

SCC0220 - Laboratório Introdução à Ciência da Computação II

Relatório de execução do trabalho prático 10

Alunos	NUSP
Alec Campos Aoki	15436800
Juan Henrique Passos	15464826

Trabalho 10 – MSC (Máxima Subsequência Crescente)

Hashtable

□ Comentário

O trabalho prático 10, consiste em dado um n , sendo $0 < n < 10^7$, e n números representados por inteiros entre 0 e 10^9 , achar a máxima subsequência crescente, ou seja, achar a maior sequência de números contínuos nesse vetor em ordem crescente. Deve ser printado o tamanho desse vetor. Nesse viés, implementou-se uma estrutura hashmap, no qual os elementos eram inseridos na estrutura, e se busca o primeiro elemento de uma sequência, ou seja, verifica-se o número $i - 1$, sendo i o número analisado, se existir um $i - 1$, então, enquanto tivermos percorrendo o vetor, chegaremos nele, então saímos da função. Caso o $i - 1$ não exista, então o número analisado é o primeiro de sua sequência, e após isso, checa-se todos os $i + 1$ até não existir, computando a quantidade máxima e guardando. Dessa forma, encontra-se a maior sequência em apenas $O(n)$, pois percorremos o vetor apenas uma vez(o que explica sairmos da função, caso não seja o primeiro da sequência). Para implementação da estrutura hashmap, utilizou-se árvores de busca binárias em cada espaço da tabela, e assim permitiu-se adotar 2 estratégias para evitar colisões: caso a tabela esteja menor que 80% de seu total e haja colisão, apenas adiciona-se o valor a estrutura árvore de busca binária presente naquele espaço, e caso esteja maior ou igual, aplica-se a técnica de Double hash, no qual irá ser aplicado uma lógica, que visa achar outro slot fácil de maneira mais eficiente possível (tal lógica de inserção também é vista na busca).

□ Código

arvorebinaria.h

```
#ifndef AB_H
#define AB_H

typedef struct arvore_ ARVORE;

#include<stdbool.h>

ARVORE *arvore_criar(void);
```

```
void arvore_apagar(ARVORE **arvore);  
void arvore_inserir(ARVORE *arvore, int chave);  
bool arvore_pertence(ARVORE *arvore, int chave);  
int arvore_tamanho(ARVORE *arvore);  
  
#endif
```

arvorebinaria.c

```
#include "arvorebinaria.h"  
  
#include <stdio.h>  
#include <stdlib.h>  
  
typedef struct no_ NO;  
  
struct no_  
{  
    int chave;  
    NO *esq, *dir;  
};  
  
struct arvore_  
{  
    NO *raiz;  
    int tamanho;  
};  
  
// Modularização  
NO* inserir(NO *raiz, int chave);  
void apagar(NO **raiz);  
bool pertence(NO *raiz, int chave);  
NO *remover(NO* raiz, int chave);  
  
ARVORE *arvore_criar()  
{  
    ARVORE *arvore = (ARVORE*) malloc(sizeof(ARVORE));  
    if(arvore != NULL)
```

```
{
    arvore->raiz = NULL;
    arvore->tamanho = 0;
}
return arvore;
}

void arvore_apagar(ARVORE **arvore)
{
    if(*arvore != NULL)
    {
        apagar(&((*arvore)->raiz));
        free(*arvore);
        *arvore = NULL;
    }
}

void apagar(NO **raiz)
{
    if(*raiz == NULL)
        return;

    apagar(&((*raiz)->esq));
    apagar(&((*raiz)->dir));

    free(*raiz);
    *raiz = NULL;
}

void arvore_inserir(ARVORE *arvore, int chave)
{
    if(arvore != NULL)
    {
        arvore->raiz = inserir(arvore->raiz, chave);
        arvore->tamanho++;
    }
}
```

```
NO* inserir(NO *raiz, int chave)
{
    // Adiciona o no quando o no NULO é encontrado.
    if (raiz == NULL) {
        NO* novo_no = (NO*) malloc(sizeof(NO));
        novo_no->chave = chave;
        novo_no->esq = NULL;
        novo_no->dir = NULL;
        return novo_no;
    }

    // Percorra a subárvore esquerda se os dados
    // forem menores que o nó atual
    if (chave < raiz->chave) {
        raiz->esq = inserir(raiz->esq, chave);
        return raiz;
    }

    // Percorra a subárvore direita se os dados
    // forem maiores que o nó atual
    else if (chave > raiz->chave) {
        raiz->dir = inserir(raiz->dir, chave);
        return raiz;
    }
    else
        return raiz;
}

// Retorna se esse elemento pertence ao arvore.
bool arvore_pertence(ARVORE *arvore, int chave)
{
    if(arvore != NULL)
    {
        return pertence(arvore->raiz, chave);
    }
    return false;
}
```

```
// Verifica se o elemento está no NO, caso não, verifica-se para os filhos.
```

```
bool pertence(NO *raiz, int chave)
{
    if (raiz == NULL)
        return false;

    if(raiz->chave == chave)
        return true;

    return pertence(raiz->esq, chave) || pertence(raiz->dir, chave);
}
```

```
// Retorna tamanho da árvore.
```

```
int arvore_tamanho(ARVORE *arvore)
{
    if(arvore != NULL)
    {
        return arvore->tamanho;
    }

    return -1; // ERRO.
}
```

hash.h

```
#ifndef HASH_H
#define HASH_H

#include "arvorebinaria.h"

typedef struct h hash_t;

hash_t *hash_criar(int tamanho);
void hash_inserir(hash_t *hash, int chave);
bool hash_busca(hash_t *hash, int chave);
void hash_apagar(hash_t **hash);

#endif
```

hash.c

```
#include "hash.h"

#include <stdio.h>
#include <stdlib.h>

struct h {
    ARVORE **tabela;
    int tamanho;
    int fator;
};

int hash_index(hash_t *hash, int chave) {
    return chave % hash->tamanho;
}

int hash2_index(hash_t *hash, int chave) {
    return ((hash->tamanho - 1) - (chave % (hash->tamanho - 1)));
}

hash_t *hash_criar(int tamanho) {
    hash_t *hash = (hash_t*) malloc(sizeof(hash_t));
    if(hash != NULL){
        hash->tabela = (ARVORE**) calloc(tamanho, sizeof(ARVORE*)); // Se for nulo, não há
nó.
        if(hash->tabela == NULL) return NULL; /*Não encerra o programa se a memória estiver
cheia*/
        hash->tamanho = tamanho;
        hash->fator = 0; // Fator de balanceamento.
        return hash;
    }
    return NULL;
}

void hash_inserir(hash_t *hash, int chave) {
    int ind = hash_index(hash, chave);
```

```
// Verifica se a posição inicial está livre
if (hash->tabela[ind] == NULL) {
    hash->tabela[ind] = arvore_criar();
    arvore_inserir(hash->tabela[ind], chave);
    hash->fator++;
    return;
}

// Verifica se a tabela está 80% preenchida para usar double hashing
if ((float) hash->fator / hash->tamanho >= 0.8) {
    for (int i = 1; i < hash->tamanho; i++) {
        int index = (ind + i * hash2_index(hash, chave)) % hash->tamanho;
        if (hash->tabela[index] == NULL) {
            hash->tabela[index] = arvore_criar();
            arvore_inserir(hash->tabela[index], chave);
            hash->fator++;
            return;
        }
    }
}

// Insere na árvore do índice encontrado inicialmente em caso de colisão
arvore_inserir(hash->tabela[ind], chave);
}

bool hash_busca(hash_t *hash, int chave) {
    if (!hash || !hash->tabela) // Verifica se a tabela hash existe.
        return false;

    for(int i = 0; i < hash->tamanho; i++) {
        int index = (hash_index(hash, chave) +
                    i*hash2_index(hash, chave)) % (hash->tamanho);

        if (index < 0 || index >= hash->tamanho) // Verificar os limites do índice.
            continue;

        if(hash->tabela[index] == NULL)
```

```
        return false;

        // Após garantir o intervalo permitido
        // e que existi tal posição, chama-se a função
        if(arvore_pertence(hash->tabela[index], chave))
            return true;
    }
    return false;
}

void hash_apagar(hash_t **hash) {
    if(*hash != NULL) {
        for(int i = 0; i < (*hash)->tamanho; i++){
            arvore_apagar(&((*hash)->tabela[i]));
        }
        free((*hash)->tabela);
        (*hash)->tabela = NULL;
        free(*hash);
        *hash = NULL;
    }
    return;
}
```

main.c

```
#include "hash.h"

#include<stdio.h>
#include<stdlib.h>

// Modularização.
int achar_sequencia(hash_t *hash, int chave);

int main(void) {
    int tamanho;

    scanf("%d", &tamanho);
```



```
int *vetor = (int*) malloc(tamanho*sizeof(int));
hash_t *hashmap = hash_criar(tamanho);

for(int i = 0; i < tamanho; i++){
    scanf("%d", &vetor[i]);
    hash_inserir(hashmap, vetor[i]);
}

int maior_seq = 0;

for(int i = 0; i < tamanho; i++){
    // Guarde possivel maior sequencial.
    int aux = achar_sequencia(hashmap, vetor[i]);
    if(maior_seq < aux){
        maior_seq = aux;
    }
}

printf("%d", maior_seq);

// Desalocação de memória.
// hash_apagar(&hashmap);
free(vetor);
vetor = NULL;

return 0;
}

int achar_sequencia(hash_t *hash, int chave){
    // Verificar se é o primeiro da sequencia.
    if(hash_busca(hash, chave-1))
        return -1; // Elemento neutro.
    else{
        int seq = 0; // Guarda tamanho sequencia.
        do{
            chave++; // Checar se existe sucessor.
            seq++; // Aumenta 1 elemento na sequencia.
        }while(hash_busca(hash, chave));
    }
}
```

```
return seq;
```

```
}
```

❏ Saída

```
→ codJuan git:(main) x gcc arvorebinaria.c hash.c main.c -o main -std=c99 -Wall
→ codJuan git:(main) x ./main < ../casosteste/9.in
92

Tempo de execucao: 0.569755s
→ codJuan git:(main) x ./main < ../casosteste/9.in
92

Tempo de execucao: 0.567031s
→ codJuan git:(main) x ./main < ../casosteste/9.in
92

Tempo de execucao: 0.574928s
→ codJuan git:(main) x ./main < ../casosteste/9.in
92

Tempo de execucao: 0.567849s
→ codJuan git:(main) x ./main < ../casosteste/9.in
92

Tempo de execucao: 0.572402s
→ codJuan git:(main) x
```

Ordenação + Busca

❏ Comentário

Para ordenar o vetor utilizamos o algoritmo Mergesort, devido à sua complexidade constante de $O(n \log(n))$. Após isso, percorremos o vetor item a item ($O(n)$), buscando recursivamente o antecessor do item analisado utilizando a busca binária ($O(\log(n))$), gerando uma complexidade total de $O(n \log(n))$. Incrementamos o maior tamanho encontrado conforme a busca encontra o antecessor do item. Notamos, pelos tempos de execução, que essa implementação é mais rápida que a utilizando hashing, apesar de ter maior complexidade. Isso se deve ao fato de que a implementação dessa solução é mais simples e direta que a de hashing, que por sua vez utiliza TADs e árvores, que tem uma implementação mais computacionalmente custosa.

❏ Código

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

```
#define ERRO -1

int buscaBinaria(int v[], int inicio, int fim, int chave);
void intercala(int esquerda[], int tamEsq, int direita[], int tamDir, int v[]);
void mergesort(int v[], int tam);
int tamSeqContinua(int v[], int tam);
void buscaRecurativa(int v[], int tam, int chave, int *tamTemp);

int main(void){
    int quantProd;
    scanf("%d", &quantProd);
    int vetProd[quantProd];

    for(int i = 0; i < quantProd; i++){
        int prod;
        scanf("%d", &prod);
        vetProd[i] = prod;
    }

    mergesort(vetProd, quantProd);

    printf("%d\n", tamSeqContinua(vetProd, quantProd));

    return EXIT_SUCCESS;
}

int tamSeqContinua(int v[], int tam){
    int tamRes = 1;

    for(int i = 0; i < tam; i++){
        int tamTemp = 1;
        buscaRecurativa(v, tam, v[i]-1, &tamTemp);
        if(tamTemp > tamRes) tamRes = tamTemp;
    }

    return tamRes;
}
```

```
void buscaRekursiva(int v[], int tam, int chave, int *tamTemp){
    if(buscaBinaria(v, 0, tam-1, chave) == ERRO) return;

    //else:
    (*tamTemp)++;
    buscaRekursiva(v, tam, chave - 1, tamTemp);

    return;
}

void intercala(int esquerda[], int tamEsq, int direita[], int tamDir, int v[]){
    int i=0, e=0, d=0;

    /*escrevendo de volta no vetor v*/
    while((e < tamEsq) && (d < tamDir)){
        if(esquerda[e] < direita[d]){
            v[i] = esquerda[e];
            e++;
        }
        else{
            v[i] = direita[d];
            d++;
        }
        i++;
    }

    /*caso não sobre elementos em um vetor para compararmos*/
    while(e < tamEsq){
        v[i] = esquerda[e];
        e++;
        i++;
    }
    while(d < tamDir){
        v[i] = direita[d];
        d++;
        i++;
    }
}
```

```
    return;
}

void mergesort(int v[], int tam){
    if(tam == 1){
        return;
    }

    int meio = tam/2;
    int esquerda[meio], direita[tam-meio];

    /*dividindo o vetor v[] em dois (esquerda[] e direita[]), e passando os valores pra cada
    um*/
    int e=0, d=0;
    for(int i=0; i<tam; i++){
        if(i < meio){
            esquerda[e] = v[i];
            e++;
        }
        else{
            direita[d] = v[i];
            d++;
        }
    }

    mergesort(esquerda, meio); //note que v[] é o vetor esquerda[]!
    mergesort(direita, tam-meio);
    intercala(esquerda, meio, direita, tam-meio, v);

    return;
}

int buscaBinaria(int v[], int inicio, int fim, int chave){
    if(inicio > fim) return ERRO;

    int meio = (inicio + fim)/2;
```

```
if(v[meio] == chave){
    return meio;
}
else if(chave < v[meio]){
    return buscaBinaria(v, inicio, meio-1, chave);
}
else{
    return buscaBinaria(v, meio+1, fim, chave);
}
}
```

❑ Saída

```
→ Aula10 git:(main) x gcc main.c -o main -std=c99 -Wall
→ Aula10 git:(main) x ./main < casosteste/9.in
92

Tempo de execucao: 0.182679s
→ Aula10 git:(main) x ./main < casosteste/9.in
92

Tempo de execucao: 0.179657s
→ Aula10 git:(main) x ./main < casosteste/9.in
92

Tempo de execucao: 0.180866s
→ Aula10 git:(main) x ./main < casosteste/9.in
92

Tempo de execucao: 0.185989s
→ Aula10 git:(main) x ./main < casosteste/9.in
92

Tempo de execucao: 0.180647s
→ Aula10 git:(main) x
```