Lista 3 de EDA 2

Arthur Luis Komatsu Aroeira
13/0102750

Pedro Kelvin de Castro M Batista
13/0129674

26 de setembro de 2017

Perguntas e Respostas

1. **Implemente o Merge Sort de forma iterativa. Apresente de forma gráfica as trocas enquanto elas ocorrem**

   Para as representações gráficas que se pede na questão, serão utilizadas tabelas para se representar os vetores, onde os números nas colunas da primeira linha indicam as posições de cada elemento do vetor. E os números nas colunas da segunda linha indicam os elementos do vetor.

   A seguir são apresentados os passos para a ordenação de um vetor de 7 posições utilizando o Merge Sort Iterativo, que foi implementado e se encontra logo após as representações.

   O Merge Sort se basea em dividir o vetor em subvetores menores (dividindo sempre pela metade deles), onde se busca fazer a ordenação com o menor número de elementos possíveis. Após serem feitas as ordenações nesses subvetores, ocorre o "merge"de cada um deles, ou seja, eles são fundidos em um só, de modo que no final o vetor original esteja ordenado.

   Na primeira parte desse algoritmo, são criadas cópias de tamanho 1 do vetor original, ou seja, vetores auxiliares de tamanho 1 são criados e se tornam cópias do vetor original. Nos exemplos a seguir eles serão chamados de L (subvetor da esquerda) e de R (subvetor da direita).

0	1	2	3	4	5	6
13	1	4	3	5	6	2

Vetor V

0
13

Vetor L

0
1

Vetor R

   Na representação acima L e R copiam os valores da primeira e da segunda posição do vetor original. Ambos comparam seus valores, e o menor deles é inserido no vetor original. Como 1 é menor que 13, o vetor original recebe o valor 1 na sua primeira posição, e o valor 13 é copiado para a próxima.

0	1	2	3	4	5	6
1	13	4	3	5	6	2

Vetor V

0
4

Vetor L

0
3

Vetor R

Acima se pode ver o vetor V com as alterações sofridas. Esse processo vai continuar até que se peguem todos os possíveis pares de subvetores de tamanho 1.

0	1	2	3	4	5	6
1	13	3	4	5	6	2

Vetor V

0
5

Vetor L

0
6

Vetor R

0	1	2	3	4	5	6
1	13	3	4	5	6	2

Vetor V

0	1
1	13

Vetor L

0	1
3	4

Vetor R

Após serem pegos os subvetores de tamanho 1, se pega agora os pares de subvetores de tamanho 2. No exemplo a cima as posições iniciais de L e R são comparadas. Como o valor 1 é menor que o valor 3, é inserido no vetor V o menor elemento correspondente ao vetor L (note que o valor 1 já era o menor elemento no vetor V, mas isso não é levado em conta, o que se leva em conta são as comparações entre L e R).

Os valores são todos comparados e inseridos no vetor V de forma ordenada. Após as comparações entre os vetores L e R o resultado é um novo vetor com suas 4 primeiras posições ordenadas.

0	1	2	3	4	5	6
1	3	4	13	5	6	2

Vetor V

0	1
5	6

Vetor L

0
2

Vetor R

O processo se repete até que se peguem todos os possíveis pares de vetores de tamanho 2 (no caso a cima, o maior tamanho possível do subvetor R era 1).

0	1	2	3	4	5	6
1	3	4	13	2	5	6

Vetor V

0	1	2	3
1	3	4	13

Vetor L

0	1	2
2	5	6

Vetor R

Em seguida, como o esperado, agora são criadas cópias para o vetor V com o dobro do tamanho dos subvetores usados nas trocas anteriores. Nesse último passo, são pegos subvetores de tamanho 4 (veja que só é possível pegar um subvetor R de tamanho no máximo igual a 3), e assim como anteriormente, são feitas as comparações e essas são enseridas no vetor V.

0	1	2	3	4	5	6
1	2	3	4	5	6	13

Vetor V

Por fim se tem o vetor V ordenado pelo Merge Sort iterativo acima.

Abaixo se encontra o código da implementação do Merge Sort Iterativo

```
#include<bits/stdc++.h>

using namespace std;

void merge(vector<int> &v, int l, int m, int r);
void mergeSort(vector<int> &v, int n);

int main()
{
    vector<int> v = {13,1,4,3,5,6,2};
    vector<int> aux = v;

    mergeSort(v, v.size());

    cout << "Vetor original : ";

    for(auto j:aux)
      cout << j << ' ';
    cout << endl;

    cout << "Vetor ordenado : ";

    for(auto i:v)
        cout << i << ' ';
    cout << endl;

    return 0;
}

void mergeSort(vector<int> &v, int n){

   for(int tam_sub_vetor = 1; tam_sub_vetor <= n-1; tam_sub_vetor = 2*
      ↪ tam_sub_vetor)
```

```
    {
        for(int inicio_esq = 0; inicio_esq < n-1; inicio_esq += 2*
            ↪ tam_sub_vetor)
        {
            int meio = inicio_esq + tam_sub_vetor - 1;
            int inicio_dir = min(inicio_esq + 2*tam_sub_vetor - 1, n-1);
            merge(v, inicio_esq, meio, inicio_dir);
        }
    }
}

void merge(vector<int> &v, int l, int m, int r){

    int i, j, k;
    int size1 = m - l + 1;
    int size2 =  r - m;

    int L[size1], R[size2];

    for (i = 0; i < size1; i++)
        L[i] = v[l + i];
    for (j = 0; j < size2; j++)
        R[j] = v[m + 1+ j];

    i = 0;
    j = 0;
    k = l;

    while (i < size1 && j < size2)
    {
        if (L[i] <= R[j])
        {
            v[k] = L[i];
            i++;
        }
        else
        {
            v[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < size1)
    {
        v[k] = L[i];
        i++;
        k++;
    }

    while (j < size2)
    {
        v[k] = R[j];
        j++;
        k++;
    }

}
```

2. **Ordene uma lista de nomes usando o Radix Sort com Counting Sort. Use**

qualquer lista pública de nomes. Lembre-se de preencher os espaços vazios com caracteres para que os nomes tenham o mesmo tamanho.

A seguir, o código implementado para ordenar um vetor de strings usando radix sort:

```
#include <bits/stdc++.h>
using namespace std;

#define debug(x) cerr << fixed << #x << " is " << x << endl;

void CountSort(vector<string> &v, int ind)
{
        int n = v.size();
        vector<string> result(n);
        map<char, int> hist; //histograma

        for (int i = 0; i < n; i++)
                if(ind < v[i].size())
                        hist[v[i][ind]]++;
                else
                        hist[' ' - 1]++;

        for (char c = ' ' - 1; c <= 'z'; c++)
                hist[c] += hist[c - 1];

        for (int i = n - 1 ; i >= 0 ; i--)
        {
                if(ind < v[i].size())
                {
                        result[hist[ v[i][ind] ] - 1] = v[i];
                        hist[ v[i][ind] ]--;
                }
                else
                {
                        result[hist[ ' '-1  ] - 1] = v[i];
                        hist[ ' '-1 ]--;
                }
        }

        for (int i = 0 ; i < n ; i++)
                v[i] = result[i];
}

void RadixSort(vector<string> &v)
{
        int n = v.size();
        int maior = v[0].size();
        for (int i = 1; i < v.size(); i++)
                if (v[i].size() > maior)
                        maior = v[i].size();

        for (int ind = maior - 1; ind >= 0; ind--)
                CountSort(v, ind);
}


int main()
{
        string s;
        vector<string> v;
        int N = 0;
```

```
        while(cin >> s && N < 100000)
        {
                v.push_back(s);
                N++;
        }
    RadixSort(v);
    for(auto i : v)
        cout << i << endl;

        return 0;
}
```

3. **Altere o código da mediana em tempo linear para utilizar grupos de sete elementos ao invés de cinco.**

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

#define TRUE 1
#define FALSE 0

const int SIZE_NUMBERS = 10003;

// sorts a small group with insertion sort
// group has 7 itens or less
void sort_group(int *numbers, int i, int group_size) {

        for (int j = i + 1; j < i + group_size; j++) {
                int k = j;
                while ((k > i) && (numbers[k] < numbers[k-1])) {
                        int aux = numbers[k-1];
                        numbers[k-1] = numbers[k];
                        numbers[k] = aux;
                        k--;
                }
        }
}

// breaks numbers into groups of five itens
// last group may be not complete
void divide_groups(int *numbers, int size) {

        int i = 0;

        /*  if size = 14
         *  two groups of 7 itens
         *  last group has 4 itens
         */

        for (i = 0; i <= (size - 7); i = i + 7) {
                sort_group(numbers, i, 7);
        }

        int last_size = size % 7; // last group size

        if (last_size > 0) {
                sort_group(numbers, i, last_size);
        }
        else {
```

```
                // do nothing, there is no incomplete group
        }

}

// prints numbers in groups of five itens
// each line is a group
// last line may be incomplete
void print_numbers(int *numbers, int size) {

        for (int i = 0; i < size; i = i + 7) {
                for (int j = 0; (j < 7) && (i + j < size); j++) {
                        printf("%d ", numbers[i + j]);
                }
                printf("\n");
        }

        printf("\n");
}

// initiates a array with random numbers
// guarantees only different numbers
void randomize_input(int *input_numbers, int size){

        // numbers are generated from 0 to MAX_NUMBER-1
        const int MAX_NUMBER = SIZE_NUMBERS * 7;

        // array to mark already generated numbers
        int generated[MAX_NUMBER];

        // marks all numbers as "not generated"
        for (int i = 0; i < MAX_NUMBER; i++) {
                generated[i] = FALSE;
        }

        // generates a different number for each position
        for (int i = 0; i < size; i++) {

        int is_different = FALSE;
        int rand_number = 0;

                // wait for a different number
                do {
                        rand_number = rand() % MAX_NUMBER;

                        if (generated[rand_number] == FALSE) {
                                is_different = TRUE;
                                generated[rand_number] = TRUE;
                        }
                } while (!is_different);

                // finally initiates position with different number
                input_numbers[i] = rand_number;
        }
}

// copy median of each group to a second array
// caution with last group
void medians_copy(int *source, int source_size, int *destiny, int
    ↪ destiny_size) {
```

```
        int number_of_complete_groups = source_size / 7;
    int i = 0;

        // copy medians of complete groups
        for (i = 0; i < number_of_complete_groups; i++) {
                destiny[i] = source[i * 7 + 2];
        }

        // from now on, deals with last group

        int last_group_size = source_size % 7;

        if (last_group_size > 0) {

            // last group first position
            int first_position = i * 7;

                int last_median = 0;

                // defines position of last median accordingly to group
                    ↪ size
                // extract its value
                switch (last_group_size) {
                        case 1:
                        case 2:
                                last_median = source[first_position];
                                break;
                        case 3:
                        case 4:
                                last_median = source[first_position + 1];
                                break;
                        case 5:
                        case 6:
                                last_median = source[first_position + 2];
                                break;
                        default: // last group is not bigger than 4 itens
                                assert(FALSE);
                                break;
                }

                // copy last median to second array
                destiny[destiny_size - 1] = last_median;
        }
        else {
                // do nothing, has only complete groups
        }
}

// copies left or right partition to a new array
int* partition_copy(int *source, int start, int end) {

        int destiny_size = (end + 1) - start;
        int *destiny = (int*) malloc(sizeof(int)*(destiny_size));

        for (int i = start; i <= end; i++) {
                destiny[i - start] = source[i];
        }

        return destiny;
```

```
}

// simple swap of two numbers of array
void swap(int *input, int first_index, int second_index) {
        int aux = input[first_index];
        input[first_index] = input[second_index];
        input[second_index] = aux;
}

// finds first occurence of value
int find_index(int *input, int size, int value) {

        int value_index = -1;

        for (int i = 0; i < size; i++) {
                if (input[i] == value) {
                        value_index = i;
                        break;
                }
        }

        return value_index;
}

// smallers numbers to the left, bigger numbers to the right
// [small numbers][mom][bigger numbers]
// has problems with very small input, 1 or 2 itens
void partition_numbers_around_mom (int *input_numbers,
                                   int input_numbers_size,
                                                                   int
                                                                       ↪ median_of_median
                                                                       ↪ )
                                                                       ↪
                                                                       ↪ {
                                                                       ↪


    // short-circuit, fix me!
        if (input_numbers_size == 2) {
                if (input_numbers[0] > input_numbers[1]) {
                        swap(input_numbers, 0, 1);
                }
                return;
        }
        else if (input_numbers_size == 1) {
                return;
        }

        int mom_index = find_index(input_numbers, input_numbers_size,
            ↪ median_of_medians);
        swap(input_numbers, 0, mom_index);

        int i = 1;
        int j = input_numbers_size - 1;

        while (i < j) {

                // finds a greater in the left side
                while (input_numbers[i] < median_of_medians) {
                        i++;
                }
```

```
                // finds a smaller in the right side
                while (input_numbers[j] > median_of_medians) {
                        j--;
                }

                swap(input_numbers, i, j);
        }

        // both numbers on the middle are inverted
        // mom is at index 0
        swap(input_numbers, 0, i-1);
        swap(input_numbers, 0, i);

        print_numbers(input_numbers, input_numbers_size);
}

// oracle implements Median of Medians algorithm
int ask_oracle_for_median(int *input_numbers, int input_numbers_size) {

        divide_groups(input_numbers, input_numbers_size);
        print_numbers(input_numbers, input_numbers_size);

        int last_medians_size = input_numbers_size / 7;

        if ((input_numbers_size % 7) > 0) {
                last_medians_size = last_medians_size + 1;
        }
        else {
                // do nothing
        }

        int *last_medians = (int*) malloc(sizeof(int)*(last_medians_size))
            ↪ ;
        medians_copy(input_numbers, input_numbers_size, last_medians,
            ↪ last_medians_size);

        while (last_medians_size > 1) {

                printf("LastMedian\n");
                print_numbers(last_medians, last_medians_size);

                divide_groups(last_medians, last_medians_size);

                printf("Sorted LastMedian\n");
            print_numbers(last_medians, last_medians_size);

                int current_medians_size = 0;

                if ((last_medians_size % 7) > 0) {
                        current_medians_size = (last_medians_size / 7) +
                            ↪ 1;
                }
                else {
                        current_medians_size = last_medians_size / 7;
                }

            int *current_medians = (int*) malloc(sizeof(int)*(
                ↪ current_medians_size));
```

```
                medians_copy(last_medians, last_medians_size,
                    ↪ current_medians, current_medians_size);

                free(last_medians);

                last_medians = current_medians;
                last_medians_size = current_medians_size;
        }

        return last_medians[0]; // this is the Median of Medians
}

/* asks the oracle for a good candidate to be the median
   partition numbers around median
   checks if candidate is real median
   if not, select left or right partition
 */
int exoteric_select (int *input_numbers, int input_numbers_size, int
    ↪ k_esimo) {

        printf("Exoteric Select\n");
        printf("K_esimo: %d\n", k_esimo);

        int mom = ask_oracle_for_median(input_numbers, input_numbers_size)
            ↪ ;

        partition_numbers_around_mom(input_numbers, input_numbers_size,
            ↪ mom);
        int mom_index = find_index(input_numbers, input_numbers_size, mom)
            ↪ ;

        printf("Median: %d\n", mom);
        printf("Median index: %d\n\n", mom_index);

        int expected_size = k_esimo;
        int median = -1;

        if (mom_index > expected_size) {

                // oracle guessed too high

                printf("Recursive on L\n");

                int left_size = mom_index;

                int *left_partition = partition_copy(input_numbers, 0,
                    ↪ left_size - 1);
                printf("Left partition:\n");
                print_numbers(left_partition, left_size);

                median = exoteric_select(left_partition, left_size,
                    ↪ k_esimo);
        }
        else if (mom_index < expected_size) {

                // oracle guessed too low

                printf("Recursive on R\n");

                int right_size = (input_numbers_size - 1) - mom_index;
```

```
                int partition_start = mom_index + 1;

                int *right_partition = partition_copy(input_numbers,
                    ↪ partition_start, input_numbers_size - 1);
                printf("Right partition:\n");
                print_numbers(right_partition, right_size);

                k_esimo = k_esimo - (mom_index + 1);
                median = exoteric_select(right_partition, (
                    ↪ input_numbers_size - 1) - mom_index, k_esimo);
        }
        else {

                // oracle is correct!

                printf("Median is %d.\n", mom);
                printf("Index is %d.\n", mom_index);
                median = mom;
        }

        return median;
}

// runs 200 times
// compares exoteric_select/MOM with the median of sorted input
int main(int argc, char *argv[]) {

    int times_correct = 0;

        for (int i = 0; i < 200; i++) {

            srand(time(NULL));

                int input_numbers_size = SIZE_NUMBERS;
                assert(input_numbers_size > 0);

                int *input_numbers = (int*) malloc(sizeof(int)*(
                    ↪ input_numbers_size));

                randomize_input(input_numbers, input_numbers_size);

                // creates copy of input
                // finds median cheating - sorts the input, O(n^2)
                int *aux = partition_copy(input_numbers, 0,
                    ↪ input_numbers_size - 1);
                sort_group(aux, 0, input_numbers_size); // sort_group also
                    ↪  works with large groups

                printf("Input\n");
                printf("Size: %d\n", input_numbers_size);
                print_numbers(input_numbers, input_numbers_size);

                int k_esimo = (input_numbers_size - 1) / 2; // median is
                    ↪ at middle

                // finds median in linear time, O(C.n)
                // C is a big constant, but still linear
                // uses exoteric select and median-of-medians algorithms
                int median = exoteric_select(input_numbers,
                    ↪ input_numbers_size, k_esimo);
```

```
            print_numbers(aux, input_numbers_size);
                printf("Expected Median: %d\n", aux[k_esimo]);
                printf("Median: %d\n", median);

                // compares O(C.n) and cheated O(n^2) solution
                if (median == aux[k_esimo]) {

                        // annotates one more right solution
                        times_correct++;
                }

                free(input_numbers);
                free(aux);
        }

        printf("Times correct: %d\n", times_correct);

        return 0;
}
```

4. **Gere 1000 números randômicos de 0 a 10000. Conte as rotações e cronometre a inserção de todos os números em uma Árvore AVL e em uma Árvore VeP. Qual fez mais rotações? Qual foi mais rápida?**

   Os seguintes dados foram obtidos:

   Tabela 1 – Dados obtidos da árvores

	Tempo	**Rotações**
Árvore Vermelha e Preta	0.000182687s	552
Árvore AVL	0.000355763s	627

   Logo, a árvore vermelha e preta foi a mais rápida e a árvore AVL fez mais rotações.