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

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>


typedef enum {RED, BLACK} color_t;

typedef struct Node
{
    /*
    Properties for the nodes
    */
    int key;
    color_t color;
    struct Node *left, *right, *parent;
}Node;

typedef struct RBT
{
    /*
    Struct for the tree
    */
    Node *root;
    Node *nil;
}RBT;

RBT* createRBT()
{
    /*
    Creates the initial structure of the RBT, NIL and root pointing to each other
    */
    RBT *tree = (RBT*)malloc(sizeof(RBT));
    tree->nil = (Node*)malloc(sizeof(Node));
    tree->nil->color = BLACK;
    tree->nil->left = tree->nil->right = tree->nil->parent = tree->nil;
    tree->root = tree->nil;
    return tree;
}

Node* createNode(RBT *tree, int key)
{
    /*
    Creates a new node into the RBT
    */
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->key = key;
    newNode->color = RED;
    newNode->left = tree->nil;
    newNode->right = tree->nil;
    newNode->parent = tree->nil;
    return newNode;
}
```

```
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 rigth 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 rigth 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 rigth 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;
```

```
                    left_rotate(T,z);
                }
                /*case 3*/
                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                right_rotate(T, z->parent->parent);
            }
        }
        else
        {
            /*same  as above, but with "right" and "left" exchanged*/
            Node *y = z->parent->parent->left; //y is z's uncle
            if (y->color == RED) //are z's parent and unce both red?
            {
                /*case 1*/
                z->parent->color = RED;
                y->color = BLACK;
                z->parent->parent->color =RED;
            }
            else
            {
                if (z == z->parent->left)
                {
                    /* case 2 */
                    z = z->parent;
                    right_rotate(T,z);
                }
                /*case 3*/
                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                left_rotate(T, z->parent->parent);
            }
        }
    }
    T->root->color = BLACK;
}


void RB_insert(RBT *T, int key)
{
    Node *z = createNode(T, key);
    Node *x = T->root; //node being compared with z
    Node *y = T->nil; //y will be parent of z

    while (x != T->nil) // descend until reaching the sentinel
    {
        y = x;
        if (z->key < x->key)
        {
            x = x->left;
        }
        else
        {
            x = x->right;
        }
    }
    z->parent = y; //found the location - insert z with parent y
    if (y == T->nil)
    {
        T->root = z; // tree T was empty
    }
    else if (z->key < y->key)
    {
        y->left = z;
    }
    else
    {
        y->right = z;
    }
    //z->left = T->nil; //both of z's children are the sentinel
    //z->right = T->nil;
    //z->color = RED; //the new node starts out red
    RB_insert_fixup(T,z);
}
```

```
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 */
```

```
                w->color = RED;
                x = x->parent;
            }
            else
            {
                if (w->right->color == BLACK)
                {
                    /* case 3 */
                    w->left->color = BLACK;
                    w->color = RED;
                    right_rotate(T, w);
                    w = x->parent->right;
                }
                /*case 4*/
                w->color = x->parent->color;
                x->parent->color = BLACK;
                w->right->color = BLACK;
                left_rotate(T, x->parent);
                x = T->root;
            }
        }
        else
        {
            /*same as above, but left and right exchanged*/
            Node *w = x->parent->left; //w is x's sibling?
            if (w->color == RED)
            {
                /*Case 1*/
                w->color = BLACK;
                x->parent->color = RED;
                right_rotate(T, x->parent);
                w = x->parent->left;
            }
            if (w->right->color == BLACK && w->left->color == BLACK)
            {
                /* case 2 */
                w->color = RED;
                x = x->parent;
            }
            else
            {
                if (w->left->color == BLACK)
                {
                    /* case 3 */
                    w->right->color = BLACK;
                    w->color = RED;
                    left_rotate(T, w);
                    w = x->parent->left;
                }
                /*case 4*/
                w->color = x->parent->color;
                x->parent->color = BLACK;
                w->left->color = BLACK;
                right_rotate(T, x->parent);
                x = T->root;
            }
        }
    }
    x->color = BLACK;
}


void RB_delete(RBT *T, Node *z)
{
    Node *y = z;
    Node *x;
    color_t y_og_color = y->color;

    if (z->left == T->nil)
    {
        x = z->right;
        RB_transplant(T, z, z->right); //replace z by its right child
    }
    else if(z->right == T->nil)
```

```
    {
        x = z->left;
        RB_transplant(T, z, z->left); //replace z by its left child
    }
    else
    {
        y = tree_minimum(T, z->right);
        y_og_color = y->color;
        x = y->right;
        if (y != z->right) //is y father down the tree?
        {
            RB_transplant(T, y, y->right); //replace y by its right child
            y->right = z->right; //z's right child becomes y's right child
            y->right->parent = y;
        }
        else
        {
            x->parent = y; // in case x is T.nil
        }
        RB_transplant(T, z, y); //replace z by its succesor y
        y->left = z->left; //and give z's left child to y, which had no left child
        y->left->parent = y;
        y->color = z->color;
    }
    if (y_og_color == BLACK) //if any red-black violations ocurred
    {
        RB_delete_fixup(T, x); //correct them
    }
    free(z);
}


void RB_delete_by_key(RBT *T, int key) {
    Node *z = tree_search(T, T->root, key);
    if (z != T->nil)
    {
        RB_delete(T, z);
    }
}


void inOrder(RBT *T, Node *x)
{
    if (x != T->nil)
    {
        inOrder(T, x->left);
        printf("%d(%s) ", x->key, (x->color == RED) ? "RED" : "BLACK");
        inOrder(T, x->right);
    }
}


void print_tree(RBT *T, Node *root, int space)
{
    /*print the RBT*/
    if (root == T->nil) return;
    space += 5;

    print_tree(T, root->right, space);
    printf("\n");

    for (int i = 5; i < space; i++)
    {
        printf(" ");
    }
    printf("%d(%c)\n", root->key, (root->color == RED) ? 'R' : 'B');
    print_tree(T, root->left, space);
}

/*
================================================================
NEW 'main' for empirical tests
================================================================
*/
```

```
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
```

```
    // We keep it constant to isolate the effect of 'n'
    const int NUM_OPS = 10000;

    printf("Starting empirical tests for RBT...\n");
    printf("================================================================\n");
    printf("%-10s | %-20s | %-25s\n", "N", "Insertion time (s)", "Search time (s)");
    printf("----------------------------------------------------------------\n");

    for (int i = 0; i < num_sizes; i++)
    {
        int N = N_values[i];
        RBT *T = createRBT();

        // --- Insertion test ---
        // Measures the time to insert N random elements
        // Large random key range (0 to N*10) to reduce collisions

        double start_time = get_time();
        for (int j = 0; j < N; j++)
        {
            RB_insert(T, rand() % (N * 10));
        }
        double end_time = get_time();
        double insert_time = end_time - start_time;

        // --- Search test ---
        // Measures the time for NUM_OPS random searches
        start_time = get_time();
        for (int j = 0; j < NUM_OPS; j++)
        {
            tree_search(T, T->root, rand() % (N * 10));
        }
        end_time = get_time();
        double search_time = end_time - start_time;

        // Print results for this N
        printf("%-10d | %-20.6f | %-25.6f\n", N, insert_time, search_time);

        // --- DELETE TEST (Optional) ---
        // Measuring this is similar to searching.
        // We focus on insertion and searching, which are the most straightforward.

        // Free the tree's memory for the next iteration
        destroyRBT(T);
    }

    printf("================================================================\n");
    printf("Tests end\n");

    return 0;
}
```

Listing 1: "RBT"

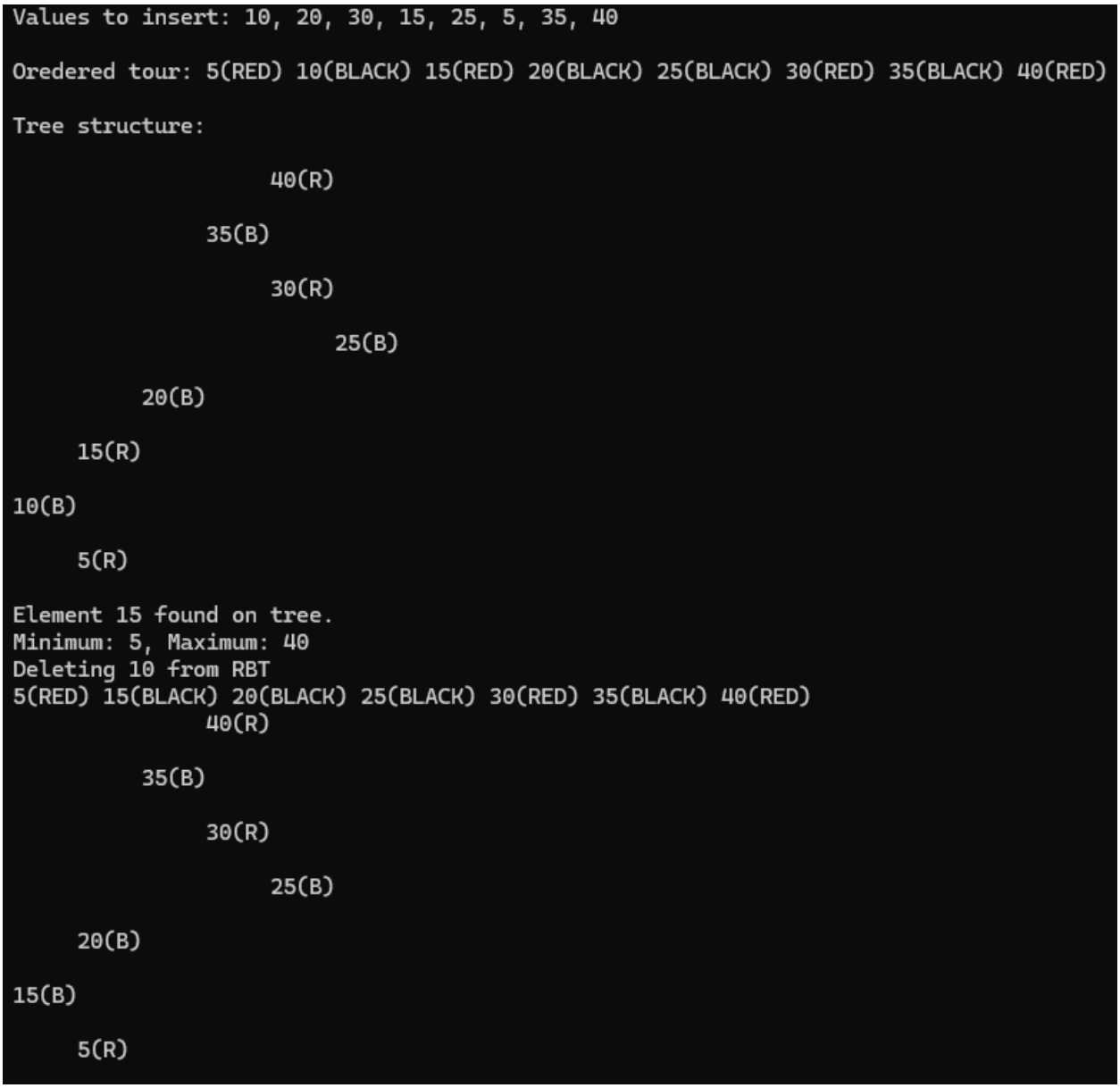

Fig. 1: RBT output

(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 $O(logn)$, 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;
```

```
    }
    free_tree(T, x->left);
    free_tree(T, x->right);
    free(x);
}

void destroyRBT(RBT *T)
{
    free_tree(T, T->root); // Free all nodes
    free(T->nil);          // Free NIL
    free(T);               // Free tree struct
}


int main()
{
    //Initialize the random number generator
    srand(time(NULL));

    //n sizes to test
    int N_values[] = {10000, 50000, 100000, 500000, 1000000};
    int num_sizes = sizeof(N_values) / sizeof(N_values[0]);

    // Number of searches/eliminations to be performed per test
    // We keep it constant to isolate the effect of 'n'
    const int NUM_OPS = 10000;

    printf("Starting empirical tests for RBT...\n");
    printf("================================================================\n");
    printf("%-10s | %-20s | %-25s\n", "N", "Insertion time (s)", "Search time (s)");
    printf("----------------------------------------------------------------\n");

    for (int i = 0; i < num_sizes; i++)
    {
        int N = N_values[i];
        RBT *T = createRBT();

        // --- Insertion test ---
        // Measures the time to insert N random elements
        // Large random key range (0 to N*10) to reduce collisions

        double start_time = get_time();
        for (int j = 0; j < N; j++)
        {
            RB_insert(T, rand() % (N * 10));
        }
        double end_time = get_time();
        double insert_time = end_time - start_time;

        // --- Search test ---
        // Measures the time for NUM_OPS random searches
        start_time = get_time();
        for (int j = 0; j < NUM_OPS; j++)
        {
            tree_search(T, T->root, rand() % (N * 10));
        }
        end_time = get_time();
        double search_time = end_time - start_time;

        // Print results for this N
        printf("%-10d | %-20.6f | %-25.6f\n", N, insert_time, search_time);

        // --- DELETE TEST (Optional) ---
        // Measuring this is similar to searching.
        // We focus on insertion and searching, which are the most straightforward.

        // Free the tree's memory for the next iteration
        destroyRBT(T);
    }

    printf("================================================================\n");
    printf("Tests end\n");

    return 0;
```

```
    }


```

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 = N=10,000, we obtain a search time of 0.006199 seconds, and for N=1000,000 (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

$$T(n) = O(log(n)) \tag{1}$$

For the insertion time, in theory we should have $O(nlog(n))$. 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

$$T(n) = O(nlog(n)) \tag{2}$$

The time increase, but not as fast as the worst case O(nlog(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

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <limits.h>


void matrix_chain_order(int p[], int n, int **m, int **s)
{
    int i, l, j, k, q;

    for (i = 1; i <= n; i++) //chain lenght 1
    {
        m[i][i] = 0;
    }
    for (l = 2; l <= n; l++) // l is the chain lenght
    {
        for (i = 1; i <= n-l+1; i++) //chain begins at Ai
        {
            j = i + l - 1; //chain ends at Aj
            m[i][j] = INT_MAX; //infinite representation
            for (k = i; k <= j-1; k++) //try Ai:kAk+1
            {
```

```
                q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
                if (q < m[i][j])
                {
                    m[i][j] = q; //remember this cost
                    s[i][j] = k; // remember this index
                }
            }
        }
    }
}


void print_optimal_parens(int **s, int i, int j)
{
    if (i == j)
    {
        printf("A%d", i);
    }
    else
    {
        printf("(");
        print_optimal_parens(s,i,s[i][j]);
        print_optimal_parens(s,s[i][j] + 1, j);
        printf(")");
    }
}


int main()
{
    /*
    matrix dimentions:
    A1 = 10x55
    A2 = 55x25
    A3 = 25x5
    A4 = 4x40
    A5 = 40x20
    A6 = 20x35
    */
    int p[] ={ 10, 55, 25, 5, 40, 20, 35};
    int n = sizeof(p) / sizeof(p[0]) - 1;
    int **m = (int **)malloc((n + 1) * sizeof(int *));
    int **s = (int **)malloc((n + 1) * sizeof(int *));

    for (int i = 0; i <= n; i++)
    {
        m[i] = (int *)malloc((n + 1) * sizeof(int));
        s[i] = (int *)malloc((n + 1) * sizeof(int));
    }
    // Init matrix
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= n; j++) {
            m[i][j] = 0;
            s[i][j] = 0;
        }
    }

    matrix_chain_order(p,n,m,s);
    printf("Minumum cost for multiplication: %d\n", m[1][n]);
    printf("Optimal order: ");
    print_optimal_parens(s, 1, n);
    printf("\n");

    printf("m table (minimum costs):\n");
    for (int i = 1; i <= n; i++)
    {
        for (int j = 0; j <= n; j++)
        {
            if (i <= j)
            {
                printf("%6d ", m[i][j]);
            }
            else
            {
```

```
                printf("       ");
            }
        }
        printf("\n");
    }

    printf("\ns table(divition points):\n");
    for (int i = 1; i < n; i++) {
        for (int j = 2; j <= n; j++) {
            if (i < j) {
                printf("%d ", s[i][j]);
            } else {
                printf("  ");
            }
        }
        printf("\n");
    }

    //free memory
    for (int i = 0; i <= n; i++) {
        free(m[i]);
        free(s[i]);
    }
    free(m);
    free(s);

    return 0;
}
```

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:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <limits.h>

void matrix_chain_order(int p[], int n, int **m, int **s) {
    int i, l, j, k, q;

    for (i = 1; i <= n; i++) {
        m[i][i] = 0;
    }
    for (l = 2; l <= n; l++) {
```

```
        for (i = 1; i <= n - l + 1; i++) {
            j = i + l - 1;
            m[i][j] = INT_MAX;
            for (k = i; k <= j - 1; k++) {
                q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
                if (q < m[i][j]) {
                    m[i][j] = q;
                    s[i][j] = k;
                }
            }
        }
    }
}

void generate_random_dimensions(int p[], int n, int max_dim) {
    for (int i = 0; i <= n; i++) {
        p[i] = rand() % max_dim + 1;
    }
}

// Function to avoid compiler optimizations
volatile int dummy_result = 0;

int main() {
    srand(time(NULL));

    // MUCH larger sizes and fewer stitches
    int sizes[] = {100, 200, 300, 400, 500, 600, 700, 800};
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);
    int max_dim = 100;
    int repetitions = 3; // Fewer repeats but larger sizes

    printf("n,Time(s)\n");

    for (int idx = 0; idx < num_sizes; idx++) {
        int n = sizes[idx];

        int *p = (int *)malloc((n + 1) * sizeof(int));
        generate_random_dimensions(p, n, max_dim);

        int **m = (int **)malloc((n + 1) * sizeof(int *));
        int **s = (int **)malloc((n + 1) * sizeof(int *));
        for (int i = 0; i <= n; i++) {
            m[i] = (int *)malloc((n + 1) * sizeof(int));
            s[i] = (int *)malloc((n + 1) * sizeof(int));
        }

        // Measure time with clock()
        clock_t total_ticks = 0;
        for (int rep = 0; rep < repetitions; rep++) {
            clock_t start = clock();
            matrix_chain_order(p, n, m, s);
            clock_t end = clock();
            total_ticks += (end - start);

            // Use the result to avoid optimizations
            dummy_result += m[1][n];
        }
        double avg_time = ((double)total_ticks / CLOCKS_PER_SEC) / repetitions;

        printf("%d,%.6f\n", n, avg_time);
        fflush(stdout); // Force immediate printing

        // free memory
        free(p);
        for (int i = 0; i <= n; i++) {
            free(m[i]);
            free(s[i]);
        }
        free(m);
        free(s);
    }

    printf("Dummy result: %d\n", dummy_result); // Prevents optimization from compiler
```

```
    return 0;
}
```

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:

$$n = 100 : T(n)/n3 = 0.002667/100^3 = 2.667 * 10^{-9} \tag{3}$$

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

(a) Implement the C min heap.
(b) Then the rest of the algorithm.
(c) Empirically Answer if the theoretical complexities are correct.

Answer

Script [(a) and (b)]

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <limits.h>

typedef struct Node
{
    /*
    Properties for the nodes
    */
    unsigned freq;
    char data;
    struct Node *left, *right;
}Node;

Node* createNode(char data, unsigned freq)
{
    /*
    Creates a new node for Huffman
    */
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->freq = freq;
    newNode->left = newNode->right = NULL;
    return newNode;
}

```

```
typedef struct min_heap {
    unsigned size;
    unsigned capacity;
    Node** array;
} min_heap;

int parent(int i)
{return (i-1)/2;}

int left(int i)
{return 2*i+1;}

int right(int i)
{return 2*i+2;}

min_heap* create_min_heap(unsigned capacity)
{
    min_heap* mini_heap = (min_heap*)malloc(sizeof(min_heap));
    mini_heap->size = 0;
    mini_heap->capacity = capacity;
    mini_heap->array = (Node**)malloc(mini_heap->capacity * sizeof(Node*));
    return mini_heap;
}

void min_heapify(min_heap *mini_heap, int i)
{
    int l = left(i);
    int r = right(i);
    int smallest;

    if (l < mini_heap->size && mini_heap->array[l]->freq < mini_heap->array[i]->freq)
    {
        smallest = l;
    }
    else
    {
        smallest = i;
    }
    if (r < mini_heap->size && mini_heap->array[r]->freq < mini_heap->array[smallest]->freq)
    {
        smallest = r;
    }
    if (smallest != i)
    {
        //exchange nodes
        Node* temp = mini_heap->array[i];
        mini_heap->array[i] = mini_heap->array[smallest];
        mini_heap->array[smallest] = temp;
        min_heapify(mini_heap, smallest);
    }
}

void build_min_heap(min_heap* mini_heap)
{
    mini_heap->size = mini_heap->capacity;
    for (int i = mini_heap->capacity/2-1; i >= 0; i--)
    {
        min_heapify(mini_heap, i);
    }
}

Node* min_heap_extract_min(min_heap* mini_heap)
{
    if (mini_heap->size < 1)
    {
        printf("heap overflow\n");
        return NULL;
    }

    Node* min = mini_heap->array[0];
    mini_heap->array[0] = mini_heap->array[mini_heap->size - 1];
    mini_heap->size--;
    min_heapify(mini_heap, 0);
    return min;
```

```
}

void min_insert(min_heap* mini_heap, Node* x)
{
    if (mini_heap->size >= mini_heap->capacity)
    {
        printf("Heap overflow\n");
        return;
    }

    int i = mini_heap->size;
    mini_heap->array[i] = x;
    mini_heap->size++;

    // Ajustar hacia arriba
    while (i > 0 && mini_heap->array[parent(i)]->freq > mini_heap->array[i]->freq)
    {
        // Intercambiar con el padre
        Node* temp = mini_heap->array[i];
        mini_heap->array[i] = mini_heap->array[parent(i)];
        mini_heap->array[parent(i)] = temp;
        i = parent(i);
    }
}


Node* huffman(char data[], unsigned freq[], int n)
{
    min_heap* Q = create_min_heap(n);

    //Q = C
    for (int i = 0; i < n; i++)
    {
        Q->array[i] = createNode(data[i], freq[i]);
    }
    Q->size = n;
    build_min_heap(Q);

    for (int i = 1; i <= n-1; i++)
    {
        /* allocate a new node z */
        Node* z = createNode('$', 0); //$ indicates internal node
        z->left = min_heap_extract_min(Q);
        z->right = min_heap_extract_min(Q);
        z->freq = z->left->freq + z->right->freq;
        min_insert(Q,z);
    }
    return min_heap_extract_min(Q);
}

void print_huffman(Node* root, int arr[], int top) {
    if (root->left) {
        arr[top] = 0;
        print_huffman(root->left, arr, top + 1);
    }

    if (root->right) {
        arr[top] = 1;
        print_huffman(root->right, arr, top + 1);
    }

    if (!root->left && !root->right) {
        printf("'%c': ", root->data);
        for (int i = 0; i < top; i++) {
            printf("%d", arr[i]);
        }
        printf("\n");
    }
}


void free_Huffman(Node* root) {
    if (root == NULL) return;
    free_Huffman(root->left);
```

```
    free_Huffman(root->right);
    free(root);
}

void run_test(int n) {
    //Generate data test
    char* data = (char*)malloc(n * sizeof(char));
    unsigned* freq = (unsigned*)malloc(n * sizeof(unsigned));

    if (!data || !freq) {
        printf("Error allocating memory for n=%d\n", n);
        free(data);
        free(freq);
        return;
    }

    for (int i = 0; i < n; i++) {
        data[i] = (char)((i % 95) + 32); //ascii characters
        freq[i] = (rand() % 1000) + 1;    // Random frequencies between 1 and 1000
    }

    //Measure the running time of huffman()
    struct timeval start, end;
    double time_taken;

    gettimeofday(&start, NULL); //Start timer

    Node* root = huffman(data, freq, n); //Execute algorithm

    gettimeofday(&end, NULL); // Detener temporizador

    //Calculate the elapsed time in seconds
    time_taken = (end.tv_sec - start.tv_sec) * 1e6; // seg t microseg
    time_taken = (time_taken + (end.tv_usec - start.tv_usec)) * 1e-6; // microseg to seg

    printf("%d, %f\n", n, time_taken);

    //free memory
    free_Huffman(root);
    free(data);
    free(freq);
}


int main()
{
    /*
    char data[] = {'a', 'b', 'c', 'd', 'e', 'f'};
    unsigned freq[] = {45, 13, 12, 16, 9, 5};
    int n = sizeof(data) / sizeof(data[0]);

    printf("Character freq: \n");
    for (int i=0; i<n; i++)
    {
        printf("%c: %u\n", data[i], freq[i]);
    }

    printf("Huffman tree\n");
    Node* root = huffman(data, freq, n);

    printf("\n");
    printf("Huffman codes: \n");
    int arr[100], top = 0;
    print_huffman(root, arr, top);

    printf("\n");
    printf("Frequency from root: %u\n", root->freq);

    free_Huffman(root);
    */
   //Initialize the random number generator
    srand(time(NULL));

    //Define the sizes of n that we want to test
```

```
    int test_sizes[] = {100, 500, 1000, 5000, 10000, 20000, 50000, 100000};
    int num_tests = sizeof(test_sizes) / sizeof(test_sizes[0]);

    printf("n, time_taken_sec\n");
    for (int i = 0; i < num_tests; i++) {
        run_test(test_sizes[i]);
    }

    return 0;
}
```

Listing 5: ”Huffman code”

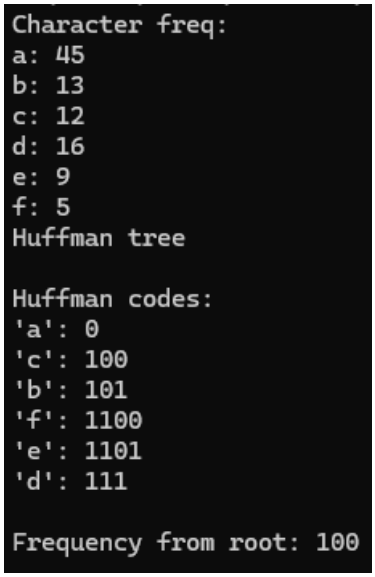

Fig. 5: Huffman Code

(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:

```
    void run_test(int n) {
      //Generate data test
      char* data = (char*)malloc(n * sizeof(char));
      unsigned* freq = (unsigned*)malloc(n * sizeof(unsigned));

      if (!data || !freq) {
        printf("Error allocating memory for n=%d\n", n);
        free(data);
        free(freq);
        return;
      }

      for (int i = 0; i < n; i++) {
        data[i] = (char)((i % 95) + 32); //ascii characters
        freq[i] = (rand() % 1000) + 1;    // Random frequencies between 1 and 1000
      }

      //Measure the running time of huffman()
      struct timeval start, end;
      double time_taken;

      gettimeofday(&start, NULL); //Start timer

      Node* root = huffman(data, freq, n); //Execute algorithm

      gettimeofday(&end, NULL); // Detener temporizador

      //Calculate the elapsed time in seconds
      time_taken = (end.tv_sec - start.tv_sec) * 1e6; // seg t microseg
      time_taken = (time_taken + (end.tv_usec - start.tv_usec)) * 1e-6; // microseg to seg
```

```
    printf("%d, %f\n", n, time_taken);

    //free memory
    free_Huffman(root);
    free(data);
    free(freq);
}


int main()
{
    //Initialize the random number generator
    srand(time(NULL));

    //Define the sizes of n that we want to test
    int test_sizes[] = {100, 500, 1000, 5000, 10000, 20000, 50000, 100000};
    int num_tests = sizeof(test_sizes) / sizeof(test_sizes[0]);

    printf("n, time_taken_sec\n");
    for (int i = 0; i < num_tests; i++) {
        run_test(test_sizes[i]);
    }

    return 0;
}
```

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 $nlogn$). If the complexity were $O(n)$, this ratio were decreasing ((because n grows slower than $nlogn$).

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

$$T(n) = O(nlogn) \tag{4}$$