At this point, we know that second smallest element is one of the lg n elements that lost to the smallest each of them is smaller than the one it has been compared to, prior to losing. In another $\lceil \lg n \rceil - 1$ comparisons we can find the smallest element out of those. Hence this is what we were looking for.

⑤
```
function randomized select (*A, p, r, i)
 while (p < r-1)
    q ← randomized_partition (A, p, r)
    k ← q-p
 if (i ← k)
  return A[q]
else if (i < k)
    r ← q
else
    p ← q+1
    i ← i-k-1
return A[p]
```

⑥ This problem can be reduced to the following inequality:

$$\frac{3n}{10} - 6 \geq \left\lceil \frac{n}{4} \right\rceil$$

$$\downarrow$$

$$\frac{3n}{10} - 6 \geq \frac{n}{4} + 1$$

$$\downarrow$$

$$\frac{3n}{10} - 7 \geq \frac{n}{4}$$

$$\downarrow$$

$$12n - 280 \geq 10n$$

$$2n \geq 280$$

$$n \geq 140$$

(7) For each pair of keys k, l where $k \neq l$, define the indicator random variable $X_{kl} = I\{h(k) = h(l)\}$. Since we assume simple uniform hashing, $Pr\{X_{kl} = 1\} = Pr\{h(k) = h(l)\} = 1/m$, and so $E[X_{kl}] = 1/m$.

Now define the random variable Y to be the total number of collisions, so that $Y = \sum_{k \neq l} X_{kl}$. The expected number of collisions is

$$E[Y] = E\left[\sum_{k \neq l} X_{kl}\right]$$

$$= \sum_{k \neq l} E[X_{kl}] \quad \text{(linearity of expectation)}$$

$$= \binom{n}{2} \frac{1}{m}$$

$$= \frac{n(n-1)}{2} \cdot \frac{1}{m}$$

$$= \frac{n(n-1)}{2m}$$

(§) The flag in each slot will indicate whether the slot is free

→ A free slot is in the free list, a doubly linked list of all free slots in the table. The slot thus contains two pointers.

→ A used slot contains an element and a pointer (possibly NIL) to the next element that hashes to this slot (Pointer points to another slot in the table.)

Operations

- Insertion

If the element hashes to a free slot, just remove the slot from the free list and store the element there (with a NIL pointer). The free list must be doubly linked in order for this deletion to run in $O(1)$ time.

→ If the element hashes to a used slot j, check whether the element x already there "belongs" there (its key also hashes to slot j).

- If so, add the new element to the chain of elements in this slot. To do so, allocate a free slot (e.g., take the head of the free list) for the new element and put this new slot at the head of the list pointed to by the hashed-to-slot(j).

- If not, E is part of another slot's chain. Move it to a new slot by allocating one from the free list, copying the old slot's (j's) contents (element x and pointer) to the new slot, and updating the pointer in the slot that pointed to j to point to the new slot. Then insert the new element in the now-empty slot as usual.

To update the pointer to j, it is necessary to find it by searching the chain of elements starting in the slot x hashed to.

Deletion: Let j be the slot the element x to be deleted hashed to.

→ If x is the only element in j (j doesn't point to any other entries) just free the slot, returning it to the head of the free list.

→ If x is in j but there's a pointer to a chain of other elements, move the first pointed-to entry to slot j and free the slot it was in.

→ If x is found by following a pointer from j, just free x's slot and splice it out of the chain

(i.e. update the slot that pointed to x to point x's successor).

Searching: Check the slot the key hashed to, and if that is not the desired element, follow the chain of pointers from the slot.

All the operations take expected $O(1)$ times for the same reason they do with the version ~~in the~~ mentioned in Cormen. The expected time to search the chains is $O(1+\alpha)$ regardless of where the chains are stored, and the fact that all the elements are stored in the table means that $\alpha \leq 1$. If the free list were singly linked, then operations that involved removing an arbitary slot from the free list would not run in $O(1)$ time.

(9) While searching for a key K, we will first check if the hash value matches if yes then only we will check if the keys actually match or not.

(10) First, we observe that we can generate any permutation by a sequence of inter-changes of pairs of characters. It is easy to proof this formally but informally, consider that both heapsort and quicksort work by interchanging pairs of elements and that they have to be able to produce any permutation of their input array. Thus, it suffices to show that if string x can be derived from string y by

interchanging a single pair of characters, then x and y hash to the same value.

Let us denote the ith character in x by x_i, and similarly for y. The interpretation of x in radix 2^p is $\sum_{i=0}^{n-1} x_i 2^{ip}$, and so $h(x) = \left(\sum_{i=0}^{n-1} x_i 2^{ip}\right) \bmod (2^p - 1)$

Likewise, $h(y) = \left(\sum_{i=0}^{n-1} y_i 2^{ip}\right) \bmod (2^p - 1)$

Suppose that x and y are identical strings of n characters in positions a and b are interchanged: $x_a = y_b$ and $y_a = x_b$. Without loss of generality, let $a < b$. We have

$$h(x) - h(y) = \left(\sum_{i=0}^{n-1} x_i 2^{ip}\right) \bmod (2^p - 1) - \left(\sum_{i=0}^{n-1} y_i 2^{ip}\right) \bmod (2^p - 1)$$

Since $0 \le h(x), h(y) < 2^p - 1$, we have that $-(2^p - 1) < h(x) - h(y) < 2^p - 1$.

If we show that $(h(x) - h(y)) \bmod (2^p - 1) = 0$, then $h(x) = h(y)$

Since the sums in the hash functions are the same except for indices a and b we have

$$(h(x) - h(y)) \bmod (2^p - 1)$$

$$= ((x_a 2^{ap} + x_b 2^{bp}) - (y_a 2^{ap} + y_b 2^{bp})) \bmod (2^p - 1)$$

$$= ((x_a 2^{aP} + x_b 2^{bP}) - (x_b 2^{aP} + x_a 2^{bP})) \bmod (2^P$$

$$= ((x_a - x_b) 2^{aP} - (x_a - x_b) 2^{bP}) \bmod (2^P - 1)$$

$$= ((x_a - x_b)(2^{aP} - 2^{bP})) \bmod (2^P - 1)$$

$$= ((x_a - x_b) 2^{bP} (2^{(a-b)P} - 1)) \bmod (2^P - 1)$$

$$\sum_{i=0}^{a-b-1} 2^{Pi} = \frac{2^{(a-b)P} - 1}{2^P - 1}$$

and multiplying both sides by $2^P - 1$, we get

$$2^{(a-b)P} - 1 = \left(\sum_{i=0}^{a-b-1} 2^{Pi}\right)(2^P - 1)$$

Thus,

$$(h(x) - h(y)) \bmod (2^P - 1)$$

$$= \left((x_a - x_b) 2^{bP} \left(\sum_{i=0}^{a-b-1} 2^{Pi}\right)(2^P - 1)\right) \bmod (2^P - 1)$$

$$= 0$$

since one of the factors is $2^P - 1$.

We have shown that $(h(x) - h(y)) \bmod (2^P - 1) =$ and so $h(x) = h(y)$.