Estrutura de Dados
Estrutura de dados Linear: Lista

Alessandro Ferreira Leite

02/04/2012

Introdução

- Uma seqüência de nós ou elementos dispostos em uma ordem estritamente linear.
- Cada elemento da lista é acessível um após o outro, em ordem.
- Pode ser implementada de várias maneiras
  1. Em um vetor
  2. Em uma estrutura que tem um vetor de tamanho fixo e uma variável para armazenar o tamanho da lista.

Definição

Definição

Um conjunto de nós, $x_1, x_2, x_3, \cdots, x_n$, organizados estruturalmente de forma a refletir as posições relativas dos mesmos. Se $n > 0$, então x_1 é o primeiro nó.
Seja L uma lista de n nós, e x_k um nó $\in$ L e k a posição do nó em L. Então, x_k é precedido pelo nó x_{k-1} e seguido pelo nó x_{k+1}. O último nó de L é x_{n-1}. Quando $n = 0$, dizemos que a lista está vazia.

Representação

- Os nós de uma lista são armazenados em endereços contínuos.
- A relação de ordem é representada pelo fato de que se o endereço do nó x_i é conhecido, então o endereço do nó x_{i+1} também pode ser determinado.
- A Figura 1 apresenta a representação de uma lista linear de n nós, com endereços representados por k

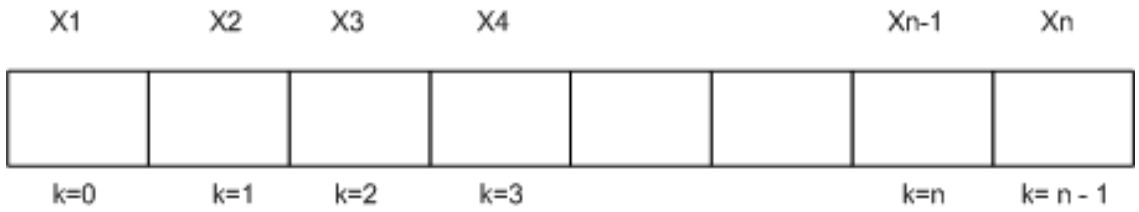

Figura: Exemplo de representação de lista.

Representação

- Para exemplificar a implementação em C, vamos considerar que o conteúdo armazenado na lista é do tipo inteiro.
- A estrutura da lista possui a seguinte representação:

```
struct lista{
  int cursor;
  int elemento[N];
}
typedef struct lista Lista;
```

- Trata-se de uma estrutura heterogênea constituída de membros distintos entre si. Os membros são as variáveis *cursor*, que serve para armazenar a quantidade de elementos da lista e o vetor *elemento* de inteiros que armazena os nós da lista.

Representação

- Para atribuirmos um valor a algum membro da lista devemos utilizar a seguinte notação:

```
Lista->elemento[0] = 1 - atribui o valor 1 ao primeiro
Lista->elemento[n-1] = 4 - atribui o valor 4 ao último
```

Operações Primitivas

- As operações básicas que devem ser implementadas em uma estrutura do tipo Lista são:

Operação	**Descrição**
criar()	cria uma lista vazia.
inserir(l,e)	insere o elemento *e* no final da lista *l*.
remover(l,e)	remove o elemento *e* da lista *l*.
imprimir(l)	imprime os elementos da lista *l*.
pesquisar(l,e)	pesquisa o elemento *e* na lista *l*.

Tabela: Operações básicas da estrutura de dados lista.

Operações auxiliares

- Além das operações básicas, temos as operações "auxiliares". São elas:

Operação	**Descrição**
empty(l)	determina se a lista *l* está ou não vazia.
destroy(l)	libera o espaço ocupado na memória pela lista *l*.

Tabela: Operações auxiliares da estrutura de dados lista.

Interface do Tipo Lista

```
/* Aloca dinamicamente a estrutura lista, inicializando seus
 * campos e retorna seu ponteiro. A lista depois de criada
 * terá tamanho igual a zero.*/
Lista* criar(void);

/* Insere o elemento e no final da lista l, desde que,
 * a lista não esteja cheia.*/
void inserir(Lista* l, int e);

/* Remove o elemento e da lista l,
 * desde que a lista não esteja vazia e o elemento
 * e esteja na lista. A função retorna 0 se o elemento
 * não for encontrado na lista ou 1 caso contrário. */
void remover(Lista* l, int e);

/* Pesquisa na lista l o elemento e. A função retorna
 * o endereço(índice) do elemento se ele pertencer a lista
 * ou -1 caso contrário.*/
int pesquisar(Lista* l, int e);

/* Apresenta os elementos da lista l. */
void imprimir(Lista* l);
```

Implementação da Lista

- A utilização de vetores para implementar a lista traz algumas vantagens como:
  1. Os elementos são armazenados em posições contíguas da memória;
  2. Economia de memória, pois os ponteiros para o próximo elemento da lista são explícitos.
- No entanto, as desvantagens são:
  1. Custo de inserir/remover elementos da lista;
  2. Limitação da quantidade de elementos da lista.

Função de Criação

- A função que cria uma lista, deve criar e retornar uma lista vazia;
- A função deve atribuir o valor zero ao tamanho da lista, ou seja, fazer $l->cursor = 0$, como podemos ver no código abaixo.
- A complexidade de tempo para criar a lista é constante, ou seja, $O(1)$.

```
/*
 * Aloca dinamicamente a estrutura lista, inicializando seus
 * campos e retorna seu ponteiro. A lista depois de criada
 * terá tamanho igual a zero.
 */
Lista* criar(void){
  Lista* l = (Lista*) malloc(sizeof(Lista));
  l->cursor = 0;
  return l;
}
```

Função de Inserção

- A inserção de qualquer elemento ocorre no final da lista, desde que a lista não esteja cheia.
- Com isso, para inserir um elemento basta atribuirmos o valor ao elemento cujo índice é o valor referenciado pelo campo *cursor*, e incrementar o valor do cursor, ou seja fazer $l->elemento[l->cursor++] = valor$, como podemos verificar no código abaixo, a uma complexidade de tempo constante, $O(1)$.

```
/*
 * Insere o elemento e no final da lista l, desde que,
 * a lista não esteja cheia.
 */
void inserir(Lista* l, int e){
   if (l == NULL || l->cursor == N){
      printf("Error. A lista esta cheia\n");
   }else{
      l->elemento[l->cursor++] = e;
   }
}
```

Função de Remoção

- Para remover um elemento da lista, primeiro precisamos verificar se ele está na lista, para assim removê-lo, e deslocar os seus sucessores, quando o elemento removido não for o último.
- A complexidade de tempo da função de remoção é $O(n)$, pois é necessário movimentar os n elementos para remover um elemento e ajustar a lista.

```
/* remove um elemento da lista */
void remover(Lista* l, int e){
  int i, d = pesquisar(l,e);
  if (d != -1){
    for(i = d; i < l->cursor; i++)
    {
      l->elemento[i] = l->elemento[i + 1];
    }
    l->cursor--;
  }
}
```

Função de Pesquisa

- Para pesquisar um elemento qualquer na lista é necessário compará-lo com os elementos existentes, utilizando alguns dos algoritmos de busca conhecidos;
- A complexidade de tempo dessa função depende do algoritmo de busca implementado. Se utilizarmos a busca seqüencial, a complexidade da função será $O(n)$. No entanto, é possível baixá-lo para $O(n \log n)$.

```
int pesquisar(Lista* l, int e){
  if (l == NULL)
    return;

  int i = 0;
  while (i <= l->cursor && l->elemento[i] != e)
    i++;

  return i > l->cursor ? -1 : i;
}
```

Função de Impressão

- A impressão da lista ocorre através da apresentação de todos os elementos compreendidos entre o intervalo: $[0..l->cursor]$.
- A complexidade de tempo da função de impressão é $O(n)$, pois no pior caso, quando lista estiver cheia, é necessário percorrer os n elementos da lista.

```
/*Apresenta os elementos da lista l. */
void imprimir(Lista* l){
  int i;
  for(i = 0; i < l->cursor; i++)
     printf("%d ", l->elemento[i]);
  printf("\n");
}
```

Exemplo de Uso da Lista

```
#include <stdio.h>
#include "list.h"

int main(void)
{
    Lista* l = criar();
    int i, j = 4;

    /* Inseri 5 elementos na lista */
    for (i = 0; i < 5; i++)
      inserir(l,j * i);

    /* Apresenta os elementos inseridos na lista*/
    imprimir(l);
    /* Remove o segundo elemento da lista*/
    remover(l,j);
    /* Apresenta os elementos da lista */
    imprimir(l);
}
```

Referências

1. Tenenbaum, A. M., Langsam, Y., and Augestein, M. J. (1995). Estruturas de Dados Usando C. MAKRON Books, pp. 207-250.
2. Wirth, N. (1989). Algoritmos e Estrutura de dados. LTC, pp. 151-165.