B05902118 陳盈如

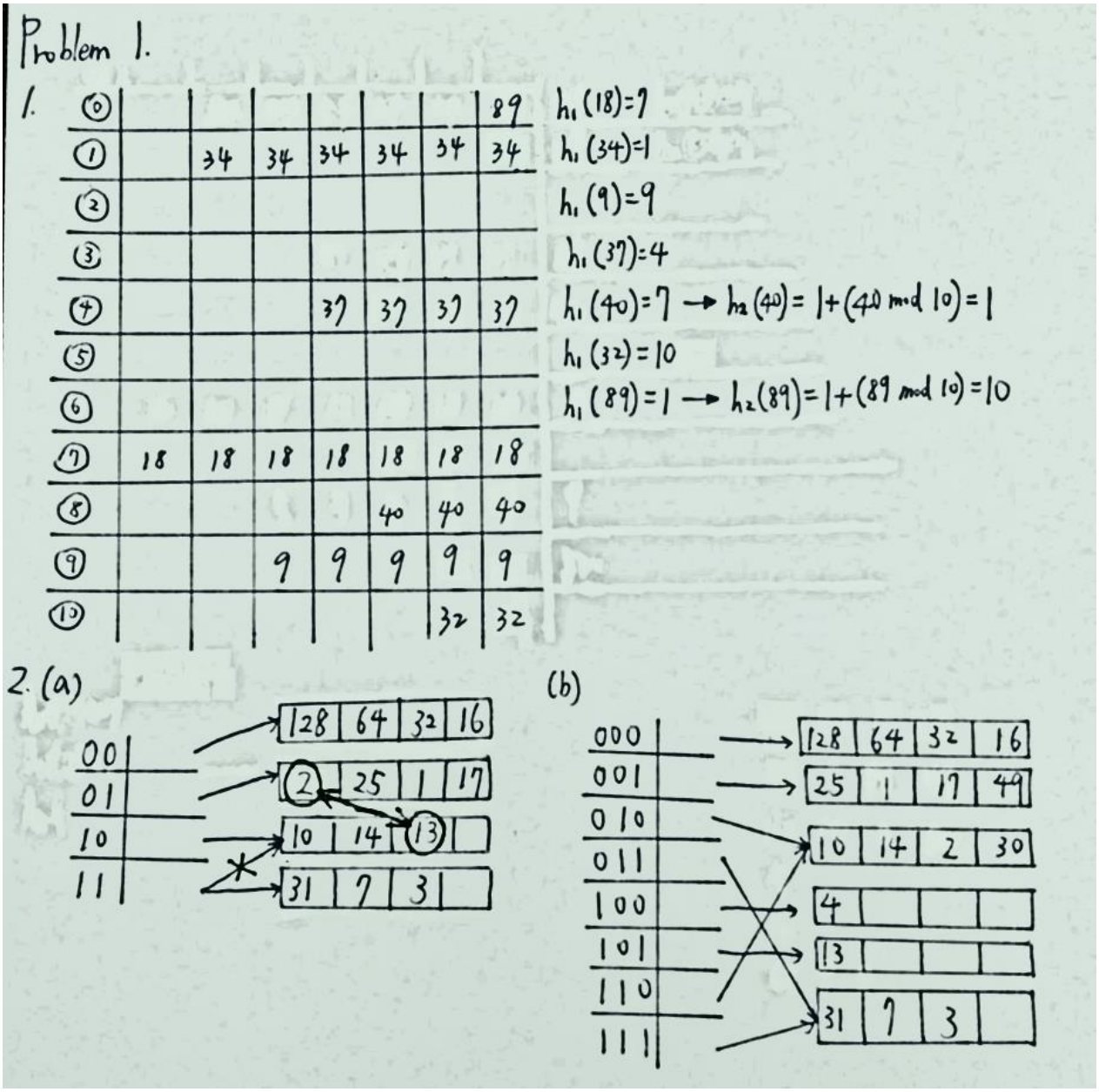
Reference：

Problem1.1：b05902066

Problem1.3：b05902041

Problem2.3：b05902120 b05902124

Problem3.4：b05902041



Problem1.3.

步驟：

1. 兩人猜拳，分別在心中想好要出甚麼
2. 然後利用 SHA-256 這個 hash function 將自己心中想要出的拳轉換成亂碼
3. 接著兩人都把亂碼貼給對方
4. 貼完之後，兩人都公布自己出甚麼拳
5. 分別將這兩種拳利用 SHA-256 再次轉換成亂碼，若亂碼與剛剛貼的符合，則可以確認兩人都是誠實地說出自己出甚麼拳，而沒有中途變拳

nice properties of SHA-256：

1. 此 hash function 發生 collision 的機率很低
2. 此 hash function 只能單向操作，經過 hash 之後無法反轉回原本的東西，如此一來，就不用擔心對方收到亂碼之後，將亂碼反轉回原本的東西後得知我出甚麼拳
3. 此 hash function 對相同的東西 hash 之後得到的亂碼都一樣，因此才能重複使用，也因此在執行此遊戲時，為了避免被對方猜出自己出甚麼拳，可以在出的拳前後加上其他沒有意義的字。
   EX：螃蟹剪刀手、我就是要出布、石頭很硬……

Problem 1.

4. (a)

T_1:

⓪	
①	
②	
③	31
④	
⑤	
⑥	6

⓪	
①	
②	2
③	31
④	
⑤	
⑥	6

⓪	
①	
②	2
③	31
④	
⑤	
⑥	6

⓪	
①	
②	30
③	31
④	
⑤	
⑥	41

⓪	
①	
②	30
③	45
④	
⑤	
⑥	6

⓪	44
①	
②	2
③	31
④	
⑤	
⑥	6

T_2:

⓪	
①	
②	
③	
④	
⑤	41
⑥	

⓪	6
①	
②	
③	
④	
⑤	41
⑥	

⓪	6
①	
②	
③	
④	31
⑤	41
⑥	

⓪	6
①	
②	2
③	
④	31
⑤	41
⑥	

⓪	6
①	
②	2
③	
④	30
⑤	41
⑥	45

$h_1(6)=6$

$h_1(31)=3$

$h_1(2)=2$

$h_1(41)=6 \rightarrow h_2(6)=0 \bmod 7=0$

$h_1(30)=2 \rightarrow h_2(2)=0 \bmod 7=0$

$\rightarrow h_1(6)=6 \rightarrow h_2(41)=5 \bmod 7=5$

$h_1(45)=3 \rightarrow h_2(31)=4 \bmod 7=4$

$h_1(44)=2 \rightarrow h_2(30)=4 \bmod 7=4 \rightarrow h_1(31)=3 \rightarrow h_2(45)=6 \bmod 7=6$

(b)

T_1:

⓪	①	②	③	④	⑤	⑥
		44	31			6

T_2:

⓪	①	②	③	④	⑤	⑥
2				30	41	45

$(h_1(6), h_2(6))=(6,0)$ $(h_1(31), h_2(31))=(3,4)$ $(h_1(2), h_2(2))=(2,0)$ $(h_1(41), h_2(41))=(6,5)$

$(h_1(30), h_2(30))=(2,4)$ $(h_1(45), h_2(45))=(3,6)$ $(h_1(44), h_2(44))=(2,6)$

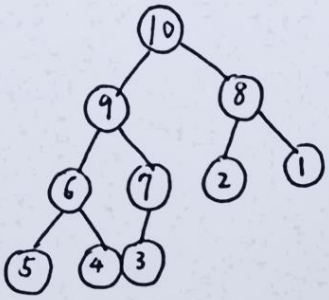
Insert 3 $\rightarrow (h_1(3), h_2(3))=(3,0)$

$\Rightarrow \boxed{h_1(3)=3} \rightarrow h_2(31)=4 \rightarrow h_1(30)=2 \rightarrow h_2(44)=6 \rightarrow h_1(45)=3 \rightarrow h_2(3)=0 \rightarrow h_1(2)=2$

$\rightarrow h_2(30)=4 \rightarrow h_1(31)=3 \rightarrow h_2(45)=6 \rightarrow h_1(44)=2 \rightarrow h_2(2)=0 \rightarrow \boxed{h_1(3)=3} \Rightarrow$ repeat

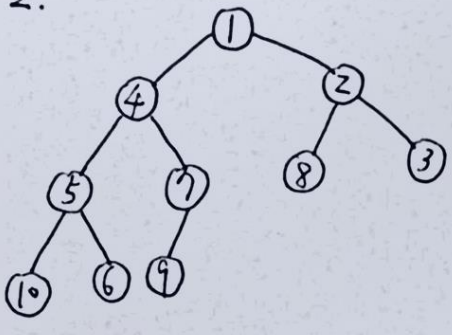
Problem 2.

1.



[5. 6. 4. 9. 3. 7. 10. 2. 8. 1].

2.



[10. 5. 6. 4. 9. 7. 1. 8. 2. 3]

Problem2.3

```
1   struct Node {
2       struct Node *left;
3       struct Node *right;
4       int data;
5   }
6
7   void check (struct Node *tree, int q)
8   {
9       if (tree -> data <= q)
10          printf(tree -> data);
11      else
12          return;
13      if (tree -> left != NULL) {
14          if (tree -> left -> data <= q)
15              check(tree -> left, q);
16      }
17      if (tree -> right != NULL) {
18          if (tree -> right -> data <= q)
19              check(tree -> right, q);
20      }
21      return;
22  }
```

從 root 開始往下找，判斷現在位置的 node 的 data 是否 <= q，若是，就將 data printf 出來，否則就 return，因為此題的 tree 是 min heap，所以此 node 的 data 若 > q，那這個 node 以下的所有 node 都不可能會 <= q。
此 node 的 data <= q 之後再判斷下面有沒有 node 也是 <= q，有的話就再進入下一個 check。
因為每一次的 check 只要判斷自己的 data 與下方兩個子 node 的 data，時間是 O(1)，因為要碰到 data <= q 才會呼叫 check，而符合這樣標準的 node 最多只會有 k 個，所以時間會是 O(k)。

Problem2.4

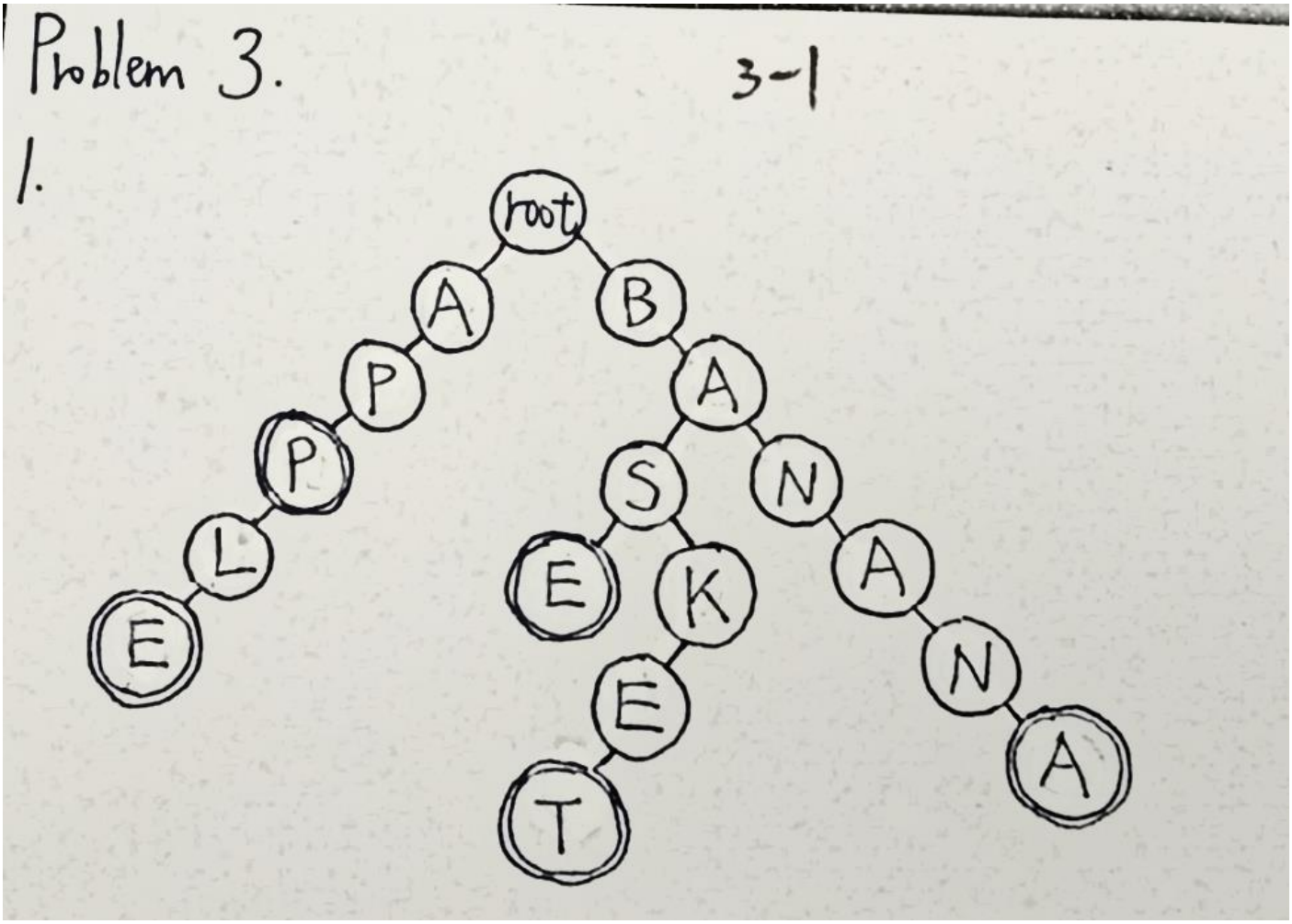
```
1   max_heapify(A, i)
2   {
3       left = 2 * i;
4       right = 2 * i + 1;
5       if ( left <= N && A[left] > A[i] )
6           largest = left;
7       else
8           largest = i;
9       if ( right <= N && A[right] > A[largest] )
10          largest = right;
11      if (largest != i) {
12          exchange A[largest] with A[i];
13          max_heapify(A, largest);
14      }
15  }
16
17  for (i = N/2; i > 0; i--)
18      max_heapify(A, i);
```

陣列的第一個 index stores the root of the heap，接下來陣列裡存的 index 就是依高度從 root 開始往下，從左至右存每一個 node 的 data

因此每一個 node 與 left node 在陣列裡面儲存位置的關係是 i & 2i，而每一個 node 與 right node 在陣列裡面儲存位置的關係則是 i & 2i + 1

- $h + 2(h-1) + 2^2(h-2) + \cdots + 2^{h-1} \cdot 1$　　最糟的情況就是每一個 node 都需要做 heapify
- $= \sum_{i=0}^{h} 2^i (h-i)$
- Let $S = \sum_{i=0}^{h} 2^i (h-i)$.
- $2S = 2h + 4(h-1) + \cdots + 2^h$
- $2S - S = -h + 2 + 4 + \cdots + 2^h$
- $S = 2^{h+1} - h - 2$
- 又$h = O(\log n)$
- $2^{\lceil \log_2 n \rceil} - O(\log n) - 2$
- $\leq 2n - O(\log n)$
- $= O(n)$　　因此可以知道 worse case 是 O(n)

故 max-heapify 的時間複雜度是 O(n)



Problem3.2

```
1   struct Node {
2   struct Node *children[26];
3   int is_word ; // Change to 1 for insert(), 0 for delete()
4   int tag ; // may be useful for prob 3 & 4
5   };
6
7   struct Node *trie;
8   trie = (struct Node *)malloc(sizeof(struct Node));
9   trie -> children = NULL;
10  trie -> is_word = trie -> tag = 0;
11  void insert (struct Node *trie, char word[], int N, int where);
12  // insert word into the trie
13  void delete(struct Node *trie, char word[], int N, int where);
14  // delete word from the trie
15  int query(struct Node *trie, char word[], int N, int where);
16  // check if word is in the trie
```

需要先開一個 root 給 trie，並且將裡面的值都初始化

INSERT()

```
1   void insert(struct Node *trie, char word[], int N, int where)
2   //where starts from 1
3   {
4       if (where > N) {
5           trie -> is_word = 1;
6           return;
7       }
8       int place;
9       place = word[where] - 'a';
10      if (trie -> children[place] == NULL) {
11          trie -> children[place] = (struct Node *)malloc(sizeof(struct
            Node));
12          trie -> children[place] -> children = NULL;
13          trie -> children[place] -> is_word = 0;
14          trie -> children[place] -> tag = 1;
15      }
16      else trie -> children[place] -> tag += 1;
17      insert(trie -> children[place], word, N, where + 1);
18      return;
19  }
```

Insert:

每一次讀取 word 的一個字母並輸入到 trie，where 紀錄現在輸入到 word 的第幾個字母，where 從 1 開始到 N (word 的長度)，tag 是紀錄這個 node 在輸入所有 word 的時候被經過幾次，利用此點可以知道這個字母以前的字串會是幾個 word 的 prefix

每讀取到一個字母後，從現在的 node 往下找 children[]裡面有沒有這個字母，利用 place 紀錄字母是 a-z 哪一個，若沒有這個字母則 malloc 一個新的 node，並將 tag 設為 1，若有，則將此字母 tag + 1 並繼續 insert 直到 word 的字母都讀取完，讀到最後一個字母時將 node 的 is_word 改成 1，代表這個 word 在這個字母結束。

DELETE()

```
1   void delete(struct Node *trie, char word[], int N, int where)
2   {
3       if (where > N) {
4           trie -> is_word = 0;
5           return;
6       }
7       int place;
8       place = word[where] - 'a';
9       if (trie -> children[place] == NULL) {
10          return;
11      }
12      else trie -> children[place] -> tag -= 1;
13      delete(trie -> children[place], word, N, where + 1);
14      return;
15  }
```

Delete:

每一次讀取 word 的一個字母，從 root 往下開始搜尋，若 children[]裡面沒有這個字母，則代表這個 trie 裡面沒有這個 word，就直接 return，若有，將 tag - 1 並往下繼續讀取下一個字母且進入下一個 delete 的 function，當讀到 word 的最後一個字母時，除了要將 tag – 1，還要將 is_word 改為 0，代表這個 word 被刪除。

QUERY()

```
1   int query(struct Node *trie, char word[], int N, int where)
2   {
3       if (where > N) {
4           if (trie -> is_word) return 1;
5       }
6       int place;
7       place = word[where] - 'a';
8       if (trie -> children[place] == NULL) {
9           return 0;
10      }
11      return query(trie -> children[place], word, N, where + 1);
12  }
```

Query:

同樣也是一次讀取 word 的一個字母，從 root 開始往下搜尋，若 children[]裡面沒有此字母就 return 0，若有，就繼續往下搜尋進入下一個 query，讀取到最後一個字母時還要多判斷 is_word 是否為 1，若不是 1 這個 word 仍然不存在。

Problem3.3

```
1   void insert (struct Node *trie, char word[], int N, int where);
2   //延續3.2的fucntion
3   void check(struct Node *trie, char word[], int N, int where);
4   //3.3新增的function
5
6   for (int i = 0; i < N; i++) {
7       insert(trie_prefix, W, W.length, where);
8       //where 從 1 到 W.length
9       insert(trie_suffix, W, W.length, where);
10      //where 從 W.length 到 1，insert recursive function 裡面的where + 1 要改成where - 1
11  }
12
13  for (int i = 0; i < Q; i++) {
14      printf("prefix:");
15      check(trie_prefix, S, S.length, where);
16      //where 從 1 到 W.length
17      printf("suffix:");
18      check(trie_suffix, S, S.length, where);
19      //where 從 W.length 到 1，insert recursive function 裡面的where + 1 要改成where - 1
20  }
21
22  void check(struct Node *trie, char word[], int N, int where)
23  {
24      if (where > N) {
25          printf(trie -> tag);
26          return;
27      }
28      int place;
29      place = word[where] - 'a';
30      if (trie -> children[place] == NULL) return;
31      query(trie -> children[place], word, N, where + 1);
32  }
```

先做兩個 trie，一個將所有的 word 從前面讀取做成 trie_prefix，另一個則是從後面讀取做成 trie_suffix，每一個 node -> tag 記錄著這個 node 的下面有幾個完整的 word。
要回答有幾個 preifx 就從 trie_prefix 的 root 開始往下搜尋配對 S 字串裡的每一個字元，當每一個都配對完之後，最後一個字元所對應到的 node -> tag 就是答案。
而要回答 suffix 與找 prefix 的方法相同，從 trie_suffix 的 root 開始搜尋。

Problem3.4

```
1   int win(struct Node *trie, int who) //who starts from 0
2   {
3       if ( trie -> children == NULL && trie -> is_word == 1 ) {
4           if ( who % 2 )
5               return 1;
6           else
7               return 2;
8       }
9       else {
10          for ( int i = 0; i < 26; i++ ) {
11              if ( win(trie -> children[i], who + 1) == 1 ) {
12                  if ( (who + 1) % 2 )
13                      return 1;
14              }
15              else {
16                  if ( (who + 1) % 2 == 0 )
17                      return 2;
18              }
19          }
20          if ( (who + 1) % 2 ) return 2;
21          else return 1;
22      }
23  }
```

從 root 開始 call win()這個 function，win()這個 function 會選擇該玩家最有利的 node 繼續往下走，走到下一個 node 的時候就再一次 call win()，每一次的 win() function 會先判斷 node 的下面還有沒有其他 node 與他的 is_word 是否等於 1，等於 1 的時候且下面沒有其他 node 則表示此玩家贏了，同時也 return 這個玩家是玩家 1 還是玩家 2。

每做一次選擇後，要 call 下一個 win()，who 會 + 1，表示現在是第幾次做選擇，先開始的玩家必定是做所有奇數次的選擇，而後開始的玩家則是做所有偶數次的選擇，利用這個性質就可以在遊戲結束時確定是哪位玩家做出最後一個選擇。

在 win() function 裡面會判斷應該往下方哪個 node 前進會最有利，並且就往那個 node 前進，若發現自身下方的 node 不論選誰都無法贏時，就直接 return 對方玩家。