Simulado da 2ª Avaliação

Algoritmos e Estrutura de Dados I (AE22CP)

Prof. Jefferson T. Oliva

1. Implemente uma função recursiva que receba, pelo menos (ou seja, se você preferir, pode colocar mais parâmetros na função, desde que explique para quê servem), um caractere e uma string. A função deverá retornar a quantidade de vezes que o caractere aparece na string. Por exemplo, para o char 'a' e a string "abaacdba", a função deverá retornar 4.

```
int cotador_rec(char c, char str[], int i){
   if (i < strlen(str))
       return (c == str[i]) + cotador_rec(c, str, i + 1);

   return 0;
}

int contador_char(char c, char str[]){
   return contador_rec(c, str, 0);
}
```

2. Dado o seguinte arranjo de caracteres que foi submetido a um algoritmo ordenação: {O, R, D, E, N, A, D, O}. Em algum momento, o arranjo encontra-se na seguinte forma: {D, E, O, R, A, D, N, O}.

   Responda:

   a) - Cite **um** dos algoritmos de ordenação que não foi aplicado. Justifique a sua resposta com a aplicação de um teste de mesa demonstrando o funcionamento do algoritmo escolhido.

   Resolução:

   Um algoritmo que não foi aplicado é o selection sort, porque, em cada passagem do for externo, poderia haver apenas uma troca. Exemplo de aplicação:

   antes da ordenação: {O, R, D, E, N, A, D, O}
   Após a 1ª passagem: {A, R, D, E, N, O, D, O}
   Após a 2ª passagem: {A, D, R, E, N, O, D, O}
   Após a 3ª passagem: {A, D, D, E, N, O, R, O}
   Após a 4ª passagem: {A, D, D, E, N, O, R, O}
   Após a 5ª passagem: {A, D, D, E, N, O, R, O}
   Após a 6ª passagem: {A, D, D, E, N, O, R, O}
   Após a 7ª passagem: {A, D, D, E, N, O, O, R}

   Dessa forma, conclui-se que em nenhum momento da ordenação o arranjo encontra-se na seguinte ordem: {D, E, O, R, A, D, N, O}

   b) - Qual algoritmo de ordenação foi aplicado? Justifique a sua resposta.

   Resolução:

   O algoritmo de ordenação aplicado foi o mergesort, pois se dividirmos os arranjos "no meio", cada metade está ordenado, o que é uma característica do algoritmo: dividir o

 Continua na próxima página...

arranjo pela metade, e depois intercalar os sub-arranjos, os quais já foram divididos e intercalados. Primeiramente, o mergesort divide, sucessivamente, o arranjo em dois (caso indutivo) até que cada sub-arranjo tenha tamanho 1 (caso base). Posteriormente, cada par de sub-arranjo é combinado por meio de intercalação, no qual os elementos são posicionados de forma ordenada. Essa combinação é feita até que o vetor inteiro esteja ordenado.

c) - Em um teste de mesa, a partir do arranjo em seu estado inicial (ordem decrescente), aplique o algoritmo de ordenação, passo-a-passo, até chegar ao estado atual {D, E, O, R, A, D, N, O}

{O, R, D, E, N, A, D, O}
divisão: {O, R, D, E} e {N, A, D, O}
divisão: {O, R} e {D, E}
divisão: {O} e {R}
intercalação: {O, R}
divisão: {D} e {E}
intercalação: {D, E}
intercalação: {D, E, O, R}
divisão: {N, A} e {D, O}
divisão: {N} e {A}
intercalação: {A, N}
divisão: {D} e {O}
intercalação: {D, O}
intercalação: {A, D, N, O}

Dessa forma, após a intercalação acima, o vetor encontra-se na seguinte ordem: {D, E, O, R, A, D, N, O}

 Continua na próxima página...

3. Dado o algoritmo de inserção no final de listas encadeadas abaixo:

```
Cell* inserir_ultimo(int x, Cell *cel){
     if (cel == NULL)
         return cel = criar_celula(x);
     else if (cel->prox == NULL)
         return cel->prox = criar_celula(x);
     else
         return inserir_ultimo(x, cel->prox);
   }

void inserir_lista(int x, Lista *l){
     if (l == NULL)
         l = criar_lista();
     l->head = inserir_ultimo(x, l->head);
}
```

Implemente uma versão alternativa à função "*inserir_ultimo*" de modo que, em vez de utilizar recursão, utilize iteração explícita para a inserção de um novo elemento ao final da lista encadeada.

Resolução:

```
Cell* inserir_ultimo(int x, Cell *cel){
     if (cel == NULL)
         cel = criar_celula(x);
     else {
         Cell *aux = cel;
         while (aux->next != NULL) aux = aux->next;
         aux->next = criar_celula(x);
}
     return cel
   }

void inserir_lista(int x, Lista *l){
     if (l == NULL)
         l = criar_lista();
     l->head = inserir_ultimo(x, l->head);
}
```

 Continua na próxima página...

4. Implemente uma função que receba três filas estáticas (f1, f2, e f3), sendo uma com elementos (f1) e as outras duas, vazias. A função deverá remover os elementos de f1 e enfileirá-las nas outras duas filas, onde cada uma deve conter a metade dos elementos de f1. Observe que na estrutura da Fila, apresentada no anexo, não há campo para representar o tamanho. Por fim, os elementos da fila podem ser acessados apenas pela função desenfileirar. Exemplo:

   - Antes:
     - f1 = {1, 2, 3, 4, 5}
     - f2 = {}
     - f3 = {}
   - Depois:
     - f1 = {}
     - f2 = {1, 2}
     - f3 = {3, 4, 5}

   Resolução:

```
void dividir(Fila *f1, Fila *f2, Fila *f3){
   int tam = 0;
   if (!fila_vazia(f1) && fila_vazia(f2) && fila_vazia(f3)){
       while (!fila_vazia(f1)){
           enfileirar(f3, desenfileirar(f1));
           tam++;
       }
       tam = tam / 2;
       if (tam == 0)
       tam++;
       while (tam > 0){
           enfileirar(f2, desenfileirar(f3));
           tam--;
       }
   }
}
```

 Continua na próxima página...

5. Implemente uma função que receba duas listas encadeadas. A função deverá retornar uma lista encadeada resultante da intercalação das duas listas. Para isso, podem ser utilizadas as estruturas e as funções definidas no anexo. Caso necessário, implemente funções auxiliares para resolver este exercício. Em seguida, faça a análise de complexidade da sua função. Exemplo de antes e depois da aplicação da função:

   - Antes:
     - l1 = {1, 2, 3}
     - l2 = {4, 5, 6}
     - l3 = { }
   - Depois:
     - l1 = { }
     - l2 = { }
     - l3 = {1, 4, 2, 5, 3, 6}

```
void inserir_primeiro(int item, ListaE *l){
   Cell *nova = criar_celula(item);
   if (l == NULL)
       l = criar_lista();
   nova->next = l->head;
   l->head = nova;
}


ListaE *intercalar(ListaE *l1, ListaE *l2){
   ListaE *criar_lista();
   if (l1 == NULL){
       if (l2 != NULL){
           l3->head = l2->head;
           l2->head = NULL;
       }
   }else if (l2 == NULL){
       l3->head = l2->head;
       l2->head = NULL;
   }else if ((l1->head != NULL) && (l2->head != NULL)){
       Cell *aux1, aux2;
       Pilha *p = criar_pilha();
       while ((l1->head != NULL) && (l2->head != NULL)){
           aux1 = l1->head;
           aux2 = l2->head;
           empilhar(p, aux1->item);
           empilhar(p, aux2->item);
           l1->head = aux1->next;
           l2->head = aux2->next;
           free(aux1);
           free(aux2);
       }
```

 Continua na próxima página...

```
        while (l1->head != NULL){
            aux1 = l1->head;
            empilhar(p, aux1->item);
            l1->head = aux1->next;
            free(aux1);
        }         while (l2->head != NULL){
            aux1 = l2->head;
            empilhar(p, aux1->item);
            l2->head = aux1->next;
            free(aux1);
        }
        while (!pilha_vazia(p))
            inserir_primeiro(desempilhar(p), l3);
        free(p);
    }
    return l3;
}
```

Como o acesso aos elementos de listas encadeadas é sequencial e temos que percorrer as duas listas, no pior caso,então a complexidade é de ordem de O(n).

 Continua na próxima página...

6. Considere:

   - Os seguintes algoritmos de pesquisa: busca sequencial; busca sequencial indexada com tabela de índices de tamanho 5; e busca binária.
   - O seguinte arranjo ordenado com 25 elementos: {1, 4, 9, 11, 14, 19, 22, 23, 27, 34, 38, 42, 55, 56, 60, 62, 67, 74, 78, 84, 88, 89, 90, 91, 95}

   Testes de mesa para a busca da chave 14:

   Busca sequencial:

   Comparação 1: {**1**, 4, 9, 11, 14, 19, 22, 23, 27, 34, 38, 42, 55, 56, 60, 62, 67, 74, 78, 84, 88, 89, 90, 91, 95}

   Comparação 2: {1, **4**, 9, 11, 14, 19, 22, 23, 27, 34, 38, 42, 55, 56, 60, 62, 67, 74, 78, 84, 88, 89, 90, 91, 95}

   Comparação 3: {1, 4, **9**, 11, 14, 19, 22, 23, 27, 34, 38, 42, 55, 56, 60, 62, 67, 74, 78, 84, 88, 89, 90, 91, 95}

   Comparação 4: {1, 4, 9, **11**, 14, 19, 22, 23, 27, 34, 38, 42, 55, 56, 60, 62, 67, 74, 78, 84, 88, 89, 90, 91, 95}

   Comparação 5: {1, 4, 9, 11, **14**, 19, 22, 23, 27, 34, 38, 42, 55, 56, 60, 62, 67, 74, 78, 84, 88, 89, 90, 91, 95}

   Foram necessárias 5 comparações para achar a chave 14

   Busca sequencial indexada com tabela de índices de tamanho 5:

   Tabela de índice

Chave	Pos. Arq.
1	0
19	5
38	10
62	15
8	20

   busca na tabela de índices:
   - primeiramente, a chave 14 é comparada com a chave 1
   - em seguida, a chave 14 é comparada com a chave 19. Como a chave 19 é maior que 14, logo, todas as demais também são. Então, a busca é continuada no arquivo a partir de uma posição a frente do elemento 1.

 Continua na próxima página...

busca no arquivo: (como cada elemento da tabela de índice "cobre" 5 elementos do arquivo, então vamos considerar a busca apenas na parte "coberta" pelo primeiro elemento da tabela de índices, que deve conter elementos >= 1 e < 19)

1, **4**, 9, 11, 14 // como a chave 14 já comparado com 1 previamente, então continuaremos a comparação a partir da chave 4
1, 4, **9**, 11, 14
1, 4, 9, **11**, 14
1, 4, 9, 11, **14**

Foram necessárias 6 comparações (duas na tabela de índice mais 4 no arquivo) para achar a chave 14

Busca binária:

ini	fim	meio	v[meio]	v[meio] == 14	v[meio] >14	v[meio] <14	Comparações
0	24	12	55	F	V	—	2
0	11	5	19	F	V	—	2
0	4	2	9	F	F	V	3
3	4	3	11	F	F	V	3
4	4	4	14	V	—	—	1

Foram necessárias 11 comparações (duas na tabela de índice mais 4 no arquivo) para achar a chave 14

Responda:

a) - Qual dos algoritmos listados acima é o mais rápido para procurar a chave 14? Quantas comparações são necessárias para esse algoritmo encontrar a chave 14? Mostre como você contou a quantidade de comparações. (10 pontos)

Resolução: busca sequencial, a qual faz 5 comparações para encontrar a chave 14.

b) - Qual dos algoritmos listados acima é o mais lento para procurar a chave 14? Quantas comparações são necessárias para esse algoritmo encontrar a chave 14? Mostre como você contou a quantidade de comparações. (10 pontos)

Resolução: busca binária, a qual faz 11 comparações para encontrar a chave 14.

Anexo

Estruturas de dados e protótipos de função para resolução do Exercício 5:

```
#define TMAX 100
typedef struct Cell{
    int item;
    struct Cell *prox;
}Cell;
   typedef struct{
    Cell *cabeca;
}Lista;
   typedef struct{
    Cell *cabeca;
}Pilha;
   typedef struct{
    int item[TMAX];
    int ini, fim;
}Fila;
```

```
Cell* criar_celula(int chave);
Lista* criar_lista();
Pilha* criar_pilha();
Fila* criar_fila();
int pilha_vazia(Pilha *p);
int fila_vazia(Fila *f);
void empilhar(Pilha *p, int chave);
int desempilhar(Pilha *p);
void enfileirar(Fila *f, int chave);
int desenfileirar(Fila *f);
void liberar_lista(Lista *l);
void liberar_pilha(Pilha *p);
void liberar_fila(Fila *f);
```