1 8.3-4

Problem
Show how to sort $n$ integers in the range $0$ to $n^3 - 1$ in $O(n)$ time.

Solution
The solution is easily obtained by considering numbers to the base $n$. Note that in a general base-$k$ number system, numbers in the range $0$ to $R - 1$ can be represented using $d = \lceil \log_k(R) \rceil$ digits. Now if we set $k = n$ and $R = n^3$, we see that we can represent $n^3$ numbers using $d = \log_n(n^3) = 3$ digits only. The numbers can be represented in this $n$-base system by solving for $a$, $b$ in $x = a \cdot n^2 + b \cdot n^1 + c \cdot n^0$, where $0 \leq a, b, c \leq n - 1$ and the number in $n$-base system is $(abc)$. For example, the (largest) number $n^3 - 1$ can be written as

$$(n - 1) \cdot n^2 + (n - 1) \cdot n + n - 1 = (n - 1) \frac{n^3 - 1}{n - 1}$$

and thus in $n$-base system the representation is $(n - 1 n - 1 n - 1)$.

We can now apply RadixSort to the converted 3-digit base $n$ numbers in $\Theta(d(n + k)) = \Theta(3(n + n)) = \Theta(n)$ time.

For general base $k$, the conversion into base $k$ numbers involves iterative mod $k$ operations (essentially division by $k$). This is expensive. However when $k$ is a power of 2 ($k = 2^r$) and your original number is given to you in binary then the conversion comes essentially for free: the $i$-last digit is the number represented by the $i$-last block of $r$ bits.

This trick is known to you from converting binary numbers into base 16 (a.k.a. Hexadecimal) numbers. In that case blocks of 4 bits form the base 16 digits 0123456789ABCDEF.

\footnote{We used the geometric sum formula.}
Problem

We wish to implement a dictionary by using direct addressing on a huge array. At the start, the array entries may contain garbage, and initializing the entire array is impractical because of its size. Describe a scheme for implementing a direct-address dictionary on a huge array. Each stored object should use $O(1)$ space; the operations SEARCH, INSERT, and DELETE should take $O(1)$ time each; and initializing the data structure should take $O(1)$ time. (Hint: Use an additional array, treated somewhat like a stack whose size is the number of keys actually stored in the dictionary, to help determine whether a given entry in the huge array is valid or not.)

Solution

We use a huge array $T$ and an verifier array $V$ together. While the array $V$ could grow up to maximum size of array $T$, the size changes dynamically as keys are added/deleted. When array $T$ is allocated, no attempt is made to initialize the entries. Instead we use the verifier array $V$ to verify the entries in $T$ as follows.

Let $n_V$ be the number of elements in array $V$, (which is same as number of keys in $T$). Let the first entry be at index 1. When adding a new object $x$ with key $k_{\text{new}}$ to $T$, we add a reference to the object to $V$, at index $j = n_V + 1$. Then we add the object $x$ to the array $T$ at location $x\cdot\text{key}$, i.e., $T[x\cdot\text{key}] = x$ and $x\cdot\text{verifier} = j$. Note that we assume existence of a field $\text{verifier}$ in object $x$. It is important to check that entry in $T$ and the corresponding entry in verifier $V$, reference each other and this cycle of reference provides us with required verification we need. When this verification is successful, we know we are dealing with a legitimate entry.

When we lookup an object with key $k$, we check for this verification ($T[k] = x$, $x\cdot\text{verifier} = j$ and $V[j] = x$ with $x\cdot\text{key} = k$) and only when the verification is successful, we obtain the object.

Deletion, of key $k$ for example, is a bit tricky. We need to perform three tasks. First is to break the verification cycle. This is pretty trivial, We just need to set the $T[k] = 0$. However the entry $j$ correponding to key $k$ in $V$ is still there. We cant just set the value to 0, as it would leave an empty space in $V$. We fix this problem by exchanging this entry with the last entry, i.e., exchange $V[j]$ with $V[n_V]$. Finally we need to fix the verification cycle for the object we just moved into index $j$. Note that all this is constant time effort.

The three key operations of a dictionary are implemented as follows.

```java
// insert object x, with key x.key into the dictionary T
Insert(T, x) {
    n_V = n_V + 1
    // adding the up reference to the array T
    V[n_V] = ptr to x
    T[x.key] = x
```
/* add a down reference to the verifier. we assume a field in object x */
x.verifer = n_V
}

// return the object corresponding to the key k. If none found return null.
Search(T, k) {
    x = T[k]
    j = x.verifer
    if ( j < 1 or j > n_V )
        return null
    obj = V[j]
    // verify that verifier points to the same object
    if ( obj.key == k )
        return obj
    else
        return null
}

// delete the key 'k' from dictionary T
Delete(T, k) {
    x = T[k]
    // reset the entry in T
    T[k] = 0
    last = n_V
    lastObj = T[last]
    // delete the entry in V..
    exchange V[j] <--> V[last]
    n_V = n_V - 1
    // fix the verifier for the lastObj
    lastObj.verifier = j
}

9.2-3

Problem
Write an iterative version of RANDOMIZED-SELECT().

Solution
// The goal is to return the ith smallest element, 1<= i <= n in the array A.
// begin is the starting index of the sub array that we currently are considering
// end is last index of the sub array.
// i is the rank of the element we are looking for in the sub array A[begin..end]
RANDOMIZED-SELECT(A, i) {
begin = 1
end = A.size
// they are two conditions for the loop to terminate.
// 1. when the desired element is the pivot. (i == k)
// 2. when the array has size 1. begin == end.
while (begin < end) {
    q = RANDOMIZED-PARTITION(A, begin, end)
    k = q - begin + 1
    if (i == k)
        return A[q]
    else if (i < k)
        end = q-1
    else
        begin = q+1
        i = i - k
}

// control comes here only if begin == end.
return A[begin]

11.3.-4

Problem
Consider a hash table of size m = 1000 and a corresponding hash function h(k) = \lfloor(m(kA \mod 1)) \rfloor for \( A = \frac{\sqrt{5} - 1}{2} \). Compute the locations to which the keys 61, 62, 63, 64, and 65 are mapped.

Solution
We are given that \( A = \frac{\sqrt{5} - 1}{2} \approx 0.618 \).

For k = 61. \( kA \mod 1 = 61 \times 0.618 \mod 1 = 37.698 \mod 1 = 0.698 \).
Note that \( kA \mod 1 \) means we take the fractional part of \( kA \). h(61) = \lfloor (m(kA \mod 1)) \rfloor = \lfloor 1000 \times 0.698 \rfloor = 698. \) Thus we have T[61] = 698.

For k = 62. \( kA \mod 1 = 62 \times 0.618 \mod 1 = 0.316 \). So h(62) = \lfloor 1000 \times 0.316 \rfloor = 316. So T(62) = 316.

For k = 63. \( 63 \times 0.618 \mod 1 = 0.934 \). So T(63) = 934.

For k = 64. \( 64 \times 0.618 \mod 1 = 0.552 \). So T(64) = 552.

For k = 65. \( 65 \times 0.618 \mod 1 = 0.17 \). So T(65) = 170.

Notice how consecutive input numbers are spread across the table.
11.4-1

Problem

Consider inserting the keys 10, 22, 31, 4, 15, 28, 17, 88, 59 into a hash table of length $m = 11$ using open addressing with the auxiliary hash function $h'(k) = k$. Illustrate the result of inserting these keys using linear probing, using quadratic probing with $c_1 = 1$ and $c_2 = 3$, and using double hashing with $h_1(k) = k$ and $h_2(k) = 1 + (k \mod (m - 1))$.

Solution

With linear probing, we use the hash function $h(k, i) = (h'(k) + i) \mod m = (k + i) \mod m$.

- $h(10, 0) = (10 + 0) \mod 11 = 10$. Thus we have $T[10] = 10$.
- $h(22, 0) = (22 + 0) \mod 11 = 0$. Thus we have $T[0] = 22$.
- $h(31, 0) = (31 + 0) \mod 11 = 9$. Thus $T[9] = 31$.
- $h(4, 0) = (4 + 0) \mod 11 = 4$. Thus $T[4] = 4$.
- $h(15, 0) = (15 + 0) \mod 11 = 4$. Since $T[4]$ is occupied, we probe again, $h(15, 1) = 5$. Thus $T[5] = 15$.
- $h(28, 0) = (28 + 0) \mod 11 = 6$. Thus $T[6] = 28$.
- $h(17, 0) = (17 + 0) \mod 11 = 6$. Since $T[6]$ is occupied, we probe again. $h(17, 1) = 7$. Thus $T[7] = 17$.
- $h(88, 0) = (88 + 0) \mod 11 = 0$. $T[0]$ is occupied, so probe again. $h(88, 1) = 1$. Thus $T[1] = 88$.
- $h(59, 0) = (59 + 0) \mod 11 = 4$. $T[4]$ is occupied. $h(59, 1) = 5, h(59, 2) = 6, h(59, 3) = 7$ are all occupied. Probing the fourth time, $h(59, 4) = 8$ works. Thus $T[8] = 59$.

The final hash table is as shown in figure 1a.

With quadratic hashing, we use the hash function $h(k, i) = (h'(k) + i + 3i^2) \mod m = (k + i + 3i^2) \mod m$.

- $h(10, 0) = (10 + 0 + 0) \mod 11 = 10$. Thus we have $T[10] = 10$.
- $h(22, 0) = (22 + 0 + 0) \mod 11 = 0$. Thus we have $T[0] = 22$.
- $h(31, 0) = (31 + 0 + 0) \mod 11 = 9$. Thus $T[9] = 31$.
- $h(4, 0) = (4 + 0 + 0) \mod 11 = 4$. Thus $T[4] = 4$.
- $h(15, 0) = (15 + 0 + 0) \mod 11 = 4$. Since $T[4]$ is occupied, we probe again, $h(15, 1) = (15 + 1 + 3) \mod 11 = 8$. Thus $T[8] = 15$. 

5
Figure 1: (a) Hash table using linear probe, with hash function $h(k, i) = (k + i) \ mod \ m$. (b) Hash table using quadratic probe, with hash function $h(k, i) = (k + i + 3i^2) \ mod \ m$. (c) Hash table using double hashing, with hash function $h(k, i) = (k + i + 3i^2) \ mod \ m$. 
\[ h(28, 0) = (28 + 0 + 0) \mod 11 = 6. \text{ Thus } T[6] = 28. \]

\[ h(17, 0) = (17 + 0 + 0) \mod 11 = 6. \text{ Since } T[6] \text{ is occupied, we probe again. } h(17, 1) = 10. \text{ We probe again as } T[10] \text{ is occupied. Similarly } h(17, 2) = 9 \text{ does not work. Finally the insert succeeds with } h(17, 3) = 3. \text{ Thus } T[3] = 17. \]

\[ h(88, 0) = (88 + 0 + 0) \mod 11 = 0. \text{ T[0] is occupied, so probe again. } h(88, 1) = 4, h(88, 2) = 3, h(88, 3) = 8, h(88, 4) = 8, h(88, 5) = 3, h(88, 6) = 4, h(88, 7) = 0 \text{ do not work. We finally succeed with } h(88, 8) = 2. \text{ Thus } T[2] = 88. \]

\[ h(59, 0) = (59 + 0 + 0) \mod 11 = 4. \text{ T[4] is occupied. We probe again. } h(59, 1) = 7. \text{ Thus } T[7] = 59. \]

The final hash table is as shown in figure 1b.

With double hashing, we use the hash function \( h(k, i) = (h_1(k) + ih_2(k)) \mod m = (k + \{1 + k \mod (m - 1)\}) \mod m. \)

\[ h(10, 0) = (10 + 0 \cdot h_2(10)) \mod 11 = 10. \text{ Thus we have } T[10] = 10. \]

\[ h(22, 0) = (22 + 0 \cdot h_2(22)) \mod 11 = 0. \text{ Thus we have } T[0] = 22. \]

\[ h(31, 0) = (31 + 0 \cdot h_2(31)) \mod 11 = 9. \text{ Thus } T[9] = 31. \]

\[ h(4, 0) = (4 + 0 \cdot h_2(4)) \mod 11 = 4. \text{ Thus } T[4] = 4. \]

\[ h(15, 0) = (15 + 0 \cdot h_2(15)) \mod 11 = 4. \text{ Since } T[4] \text{ is occupied, we probe again. } h(15, 1) = (15 + 1 \cdot h_2(15)) \mod 11 = (15 + (15 \mod 10)) = 10. \text{ Since } T[10] \text{ is occupied we probe again. } h(15, 2) = 5. \text{ Thus } T[5] = 15. \]

\[ h(28, 0) = (28 + 0 \cdot h_2(28)) \mod 11 = 6. \text{ Thus } T[6] = 28. \]

\[ h(17, 0) = (17 + 0 \cdot h_2(17)) \mod 11 = 6. \text{ Since } T[6] \text{ is occupied, we probe again. } h(17, 1) = 3. \text{ Thus } T[3] = 17. \]

\[ h(88, 0) = (88 + 0 \cdot h_2(88)) \mod 11 = 0. \text{ T[0] is occupied, so probe again. } h(88, 1) = 9 \text{ which is occupied. } h(88, 2) = 7. \text{ Thus } T[7] = 88. \]

\[ h(59, 0) = (59 + 0 \cdot h_2(59)) \mod 11 = 4. \text{ T[4] is occupied. We probe again. } h(59, 1) = 3 \text{ does not work. } h(59, 2) = 2. \text{ Thus } T[2] = 59. \]

The final hash table is as shown in figure 1c.

**11.4-2**

**Problem**

Write pseudocode for HASH-DELETE as outlined in the text, and modify HASH-INSERT to handle the special value DELETED.
Solution

HASH-DELETE(T, k)

i = 0

repeat
    j = h(k, i)
    if ( T[j] == k )
        T[j] = DELETED
        return
    i = i + 1
until T[j] == NIL or i == m

return

Note that the Deletion code cannot simply mark a slot as empty by storing NIL in it. If we did so, key retrieval will fail for any key k for which insertion code found the slot occupied and probed beyond it. This issue is solved by marking the slot with a special DELETED value. Inserts can treat such slots as empty, while search simply skips this slot.

The Hash-Insert() code that handles this modified case is as follows.

HASH-INSERT(T, k)

i = 0

repeat
    j = h(k, i)
    if ( T[j] == NIL or T[j] == DELETED )
        T[j] = k
        return j
    else i = i + 1
until i == m

error "hash table overflow"