CES-11

Algoritmos e Estruturas de Dados

Listas encadeadas lineares

RESPOSTAS
DOS
EXERCÍCIOS

Carlos Forster
Carlos Alonso
Juliana Bezerra

Exercícios

- Dadas duas listas lineares, deseja-se implementar a operação de *concatenação*, isto é, acrescentar a segunda lista no final da primeira. Qual seria a melhor implementação para realizar essa operação?
- Escreva o código das principais operações (criação, impressão, inserção, eliminação e esvaziamento) em uma:
  - lista encadeada sem nó líder;
  - lista duplamente encadeada;
  - lista encadeada circular (sem nó líder).

Exercícios

Dica: as funções recursivas devem ser aplicadas à lista simples sem cabeça

- Definir funções recursivas para:
  - Retornar uma lista dado o primeiro elemento e uma lista com os demais elementos;
  - Criar uma cópia de uma lista ligada;
  - Buscar um elemento de uma lista ligada (retorna o apontador para o nó com o elemento ou NULL);
  - Imprimir a lista de trás para frente;
  - Inserir elemento no fim da lista;
  - Obter a lista em ordem inversa;
  - Dividir a lista em uma com elementos menores que o valor dado e outra com os maiores ou iguais;
  - Combinar duas listas ordenadas numa só (ordenada);
  - Ordenar inserindo um elemento de cada vez.

Exercícios

- Definir funções iterativas para:
  - Inverter a lista (com inversão de apontadores)
  - Ordenar a lista selecionando o maior elemento e removendo da lista, até esgotar a lista.

Lista sem cabeca para implementar mais facilmente as funções recursivas

```
typedef struct node node;
struct node
{
    char elem;
    node *prox;
};

typedef node *lista;
typedef node *posicao;
```

abreviação para o uso de malloc e teste do apontador obtido

```
node *new_node()
{
    node *n;
    n = (node *) malloc(sizeof(node));
    if (n==NULL) erro("Acabou o espaco em memoria");
    return n;
}
```

Funções de Manipulação das listas

- Identifique quais dessas funções:
  - 1- são recursivas ou iterativas
  - 2- são destrutivas (modificam o parâmetro passado) ou construtivas (constroem uma nova lista deixando o parâmetro intacto)
  - 3- são O(1), O(N), etc

/* construtor de lista, dado primeiro elemento e resto da lista */

```
lista cons(char info, lista resto)
{
    lista x;
    x=new_node();
    x->elem=info;
    x->prox=resto;
    return x;
}
```

/* copia lista, criando novos nos */

```
lista copia_lista(lista x)
{
    lista y;
    if(x==NULL) return x;
    y=new_node();
    y->elem=x->elem;
    y->prox=copia_lista(x->prox);
    return y;
}
```

/* versao simplificada utilizando a função cons() */

```
lista copia_lista2(lista x)
{
    lista y;
    if(x==NULL) return x;
    y=cons(x->elem,copia_lista(x->prox));
    return y;
}
```

/* liberar memória da lista toda */

```
void free_lista(lista x)
{
    if(x!=NULL)
    {
        free_lista(x->prox);
        free(x);
    }
}
```

/* busca elemento na lista e retorna apontador ou NULL se nao encontrou */

```
posicao busca_lista(lista x, char elt)
{
    if(x==NULL) return x;
    if(x->elem==elt) return x;
    else return busca_lista(x->prox,elt);
}
```

/* imprimir lista */

```
void imprime_lista(lista x)
{
    if(x==NULL) return;
    printf("%c",x->elem);
    imprime_lista(x->prox);
}
```

/* imprimir lista ao contrário */

```
void imprime_rev_lista(lista x)
{
    if(x==NULL) return;
    imprime_rev_lista(x->prox);
    printf("%c",x->elem);
}
```

/* imprime com parenteses e virgula */

```
/***** utiliza função auxiliar **/

void _imprime_lista_virgulas_aux(lista x)
{
   if(x==NULL) {printf(")");return;}
   printf(",%c",x->elem);
   _imprime_lista_virgulas_aux(x->prox);
}

void imprime_lista_virgulas(lista x)
{
   if(x==NULL) printf("(VAZIA)");
   else
   {
      printf("(%c",x->elem);
     _imprime_lista_virgulas_aux(x->prox);
   }
}
```

/* inversao recursiva da lista */

```
/* a função auxiliar _lista_inv recebe a lista a ser invertida
 * e mais o pedaço já invertido que deve ficar no fim da lista
 * @param p o endereço do apontador da parte da lista a inverter
 * @param final o endereço do apontador da porção da lista já invertida
 */
void _lista_inv(lista *final, lista *p)
{
   lista r;
   if((*p)==NULL) return; /* final já contem todos os nós*/
   r=(*p)->prox;
   (*p)->prox=(*final);
   (*final)=(*p);
   (*p)=r;
   _lista_inv(final, p);
}

void inverte_lista(lista *q)
{
   lista s;
   s=NULL;
   _lista_inv(&s,q);
   (*q)=s;
}
```

/* inversao iterativa */

```
void inverte_lista2(lista *p)
{
   /* (*p) contem a lista a ser invertida
      q a porção já invertida
   */
   lista q,r;
   q=NULL;
   while((*p)!=NULL) /* enquanto há nós para inverter*/
   {
      r=(*p)->prox;
      (*p)->prox=q; /*inversão de apontador*/
      q=(*p);
      (*p)=r;
   }
   (*p)=q;
}
```

/* inserir no final */

```
void insere_final(lista *p, char elt)
{
    if((*p)==NULL) (*p)=cons(elt,NULL);
    else insere_final(&((*p)->prox),elt);
}
```

/* concatenar - parecido com inserir no final */

```
void concatenar_listas(lista *p, lista q)
{
    if((*p)==NULL) (*p)=q;
    else concatenar_listas(&((*p)->prox),q);
}
```

/* concatenar iterativo */

```
void concatenar_listas2(lista *p, lista q)
{
    while((*p)!=NULL) p=&((*p)->prox);
    (*p)=q;
}
```

/* circularizar - cria lista circular, inserindo lista em seu próprio final */

```
void circulariza_lista(lista *p)
{
    concatenar_listas(p,*p);
}
```

/* construir lista a partir de cadeia terminada em zero */

```
lista lista_de_cadeia(char *s)
{
    if(s[0]=='\0') return NULL;
    else return
cons(s[0],lista_de_cadeia(&(s[1])));
}
```

/* obter tamanho da lista */

```
int comprimento_lista(lista p)
{
    if(p==NULL) return 0;
    else return 1+comprimento_lista(p->prox);
}
```

/* comparar duas listas elemento a elemento e retorna 1 se todos iguais*/

```
int compara_listas(lista p, lista q)
{
    if (p==NULL)
        if (q==NULL) return 1;
        else return 0;
    else
        if(q==NULL) return 0;
        else if(p->elem==q->elem) return
            compara_listas(p->prox, q->prox);
        else return 0;
}
```

/* particionar lista em elementos menores e maiores ou iguais que um dado valor */

```
void particiona(lista *p, lista *menores, lista *maiores, char val)
{
   lista r;
   if((*p)==NULL) return;
   r=(*p)->prox;
   if( (*p)->elem >= val )
   {
      (*p)->prox=(*maiores);
      (*maiores)=(*p);
   }
   else
   {
      (*p)->prox=(*menores);
      (*menores)=(*p);
   }
   (*p)=r;
   particiona(p,menores,maiores,val);
}
```

/* combinar duas listas ordenadas p e q numa unica m, ordenada*/

```
void _passa_elemento(lista *from, lista *to) /*move elemento de from para to */
{
   lista r;
   r=(*from)->prox;
   (*from)->prox=(*to);
   (*to)=(*from);
   (*from)=r;
}

void merge(lista *p, lista *q, lista *m)
{
   if ((*p)==NULL)
      if ((*q)==NULL) return;
      else _passa_elemento(q,m);
   else
      if((*q)==NULL) _passa_elemento(p,m);
      else if ((*q)->prox < (*p)->prox) _passa_elemento(q,m);
      else _passa_elemento(p,m);
   merge(p,q,m); /* a lista combinada vai ficar reversa :'-( */
}
```

/* inserir elemento ordenando */

```
void insere_ordenado(lista *p, char elt)
{
   if ((*p)==NULL) (*p)=cons(elt,NULL);
   else if((*p)->elem > elt) (*p)=cons(elt,(*p));
   else insere_ordenado (&((*p)->prox),elt);
}
```

/* copia ordenada usando insere_ordenado */

```
lista copia_ordenado(lista p)
{
   lista q=NULL;
   while(p!=NULL)
   {
      insere_ordenado(&q, p->elem);
      p=p->prox;
   }
   return q;
}
```