Gabarito Lista 2 – Algoritmos e Estruturas de Dados
DEL-UFRJ

Pilhas, Filas e Listas

1.Basta criar uma estrutura de dados contendo um vetor com as informações de tamanho de todo o vetor, do tamanho n1 da pilha 1 e do tamanho n2 da pilha 2. A pilha 1 teria então seus valores armazenados nas posições iniciais do vetor e a pilha 2 nas suas posições finais. Com isso, o vetor ficaria cheio quando n1+n2=n.

2. (C++)

// Deque.h

```
const int maxDeque = 100;
class Deque {
public:
        Deque();
        void append( int x ); //insere no fim
        void insert( int x ); //insere no inicio
        void remove_head( int *x );
        void remove_tail( int *x );
private:
        int head, tail;
        int size;
        int entry[maxDeque+1];
};
```

//Deque.cpp

```
#include "Deque.h"
Deque::Deque() {
        size = 0;
        head = 1;
        tail = 0;
}
void Deque::append(int x) {
        if (size != maxDeque) {
                size++;
                tail = (tail % maxDeque) + 1;
                entry[tail] = x;
        }
}
void Deque::insert(int x) {
        if (size != maxDeque) {
                size++;
                head = (head == 1) ? maxDeque : head - 1;
                entry[head] = x;
        }
}
void Deque::remove_head(int *x) {
        if ( size != 0 ) {
                size--;
                *x = entry[head];
                head = (head % maxDeque) + 1;
        }
}
void Deque::remove_tail(int *x) {
        if ( size != 0 ) {
```

```
                size--;
                *x = entry[tail];
                tail = (tail == 1) ? maxDeque : tail - 1;
        }
}
```

3. Para implementar uma fila com duas pilhas:

Manter os elementos sendo adicionados na pilha1, e quando for retirar algum elemento da fila, passar todos da pilha1 para a pilha2 e dar um "pop" na pilha2.
Exemplo:
Acrescentar o valor 1 na fila e retirar o próximo a sair da fila.

Situação inicial:
Pilha1: 3 2
Pilha2: <vazia>

Acrescentado 1:
Pilha1: 3 2 1
Pilha2: <vazia>

Para retirar o próximo elemento a sair da fila deve-se transferir tudo da pilha1 para a pilha2:
Pilha1: <vazia>
Pilha2: 1 2 3

O valor a ser retirado da fila então é o 3, a seguir pode-se transferir de volta tudo para a pilha1:
Pilha1: 1 2
Pilha2: <vazia>

Temos então que acrescentar um elemento na fila fica como O(1) e retirar passa a ser O(n).

4. Para implementar uma pilha com duas filas:

A cada "push" para a pilha, transferir todos os elementos da fila1 para a fila2. Em seguida, adicionar o elemento para a fila1. Então deve-se passar todos os elementos da fila2 de volta para a fila1. O "pop" da pilha seria então retirar um elemento da fila1, que seria justamente o último a ser adicionado, caracterizando uma pilha.
Exemplo:
Push no valor 3, em uma pilha com os valores 1 e 2

Situação inicial
Fila1: 1 2
Fila2: <vazia>

Transferir tudo da Fila1 para a Fila2:

Fila1: <vazia>
Fila2: 1 2

Acrescentar elemento 3 na Fila1:
Fila1: 3
Fila2: 1 2

Passar de volta os elementos da Fila2 para a Fila1:
Fila1: 1 2 3
Fila2: <vazia>

Se for dado um pop em seguida, o elemento retirado será o 3, caracterizando uma pilha(last in – first out).

Como há n deslocamentos, o tempo de execução do push é O(n).
O tempo do pop é O(1).

5. Ver Aula09_Pilhas.pdf, slides 46 a 71

6. Ver Aula10_Filas, slides 72 a 100, com append no lugar de enqueue e serve no lugar de dequeue.

7.

```
Node* reverse_list( Node **List )
{

        Node * current = *List; // nó atual
        Node * next = NULL;
        Node * prev = NULL;

        while ( current )
        {
                next = current->pNext; // armazena prox nó
                current->pNext = prev; // passa a apontar ponteiro do prox p/ o anterior
                prev = current;
                current = next;
        }
        *List = prev; // ultimo no passa a ser o primeiro da nova lista
        return *List;
}
```

8. Cada elemento armazena o XOR do endereço do elemento anterior com o endereço do próximo. Para percorrer a lista do fim ao início, basta fazer o XOR do próximo endereço da lista (no fim da lista, o próximo é NULL, 0) com o atual, o que dá como resultado o endereço do anterior. Se essa operação der como resultado o próximo endereço, significa que foi feito um XOR com 0, então esse endereço resultante é o início. Para inserir, basta fazer um XOR do endereço do elemento a ser inserido com o endereço do último elemento, e armazenar o resultado no último elemento. O novo elemento recebe o endereço anteriormente armazenado no último elemento. Para remover um elemento, deve-se percorrer a lista até ele, recuperar os 2 endereços anteriores, e próximos, e fazer os ajustes necessários com operações XOR. Para

recuperar um elemento, basta percorrer a lista conforme o método já descrito. Para inverter a ordem, basta trocar o último com o primeiro elemento da lista.

9.a) Tamanho ALUNO = 41+11+4=56 bytes
Tamanho LISTA_ALUNOS= 200*56 + 4= 11204 bytes

b)

```
int numAlunosCrAlto(LISTA_ALUNOS *pLista) {
        int i;
        int count=0;
        if ( !( pLista ) )
                return 0;
        for( i=0; i < pLista->numAlunos; i++ ) {
                if ( pLista->aluno[i].cr >= 7 )
                        count++;
        }
        return count;
}
```

c)

```
int excluirAluno(LISTA_ALUNOS *pLista, char *matricula) {
        int i, found;
        if ( !( pLista ) )
                return 1;
        for( i = 0, found = 0; i < pLista->numAlunos; i++ ) {
                if ( found == 1 )
                        pLista->aluno[i-1] = pLista->aluno[i];
                else {
                        if ( strcmp(pLista->aluno[i].matricula, matricula) == 0 )
                        found = 1;
                }
        }
        if ( found == 1 ) {
                pLista->numAlunos--;
                return 0;
        }
return 1;
}
```

10. a)

```
int pesquisar(LISTA_ITENS *pLista, int cod) {
	int i;
	int count=0;
	if ( !( pLista ) )
		return 0;
	for( i=0; i < pLista->numItens; i++ ) {
		if ( pLista->item[i].codItem == cod )
			return pLista->item[i].qtdEstoque;
	}
	return 0;
}
```

b)

```
int pesquisa_binaria(LISTA_ITENS *pLista, int cod) {
	int ini,meio,fim;
	if ( !( pLista ) )
		return 0;
	for ( ini=0, fim = pLista->numItens - 1, meio = fim/2; ( (ini < fim) && (pLista->item[meio].codItem != cod) ) ;
meio = (ini+fim)/2 ) {
		if ( pLista->item[meio].codItem < cod ) //procurar na metade direita
			ini = meio + 1;
		else if ( pLista->item[meio].codItem < cod ) //procurar na metade esquerda
			fim = meio - 1;
	}
	if ( pLista->item[meio].codItem == cod )
		return pLista->item[meio].qtdEstoque;
return 0;
}
```

11.a)

```
int empilhar(PILHA *pPilha, char c) {
	if ( !( pPilha ) )
		return 1;
	if ( pPilha->topo > MAX - 1) //pilha cheia
		return 2;
	pPilha->c[++pPilha->topo] = c;
	return 0;
}
int desempilhar(PILHA *pPilha, char *c) {
	if ( !( pPilha ) )
		return 1;
	if ( pPilha->topo < 0) //pilha vazia
		return 2;
	*c = pPilha->c[pPilha->topo++];
	return 0;
}
```

b)

```
int palindroma(char *palavra) {
	int tamanho, meio, i;
	char c;
	PILHA pilha;
	pilha.topo = -1;
	tamanho = strlen(palavra);
	meio = tamanho / 2;
	if (meio > MAX) // estouro da pilha
		return 2;
	for (i=0; i <= meio - 1; i++)
		empilhar( &pilha, palavra[i] );

	meio = ( ( tamanho % 2 ) ? meio + 2 : meio + 1 );
```

```
        for (i=meio; i <= tamanho - 1; i++) {
                desempilhar( &pilha, &c );
                if ( c != palavra[i] ) // nao palindromo
                        return 1;
        }
return 0;
}
```

12.

a)

```
typedef struct
{
   CHAMADO chamado[500];
   int inicio;
   int numChamados;
} FILA_CHAMADOS;
```

b)

```
int cadastrarChamado(FILA_CHAMADOS *pFila, CHAMADO *pChamado)
{
   int fim;

   // Verificar se a fila está cheia
   if (pFila->numChamados == 500) return 1;

   // Determinar a posição onde o chamado será enfileirado
   fim = pFila->inicio + pFila->numChamados;
   if (fim >= 500) fim -= 500;

   // Gravar o novo chamado no fim da fila
   pFila->chamado[fim] = *pChamado;

   // Atualizar o número de chamados da fila
   pFila->numChamados++;

   // retornar indicando operação bem sucedida
   return 0;
}
```

c)

```
int atenderChamado(FILA_CHAMADOS *pFila, CHAMADO *pChamado)
{
   int ultimo;

   // Verificar se a fila está vazia
   if (pFila->numChamados == 0) return 2;

   // Desenfileirar o primeiro chamado da fila
   *pChamado = pFila->chamado[inicio];

   // Ajustar o início da fila
   pFila->inicio++;
   if (pFila->inicio == 500) pFila->inicio = 0;

   // Atualizar o número de chamados da fila
   pFila->numChamados--;

   // retornar indicando operação bem sucedida
   return 0;
}
```

13.

a)

```
#include <string.h>
#include <stdlib.h>
#include "qsort.h"

void swap(CLIENTE* a, CLIENTE* b) {
  CLIENTE* tmp;
  tmp = (CLIENTE*) malloc(sizeof(CLIENTE));
  memcpy (tmp,a,sizeof(*a));
  memcpy(a,b,sizeof(*b));
  memcpy(b,tmp,sizeof(*tmp));
  free(tmp);
}

int partitionNome(CLIENTE* vec, int left, int right) {
  int i, j;

  i = left;
  for (j = left + 1; j <= right; ++j) {
              if ( strcmp(vec[j].nome,vec[left].nome) < 0 ) {
     ++i;
              swap(&vec[i], &vec[j]);
    }
  }
  swap(&vec[left], &vec[i]);

  return i;
}

void quickSortNome(CLIENTE* vec, int left, int right) {
  int r;

  if ( right > left ) {
    r = partitionNome(vec, left, right);
    quickSortNome(vec, left, r - 1);
    quickSortNome(vec, r + 1, right);
  }
}

void ordenarNome( LISTA_CLIENTES *pLista ) {
            quickSortNome(pLista->cliente, 0 , pLista->numClientes - 1 );
}
```

b)

```
int partitionIdadeRenda(CLIENTE* vec, int left, int right) {
  int i, j;

  i = left;
  for (j = left + 1; j <= right; ++j) {
              if ( ( ( vec[j].idade < vec[left].idade ) ) || ( (vec[j].idade == vec[left].idade) && (vec[j].rendaMensal >
vec[left].rendaMensal) ) ){

                          ++i;
                          swap(&vec[i], &vec[j]);

    }
  }
  swap(&vec[left], &vec[i]);

  return i;
```

```
}

void quickSortIdadeRenda(CLIENTE* vec, int left, int right) {
  int r;

  if ( right > left ) {
    r = partitionIdadeRenda(vec, left, right);
    quickSortIdadeRenda(vec, left, r - 1);
    quickSortIdadeRenda(vec, r + 1, right);
  }
}

void ordenarIdadeRenda( LISTA_CLIENTES *pLista ) {
          quickSortIdadeRenda(pLista->cliente, 0 , pLista->numClientes - 1 );
}
```

14.

a) Similar à solução 9.b

b)

```
CLIENTE *clienteMaisVelho(LISTA_ENCADEADA *pLista) {

int max=0;
CLIENTE *atual, *maisVelho;

if ( pLista->pPrimeiro == NULL)
   return NULL;

atual = pLista->Primeiro;
maisVelho = NULL;

while (atual != NULL) {
if (atual->cliente.idade > max ) {
   max =  atual->cliente.idade;
   maisVelho = atual;
}
atual = atual->pProximo;
}
return ( maisVelho );
```

15. Similar às questões 2 e 18.

16.

a)

```
typedef struct
{
  char caracter [200];
  int  topo 0;
}

void empilhar(Pilha *p, char c)
{
  if (topo == 200) return;
  p->caracter[p->topo] = c;
  p->topo++;
}

void desempilhar(Pilha *p, char *c)
{
  if (topo == 0) return;
  p->topo--;
  *c = p->caracter[p->topo];
}
```

b)

```
int sintaxeCorreta(char *s)
{
  Pilha p;
  char c;

  p->topo=0;
  for (i=0; i < strlen(s); i++)
  {
    if (s[i]=='(') empilhar(p,')');
    if (s[i]=='[') empilhar(p,']');
```

```
    if (s[i]==')') || s[i]==']')
    {
       if (p->topo==0) return 0; // erro
       desempilhar(p,&c);
       if (c!=s[i]) return 0;    // erro
    }
  }
  if (p->topo!=0) return 0;      // erro
}
```

17. Similar às questões 9 e 14.

18.

a)

```
#define MAX 50
#define FILA_CHEIA 1
#define FILA_VAZIA 2

typedef struct
{
        FILE* arq[MAX+1];
        int head, tail, count;
} FILA;


int enfileirar ( FILA *fila, FILE *arq) {
        if ( fila->count == MAX )
                return FILA_CHEIA;
        fila->count++;
        fila->tail = ( fila->tail % MAX ) + 1;
        fila->arq[fila->tail] = arq;
        return ( 0 );
}
int desenfileirar ( FILA *fila, FILE **arq ) {
        if ( fila->count == 0 )
                return FILA_VAZIA;

        fila->count--;
        *arq = fila->arq[fila->head];
        fila->head = ( fila->head % MAX ) +1;
        return ( 0 );
}
```

b)

```
int inserirSpooler ( SPOOLER *spooler, FILE *arq, char prioridade ) {
        int retorno;

        if (!(spooler))
                return SPOOLER_NULL;

        switch (prioridade) {
                case 'A':
                case 'a':
                        retorno = enfileirar(&spooler->FilaAlta, arq);
                        break;
                case 'M':
                case 'm':
                        retorno = enfileirar(&spooler->FilaMedia, arq);
                        break;
                case 'B':
                case 'b':
                        retorno = enfileirar(&spooler->FilaBaixa, arq);
                        break;
                default:
                        return PRIORIDADE_INVALIDA;
        }
        return retorno;
```

```
}
int retirarSpooler ( SPOOLER *spooler, FILE **arq ) {
	int retorno;

	if (!(spooler))
		return SPOOLER_NULL;
	if ( retorno =	desenfileirar( &spooler->FilaAlta, arq ) == FILA_VAZIA )
		if ( retorno =	desenfileirar( &spooler->FilaMedia, arq ) == FILA_VAZIA )
			if ( retorno =	desenfileirar( &spooler->FilaBaixa, arq ) == FILA_VAZIA )
				return SPOOLER_VAZIO;
	return 0;
}
```

19.a)

```
void trocarPosicao( ELEMENTO *pElemento ) {
	ELEMENTO *tmp, *sucessor;
	if (!(pElemento))
		exit (1);
	sucessor = pElemento->pProximoCliente;
	tmp = malloc( sizeof (*tmp) );

	memcpy( &tmp->dados, &sucessor->dados, sizeof(sucessor->dados) );
	memcpy( &sucessor->dados, &pElemento->dados , sizeof(sucessor->dados) );
	memcpy( &pElemento->dados, &tmp->dados , sizeof(sucessor->dados) );

	free( tmp );
}
```

b)

```
void ordenarBolha( ELEMENTO *pLista ) {
int i,trocou;
ELEMENTO *ele, *proximo;


if (!(pLista))
	exit(1);

do
	{
		ele = pLista->pProximoCliente;
		trocou = 0;
		for(i = 0; ele->pProximoCliente != NULL ; i++) {
			proximo = (ELEMENTO*)ele->pProximoCliente;
			if(ele->dados.idade > proximo->dados.idade)
			{
				trocarPosicao(ele);
				trocou = 1;
			}
			ele = ele->pProximoCliente;
		}
	}while(trocou);

}
```

20.a)

```
#include "cliente.h"
int numElementos(LISTA_CLIENTES *pLista) {
	int count = 0;
	CLIENTE *temp;
	temp = pLista->pPrimeiroCliente;

	while( temp ) {
		count++;
		temp = temp->pProximoCliente;
	}
```

```
return count;
}
```

b)

```
DADOS_CLIENTE *dadosClienteMaisVelho(LISTA_CLIENTES *pLista) {
	int max = 0;
	CLIENTE *temp, *temp2;
	temp2 = pLista->pPrimeiroCliente;

	while( temp2 ) {
		if ( temp2->dados.idade > max ) {
			temp = temp2;
			max = temp2->dados.idade;
		}
		temp2 = temp2->pProximoCliente;
	}
	return &(temp->dados);
}
```

c)

```
void excluirIdosos(LISTA_CLIENTES *pLista) {
	CLIENTE *curr,*prev;

	curr = pLista->pPrimeiroCliente;
	prev = NULL;

	while( curr ) {
		if ( curr->dados.idade < 65 ) {
			prev = curr;
			curr = curr->pProximoCliente;
		}
		else { //exclui idosos
			if ( prev )
				prev->pProximoCliente = curr->pProximoCliente;
			free( curr );
		}
	}
}
```

Árvores

1. Basta empilhar os nós filhos à esquerda até que o extremo esquerdo seja atingido. A partir daí, desempilha-se e segue-se o mesmo procedimento com a subárvore direita do nó recém-desempilhado.

2.

PREORDEM(x)
se $x \neq$ NIL
	então print key[x]
		PREORDEM*(esquerda*[x]*)*
		PREORDEM*(direita*[x]*)*
POSORDEM(x)
se $x \neq$ NIL
	então PREORDEM*(esquerda*[x]*)*
		PREORDEM*(direita[x])*
		print key[x]

3. Em uma ABB cada nó tem todos os elementos da sua subárvore direita de valor maior que o do próprio nó. Dessa forma, o sucessor deve estar na subárvore direita. Além disso, em cada nó, o filho à esquerda tem um valor menor do que o nó pai. Logo, o sucessor imediato de um nó com 2 filhos deve estar na subárvore direita, e não ter nenhum filho à sua esquerda(pois se o tivesse, ele deixaria de ser o sucessor imediato). Para o antecessor imediato a demonstração é análoga.

4. O procedimento funciona como o percurso em ordem. Dessa forma, cada nó é acessado no máximo O(1) vezes. Com o acesso aos n nós, o limite assintótico é de O(n).

5. O algoritmo Tree-insert tem como pior caso O(n) (árvore desbalanceada) e caso médio O(logn). Como ele é chamado para n elementos, os limites são $O(n^2)$ em pior caso e O(n logn) no caso médio.

6.

Solution to Problem 12-2

To sort the strings of S, we first insert them into a radix tree, and then use a preorder tree walk to extract them in lexicographically sorted order. The tree walk outputs strings only for nodes that indicate the existence of a string (i.e., those that are lightly shaded in Figure 12.5 of the text).

Correctness: The preorder ordering is the correct order because:

- Any node's string is a prefix of all its descendants' strings and hence belongs before them in the sorted order (rule 2).
- A node's left descendants belong before its right descendants because the corresponding strings are identical up to that parent node, and in the next position the left subtree's strings have 0 whereas the right subtree's strings have 1 (rule 1).

Time: $\Theta(n)$.

- Insertion takes $\Theta(n)$ time, since the insertion of each string takes time proportional to its length (traversing a path through the tree whose length is the length of the string), and the sum of all the string lengths is n.
- The preorder tree walk takes $O(n)$ time. It is just like INORDER-TREE-WALK (it prints the current node and calls itself recursively on the left and right subtrees), so it takes time proportional to the number of nodes in the tree. The number of nodes is at most 1 plus the sum (n) of the lengths of the binary strings in the tree, because a length-i string corresponds to a path through the root and i other nodes, but a single node may be shared among many string paths.

7.a)

```
Pré-Ordem: 26 - 18 - 12 - 21 - 39 - 31 - 58 - 52 - 65
Pós-Ordem: 12 - 21 - 18 - 31 - 52 - 65 - 58 - 39 - 26
```

b)

```
Nó hE hD Balanco
12  0  0   0
21  0  0   0
18  1  1   0
31  0  0   0
```

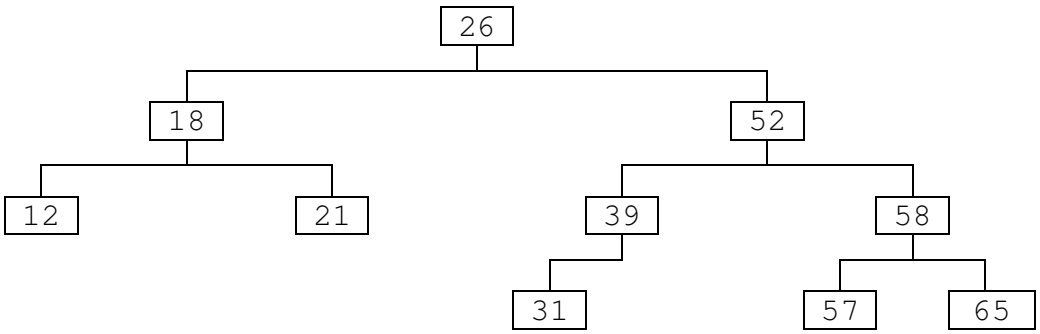
```
52  0  0   0
65  0  0   0
58  1  1   0
39  1  2  -1
26  2  3  -1
```

c)

```
Igual à árvore original, com o novo nó à direita do 52.

Nó hE hD Balanco
39  1  3  -2 (desregulado)
26  2  4  -2 (desregulado)
```

d)



Rotação Dupla à Esquerda

8.a)

```
char * pesquisarTelefone(NO * pNoRaiz, char *nomeAssinante) {
        int comp;
        while ( (pNoRaiz != NULL) && ( comp = (strcmp( pNoRaiz->a.nome, nomeAssinante )) != 0 ) )
                pNoRaiz = ( comp > 0 ) ? pNoRaiz->pEsq : pNoRaiz->pDir;
        if (pNoRaiz == NULL)
                return NULL;
        return pNoRaiz->a.telefone;
}
```

b)

```
void exibirLista(NO *pNoRaiz) {
        if (pNoRaiz == NULL)
                return;
        exibirLista(pNoRaiz->pEsq);
        printf("%s\n%s\n", pNoRaiz->a.nome, pNoRaiz->a.telefone);
        exibirLista(pNoRaiz->pDir);
}
```

c)

```
int determinarAltura(NO *pNoRaiz) {
        int aDir, aEsq;
        if (pNoRaiz == NULL)
                return 0;
        aDir = determinarAltura(pNoRaiz->pDir);
        aEsq = determinarAltura(pNoRaiz->pDir);
        if (aDir < aEsq)
                return (aDir + 1);
        return (aEsq + 1);
}
```

9.a) Similar ao item 8.a).

b) Similar ao item 8.b).

c)

```
void rotacaoDuplaDireita(ARVORE *pArvore) {
```

```
        ARVORE *aux;

        aux = pArvore->pSubArvDir;
        pArvore->pSubArvDir = aux->pSubArvEsq;
        aux->pSubArvEsq = pArvore;
}
```

10.

```
float salarioFuncionario(Arvore *pArvore, int matricula)
{
  if (!pArvore) return 0;
  if (matricula < pArvore->funcionario.matricula) return pesquisarSalario(pArvore->pEsq,matricula);
  if (matricula > pArvore->funcionario.matricula) return pesquisarSalario(pArvore->pDir,matricula);
  return pArvore->funcionário.salario;
}
```