ACH2023 - Algoritmos e Estruturas de Dados I

Exercícios sobre listas, pilhas e filas

Exercícios conceituais

1) Durante o semestre estudamos dois tipos principais de listas: listas sequenciais (usam um vetor - ou *array* - como espaço de armazenamento dos seus elementos) e listas ligadas (cada elemento é armazenado em uma estrutura denominada nó, e a lista é representada pelo encadeamento de diversos nós). Comparando estas duas abordagens principais de implementação de listas, o que considerações podemos fazer em relação a(o):

    a) consumo de memória em cada uma das abordagens.
    b) acesso a um elemento dado um índice.
    c) eficiência da operação de inserção (discuta também se faz diferença a inserção ser baseada em posição, ou baseada em ordem de valor dos elementos).
    d) eficiência da operação de remoção.

2) São denominadas listas ordenadas, aquelas listas em que a inserção ocorre de tal modo que os elementos sejam mantidos em ordem de valor. A organização ordenada dos elementos pode acelerar a operação de busca de um elemento na lista? Em caso positivo explique como ocorre essa melhoria, e se ela se aplica tanto a listas sequenciais quanto listas ligadas.

3) Explique como funciona a busca apoiada por um elemento sentinela em listas não ordenadas (tanto para listas sequenciais, como para listas ligadas). Discuta qual a vantagem do uso da sentinela na busca.

4) A partir de uma implementação básica de lista ligada (com encadeamento simples e ponteiro apenas para o primeiro nó da lista), explique quais as eventuais vantagens que as seguintes modificações podem oferecer:

    a) uso de um nó cabeça (sem conteúdo útil).
    b) último nó apontando para o primeiro, tornando a lista circular.
    c) uso de um ponteiro adicional que aponta para o último nó.
    d) nós apontando para o elemento anterior, além do próximo, tornando a lista duplamente ligada.

5) A partir das implementações de referência estudadas de listas sequenciais e listas ligadas indique qual delas você escolheria para usar como base da implementação (sem realizar qualquer modificação, ou realizando o mínimo de modificações) de uma:

a) fila.
b) pilha.

6) Suponha uma implementação melhorada de uma lista ligada, que é duplamente ligada e circular, e para a qual se mantém apenas o ponteiro para o primeiro nó:

    a) tal implementação seria adequada para implementar uma pilha? Haveria alguma vantagem em relação ao uso de uma lista ligada simples como base da implementação de uma pilha? Justifique.

    b) tal implementação seria adequada para implementar uma fila? Haveria alguma vantagem em relação ao uso de uma lista ligada simples como base da implementação de uma fila? Justifique.

Exercícios práticos (com algumas perguntas conceituais sobre eles)

Tomando como base as implementações de listas disponibilizadas, compile e teste as implementações disponíveis nos arquivos **lista_sequencial_dinamica.c** e **lista_ligada.c**. Em seguinte, implemente as seguintes funcionalidades:

1) Para as duas versões de lista, implemente a função **`valor_indice`** que recebe um ponteiro para uma lista, um índice e devolva o valor armazenado na lista naquele índice (lembrando que convencionamos que o primeiro elemento está no índice zero, o segundo no índice 1, o terceiro no 2, e assim sucessivamente).

2) Crie um programa de teste, que crie uma lista, insira alguns elementos, e depois calcule a soma de todos os valores armazenados na lista da seguinte forma:

```
int i;
int soma = 0;
int n = tamanho(lista);

for(i = 0; i < n; i++) soma += valor_indice(lista, i);

printf("Soma: %d\n", soma);
```

   Execute o programa de teste, variando a quantidade de elementos inseridos na lista, e também as versões de lista. O que é possível observar? Como explicar o fenômeno observado?

3) Considerando apenas a implementação da lista ligada (e considerando apenas o cenário em que as inserções são baseadas em posição):

    a) Crie a função **junta_listas_1** que recebe o ponteiro para duas listas, junta o conteúdo das duas listas em uma única lista, e devolve o endereço da lista resultante. Implemente a função de modo que o conteúdo das listas passadas como parâmetro da função não sejam alterados.

b) Crie uma função **junta_listas_2** que recebe o ponteiro para duas listas, junta o conteúdo das duas listas em uma única lista, e devolve o endereço da lista resultante. Para esta segunda versão, é permitido alterar o conteúdo de uma das listas passadas como parâmetro para a função.

c) Que considerações podemos fazer em relação às duas abordagens para implementar a junção de duas listas?

d) É possível realizar alguma alteração da estrutura **ListaLigada** de modo a acelerar o desempenho da função **junta_listas_2**?

e) Cria uma função **divide_lista**, que recebe o ponteiro de uma lista, e a divida em duas partes de igual tamanho (se a quantidade de elementos for ímpar, faça com que a primeira parte tenha 1 elemento a mais do que a segunda). A lista passada como parâmetro deve ser modificada para corresponder à primeira metade da divisão, enquanto o endereço da segunda lista deve ser devolvido como retorno da função.

f) Crie uma função chamada **processa_lista** que recebe um ponteiro para uma lista, um parâmetro **max** (do tipo Elemento - que podemos assumir que será um tipo numérico), e modifique a lista, de tal modo que qualquer valor **x** maior que **max** seja decomposto em parcelas **p1, p2, … pn** de tal modo que **x = p1 + p2 + … + pn**. Cada parcela não pode exceder o valor **max**, e também deve ter o maior valor possível (ou seja, com exceção da última parcela, todas as demais terão valor igual a **max**). Escreva uma função que modifique a própria lista recebida (ou seja, você não pode criar uma lista nova vazia, e preenchê-la com os valores adequados). Além disso, implemente a função de modo que ela preserve a posição relativa das parcelas na lista modificada em relação aos demais elementos da lista.

   Exemplo: dada a lista [20, 40, 30, 80, 90, 120, 10] e max = 50, sua função deve modificar a lista de modo que seu conteúdo passe a ser: [20, 40, 30, 50, 30, 50, 40, 50, 50, 20, 10].

g) Crie a função chamada **no_indice**, que recebe um ponteiro para uma lista, um índice e devolve o endereço do nó correspondente ao elemento que ocupa o índice especificado, ou o endereço nulo, caso o índice seja inválido.

h) Modifique a função **insere** para que ela passe a usar a função **no_indice**.

i) Modifique a função **valor_indice** para que ela passe a usar a função **no_indice**.

j) Cria a função **busca_no_anterior**, que funciona de modo análogo à busca, mas devolve o nó imediatamente anterior ao nó que contém o elemento procurado. Caso o elemento procurado seja o primeiro da lista, a função

deve devolver o endereço nulo. Caso o elemento procurado não exista, deve ser devolvido o endereço do último nó da lista.

k) Modifique a função **busca**, para que ela continue realizando a mesma operação, mas tirando proveito da função **busca_no_anterior**.

l) Modifique a função **remove_elemento**, para que ela continue realizando a mesma operação, mas usando proveito da função **busca_no_anterior**.

m) O que podemos dizer sobre a eficiência da nova versão da função **remove_elemento**?

4) Discuta a respeito da implementação de uma pilha, baseada em uma lista sequencial, apresentada abaixo. Discuta se é uma implementação correta de uma pilha (ou seja, realiza as operações esperadas corretamente), e o quão boa é esta implementação (em termos de eficiência).

```
#include <stdlib.h>
#include "lista_sequencial_dinamica.h"

typedef struct {

      ListaSequencial * lista;

} Pilha;

Pilha * cria_pilha(int n){

      Pilha * p = (Pilha *) malloc(sizeof(Pilha));
      p->lista = cria_lista(n);
      return p;
}

void imprime_pilha(Pilha * p){

      imprime(p->lista);
}

Boolean vazia(Pilha * p){

      return tamanho(p->lista) == 0;
}

Boolean empilha(Pilha * p, Elemento e){

      return insere(p->lista, e, 0);
}
```

```
Boolean desempilha(Pilha * p, Elemento * e){

      if(tamanho(p->lista) > 0){

            *e = p->lista->a[0];
            return remove_elemento(p->lista, *e);
      }

      return FALSE;
}
```

5) Escreva a função **inverte_lista** que recebe o endereço de uma lista ligada (lista simplesmente ligada, com ponteiro apenas para o nó inicial) como parâmetro, e devolve o endereço de uma outra lista ligada contendo os mesmos elementos, porém em ordem invertida.