UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA

Disciplina: Organização e Recuperação de Dados
Semestre: 2020.2
Professor: Leandro Carlos de Souza
Data: 24/05/2021
Nome: Thaís Gabrielly Marques de Andrade
Matrícula: 20180135293

Exercícios de Fixação e Aprendizagem II

Questão 1 (2,5 pts)

Sobre Listas de Prioridades:

(a) Explique a sua relação com heaps. Dê um exemplo.

Heap é um método de implementação das listas de prioridades. Porque ele guarda o valor que gere a prioridade das listas, sendo um bom casamento e ainda fornece eficiência. Por exemplo: Um sistema para organizar filas de banco, implementado pelo heap, daria para exibir no telão de saída, qual fila iria para o caixa livre, decidindo pela prioridade da senha.

(b) Considerando a implementação por heap, explique como funciona a inserção de um elemento. Dê um exemplo.

Coloca um elemento no final do vetor, depois verifica se o valor que está sendo inserido é maior que o valor do pai, se for maior acontece a troca. Repete o teste com o novo valor do nó pai. Assim a verificação termina se for encontrado o pai de maior valor ou se alcança a raiz. Por exemplo, seguiria por essa linha de raciocínio:

```
int insere(int a, Heap* h){
        h->vetor[h->n++] = a;
        sobe(a, h->n-1); // nessa função ocorre as verificações
}
```

(c) Considerando a implementação por heap, explique como funciona a remoção de um elemento. Dê um exemplo.

Funciona seguindo esses passos:

1 - Retira o maior valor associado à raiz e coloca no lugar o valor da última folha.

2 - Reduz em uma unidade o número dos elementos do vetor.

3 - Se o valor novo da raiz não for o máximo do conjunto ele desce.

4- Compara o novo valor com o maior valor dos dois filhos e se o maior valor dos filhos for maior que o valor do pai, acontece a troca.

5- A verificação vai acontecer até que alcance um nó que não tem filhos ou se encontrarem filhos com valores menores do que o valor-novo.

Por exemplo:

```
int heap_retira(Heap* h){
        int b = h->vetor[0];
        h->vetor[0] =  h->vetor[--h->n];
        desce(h,0); // nessa função ocorre algumas verificações
        return b;
}
```

Questão 2 (2,5 pts)

(a) Explique os métodos da divisão e da multiplicação para mapeamento de chaves em tabelas de dispersão.

- Método da divisão:

A chave k é dividida pela dimensão da tabela N, e o resto da divisão é utilizado como endereço da chave. Isto é h(k) = k mod N. Resultando em endereços no intervalo.

Mas precisa se atentar ao valor de N, pois nem sempre é a solução ideal, por exemplo existem alguns critérios que podem ajudar, como: para um bom resultado é sensato que N seja um número primo não próximo a uma potência de 2.

- Método da multiplicação:

Esse método é dividido em duas partes:

- ➢ Na primeira multiplica-se a chave K por uma constante fracionária A(0<A<1) e extrai AK.
- ➢ Na segunda, multiplica o valor encontrado por N.

Após esses passos resulta no endereço da chave. Uma vantagem desse método é que ele não é tão crítico quanto o método anterior, pode por exemplo escolher uma potência de 2. E enquanto ao valor de A, para melhor funcionamento, utilizamos A= 0.6180.

(b) Explique como funciona o endereçamento aberto para o tratamento de colisão em tabelas de dispersão. Dê um exemplo explicando a inserção e a busca de uma chave.

Usamos o endereçamento aberto quando a função de dispersão manda a chave de busca para um índice já ocupado, aí é preciso procurar outro índice livre utilizando o incremento circular para armazenar o novo elemento (a chave).

Por exemplo na inserção:

Inserimos as chaves a,b,c na tabela (tab=3).

1. Insere a ->h(a) = 2 e c -> h(c) = 4.
2. Agora insere b -> h(b) = 2.
3. Temos uma colisão com a ->h(a) = 2.
4. A próxima posição vazia é 3.
5. Insere b -> h(b) = 3.

Por exemplo na busca de uma chave:

A forma mais simples de busca é percorrer linearmente até encontrar o registro buscado. Utilizando o exemplo anterior seria preciso percorrer do 0 até a posição 3 para achar o b.

(c) Implemente um TAD para hash com tratamento de colisões utilizando listas encadeadas. Utilize uma hash universal, que muda a cada execução. Não esqueça de fazer uma aplicação testando o seu TAD.

Nome do arquivo: Q2_C.h

```
#include <stdio.h>//responsável por printf
#include <stdlib.h> //responsável por malloc, free...
#include <locale.h> //serve para exibir acento.
```

```c
#define M 20 // tamanho da tabela



typedef struct { //dados das pessoas
    int matricula;
    char nome[50];
} Pessoa;

// nó usado na lista encadeada
typedef struct no {
    Pessoa pessoa;
    struct no *proximo;
} No;

// Lista com um ponteiro para o primeiro nó
typedef struct {
    No *inicio;
    int tam;
} Lista;

// Tabela
Lista *tabela[M];


// Recebe os dados e retorna
Pessoa criarPessoa() {
     setlocale(LC_ALL, "Portuguese"); //Consegue exibir os acentos
nos printf.

    Pessoa p;
    printf("\nDigite o nome: ");
    scanf("%*c");
    fgets(p.nome, 50, stdin);
    printf("\nDigite a matricula: ");
    scanf("%d", &p.matricula);
    return p;
}

// --------------Funções -----------//

// Responsável por exibir
```

```c
void imprimirPessoa(Pessoa p) {
      printf("\tNome: %s\tMatricula: %d\n", p.nome, p.matricula);
}

// Cria uma lista vazia e retorna seu endereço
Lista* criarLista() {
    Lista *l = malloc(sizeof(Lista));
    l->inicio = NULL;
    l->tam = 0;
    return l;
}


void inserirInicio(Pessoa p, Lista *lista) {
    No *no = malloc(sizeof(No));
    no->pessoa = p;
    no->proximo = lista->inicio;
    lista->inicio = no;
    lista->tam++;
}

// Busca um elemento na lista
No* buscarNo(int mat, No *inicio) {

    while(inicio != NULL) {
        if(inicio->pessoa.matricula == mat)
            return inicio;
        else
            inicio = inicio->proximo;
    }
    return NULL;// matricula não encontrada
}

void imprimirLista(No *inicio) {
    while(inicio != NULL) {
        imprimirPessoa(inicio->pessoa);
        inicio = inicio->proximo;
    }
}

// Inicializa a tabela com uma lista vazia em cada posição.
void inicializar(){
    int i;
```

```
    for(i = 0; i < M; i++)
        tabela[i] = criarLista();
}


// Função de espalhamento
int funcaoEspalhamento(int mat){ //também conhecido como hash
    return mat % M;
}

// Cria e insere na tabela
void inserTabela(){
    Pessoa pes = criarPessoa();
    int indice = funcaoEspalhamento(pes.matricula);
    inserirInicio(pes, tabela[indice]);
}

// Busca uma pessoa. Seu retorno é um endereço ou NULL.
Pessoa* buscarPessoaTabela(int mat){
    int indice = funcaoEspalhamento(mat);
    No *no = buscarNo(mat, tabela[indice]->inicio);
    if(no)
        return &no->pessoa;
    else
        return NULL;
}

// Imprime a tabela
void imprimirTabela(){
     setlocale(LC_ALL, "Portuguese"); //Consegue exibir os acentos
nos printf.

    int i;

printf("\n---------------------TABELA-------------------------\n")
;
    for(i = 0; i < M; i++){
        printf("%d Lista, tamanho: %d\n", i, tabela[i]->tam);
        imprimirLista(tabela[i]->inicio);
    }

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

Nome do arquivo: Q2_C.c

```c
/* -----Nome: Thaís Gabrielly Marques
--------Questão 2- C)*/

#include <stdio.h>
#include <stdlib.h>
#include <locale.h> //serve para exibir acento.
#include "Q2_C.h"

int main() {
    setlocale(LC_ALL, "Portuguese"); //Consegue exibir os acentos nos printf.
    int op, mat;
    Pessoa *p;

    inicializar();

    do {
            printf("\n\tBanco de dados dos Funcionários, escolha uma opção:\n");
         printf("\n1 - Imprimir tabela\n2 - Inserir\n3 - Buscar\n0 - Sair\n");
        printf("\nOpção: ");
        scanf("%d", &op);

        switch(op){
        case 1:
            imprimirTabela();
            break;
        case 2:
            inserTabela();
            break;
        case 3:
            printf("Qual a matricula a ser buscada? ");
            scanf("%d", &mat);
            p = buscarPessoaTabela(mat);
            if(p) {
                  printf("\nPessoa encontrada:\n Matrícula: %d\tNome: %s", p->matricula, p->nome);
            } else
                            printf("\nMatrícula não encontrada, tente novamente!\n");
            break;
```

```
        case 0:
            printf("\nAté logo...\n");
            break;

        default:
            printf("\nOpção invalida, tente novamente!\n");
        }
    } while(op != 0);

    return 0 ;
}
```

Observação: Não consegui implementar a segunda parte utilizando o Hash universal propriamente dito.

Questão 3 (5,0pts)

Para cada um dos seguintes métodos de ordenação:

Explique a idéia aplicada em seu funcionamento e escreva um pseudo-código:

(a) Bubble sort.

A ideia do Bubble é ordenar valores, onde o elemento da posição atual é comparado com o elemento da posição próxima e se o elemento da atual for maior é realizado uma troca de posições. Caso contrário, passa para o próximo par de comparações.

Pseudo-código:

```
Bubble_sort(t: tabela, n: inteiro)
  para j -> 1 até n-1 faça
    para i -> até n-1 faça
      se t[i]>t[i+1] então
        aux -> t[i];
        t[i] -> t[i+1];
        t[i+1] -> aux;
      fim-se
```

```
      fim-para
   fim-para
fim-Bubble_sort
```

(b) Selection sort.

A ideia do Selection Sort é passar o menor valor do array (ou maior dependendo da ordem adotada) para a primeira posição, depois o próximo menor valor para a segunda posição, sucessivamente até n-1 elementos restantes.

Pseudo-código:

```
SelectionSort(t[1...n])
        para i -> 1 até n - 1 faça
              minIndex -> i
              para j -> i + 1 até n faça
                 se(t[minIdex] > t[j]) então
                    minIndex -> j
                 fim-se
              fim-para
              temp -> t[i]
              t[i] -> A[minIndex]
             t[minIndex] -> temp
       fim-para
 fim-SelectionSort
```

(c) Insertion sort.

A ideia do Insertion sort é inserir ordenadamente, ele constrói o array com um elemento por vez, executando até n-1 vezes.

Pseudo-código:

```
InsertionSort(t[0...n-1])
        para i->1 até n-1 faça
           valor-> t[i]
            j->i-1
           enquanto (j >=0 e t[j]>valor) // essa parte é responsável pelo
                                            deslocamento para a direita.
```

```
                t[j+1] -> t[j]
                j-> j -1
            fim-enquanto
            t[j+1] = valor //inserindo ordenadamente.
        fim-para
fim-InsertionSort
```

(d) Quicksort.

A ideia do QuickSort é dividir para conquistar o problema de ordenação em um array qualquer. Os passos para a execução são: escolher um pivô(um elemento da lista) para começar as comparações, ir reajustando a lista para que os elementos anteriores ao pivô sejam menores que ele e os posteriores sejam maiores. Quando o primeiro pivô termina as comparações, passa para o próximo pivô ordenado, geralmente é o último elemento da lista e recomeça as comparações, repetindo até que o array esteja ordenado.

Pseudo-código:

```
Partição(t[0...n - 1], inicio, fim)
    pivo -> t[inicio]
    i -> inicio + 1
    j -> fim
   enquanto(i ≤ j) faça
            enquanto(i ≤ j E t[i] ≤ pivo) faça
                i -> i + 1
            fim-enquanto
        enquanto(i ≤ j E t[j] > pivo) faça
            j-> j - 1
        fim-enquanto
        se(i < j) então
         trocar(t, i, j)
        fim-se
    fim-enquanto
    trocar(t, inicio, j) //coloca o pivô na posição de ordenação
    retorne j;
fim-Partição
```

```
QuickSort(t[0...n-1], inicio, fim)
        se (inicio < fim) então
           a = Partição(t, inicio, fim)
           QuickSort(t, inicio, a-1) // ordena a esquerda
           QuickSort(t, a+1, fim) // ordena a direita
         fim-se
fim-QuickSort
```

(e) Mergesort.

A ideia do MergeSort é semelhante a do QuickSort, ambos utilizam do método dividir e conquistar. Mas o Merge divide o array em duas metades, chamando a si mesmo para ambas e ao final mescla as duas metades para produzir um único array ordenado.

Há um gasto considerável de memória e de tempo de execução nesse algoritmo.

Pseudo-código:

```
Mescla(A[0...n - 1], inicio, meio, fim)
        tamanho_Esq -> meio - inicio + 1
        tamanho_Dir -> fim - meio
         vetor Esquerda[0..tamanho_Esq - 1]
         vetor Direita[0...tamanho_Dir - 1]
          para i -> 0 até tamanho_Esq - 1
                  Esquerda[i] -> A[inicio + i]
          fim-para
          para j -> 0 até tamanho_Dir - 1
                   Direita[j] -> A[meio + 1 + j]
          fim-para
          //São índices do vetor auxiliar
          id_Esq -> 0
          id_Dir -> 0

          para k -> inicio até fim
                 se(  id_Esq < tamanho_Esq)
```

```
                    se( id_Dir < tamanho_Dir)
                        se(Esq[id_Esq] < Dir[id_Dir])
                          A[k] -> Esquerda[  id_Esq]
                          id_Esq ->   id_Esq + 1
                        senão
                          A[k] -> Direita[id_Dir]
                          id_Dir ->  id_Dir + 1
                        fim-se
                    senão
                        A[k] -> Esquerda[ id_Esq]
                        id_Esq ->   id_Esq + 1
                    fim-se
            senão
               A[k] -> Direita[ id_Dir]
               id_Dir->  id_Dir + 1
            fim-se
        fim-para
fim-mescla

MergeSort(t[0...n - 1], inicio, fim)
        se(inicio < fim)
            meio ← (inicio + fim) / 2
           MergeSort(t, inicio, meio)
           MergeSort(t, meio + 1, fim)
           MergeSort(t, inicio, meio, fim)
        fim-se
fim-MergeSort
```