DSA Homework 3

B00401062 羅文斌

Problem 1.1

The formula of double addressing is $h(k,i) = (h_1(k) + i \cdot h_2(k))\%m = (k\%m + i(1 + k\%(m-1)))\%m$.

Each step is shown as follows. Collision is colored **red**.

Insert	Try	Hash Table
18	$h(18,0) = 7$	[__ __ __ __ __ __ __ 18 __ __ __]
34	$h(34,0) = 1$	[__ 34 __ __ __ __ __ 18 __ __ __]
9	$h(9,0) = 9$	[__ 34 __ __ __ __ __ 18 __ 9 __]
37	$h(37,0) = 4$	[__ 34 __ __ 37 __ __ 18 __ 9 __]
40	$h(40,0) = 7$	[__ 34 __ __ 37 __ __ **18** __ 9 __]
	$h(40,1) = 8$	[__ 34 __ __ 37 __ __ 18 40 9 __]
32	$h(32,0) = 10$	[__ 34 __ __ 37 __ __ 18 40 9 32]
89	$h(89,0) = 1$	[__ **34** __ __ 37 __ __ 18 40 9 32]
	$h(89,1) = 0$	[89 34 __ __ 37 __ __ 18 40 9 32]

Problem 1.2

a.

Directory	Keys
00	[128 64 32 16]
01	[25 1 17 13]
10	[2 10 14 ___]
11	[31 7 3 ___]

b.

Directory	Keys
000	A = [128 64 32 16]
001	B = [25 1 17 49]
010	C = [2 10 14 30]
011	D = [31 7 3 ___]
100	E = [4 ___ ___ ___]
101	F = [13 ___ ___ ___]
110	C
111	D

Problem 1.3

Instead of broadcasting one of rock-paper-scissors directly, players should first use their own hash function, e.g. SHA-256, to generate an encrypted rock-paper-scissors and send the encrypted one to other players. The encrypted rock-paper-scissors is one of $h(\text{rock})$, $h(\text{paper})$, and $h(\text{scissors})$. After receiving the encrypted rock-paper-scissors from all the other players, players then broadcast their own hash function to prove that they use the exact function to generate the encrypted rock-paper-scissors.

A hash function like SHA-256 has some good features. First, it must be hard for cheaters to decrypt the ciphertext to unveil the plaintext. Second, few collisions occur under the hash function, i.e. the plaintext and ciphertext pair should be unique under a certain hash function. Also, different hash functions should not hash different plaintexts to the same ciphertext, for example, it must be essentially impossible to find h_1 and h_2 such that $h_1(\text{rock})$ equals $h_2(\text{paper})$.

Problem 1.4

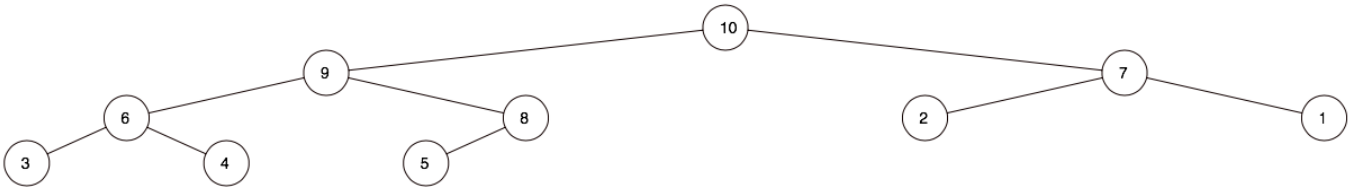
a. Collision is colored **red**.

Insert	Try	T1	T2
6	$h_1(6) = 6$	[__ __ __ __ __ __ 6]	[__ __ __ __ __ __ __]
31	$h_1(31) = 3$	[__ __ __ 31 __ __ 6]	[__ __ __ __ __ __ __]
2	$h_1(2) = 2$	[__ __ 2 31 __ __ 6]	[__ __ __ __ __ __ __]
41	$h_1(41) = 6$	[__ __ 2 31 __ __ **41**]	[__ __ __ __ __ __ __]
	$h_2(6) = 0$	[__ __ 2 31 __ __ 41]	[6 __ __ __ __ __ __]
30	$h_1(30) = 2$	[__ __ **30** 31 __ __ 41]	[6 __ __ __ __ __ __]
	$h_2(2) = 0$	[__ __ 30 31 __ __ 41]	[**2** __ __ __ __ __ __]
	$h_1(6) = 6$	[__ __ 30 31 __ __ **6**]	[2 __ __ __ __ __ __]
	$h_2(41) = 5$	[__ __ 30 31 __ __ 6]	[2 __ __ __ __ 41 __]
45	$h_1(45) = 3$	[__ __ 30 **45** __ __ 6]	[2 __ __ __ __ 41 __]
	$h_2(31) = 4$	[__ __ 30 45 __ __ 6]	[2 __ __ __ 31 41 __]
44	$h_1(44) = 2$	[__ __ **44** 45 __ __ 6]	[2 __ __ __ 31 41 __]
	$h_2(30) = 4$	[__ __ 44 45 __ __ 6]	[2 __ __ __ **30** 41 __]
	$h_1(31) = 3$	[__ __ 44 **31** __ __ 6]	[2 __ __ __ 30 41 __]
	$h_2(45) = 6$	[__ __ 44 31 __ __ 6]	[2 __ __ __ 30 41 45]

b. The element to be inserted is 36. The sequence of displacements is [36, 6, 2, 44, 45, 31, 30, 2, 6, 36, 44, 30, 31, 45], which will cycle again and again.

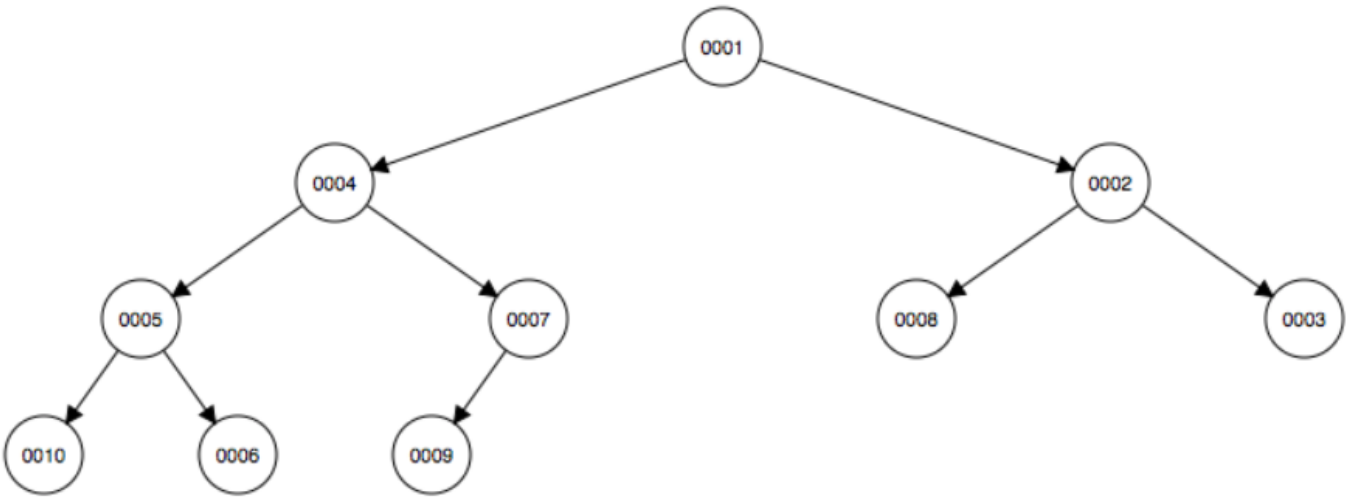
Problem 2.1

Sequence: [10, 9, 7, 6, 8, 2, 1, 3, 4, 5]



Problem 2.2

Sequence: [1, 4, 2, 5, 7, 8, 3, 10, 6, 9]



Problem 2.3

```
1  function smaller_keys(root, q) {
2      if (root.key > q) // if the key is smaller than q
3          return [ ]; // return an empty array
4      if (root.key == q) // if the key equals q
5          return [ root.key ]; // return an array with the key
6      let keys = [ ]; // create an empty array
7      keys.push(root.key); // push the key into the array
8      if (root.left) // search in the left child
9          keys.concat(smaller_keys(root.left, q)); // append keys to the array
10     if (root.right) // search in the right child
11         keys.concat(smaller_keys(root.right, q)); // append keys to the array
12     return keys; // return keys
13 }
```

This function calls itself recursively whenever its current key is smaller than q and will return until the key is greater than or equal to q. Therefore, the number of recursive calls at line 9 and line 11 will be no more than q. Time complexity = $O(q) = O(k)$

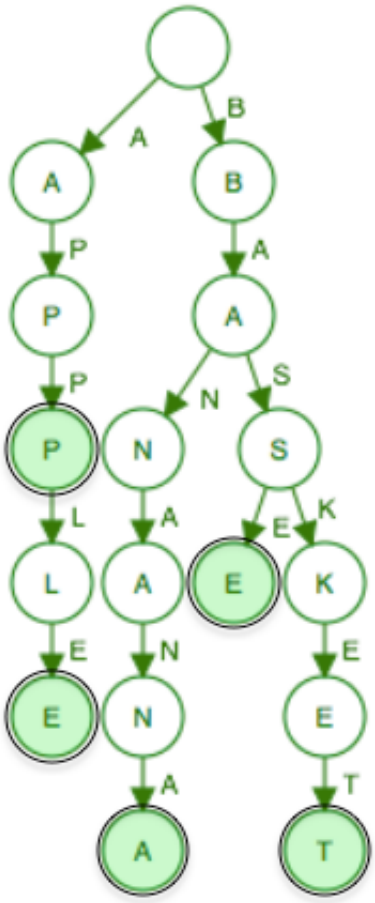
Problem 2.4

The correspondent array implementation of a heap has the following properties. Two children of the i-th node are $2i + 1$ and $2i + 2$. The parent of the i-th node is $\lfloor \frac{i-1}{2} \rfloor$.

We define a heapify function which heapifies all descendents of a given node by sinking the given node until (1) none of its children is greater (smaller) than its key value, or (2) it hits the bottom of the heap and becomes a leaf. This function takes time proportional to the current height of the heap, i.e. $O(h)$. We then apply this function on each element in the array backward (from the last element to the first element). When we reach the first index of the array (i.e. the root of the binary heap), the array will be heapified already since both of its children are also valid binary heaps.

The time these operations take can be expressed as follows. Assume we are given an array of N entries whose correspondent binary heap has height $h = O(\lg(N))$. Each entry on the i-th level of the heap takes $O(h - i)$ to finish the heapify function, and there are 2^i entries on the i-th level. The total time complexity is represented as $S = \sum_{i=0}^{h} 2^i(h - i)$. By substracting S from $2S$, we get $2S - S = -h + 2 + 2^2 + \ldots + 2^h = 2^{h+1} - h - 2$. Plug $h = O(\lg(N))$ into the formula, we get $2^{h+1} - h - 2 = 2N - \lg(N) - 2 \leq 2N = O(N)$

Problem 3.1



Problem 3.2

Given a word of length N to insert, delete or query,

- **Insert**. Given a root node and among all of its direct children, we have to first search for the target alphabet `w[i]` at `children[w[i]-'a']` and then insert a new alphabet at `children[w[i]-'a']` if the target alphabet has not appeared. We recursively locate or put one alphabet at one of the children as we process down the word until the whole word forms a path in the trie. At the last alphabet, `is_word` is changed to 1 to indicate the end of the word. For each alphabet in the word, searching and inserting take constant time. The total time complexity to insert a word of length N is $O(N)$
- **Delete**. Assume the word is in the trie. First, we have to search for the target alphabet `w[i]` at `children[w[i]-'a']` as we process down the word. If the word is in the trie, we will traverse all the way down the trie until we arrive at the last alphabet of the word. The time complexity is $O(N)$. Once we reach the node of the last alphabet, we have to do either one of the following things: (1) If the node to be deleted is a leaf of the trie, we have to free the space of the nodes backward recursively until we meet a node with branches. This takes at most $O(N)$ (i.e. the worst case is that the word itself occupies a whole path without branching). (2) If the node to be deleted is not a leaf, we just change `is_word` to 0 and the deletion is done. This takes $O(1)$. To sum up, deletion takes $O(N) + O(N) = O(N)$ in the worst case.
- **Query**. First, we have to search for the target alphabet among all the children with respect to a node as we process down the word. If the word is in the trie, we will traverse all the way down the trie until we arrive at the last alphabet and make sure `is_word` is 1. The time complexity is $O(N)$. If the word is not in the trie, we don't even necessarily process all the alphabets in some cases. As long as the target alphabet doesn't appear at a certain level, or `is_word` is 0 when we reach the last alphabet, the word has no way to be in the trie. The time complexity in the worst case is still $O(N)$. In conclusion, the time complexity of query is $O(N)$

Problem 3.3

```
1  function insert(root, w) { // insert a new word w
2      let ptr = root;
3      for (let i = 0; i < w.length; i++) {
4          if (!ptr.children[w[i]-'a'])
5              ptr.children[w[i]-'a'] = new Node(); // tag initialized to 0
6          ptr = ptr.children[w[i]-'a'];
7          ptr.tag++; // tag increases by 1 whenever a word goes pass it
8      }
9      ptr.is_word = true;
10 }
11 function prefix(root, s) { // process a query s
12     let ptr = root;
13     for (let i = 0; i < s.length; i++)
14         ptr = ptr.children[s[i]-'a'];
15     return ptr.tag; // tag is exactly the number of words from the ptr
16 }
```

`tag` is introduced to keep track of the number of words going pass a pointer, i.e. the number of words sharing the same prefix. In the pseudocode, we assume `tag` is always initialized to 0 and `is_word` is always initialized to `false` to make the pseudocode less trivial. From problem 3.2, we know insertion takes $O(|\mathbf{word}|)$, so the total time complexity to build a complete trie w.r.t. a set of words is $O(\sum_{i=1}^{N} |W_i|)$. For each query `s`, the `for` loop at line 12 runs at most |s|, i.e. $O(|S|)$, to reach the last alphabet of `s`. Then we return the `tag` of that node, which records how many words go pass it, i.e. how many words share the same prefix with the query `s`.

Also note the pseudocode given above only deals with prefix. If now we want to deal with suffix, we just process each word `w` and query `s` backward, i.e. the `for` loop will go from `w.length-1` to 0.

Problem 3.4

The basic idea to win the game is to lead the word to a path that you are able to fill the last character. There are two conditions. If you start first, then you want a path in the trie that contains odd number of alphabets. Likewise, if your opponent starts first, then you will need a path that contains even number of alphabets. However, if there is no such a path satisfying the condition in the beginning of the game, any winning algorithm still loses at last... The description "that means he/she can always win no matter how the other player moves" is kind of weird...

To put it more clearly, we assume now is your turn and a word w has been built. The following steps may lead to win:

1. Remove those not sharing the same prefix with w from the list to get a list of candidates.
2. If any candidate is a prefix of one another, then that candidate is also removed.
3. Compute the length of the remaining alphabets of each candidate and partition them into two groups: odd number and even number
4. Choose the next alphabet from the group with even number of remaining alphabets but not in the group with odd number of remaining alphabets.
5. If there is no such alphabet satisfying the last step, then a win cannot be guaranteed.