

Homework IV - Analysis of Algorithms

Edwing Alexis Casillas Valencia

October 27, 2025

Abstract

In this homework, we implement and analyze the RBT algorithm, the Matrix Multiplication (Dynamic Programming) and the Huffman code.

PROBLEM 1

Implement using C language, the RBT insert, search, minimum, maximum, delete operations, then

(a) Empirically Answer if the theoretical complexities are correct

Answer

Script

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/time.h>
4
5
6 typedef enum {RED, BLACK} color_t;
7
8 typedef struct Node
9 {
10     /*
11     Properties for the nodes
12     */
13     int key;
14     color_t color;
15     struct Node *left, *right, *parent;
16 }Node;
17
18 typedef struct RBT
19 {
20     /*
21     Struct for the tree
22     */
23     Node *root;
24     Node *nil;
25 }RBT;
26
27 RBT* createRBT()
28 {
29     /*
30     Creates the initial structure of the RBT, NIL and root pointing to each other
31     */
32     RBT *tree = (RBT*)malloc(sizeof(RBT));
33     tree->nil = (Node*)malloc(sizeof(Node));
34     tree->nil->color = BLACK;
35     tree->nil->left = tree->nil->right = tree->nil->parent = tree->nil;
36     tree->root = tree->nil;
37     return tree;
38 }
39
40 Node* createNode(RBT *tree, int key)
41 {
42     /*
43     Creates a new node into the RBT
44     */
45     Node* newNode = (Node*)malloc(sizeof(Node));
46     newNode->key = key;
47     newNode->color = RED;
48     newNode->left = tree->nil;
49     newNode->right = tree->nil;
50     newNode->parent = tree->nil;
51     return newNode;
52 }
```

```

53
54
55 void left_rotate(RBT *T, Node *x)
56 {
57     Node *y = x->right;
58     x->right = y->left; //turn y's left subtree into x's right subtree
59     if (y->left != T->nil) //if y's left subtree is not empty...
60     {
61         y->left->parent = x; //... then x becomes y's parent
62     }
63     y->parent = x->parent; //x's parent becomes y's parent
64     if (x->parent == T->nil) //if x was the root...
65     {
66         T->root = y; //then y becomes the root
67     }
68     else if (x == x->parent->left) //otherwise, if x was a left child...
69     {
70         x->parent->left = y; //...then y becomes a left child
71     }
72     else
73     {
74         x->parent->right = y; // otherwise, x was a righth child, and now y is
75     }
76     y->left = x; //make x become y's left child
77     x->parent = y;
78 }
79
80
81 void right_rotate(RBT *T, Node *y)
82 {
83     Node *x = y->left;
84     y->left = x->right; //turn y's left subtree into x's right subtree
85     if (x->right != T->nil) //if y's righth subtree is not empty...
86     {
87         x->right->parent = y; //... then x becomes y's parent
88     }
89     x->parent = y->parent; //x's parent becomes y's parent
90     if (y->parent == T->nil) //if x was the root...
91     {
92         T->root = x; //then y becomes the root
93     }
94     else if (y == y->parent->left) //otherwise, if x was a left child...
95     {
96         y->parent->left = x; //...then y becomes a left child
97     }
98     else
99     {
100         y->parent->right = x; // otherwise, x was a righth child, and now y is
101     }
102     x->right = y; //make x become y's right child
103     y->parent = x;
104 }
105
106
107 void RB_insert_fixup(RBT *T, Node *z)
108 {
109     while (z->parent->color == RED)
110     {
111         if (z->parent == z->parent->parent->left) //if z's parent a left child?
112         {
113             Node *y = z->parent->parent->right; //y is z's uncle
114             if (y->color == RED) //are z's parent and unce both red?
115             {
116                 /*case 1*/
117                 z->parent->color = RED;
118                 y->color = BLACK;
119                 z->parent->parent->color = RED;
120             }
121             else
122             {
123                 if (z == z->parent->right)
124                 {
125                     /* case 2 */
126                     z = z->parent;

```

```

127         left_rotate(T,z);
128     }
129     /*case 3*/
130     z->parent->color = BLACK;
131     z->parent->parent->color = RED;
132     right_rotate(T, z->parent->parent);
133 }
134 }
135 else
136 {
137     /*same as above, but with "right" and "left" exchanged*/
138     Node *y = z->parent->parent->left; //y is z's uncle
139     if (y->color == RED) //are z's parent and unce both red?
140     {
141         /*case 1*/
142         z->parent->color = RED;
143         y->color = BLACK;
144         z->parent->parent->color = RED;
145     }
146     else
147     {
148         if (z == z->parent->left)
149         {
150             /* case 2 */
151             z = z->parent;
152             right_rotate(T,z);
153         }
154         /*case 3*/
155         z->parent->color = BLACK;
156         z->parent->parent->color = RED;
157         left_rotate(T, z->parent->parent);
158     }
159 }
160 }
161 T->root->color = BLACK;
162 }
163
164
165 void RB_insert(RBT *T, int key)
166 {
167     Node *z = createNode(T, key);
168     Node *x = T->root; //node being compared with z
169     Node *y = T->nil; //y will be parent of z
170
171     while (x != T->nil) // descend until reaching the sentinel
172     {
173         y = x;
174         if (z->key < x->key)
175         {
176             x = x->left;
177         }
178         else
179         {
180             x = x->right;
181         }
182     }
183     z->parent = y; //found the location - insert z with parent y
184     if (y == T->nil)
185     {
186         T->root = z; // tree T was empty
187     }
188     else if (z->key < y->key)
189     {
190         y->left = z;
191     }
192     else
193     {
194         y->right = z;
195     }
196     //z->left = T->nil; //both of z's children are the sentinel
197     //z->right = T->nil;
198     //z->color = RED; //the new node starts out red
199     RB_insert_fixup(T,z);
200 }

```

```

201
202
203 Node* tree_search(RBT *T, Node *x, int key)
204 {
205     if (x == T->nil || key == x->key)
206     {
207         return x;
208     }
209     if (key < x->key)
210     {
211         return tree_search(T, x->left, key);
212     }
213     else
214     {
215         return tree_search(T, x->right, key);
216     }
217 }
218
219
220 Node* tree_minimum(RBT *T, Node *x)
221 {
222     while (x->left != T->nil)
223     {
224         x = x->left;
225     }
226     return x;
227 }
228
229 Node* tree_maximum(RBT *T, Node *x)
230 {
231     while (x->right != T->nil)
232     {
233         x = x->right;
234     }
235     return x;
236 }
237
238
239 void RB_transplant(RBT *T, Node *u, Node *v)
240 {
241     if (u->parent == T->nil)
242     {
243         T->root = v;
244     }
245     else if (u == u->parent->left)
246     {
247         u->parent->left = v;
248     }
249     else
250     {
251         u->parent->right = v;
252     }
253     v->parent = u->parent;
254 }
255
256
257 void RB_delete_fixup(RBT *T, Node *x)
258 {
259     while (x != T->root && x->color == BLACK)
260     {
261         if (x == x->parent->left) //is x a left child
262         {
263             Node *w = x->parent->right; //w is x's sibling?
264             if (w->color == RED)
265             {
266                 /*Case 1*/
267                 w->color = BLACK;
268                 x->parent->color = RED;
269                 left_rotate(T, x->parent);
270                 w = x->parent->right;
271             }
272             if (w->left->color == BLACK && w->right->color == BLACK)
273             {
274                 /* case 2 */

```

```

275         w->color = RED;
276         x = x->parent;
277     }
278     else
279     {
280         if (w->right->color == BLACK)
281         {
282             /* case 3 */
283             w->left->color = BLACK;
284             w->color = RED;
285             right_rotate(T, w);
286             w = x->parent->right;
287         }
288         /*case 4*/
289         w->color = x->parent->color;
290         x->parent->color = BLACK;
291         w->right->color = BLACK;
292         left_rotate(T, x->parent);
293         x = T->root;
294     }
295 }
296 else
297 {
298     /*same as above, but left and right exchanged*/
299     Node *w = x->parent->left; //w is x's sibling?
300     if (w->color == RED)
301     {
302         /*Case 1*/
303         w->color = BLACK;
304         x->parent->color = RED;
305         right_rotate(T, x->parent);
306         w = x->parent->left;
307     }
308     if (w->right->color == BLACK && w->left->color == BLACK)
309     {
310         /* case 2 */
311         w->color = RED;
312         x = x->parent;
313     }
314     else
315     {
316         if (w->left->color == BLACK)
317         {
318             /* case 3 */
319             w->right->color = BLACK;
320             w->color = RED;
321             left_rotate(T, w);
322             w = x->parent->left;
323         }
324         /*case 4*/
325         w->color = x->parent->color;
326         x->parent->color = BLACK;
327         w->left->color = BLACK;
328         right_rotate(T, x->parent);
329         x = T->root;
330     }
331 }
332 }
333 x->color = BLACK;
334 }
335
336
337 void RB_delete(RBT *T, Node *z)
338 {
339     Node *y = z;
340     Node *x;
341     color_t y_og_color = y->color;
342
343     if (z->left == T->nil)
344     {
345         x = z->right;
346         RB_transplant(T, z, z->right); //replace z by its right child
347     }
348     else if (z->right == T->nil)

```

```

349 {
350     x = z->left;
351     RB_transplant(T, z, z->left); //replace z by its left child
352 }
353 else
354 {
355     y = tree_minimum(T, z->right);
356     y->og_color = y->color;
357     x = y->right;
358     if (y != z->right) //is y father down the tree?
359     {
360         RB_transplant(T, y, y->right); //replace y by its right child
361         y->right = z->right; //z's right child becomes y's right child
362         y->right->parent = y;
363     }
364     else
365     {
366         x->parent = y; // in case x is T.nil
367     }
368     RB_transplant(T, z, y); //replace z by its sucesor y
369     y->left = z->left; //and give z's left child to y, which had no left child
370     y->left->parent = y;
371     y->color = z->color;
372 }
373 if (y->og_color == BLACK) //if any red-black violations occurred
374 {
375     RB_delete_fixup(T, x); //correct them
376 }
377 free(z);
378 }
379
380
381 void RB_delete_by_key(RBT *T, int key) {
382     Node *z = tree_search(T, T->root, key);
383     if (z != T->nil)
384     {
385         RB_delete(T, z);
386     }
387 }
388
389
390 void inOrder(RBT *T, Node *x)
391 {
392     if (x != T->nil)
393     {
394         inOrder(T, x->left);
395         printf("%d(%s) ", x->key, (x->color == RED) ? "RED" : "BLACK");
396         inOrder(T, x->right);
397     }
398 }
399
400
401 void print_tree(RBT *T, Node *root, int space)
402 {
403     /*print the RBT*/
404     if (root == T->nil) return;
405     space += 5;
406
407     print_tree(T, root->right, space);
408     printf("\n");
409
410     for (int i = 5; i < space; i++)
411     {
412         printf(" ");
413     }
414     printf("%d(%c)\n", root->key, (root->color == RED) ? 'R' : 'B');
415     print_tree(T, root->left, space);
416 }
417
418 /*
419 =====
420 NEW 'main' for empirical tests
421 =====
422 */

```

```

423
424 // Function to get the current time in seconds
425 double get_time()
426 {
427     struct timeval tv;
428     gettimeofday(&tv, NULL);
429     return tv.tv_sec + tv.tv_usec / 1000000.0;
430 }
431
432 // Function to free all tree memory (post-order traversal)
433 void free_tree(RBT *T, Node *x)
434 {
435     if (x == T->nil)
436     {
437         return;
438     }
439     free_tree(T, x->left);
440     free_tree(T, x->right);
441     free(x);
442 }
443
444 void destroyRBT(RBT *T)
445 {
446     free_tree(T, T->root); // Free all nodes
447     free(T->nil);          // Free NIL
448     free(T);              // Free tree struct
449 }
450
451
452 int main()
453 {
454     /*RBT *T = createRBT();
455
456     printf("Values to insert: 10, 20, 30, 15, 25, 5, 35, 40\n");
457     int values[] = {10, 20, 30, 15, 25, 5, 35, 40};
458     for (int i = 0; i < 8; i++)
459     {
460         RB_insert(T, values[i]);
461     }
462
463     printf("\nOredered tour: ");
464     inOrder(T, T->root);
465
466     printf("\n\nTree structure:\n");
467     print_tree(T, T->root, 0);
468
469     // find element on tree
470     int searchKey = 15;
471     Node *found = tree_search(T, T->root, searchKey);
472     if (found != T->nil) {
473         printf("\nElement %d found on tree.\n", searchKey);
474     } else {
475         printf("\nElement %d NOT found on tree.\n", searchKey);
476     }
477
478     Node *minNode = tree_minimum(T, T->root);
479     Node *maxNode = tree_maximum(T, T->root);
480     printf("Minimum: %d, Maximum: %d\n", minNode->key, maxNode->key);
481
482     printf("Deleting 10 from RBT\n");
483     RB_delete_by_key(T, 10);
484     inOrder(T, T->root);
485     print_tree(T, T->root, 0);
486     */
487
488     //Mod for test the complexity
489     //Initialize the random number generator
490     srand(time(NULL));
491
492     //n sizes to test
493     int N_values[] = {10000, 50000, 100000, 500000, 1000000};
494     int num_sizes = sizeof(N_values) / sizeof(N_values[0]);
495
496     // Number of searches/eliminations to be performed per test

```

```

497 // We keep it constant to isolate the effect of 'n'
498 const int NUM_OPS = 10000;
499
500 printf("Starting empirical tests for RBT...\n");
501 printf("=====\n");
502 printf("%-10s | %-20s | %-25s\n", "N", "Insertion time (s)", "Search time (s)");
503 printf("-----\n");
504
505 for (int i = 0; i < num_sizes; i++)
506 {
507     int N = N_values[i];
508     RBT *T = createRBT();
509
510     // --- Insertion test ---
511     // Measures the time to insert N random elements
512     // Large random key range (0 to N*10) to reduce collisions
513
514     double start_time = get_time();
515     for (int j = 0; j < N; j++)
516     {
517         RB_insert(T, rand() % (N * 10));
518     }
519     double end_time = get_time();
520     double insert_time = end_time - start_time;
521
522     // --- Search test ---
523     // Measures the time for NUM_OPS random searches
524     start_time = get_time();
525     for (int j = 0; j < NUM_OPS; j++)
526     {
527         tree_search(T, T->root, rand() % (N * 10));
528     }
529     end_time = get_time();
530     double search_time = end_time - start_time;
531
532     // Print results for this N
533     printf("%-10d | %-20.6f | %-25.6f\n", N, insert_time, search_time);
534
535     // --- DELETE TEST (Optional) ---
536     // Measuring this is similar to searching.
537     // We focus on insertion and searching, which are the most straightforward.
538
539     // Free the tree's memory for the next iteration
540     destroyRBT(T);
541 }
542
543 printf("=====\n");
544 printf("Tests end\n");
545
546 return 0;
547 }

```

Listing 1: "RBT"


```

Values to insert: 10, 20, 30, 15, 25, 5, 35, 40

Ordered tour: 5(RED) 10(BLACK) 15(RED) 20(BLACK) 25(BLACK) 30(RED) 35(BLACK) 40(RED)

Tree structure:
          40(R)
         35(B)
        30(R)
       25(B)
      20(B)
     15(R)
    10(B)
   5(R)

Element 15 found on tree.
Minimum: 5, Maximum: 40
Deleting 10 from RBT
5(RED) 15(BLACK) 20(BLACK) 25(BLACK) 30(RED) 35(BLACK) 40(RED)
          40(R)
         35(B)
        30(R)
       25(B)
      20(B)
     15(B)
    5(R)

```

Fig. 1: RBT output

(a)

To verify the complexity, we can measure the execution time of each tree operation with arrays that grows at n size, if the complexity of the script is correct $O(\log n)$, we should see that the execution time grows very slow. For example, if n grows 10 times, we should see that the execution time with 2 arrays of sizes 100,000 and 1,000,000 grows $O(\log(10))$.

To measure this, we need to add the following functions to the script, and change the main() for the following one.

```

1  // Function to get the current time in seconds
2  double get_time()
3  {
4      struct timeval tv;
5      gettimeofday(&tv, NULL);
6      return tv.tv_sec + tv.tv_usec / 1000000.0;
7  }
8
9  // Function to free all tree memory (post-order traversal)
10 void free_tree(RBT *T, Node *x)
11 {
12     if (x == T->nil)
13     {
14         return;

```

```

15     }
16     free_tree(T, x->left);
17     free_tree(T, x->right);
18     free(x);
19 }
20
21 void destroyRBT(RBT *T)
22 {
23     free_tree(T, T->root); // Free all nodes
24     free(T->nil);          // Free NIL
25     free(T);              // Free tree struct
26 }
27
28
29 int main()
30 {
31     //Initialize the random number generator
32     srand(time(NULL));
33
34     //n sizes to test
35     int N_values[] = {10000, 50000, 100000, 500000, 1000000};
36     int num_sizes = sizeof(N_values) / sizeof(N_values[0]);
37
38     // Number of searches/eliminations to be performed per test
39     // We keep it constant to isolate the effect of 'n'
40     const int NUM_OPS = 10000;
41
42     printf("Starting empirical tests for RBT...\n");
43     printf("=====\n");
44     printf("%-10s | %-20s | %-25s\n", "N", "Insertion time (s)", "Search time (s)");
45     printf("-----\n");
46
47     for (int i = 0; i < num_sizes; i++)
48     {
49         int N = N_values[i];
50         RBT *T = createRBT();
51
52         // --- Insertion test ---
53         // Measures the time to insert N random elements
54         // Large random key range (0 to N*10) to reduce collisions
55
56         double start_time = get_time();
57         for (int j = 0; j < N; j++)
58         {
59             RB_insert(T, rand() % (N * 10));
60         }
61         double end_time = get_time();
62         double insert_time = end_time - start_time;
63
64         // --- Search test ---
65         // Measures the time for NUM_OPS random searches
66         start_time = get_time();
67         for (int j = 0; j < NUM_OPS; j++)
68         {
69             tree_search(T, T->root, rand() % (N * 10));
70         }
71         end_time = get_time();
72         double search_time = end_time - start_time;
73
74         // Print results for this N
75         printf("%-10d | %-20.6f | %-25.6f\n", N, insert_time, search_time);
76
77         // --- DELETE TEST (Optional) ---
78         // Measuring this is similar to searching.
79         // We focus on insertion and searching, which are the most straightforward.
80
81         // Free the tree's memory for the next iteration
82         destroyRBT(T);
83     }
84
85     printf("=====\n");
86     printf("Tests end\n");
87
88     return 0;

```

```

89 }
90
91

```

Listing 2: "Empirical test"

With this modification, we can obtain results similar to the following

```

Starting empirical tests for RBT...
=====
N          | Insertion time (s) | Search time (s)
=====
10000      | 0.008300           | 0.006199
50000      | 0.079592           | 0.015683
100000     | 0.124230           | 0.008987
500000     | 0.486644           | 0.004498
1000000    | 0.602463           | 0.009250
=====
Tests end

```

Fig. 2: Empirical results for RBT

As we can see, for $N = 10,000$, we obtain a search time of 0.006199 seconds, and for $N=1,000,000$ (a tree 100 times bigger) the search time was 0.009250. The time was practically the same!. Increasing the size of N 100 times didn't create a big impact on the search time, so we can confirm that the execution time for the search, was similar to

$$T(n) = O(\log(n)) \quad (1)$$

For the insertion time, in theory we should have $O(n \log(n))$. For the sizes 100,000 and 1,000,000, we obtained 0.12430 seconds and 0.602463 seconds, which is an increment of 4.84684634 times, instead of 10. With this result, we can confirm that the increase is almost linear, similar to

$$T(n) = O(n \log(n)) \quad (2)$$

The time increase, but not as fast as the worst case $O(n \log(n))$, this could be due some noise caused by the processor cache.

PROBLEM 2

Implement the Matrix Multiplication Algorithm (Dynamic Programming) in C language, then

(a) Empirically Answer if the theoretical complexities are correct.

Answer

Script

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/time.h>
4 #include <limits.h>
5
6
7 void matrix_chain_order(int p[], int n, int **m, int **s)
8 {
9     int i, l, j, k, q;
10
11     for (i = 1; i <= n; i++) //chain lenght 1
12     {
13         m[i][i] = 0;
14     }
15     for (l = 2; l <= n; l++) // l is the chain lenght
16     {
17         for (i = 1; i <= n-l+1; i++) //chain begins at Ai
18         {
19             j = i + l - 1; //chain ends at Aj
20             m[i][j] = INT_MAX; //infinite representation
21             for (k = i; k <= j-1; k++) //try Ai:kAk+1
22             {

```

```

23         q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
24         if (q < m[i][j])
25         {
26             m[i][j] = q; //remember this cost
27             s[i][j] = k; // remember this index
28         }
29     }
30 }
31 }
32 }
33
34
35 void print_optimal_parens(int **s, int i, int j)
36 {
37     if (i == j)
38     {
39         printf("A%d", i);
40     }
41     else
42     {
43         printf("(");
44         print_optimal_parens(s, i, s[i][j]);
45         print_optimal_parens(s, s[i][j] + 1, j);
46         printf(")");
47     }
48 }
49
50
51 int main()
52 {
53     /*
54     matrix dimentions:
55     A1 = 10x55
56     A2 = 55x25
57     A3 = 25x5
58     A4 = 4x40
59     A5 = 40x20
60     A6 = 20x35
61     */
62     int p[] = { 10, 55, 25, 5, 40, 20, 35};
63     int n = sizeof(p) / sizeof(p[0]) - 1;
64     int **m = (int **)malloc((n + 1) * sizeof(int *));
65     int **s = (int **)malloc((n + 1) * sizeof(int *));
66
67     for (int i = 0; i <= n; i++)
68     {
69         m[i] = (int *)malloc((n + 1) * sizeof(int));
70         s[i] = (int *)malloc((n + 1) * sizeof(int));
71     }
72     // Init matrix
73     for (int i = 0; i <= n; i++) {
74         for (int j = 0; j <= n; j++) {
75             m[i][j] = 0;
76             s[i][j] = 0;
77         }
78     }
79
80     matrix_chain_order(p, n, m, s);
81     printf("Minumum cost for multiplication: %d\n", m[1][n]);
82     printf("Optimal order: ");
83     print_optimal_parens(s, 1, n);
84     printf("\n");
85
86     printf("m table (minimum costs):\n");
87     for (int i = 1; i <= n; i++)
88     {
89         for (int j = 0; j <= n; j++)
90         {
91             if (i <= j)
92             {
93                 printf("%6d ", m[i][j]);
94             }
95             else
96             {

```

```

97         printf(" ");
98     }
99 }
100 printf("\n");
101 }
102
103 printf("\ns table(divition points):\n");
104 for (int i = 1; i < n; i++) {
105     for (int j = 2; j <= n; j++) {
106         if (i < j) {
107             printf("%d ", s[i][j]);
108         } else {
109             printf(" ");
110         }
111     }
112     printf("\n");
113 }
114
115 //free memory
116 for (int i = 0; i <= n; i++) {
117     free(m[i]);
118     free(s[i]);
119 }
120 free(m);
121 free(s);
122
123 return 0;
124 }

```

Listing 3: "Matrix Multiplication"

```

Minumum cost for multiplication: 18875
Optimal order: ((A1(A2A3))((A4A5)A6))
m table (minimum costs):
      0 13750  9625 11625 14625 18875
        0  6875 17875 16375 24000
          0  5000  6500 11875
            0  4000  7500
              0 28000
                0

s table(divition points):
1 1 3 3 3
  2 3 3 3
    3 3 3
      4 5
        5

```

Fig. 3: Matrix Multiplication

(a)

A simple inspection in the script ,the matrix chain order function, shows us that the complexity of itself it's $O(n^3)$ due the 3 for cycles, and $\Theta(n^2)$ space to store m and s.

To prove it, we can generate a few test cases with different matrix sizes and measure the execution time for each one. For that, we can modify the script as follows:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #include <limits.h>
5
6 void matrix_chain_order(int p[], int n, int **m, int **s) {
7     int i, l, j, k, q;
8
9     for (i = 1; i <= n; i++) {
10         m[i][i] = 0;
11     }
12     for (l = 2; l <= n; l++) {

```

```

13     for (i = 1; i <= n - 1 + 1; i++) {
14         j = i + 1 - 1;
15         m[i][j] = INT_MAX;
16         for (k = i; k <= j - 1; k++) {
17             q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
18             if (q < m[i][j]) {
19                 m[i][j] = q;
20                 s[i][j] = k;
21             }
22         }
23     }
24 }
25 }
26
27 void generate_random_dimensions(int p[], int n, int max_dim) {
28     for (int i = 0; i <= n; i++) {
29         p[i] = rand() % max_dim + 1;
30     }
31 }
32
33 // Function to avoid compiler optimizations
34 volatile int dummy_result = 0;
35
36 int main() {
37     srand(time(NULL));
38
39     // MUCH larger sizes and fewer stitches
40     int sizes[] = {100, 200, 300, 400, 500, 600, 700, 800};
41     int num_sizes = sizeof(sizes) / sizeof(sizes[0]);
42     int max_dim = 100;
43     int repetitions = 3; // Fewer repeats but larger sizes
44
45     printf("n,Time(s)\n");
46
47     for (int idx = 0; idx < num_sizes; idx++) {
48         int n = sizes[idx];
49
50         int *p = (int *)malloc((n + 1) * sizeof(int));
51         generate_random_dimensions(p, n, max_dim);
52
53         int **m = (int **)malloc((n + 1) * sizeof(int *));
54         int **s = (int **)malloc((n + 1) * sizeof(int *));
55         for (int i = 0; i <= n; i++) {
56             m[i] = (int *)malloc((n + 1) * sizeof(int));
57             s[i] = (int *)malloc((n + 1) * sizeof(int));
58         }
59
60         // Measure time with clock()
61         clock_t total_ticks = 0;
62         for (int rep = 0; rep < repetitions; rep++) {
63             clock_t start = clock();
64             matrix_chain_order(p, n, m, s);
65             clock_t end = clock();
66             total_ticks += (end - start);
67
68             // Use the result to avoid optimizations
69             dummy_result += m[1][n];
70         }
71         double avg_time = ((double)total_ticks / CLOCKS_PER_SEC) / repetitions;
72
73         printf("%d,%.6f\n", n, avg_time);
74         fflush(stdout); // Force immediate printing
75
76         // free memory
77         free(p);
78         for (int i = 0; i <= n; i++) {
79             free(m[i]);
80             free(s[i]);
81         }
82         free(m);
83         free(s);
84     }
85
86     printf("Dummy result: %d\n", dummy_result); // Prevents optimization from compiler

```

```

87
88     return 0;
89 }

```

Listing 4: "Empirical Matrix Multiplication"

The above script generate some results similar to this ones:

```

n,Time(s)
100,0.002667
200,0.021333
300,0.059000
400,0.090667
500,0.213000
600,0.813333
700,1.016667
800,0.994333
Dummy result: 29380365

```

Fig. 4: Empirical results for Matrix Multiplication

For the analysis, we need to divide the execution time by n^3 , this give us a constant value which is going to be maintained for the different sizes of n . Taking the first row we have this:

$$n = 100 : T(n)/n^3 = 0.002667/100^3 = 2.667 * 10^{-9} \quad (3)$$

This constant is maintained from the first row to the second one, telling us that we have a complexity of $O(n^3)$. On the rest of the results we can't observe this behavior because my computer is giving the results very quick (even for the analysis) that's why I added some modifications to make it more slow and prevent optimizations from the compilation, but in general, we verify the correct behavior.

PROBLEM 3

Implement the Hoffman compression code by

- Implement the C min heap.
- Then the rest of the algorithm.
- Empirically Answer if the theoretical complexities are correct.

Answer

Script [(a) and (b)]

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/time.h>
4 #include <limits.h>
5
6 typedef struct Node
7 {
8     /*
9     Properties for the nodes
10    */
11     unsigned freq;
12     char data;
13     struct Node *left, *right;
14 }Node;
15
16 Node* createNode(char data, unsigned freq)
17 {
18     /*
19     Creates a new node for Huffman
20    */
21     Node* newNode = (Node*)malloc(sizeof(Node));
22     newNode->data = data;
23     newNode->freq = freq;
24     newNode->left = newNode->right = NULL;
25     return newNode;
26 }
27

```

```

28 typedef struct min_heap {
29     unsigned size;
30     unsigned capacity;
31     Node** array;
32 } min_heap;
33
34 int parent(int i)
35 {return (i-1)/2;}
36
37 int left(int i)
38 {return 2*i+1;}
39
40 int right(int i)
41 {return 2*i+2;}
42
43 min_heap* create_min_heap(unsigned capacity)
44 {
45     min_heap* mini_heap = (min_heap*)malloc(sizeof(min_heap));
46     mini_heap->size = 0;
47     mini_heap->capacity = capacity;
48     mini_heap->array = (Node**)malloc(mini_heap->capacity * sizeof(Node*));
49     return mini_heap;
50 }
51
52 void min_heapify(min_heap *mini_heap, int i)
53 {
54     int l = left(i);
55     int r = right(i);
56     int smallest;
57
58     if (l < mini_heap->size && mini_heap->array[l]->freq < mini_heap->array[i]->freq)
59     {
60         smallest = l;
61     }
62     else
63     {
64         smallest = i;
65     }
66     if (r < mini_heap->size && mini_heap->array[r]->freq < mini_heap->array[smallest]->freq)
67     {
68         smallest = r;
69     }
70     if (smallest != i)
71     {
72         //exchange nodes
73         Node* temp = mini_heap->array[i];
74         mini_heap->array[i] = mini_heap->array[smallest];
75         mini_heap->array[smallest] = temp;
76         min_heapify(mini_heap, smallest);
77     }
78 }
79
80 void build_min_heap(min_heap* mini_heap)
81 {
82     mini_heap->size = mini_heap->capacity;
83     for (int i = mini_heap->capacity/2-1; i >= 0; i--)
84     {
85         min_heapify(mini_heap, i);
86     }
87 }
88
89 Node* min_heap_extract_min(min_heap* mini_heap)
90 {
91     if (mini_heap->size < 1)
92     {
93         printf("heap overflow\n");
94         return NULL;
95     }
96
97     Node* min = mini_heap->array[0];
98     mini_heap->array[0] = mini_heap->array[mini_heap->size - 1];
99     mini_heap->size--;
100    min_heapify(mini_heap, 0);
101    return min;

```



```

102 }
103
104 void min_insert(min_heap* mini_heap, Node* x)
105 {
106     if (mini_heap->size >= mini_heap->capacity)
107     {
108         printf("Heap overflow\n");
109         return;
110     }
111
112     int i = mini_heap->size;
113     mini_heap->array[i] = x;
114     mini_heap->size++;
115
116     // Ajustar hacia arriba
117     while (i > 0 && mini_heap->array[parent(i)]->freq > mini_heap->array[i]->freq)
118     {
119         // Intercambiar con el padre
120         Node* temp = mini_heap->array[i];
121         mini_heap->array[i] = mini_heap->array[parent(i)];
122         mini_heap->array[parent(i)] = temp;
123         i = parent(i);
124     }
125 }
126
127
128 Node* huffman(char data[], unsigned freq[], int n)
129 {
130     min_heap* Q = create_min_heap(n);
131
132     //Q = C
133     for (int i = 0; i < n; i++)
134     {
135         Q->array[i] = createNode(data[i], freq[i]);
136     }
137     Q->size = n;
138     build_min_heap(Q);
139
140     for (int i = 1; i <= n-1; i++)
141     {
142         /* allocate a new node z */
143         Node* z = createNode('$', 0); // $ indicates internal node
144         z->left = min_heap_extract_min(Q);
145         z->right = min_heap_extract_min(Q);
146         z->freq = z->left->freq + z->right->freq;
147         min_insert(Q, z);
148     }
149     return min_heap_extract_min(Q);
150 }
151
152 void print_huffman(Node* root, int arr[], int top) {
153     if (root->left) {
154         arr[top] = 0;
155         print_huffman(root->left, arr, top + 1);
156     }
157
158     if (root->right) {
159         arr[top] = 1;
160         print_huffman(root->right, arr, top + 1);
161     }
162
163     if (!root->left && !root->right) {
164         printf("%c: ", root->data);
165         for (int i = 0; i < top; i++) {
166             printf("%d", arr[i]);
167         }
168         printf("\n");
169     }
170 }
171
172
173 void free_Huffman(Node* root) {
174     if (root == NULL) return;
175     free_Huffman(root->left);

```

```

176     free_Huffman(root->right);
177     free(root);
178 }
179
180 void run_test(int n) {
181     //Generate data test
182     char* data = (char*)malloc(n * sizeof(char));
183     unsigned* freq = (unsigned*)malloc(n * sizeof(unsigned));
184
185     if (!data || !freq) {
186         printf("Error allocating memory for n=%d\n", n);
187         free(data);
188         free(freq);
189         return;
190     }
191
192     for (int i = 0; i < n; i++) {
193         data[i] = (char)((i % 95) + 32); //ascii characters
194         freq[i] = (rand() % 1000) + 1;    // Random frequencies between 1 and 1000
195     }
196
197     //Measure the running time of huffman()
198     struct timeval start, end;
199     double time_taken;
200
201     gettimeofday(&start, NULL); //Start timer
202
203     Node* root = huffman(data, freq, n); //Execute algorithm
204
205     gettimeofday(&end, NULL); // Detener temporizador
206
207     //Calculate the elapsed time in seconds
208     time_taken = (end.tv_sec - start.tv_sec) * 1e6; // seg t microseg
209     time_taken = (time_taken + (end.tv_usec - start.tv_usec)) * 1e-6; // microseg to seg
210
211     printf("%d, %f\n", n, time_taken);
212
213     //free memory
214     free_Huffman(root);
215     free(data);
216     free(freq);
217 }
218
219
220 int main()
221 {
222     /*
223     char data[] = {'a', 'b', 'c', 'd', 'e', 'f'};
224     unsigned freq[] = {45, 13, 12, 16, 9, 5};
225     int n = sizeof(data) / sizeof(data[0]);
226
227     printf("Character freq: \n");
228     for (int i=0; i<n; i++)
229     {
230         printf("%c: %u\n", data[i], freq[i]);
231     }
232
233     printf("Huffman tree\n");
234     Node* root = huffman(data, freq, n);
235
236     printf("\n");
237     printf("Huffman codes: \n");
238     int arr[100], top = 0;
239     print_huffman(root, arr, top);
240
241     printf("\n");
242     printf("Frequency from root: %u\n", root->freq);
243
244     free_Huffman(root);
245     */
246     //Initialize the random number generator
247     srand(time(NULL));
248
249     //Define the sizes of n that we want to test

```

```

250 int test_sizes[] = {100, 500, 1000, 5000, 10000, 20000, 50000, 100000};
251 int num_tests = sizeof(test_sizes) / sizeof(test_sizes[0]);
252
253 printf("n, time_taken_sec\n");
254 for (int i = 0; i < num_tests; i++) {
255     run_test(test_sizes[i]);
256 }
257
258 return 0;
259 }

```

Listing 5: "Huffman code"

```

Character freq:
a: 45
b: 13
c: 12
d: 16
e: 9
f: 5
Huffman tree

Huffman codes:
'a': 0
'c': 100
'b': 101
'f': 1100
'e': 1101
'd': 111

Frequency from root: 100

```

Fig. 5: Huffman Code

(c) To verify the complexity, we need to measure the execution time of the huffman function for different values of n , so for this, we need to do some modifications to use n with different increasing values. This can be done adding the following functions into the script and modifying the main as follows:

```

1 void run_test(int n) {
2     //Generate data test
3     char* data = (char*)malloc(n * sizeof(char));
4     unsigned* freq = (unsigned*)malloc(n * sizeof(unsigned));
5
6     if (!data || !freq) {
7         printf("Error allocating memory for n=%d\n", n);
8         free(data);
9         free(freq);
10        return;
11    }
12
13    for (int i = 0; i < n; i++) {
14        data[i] = (char)((i % 95) + 32); //ascii characters
15        freq[i] = (rand() % 1000) + 1; // Random frequencies between 1 and 1000
16    }
17
18    //Measure the running time of huffman()
19    struct timeval start, end;
20    double time_taken;
21
22    gettimeofday(&start, NULL); //Start timer
23
24    Node* root = huffman(data, freq, n); //Execute algorithm
25
26    gettimeofday(&end, NULL); // Detener temporizador
27
28    //Calculate the elapsed time in seconds
29    time_taken = (end.tv_sec - start.tv_sec) * 1e6; // seg t microseg
30    time_taken = (time_taken + (end.tv_usec - start.tv_usec)) * 1e-6; // microseg to seg

```

```

31     printf("%d, %f\n", n, time_taken);
32
33     //free memory
34     free_Huffman(root);
35     free(data);
36     free(freq);
37 }
38
39
40
41 int main()
42 {
43     //Initialize the random number generator
44     srand(time(NULL));
45
46     //Define the sizes of n that we want to test
47     int test_sizes[] = {100, 500, 1000, 5000, 10000, 20000, 50000, 100000};
48     int num_tests = sizeof(test_sizes) / sizeof(test_sizes[0]);
49
50     printf("n, time_taken_sec\n");
51     for (int i = 0; i < num_tests; i++) {
52         run_test(test_sizes[i]);
53     }
54
55     return 0;
56 }
57
58

```

Listing 6: "Empirical test"

Running the script with this additions we get the following results

```

n, time_taken_sec
100, 0.000129
500, 0.000697
1000, 0.001579
5000, 0.021852
10000, 0.051352
20000, 0.083478
50000, 0.270347
100000, 0.599437

```

Fig. 6: Empirical Huffman Code

Now, calculating the ratio for each one, we have the following

n	time_taken_sec	ratio: $T(n) / (n * \log_2 n)$
100	0.000129	1.94164E-07
500	0.000697	1.5548E-07
1000	0.001579	1.58442E-07
5000	0.021852	3.55672E-07
10000	0.051352	3.86462E-07
20000	0.083478	2.92132E-07
50000	0.270347	3.46385E-07
100000	0.599437	3.60897E-07

Fig. 7: Huffman ratio

As we see, the ratio is relatively constant for each n , although there's a fluctuation (and a noticeable jump between $n = 1000$ and $n = 5000$), the ratio values do not systematically increase or decrease. From $n = 5000$ onwards, all the values (3.55e-7, 3.86e-7, 2.92e-7, 3.46e-7, 3.60e-7) are in the same order of magnitude and fluctuate around a constant value (approx. 3.5×10^{-7}).

If the complexity were $O(n^2)$, this ratio were increasing (because n^2 grows faster than $n \log n$). If the complexity were $O(n)$, this ratio were decreasing (because n grows slower than $n \log n$).

Since the ratio tends to be constant, the empirical results confirm the theoretical complexity for the Huffman Code

$$T(n) = O(n \log n) \quad (4)$$