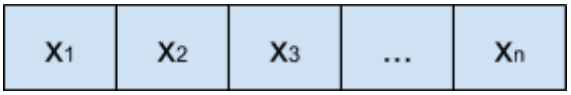
Notas de aula 04/04

Listas

Uma lista é um tipo de estrutura de dados que guarda uma sequência de itens:



onde x_1 é o primeiro item da lista, x_2 é o segundo item da lista e assim por diante até x_n, que é o último item da lista. O *n* representa o tamanho da lista.

Em uma lista, x_i é o item que precede o item x_{i+1} e que sucede o item x_{i-1}. Chamamos x_i de i-ésimo item da lista.

- A lista é uma das estruturas mais simples para guardar elementos;
- É flexível: dependendo do método de implementação, seu tamanho pode aumentar ou diminuir durante a execução do programa, conforme a demanda. Por esta característica, listas são adequadas para aplicações nas quais não é possível prever a demanda por memória;
- Aplicações: cadastros em geral, armazenar dados que precisam estar ordenados (mais adiante veremos como ordenar os itens de uma lista).

TAD Lista

É formado pela estrutura que representa a lista e por um conjunto de operações para manipular esta estrutura. Geralmente, uma lista precisa das seguintes operações:

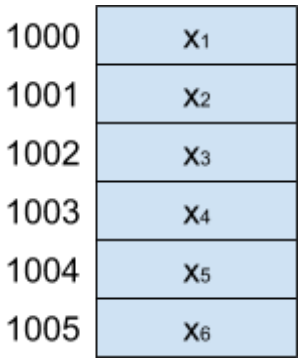
- Criar uma lista vazia;
- Inserir um item;
- Remover um item;
- Acessar um item;
- Verificar se a lista está vazia;
- Verificar se a lista está cheia;
- Imprimir a lista;
- Liberar a lista.

Implementações de lista

Os métodos mais comuns de implementar listas são por meio de arranjos (vetores) ou por meio de ponteiros (listas encadeadas). Primeiramente estudaremos a implementação por arranjos ou vetores.

Implementação por arranjos ou vetores

Seguindo a lógica de vetores, os itens da lista são armazenados em posições contíguas de memória. Considere, por exemplo, uma lista de 6 itens:



Ainda seguindo a ideia de vetores, é preciso definir um tamanho máximo para a lista e deve-se respeitar este tamanho.

Implementação de lista usando vetores em C

Um item de uma lista normalmente é uma estrutura que contém uma chave (um identificador para o item) e os campos desejados para o item:

```
typedef struct item Item;
struct item {
      int chave;
      // demais campos desejados
};
```

Se o objetivo fosse manter uma lista de livros, por exemplo, além do campo chave, poderíamos ter os campos: título, autor, editora, genero, ano, edição, número de páginas, etc.

Uma lista geralmente é representada por uma estrutura que contém um vetor de itens e um campo que guarda a quantidade de itens da lista (para depois poder percorrer os itens). Assim, temos:

```
#define MAXTAM 100
typedef struct item Item;
typedef struct lista Lista;

struct item {
      int chave;
      // demais campos desejados
};

struct lista {
      Item item[MAXTAM];
      int quantidade; // das 100 posições reservadas, quantas já foram ocupadas?
};
```

Definida a estrutura da lista, iremos implementar primeiramente a função para criação da lista:

```
// criar uma lista vazia
Lista * cria_lista_vazia() {
      Lista *l = malloc(sizeof(Lista));
      l->ultimo = -1;
      return l;
}
```

A função não precisa de nenhum parâmetro. Primeiramente a memória para guardar a lista é alocada (a função malloc aloca um bloco de endereços contíguos de memória para a nova lista). Depois podemos atribuir o valor -1 para o campo último, como uma forma de indicar que a lista está vazia. Finalmente, a função retorna um ponteiro para uma lista vazia pronta para uso.

Com a lista pronta podemos pensar na inserção de itens. A inserção de itens só deve ser possível enquanto a lista não estiver cheia, ou seja, enquanto o campo último não apontar para o tamanho máximo da lista. Sendo assim, antes de criar uma função para inserir itens, convém criar uma função para verificar se a lista está cheia:

```
// retorna 1 se a lista está cheia ou 0 se não está cheia
int verifica_lista_cheia(Lista *l) {
      return l->ultimo == MAXTAM - 1;
}
```

A lista estará cheia quando o campo último apontar para o tamanho máximo da lista menos um, se formos considerar que a lista inicia no índice 0.

Vamos agora inserir um item no final da lista:

```
// adiciona um elemento no fim da lista
void adiciona_item_fim_lista(Lista *l, int chave) {
      if(verifica_lista_cheia(l)){
            printf("Erro: a lista está cheia.\n");
            return;
      }
      else {
            Item novo_item;
            novo_item.chave = chave;
            l->ultimo++;
            l->item[l->ultimo] = novo_item;
      }
}
```

A função recebe como parâmetro um ponteiro para a lista e também os dados do item a ser adicionado (neste caso, o item só contém uma chave). Um novo tipo Item é criado para guardar o novo item. Se a lista estiver cheia, mostramos uma mensagem

informando essa situação e retornamos. Se não, é possível inserir o item. O novo item criado anteriormente recebe os dados, incrementados o campo último (pois agora mais uma posição da lista será ocupada) e inserimos então o novo item na última posição da lista. Como a lista foi passada por referência, não é preciso de retorno.

Agora que já podemos adicionar alguns itens na lista, é conveniente criar uma função para imprimir a lista:

```
// imprime a lista
void imprime_lista(Lista *l) {
    int tam = l->ultimo + 1;
    int i;
    for(i = 0; i < tam; i++)
        printf("Chave: %d\n", l->item[i].chave);
}
```

A função recebe um ponteiro para a lista, calcula o tamanho da lista com base no campo último e assim consegue percorrer a lista imprimindo os campos de cada item.

Outra função importante em um TAD lista é excluir um item. A exclusão de um item só pode ser feita se a lista não estiver vazia. Portanto, vamos primeiramente criar uma função que verifica se a lista está vazia. Para isso, basta receber um ponteiro para a lista e verificar se o campo último ainda é igual a -1 (valor que ele recebeu no momento da criação da lista vazia):

```
// retorna 1 se a lista está vazia ou 0 se não está vazia
int verifica_lista_vazia(Lista *l) {
      return l->ultimo == -1;
}
```

Podemos remover um item da lista de duas formas: passando o índice do item que deve ser removido ou então passando a chave do item que deve ser removido. A segunda opção tem mais utilidade. Imagine que você tem um cadastro de alunos. Você não lembra qual a posição de cada aluno dentro da lista, mas sabe o RA de cada um, então você pode usar o RA para indicar qual aluno quer excluir da lista. O RA nada mais seria do que a chave do item.

Sendo assim, antes de criar a função para excluir um item, podemos criar uma função de busca, que recebe a chave do item que deve ser excluído e retorna seu índice:

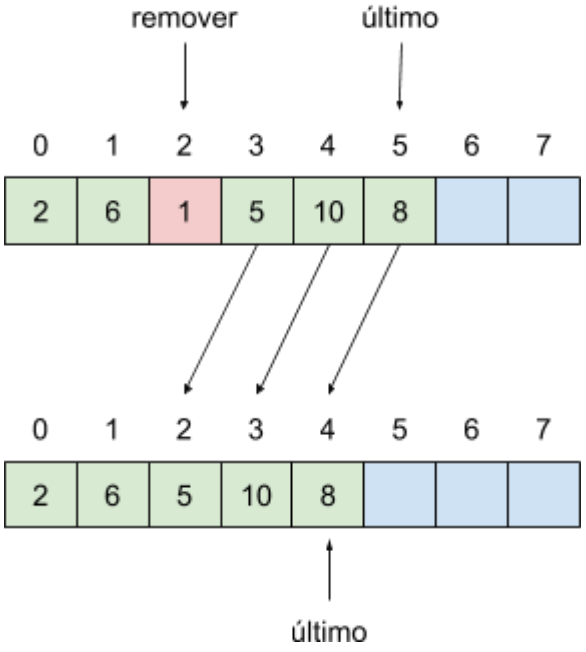
```
// retorna o indice do item com a chave buscada ou -1 se não encontrou
int busca_item_por_chave(Lista *l, int chave) {
      int tam, i, posicao;
      tam = l->ultimo + 1;
      posicao = -1;
      for(i = 0; i < tam; i++)
            if(l->item[i].chave == chave)
                  posicao = i;
      return posicao;
}
```

Criamos uma variável chamada posição que é inicializada com -1. Depois percorremos a lista verificando se a chave informada confere com a chave de algum item da lista. Em caso afirmativo, a variável posição é atualizada com o índice do item que possui aquela chave. A variável posição é retornada.

Com a criação das duas funções anteriores fica mais fácil fazer a função para exclusão de um item:

```
// remove um item qualquer da lista
void remove_item(Lista *l, int chave) {
      int posicao, vazia, i, tam;
      vazia = verifica_lista_vazia(l);
      posicao = busca_item_por_chave(l, chave);
      if (vazia || posicao == -1) {
            printf("Erro: a lista está vazia ou o item não existe.\n");
            return;
      }
      else {
            tam = l->ultimo + 1;
            for(i = posicao; i < tam - 1; i++)
                  l->item[i] = l->item[i + 1];
                  l->ultimo--;
      }
}
```

A função recebe um ponteiro para a lista e a chave do item a ser excluído. Se a lista estiver vazia ou a chave não pertencer a nenhum item da lista, mostramos uma mensagem de erro informando essa situação e retornamos. Caso contrário, precisamos percorrer a lista deslocando todos os itens após o índice do item que desejamos remover. Precisamos também decrementar o campo último, para indicar que uma posição foi liberada. Por exemplo:



Por fim devemos lembrar que se a memória para a lista foi alocada usando malloc, é preciso liberar a memória manualmente:

```
void libera_lista(Lista *l) {
      free(l);
}
```

Na função principal podemos chamar as funções criadas para fazer algumas manipulações na lista. Para facilitar considere uma lista com tamanho máximo = 6:

```
main() {
      Lista *l;
      int vazia, cheia, chave;

      printf("Foi criada a lista vazia.\n");
      l = cria_lista_vazia();

      vazia = verifica_lista_vazia(l);
      printf("Lista vazia? %d\n", vazia);

      cheia = verifica_lista_cheia(l);
      printf("Lista cheia? %d\n", cheia);

      printf("\nAdicionando o 1o item.\n");
      chave = 1;
      adiciona_item_fim_lista(l, chave);
      vazia = verifica_lista_vazia(l);
      printf("Lista vazia? %d\n", vazia);

      printf("\nAdicionando o 2o item.\n");
      chave = 2;
      adiciona_item_fim_lista(l, chave);

      printf("Adicionandoo 3o item.\n");
      chave = 3;
      adiciona_item_fim_lista(l, chave);

      printf("Adicionando o 4o item.\n");
      chave = 4;
      adiciona_item_fim_lista(l, chave);

      printf("Adicionando o 5o item.\n");
      chave = 5;
      adiciona_item_fim_lista(l, chave);

      cheia = verifica_lista_cheia(l);
```

```
      printf("Lista cheia? %d\n", cheia);

      printf("\nTentando adicionar 6o item.\n");
      chave = 6;
      adiciona_item_fim_lista(l, chave);

      printf("\nItens da lista:\n");
      imprime_lista(l);

      printf("\nBuscando posição do item a partir da chave.\n");
      printf("Posição do item com chave = 1: %d\n",
busca_item_por_chave(l, 1));
      printf("Posição do item com chave = 2: %d\n",
busca_item_por_chave(l, 2));
      printf("Posição do item com chave = 3: %d\n",
busca_item_por_chave(l, 3));
      printf("Posição do item com chave = 4: %d\n",
busca_item_por_chave(l, 4));
      printf("Posição do item com chave = 5: %d\n",
busca_item_por_chave(l, 5));

      printf("\nTentando remover item com chave = 6.\n");
      remove_item(l, 6);
      imprime_lista(l);

      printf("\nTentando remover item com chave = 3.\n");
      remove_item(l, 3);
      imprime_lista(l);

      libera_lista(l);
}
```

Confira o código completo separado em Lista.h, Lista.c e usa_lista.c no github.

Atividade

No tutorial acima vimos como adicionar um item no final da lista. Como você faria para adicionar um item em alguma posição do meio da lista?

Vantagens da implementação por arranjos

- Como é implementada como um vetor, o acesso aos itens da lista é fácil, basta acessar os itens através dos índices.
- O custo para acessar cada item é constante: o custo para acessar o primeiro item da lista é igual ao custo para acessar o último item da lista. Lembre-se: um vetor *p* é um ponteiro para o primeiro endereço de memória ocupado. Então, para acessar o primeiro item tenho *p + 0*, para acessar o segundo item tenho *p + 1*, para acessar o

último item, tenho $p + n$. Em todos os casos, a mesma operação de soma precisa ser feita para encontrar o endereço de memória de um determinado item.
- Implementação mais fácil do que uma implementação de lista encadeada.

Desvantagens da implementação por arranjos

- Operações de inserção e remoção no meio da lista são custosas: para adicionar um item na posição *i* da lista, é preciso deslocar todos os itens após *i* uma posição à frente. Para remover um item na posição *i* da lista, é preciso deslocar todos os itens após *i* uma posição atrás. Por exemplo, se a lista tiver 100 itens e eu quiser remover o primeiro, terei que fazer 99 deslocamentos.
- Tamanho limitado: no momento em que escrevo o programa preciso definir quantos itens a lista terá. Se eu definir que a lista terá 1000 itens, essa é a quantidade máxima de itens que poderei ter.

Concluindo

- Se a lista for pequena ou existir uma estimativa da quantidade de itens necessários, usar a implementação por arranjos é uma opção, pois não haverá falta nem excesso de memória.
- Se os itens da lista puderem ser inseridos e removidos sempre do final, também é uma opção usar a implementação por arranjos, visto que inserções e remoções no final da lista tem um baixo custo (não é preciso fazer nenhum deslocamento, diferente de inserções e remoções no meio da lista).

Materiais interessantes sobre listas

Série de vídeos sobre listas prof. André Backes: https://www.youtube.com/watch?v=S6rOYN-UiAA

Capítulo sobre estruturas de dados básicas do livro Projeto De Algoritmos Com Implementações Em Pascal E C - Nivio Ziviani.

Slides sobre estruturas de dados básicas dos prof. Nivio Ziviani e Charles Ornelas Almeida: http://www2.dcc.ufmg.br/livros/algoritmos/cap3/slides/c/completo1/cap3.pdf