Lista 4 - MAC0122 Princípios de Desenvolvimento de Algoritmos - POLI

Prof. Ronaldo Fumio Hashimoto

1 Exercícios

Os exercícios de 34 a 46 correspondem, respectivamente, aos exercícios de 3.34 a 3.46 do livro de Robert Sedgewick [1].

34. Escreva uma função que move o maior item de uma dada lista para o nó final da lista.

35. Escreva uma função que move o menor elemento de uma dada lista para o primeiro nó da lista.

36. Escreva uma função que rearranja uma lista ligada deixando os nós de posição pares depois dos nós de posição impares na lista, preservando a ordem relativa inicial destes nós pares e ímpares.

37. Implemente um fragmento de código para uma lista ligada que troca as posições dos nós posteriores aos referenciados pelos *links* `t` e `u`.

38. Escreva uma função que recebe um *link* para uma lista como argumento e retorna um *link* para uma cópia da lista (uma nova lista que contém os mesmos itens, na mesma ordem).

39. Escreva uma função que recebe dois argumentos - um *link* para uma lista e uma função que recebe um *link* como argumento - e remove todos os itens da lista dada para o qual a função (recebida como argumento) retorna um valor diferente de zero.

40. Resolva o exercicio 39, mas faça uma cópia de todos os nós que passam pelo teste e retorne um *link* para esta lista contendo esses nós, na ordem que eles aparecem na lista original.

41. Implemente uma versão do programa abaixo (Program 3.10 do livro) que usa um nó cabeça (*head node*).

Program 3.10

```
link reverse(link x)
   {  link t, y = x, r = NULL;
      while (y != NULL)
         { t = y->next; y->next = r; r = y; y = t; }
      return r;
   }
```

42. Implemente uma versão do programa abaixo (Programa 3.11 do livro) que não usa nós cabeça (*head nodes*).

Programa 3.11

```
struct node heada, headb;
link t, u, x, a = &heada, b;
for (i = 0, t = a; i < N; i++)
  {
    t->next = malloc(sizeof *t);
    t = t->next; t->next = NULL;
    t->item = rand() % 100;
  }
b = &headb; b->next = NULL;
for (t = a->next; t != NULL; t = u)
   {
      u = t->next;
      for (x = b; x->next != NULL; x->next)
         if (x->next->item > t->item) break;
      t->next = x->next; x->next = t;
   }
```

43. Implemente uma versão do programa abaixo (Programa 3.9 do livro) que usa um nó cabeça (*head node*)

    Programa 3.9

```
#include <stdio.h>
typedef struct node* link;
struct node { int item; link next; };
main(int argc, char* argv[])
{ int i, N = atoi(argv[1]), M = atoi(argv[2]);
  link t = malloc(sizeof(*t)), x = t;
  t->item = 1; t->next = t;
  for (i = 2; i <= N; i++)
    {
      x = (x->next = malloc(sizeof(*x)));
      x->item = i; x->next = t;
    }
  while (x != x->next)
    {
      for (i = 1; i < M; i++) x = x->next;
      x->next = x->next->next; N--;
    }
  printf("%d\n", x->item);
}
```

44. Implemente uma função que troca dois dados nós em uma lista duplamente ligada.

45. Escreva códigos nos moldes da Tab.1 (Tabela 3.1 do livro), descrevendo uma lista que nunca é vazia, é referênciada pelo ponteiro para o primeiro nó, e cujo nó final tenha um ponteiro para ele mesmo.

46. Escreva códigos nos moldes da Tab.1 (Tabela 3.1 do livro), descrevendo uma lista circular que tem um nó falso (*dummy node*), que funciona tanto como o nó cabeça (*head node*) quanto como nó cauda (*tail node*).

Tabela 1: Tabela retirada de [2]
Convenções sobre cabeça (*head*) e cauda (*tail*) em lista ligada.

Essa tabela exibe a implementação das operações de processamento de lista básicas com cinco convenções normalmente utilizadas. Este tipo de código é usado em aplicações simples onde o código do processamento de lista é *inline*.

Circular, nunca vazia

insere primeiro:	`head->next = head;`
insere t depois de x:	`t->next = x->next; x->next = t;`
remove depois de x:	`x->next = x->next->next;`
laço do percurso:	`t = head` `do { ... t = t->next;} while (t != head);`
testa se está vazio:	`if (head->next == head)`

Ponteiro para cabeça (*head pointer*), cauda nula (*null tail*)

inicializa:	`head = NULL`
insere t depois de x:	`if (x == NULL) { head = t; head->next = NULL; }` `else { t->next = x->next; x->next = t; }`
remove depois de x:	`t = x->next; x->next = t->next;`
laço do percurso:	`for (t = head; t != NULL; t = t->next)`
testa se está vazio:	`if (head == NULL)`

Nó cabeça falso (*dummy head node*), cauda nula (*null tail*)

inicializa:	`head = malloc(sizeof *head);` `head->next = NULL;`
insere t depois de x:	`t->next = x->next; x->next = t`
remove depois de x:	`t = x->next; x->next = t->next;`
laço de percuso:	`for (t = head->next; t != NULL; t = t->next)`
testa se está vazio:	`if (head->next == NULL)`

Nó cabeça falso (*dummy head node*), nó de cauda nula (*tail node*)

inicializa:	`head = malloc(sizeof *head);`
	`z = malloc(sizeof *z);`
	`head->next = z; z->next = z;`

insere t depois de x:	`t->next = x->next; x->next = t;`
remove depois de x:	`x->next = x->next->next;`
laço de percurso:	`for (t = head->next; t != z; t = t->next)`
testde se está vazio:	`if (head->next == z)`

Referências

[1] R. Sedgewick, "Algorithms in c—third edition," p. 82, 1998.

[2] R. Sedgewick, "Algorithms in c—third edition," p. 101, 1998.