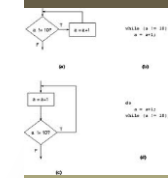
Estruturas Dinâmicas Básicas na Linguagem C (Listas e Pilhas)

Edkallenn Lima

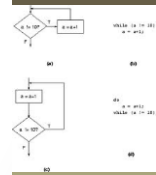
Agente de Polícia Federal

Chefe do Núcleo de Tecnologia da Informação da Polícia Federal- NTI/AC



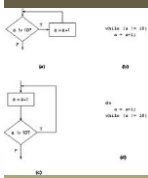
Estruturas de Dados

- Prof. Edkallenn Lima
- edkallenn@bol.com.br (somente para dúvidas)
- Blogs:
  - http://professored.wordpress.com (Computador de Papel – O conteúdo da forma)
  - http://professored.tumblr.com/ (Pensamentos Incompletos)
  - http://umcientistaporquinzena.tumblr.com/
  - http://eulinoslivros.tumblr.com/
  - http://linabiblia.tumblr.com/
- Redes Sociais:
  - http://www.facebook.com/edkallenn
  - http://twitter.com/edkallenn
  - https://plus.google.com/u/0/113248995006035389558/posts
  - Instagram: http://instagram.com/edkallenn ou @edkallenn
  - Foursquare: https://pt.foursquare.com/edkallenn
- Telefones:
  - 68 8401-2103 (VIVO) e 68 3212-1211.
- Os exercícios devem ser enviados SEMPRE para o e-mail: edkevan@gmail.com ou para o e-mail: edkallenn.lima@uninorteac.com.br



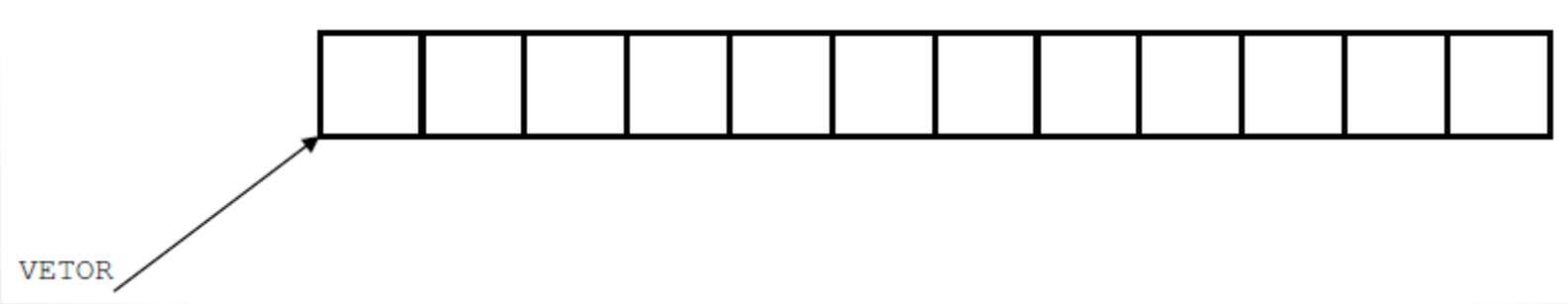
Agenda

- Introdução
- Estruturas autorreferenciadas
- Listas encadeadas
  - Função de criação
  - Função de Inserção
  - Função que percorre os elementos
  - Função que verifica se a lista está vazia
  - Função de Busca
  - Função que retira um elemento da lista
  - Função para liberar a lista
  - Manutenção da Lista Ordenada
  - Definição Recursiva de lista
  - Implementações Recursivas
- Listas Duplamente Encadeadas
  - Função de Inserção
  - Função de Busca
  - Função para Retirar um elemento da Lista
- Pilhas
  - Introdução
  - Implementação de pilha com vetor
  - Implementação de pilha com lista



Introdução

- Para representar conjuntos de dados podemos usar, como vimos, vetores.
- O vetor é a forma mais primitiva de representar dados agrupados.
- O vetor ocupa um espaço contíguo de memória
- Permite acesso randômico (podemos acessar qualquer elemento aleatoriamente)
- O vetor deve ser dimensionado com um número máximo de elementos

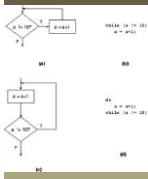


Introdução

- Entretanto, o vetor não é uma estrutura de dados muito flexível
- Pois precisamos dimensioná-lo com um número máximo de elementos

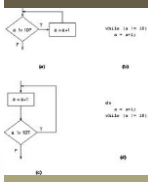
```
#define MAX 1000
int vet[MAX];
```

- Se o número de elementos exceder a dimensão do vetor há o problema de alterar a dimensão do vetor em tempo de execução
- Por outro lado, se o número de elementos for muito inferior à sua dimensão, espaço de memória será subutilizado



Introdução

- A solução é utilizar estruturas de dados que cresçam conforme precisem armazenar novos elementos
- E diminuam conforme precisarmos retirar elementos armazenados anteriormente
- Essas são chamadas Estruturas de dados dinâmicas
- Um exemplo são as listas encadeadas (que são usadas para implementar outras estruturas de dados)



Estruturas auto-referenciadas

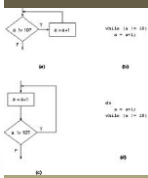
- Uma estrutura autorreferenciada contém um membro ponteiro que aponta para uma estrutura do mesmo tipo.
- Por exemplo, a definição:


```
struct no{
    int dado;
    struct no *proxPtr;
};
```

- Define um tipo, **`struct no`** com dois membros: um membro **`dado`** e um membro ponteiro **`proxPtr`**
- **`proxPtr`** aponta para uma estrutura **`struct no`** (a mesma declarada aqui)
- Daí o termo “auto-referenciada”

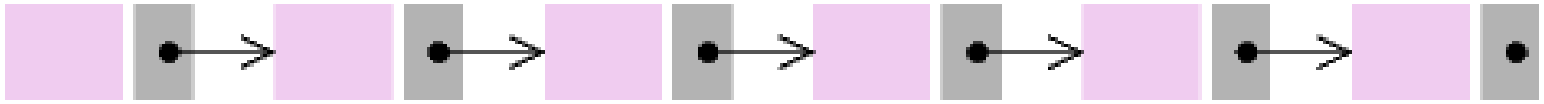
Estruturas auto-referenciadas

- O ponteiro **`proxPtr`** é usado para "ligar" uma estrutura do tipo **`struct no`** a outra do mesmo tipo
- É chamado de "link"
- As estruturas auto-referenciadas podem ser ligadas entre si para formar estruturas úteis de dados como **listas**, **filas**, **pilhas** e **árvores**
- As chamadas estruturas de dados dinâmicas.



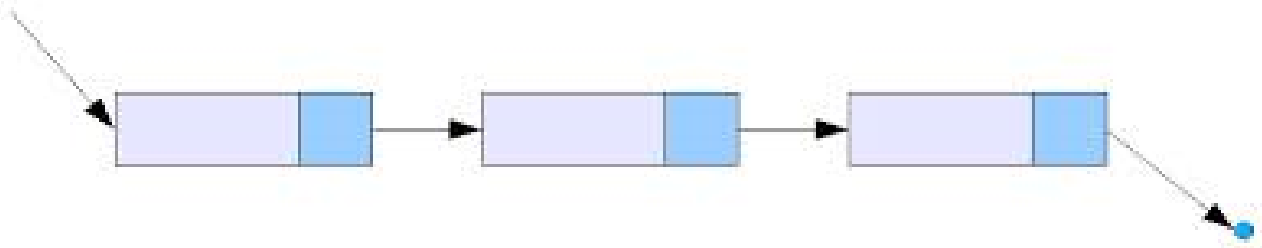
Listas encadeadas

- Uma Lista Encadeada (ou lista linear, ou linked list) é um conjunto linear de estruturas autorreferenciadas, chamadas nós, conectadas por links de ponteiros
- Uma lista encadeada é acessada por meio de um ponteiro para o primeiro nó da lista



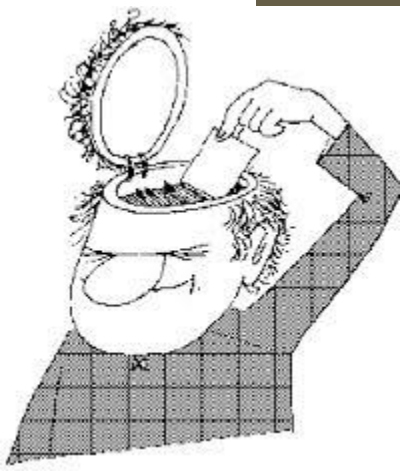
Listas encadeadas

- Os dados são armazenados dinamicamente em uma lista encadeada (cada nó é criado quando necessário)
- O ponteiro do último elemento é NULL

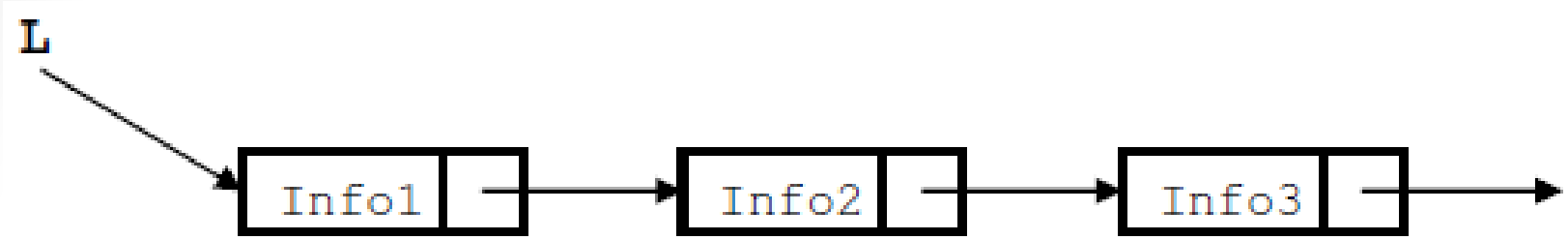


Listas encadeadas

- Para cada novo elemento inserido na estrutura, aloca-se um espaço de memória para armazená-lo
- O espaço é, portanto, proporcional ao número de elementos armazenado
- Não há como garantir que os elementos da lista ocuparão um espaço contíguo
- Não temos acesso direto aos elementos da lista
- Para percorrer a lista deve-se guardar o encadeamento, que é feito guardando-se o ponteiro para o próximo elemento da lista.



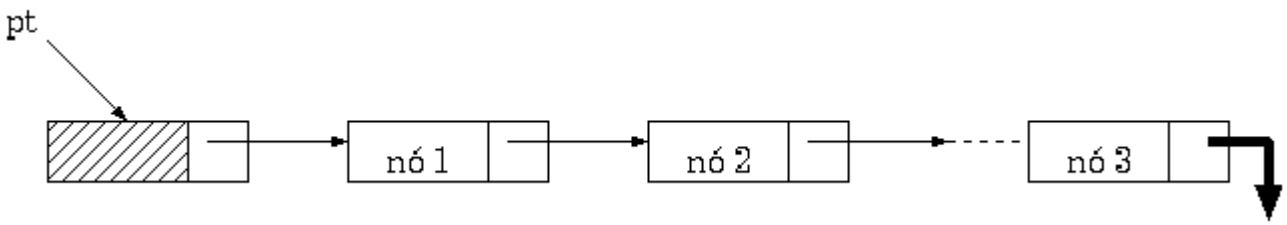
Listas encadeadas



- Conceitualmente, cada nó da lista é representado por uma estrutura que contém a informação (ou informações) e o ponteiro para o próximo elemento da lista.
- A lista é representada por um ponteiro para o primeiro elemento (ou nó)
- Do primeiro elemento, podemos alcançar o segundo, seguindo o encadeamento e assim por diante
- O último elemento é um ponteiro inválido com valor NULL, e sinaliza que não há outros elementos.

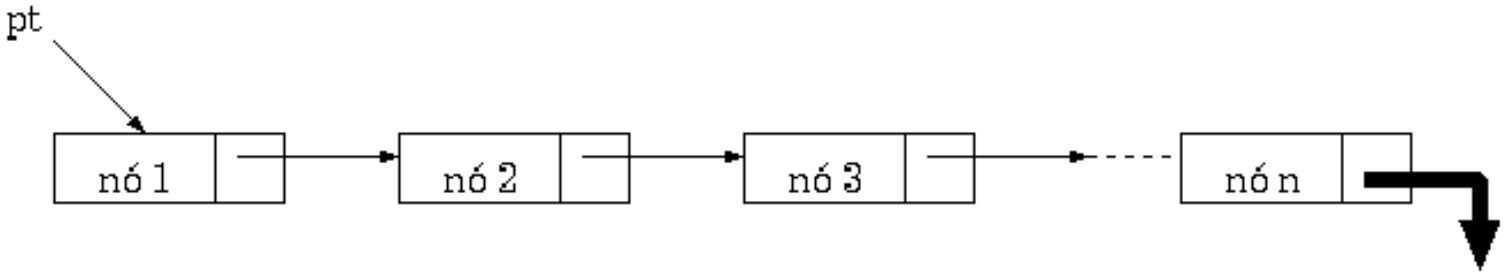
Listas com e sem cabeça

- Lista *com* cabeça. O conteúdo da primeira célula é irrelevante: ela serve apenas para marcar o início da lista. A primeira célula é a **cabeça** (= *head cell* = *dummy cell*) da lista.
- A primeira célula está sempre no mesmo lugar na memória, mesmo que a lista fique vazia



Lista com e sem cabeça

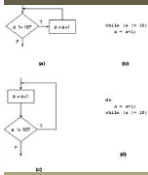
- Lista *sem* cabeça. O conteúdo da primeira célula é tão relevante quanto o das demais. Nesse caso, a lista está vazia se o endereço de sua primeira célula é NULL



- Utilizaremos no exemplo uma lista sem cabeça (por ser mais "pura).

TAD Lista Encadeada (inteiros)

- Interface:
  - Define (Declara) da estrutura autorreferenciada Lista de inteiros.
- Funções:
  - lst_cria – Cria a lista
  - lst_libera – Libera a lista
  - lst_retira – Retira um elemento da lista
  - lst_vazia – Verifica se a lista está vazia
  - lst_busca – Busca um elemento na lista
  - lst_imprime – Imprime a lista

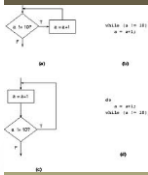


Listas encadeadas - exemplo

- Como exemplo vamos usar um exemplo simples em que queremos armazenar valores inteiros em uma lista encadeada.
- Cada nó da lista pode ser representado pela estrutura a seguir:

```
//definicao
struct lista{
    int info;
    struct lista* prox;
};
//sinonimo
typedef struct lista Lista;
```

- Pode se notar que é uma estrutura autorreferenciada.
- Embora não seja essencial, é uma boa estratégia definir o tipo **`Lista`** como sinônimo de **`struct lista`**
- O tipo **`Lista`** representa um nó da lista e a estrutura encadeada é representada pelo ponteiro para seu primeiro elemento (tipo **`Lista*`**)
- Assim, com esta definição, pode-se definir as principais funções necessárias para implementar uma lista encadeada.



Antes, uma observação:

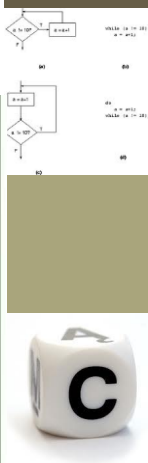
- **Endereço de uma lista:**
- O **endereço** de uma lista encadeada é o endereço de seu primeiro nó.
- Se p é o endereço de uma lista, convém, às vezes, dizer simplesmente:

"p é uma lista".

- Listas são eminentemente recursivas.
- Para tornar isso evidente, basta fazer a seguinte observação: se p é uma lista então vale uma das seguintes alternativas:

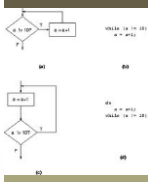
`p == NULL` ou

`p->prox` é uma lista.



Função de Criação

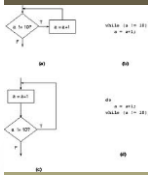
- A função que cria uma lista vazia deve ter como valor de retorno uma lista sem nenhum elemento
- Como a lista é representada para o primeiro elemento, uma lista vazia é representada pelo ponteiro **NULL**, pois não há elementos na lista
- O valor de retorno é a lista vazia inicializada, isto é, o valor de retorno é **NULL**.



Função de Criação

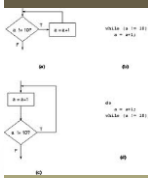
- Cria uma lista vazia, representada pelo ponteiro NULL

```
//Função de criação
//Retorna uma lista vazia
Lista* lst_cria(void)
{
    return NULL;
}
```



Função de Inserção

- Uma vez criada a lista vazia, pode-se inserir nela novos elementos
- Para cada elemento inserido, deve-se alocar dinamicamente a memória necessária para armazenar o elemento e encadeá-lo na lista existente
- A função de inserção mais simples insere o elemento no início da lista
- A seguir, uma possível implementação desta função



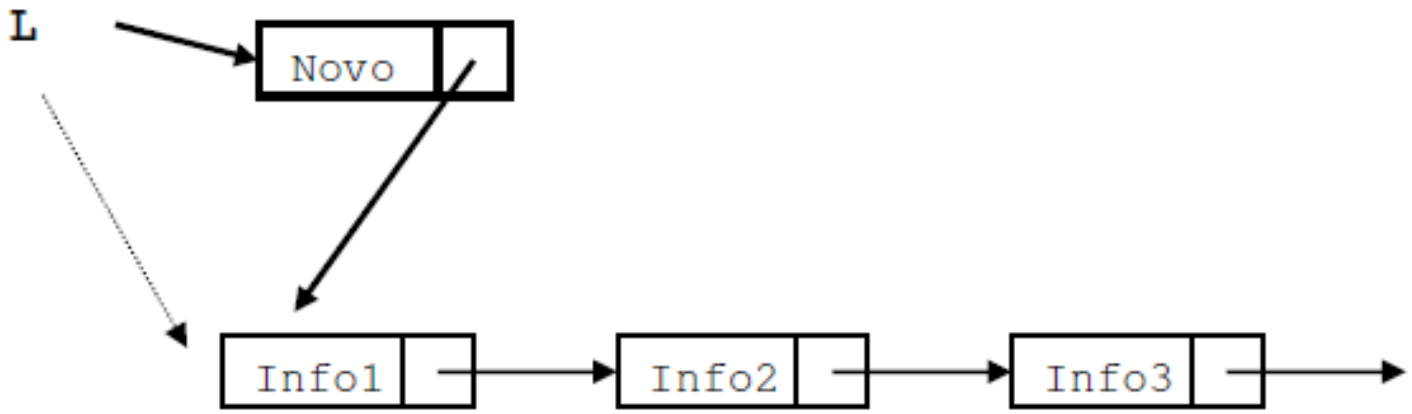
Função de Inserção

- Aloca memória para armazenar o elemento
- Encadeia o elemento na lista existente

```
//Insercao no inicio - retorna a lista atualizada
Lista* lst_insere(Lista* l, int i){
    Lista* novo = (Lista*)malloc(sizeof(Lista));
    novo->info = i;
    novo->prox = l;
    return novo;
}
```

- O ponteiro que representa a lista deve ter seu valor atualizado, pois a lista deve passar a ser representada pelo ponteiro para o novo primeiro elemento
- Os parâmetros de entrada são a lista na qual será inserido o novo elemento e a informação do novo elemento
- O valor de retorno é a nova lista, representada pelo ponteiro para o novo elemento.

Função de Inserção



- A função aloca dinamicamente o espaço para armazenar o nó da lista, guarda a informação no novo nó e faz ele apontar (isto é, tenha como próximo elemento) para o elemento que era o primeiro da lista.

Exemplo - Trecho de código

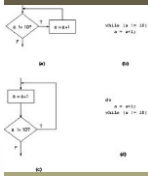
- Este trecho de código cria uma lista inicialmente vazia e insere novos elementos.

```
int main (void)
{
    Lista* l;                /* declara uma lista não inicializada */
    l = lst_cria();          /* cria e inicializa lista como vazia */
    l = lst_insere(l, 23);   /* insere na lista o elemento 23 */
    l = lst_insere(l, 45);   /* insere na lista o elemento 45 */
    ...
    return 0;
}
```

- Deve-se atualizar a variável que representa a lista a cada inserção de um novo elemento.
- Se o valor de l não fosse atualizado após a inserção do primeiro elemento, estaríamos passando na segunda chamada da função insere o valor de uma lista vazia, como se o primeiro elemento não tivesse sido inserido.

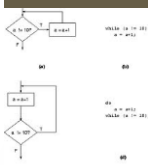
```c
#include <stdio.h>
#include <stdlib.h>
//definicao
struct lista{
    int info;
    struct lista* prox;
};
//sinonimo
typedef struct lista Lista;
//prototipos
Lista* lst_cria(void);
Lista* lst_insere(Lista* l, int i);
main(){
    Lista *l_nova;

    l_nova = lst_cria();
    l_nova = lst_insere(l_nova, 13);
    l_nova = lst_insere(l_nova, 45);
    l_nova = lst_insere(l_nova, 666);
}

//Função de criação
//Retorna uma lista vazia
Lista* lst_cria(void){
    return NULL;
}
//Insercao no inicio - retorna a lista atualizada
Lista* lst_insere(Lista* l, int i){
    Lista* novo = (Lista*)malloc(sizeof(Lista));
    novo->info = i;
    novo->prox = l;
    return novo;
}
```

- Programa completo exemplificando as funções de criação e de inserção em uma lista singularmente encadeada.
- Programa nada mostra na tela
- Ainda não temos uma função para exibir os elementos
- Veremos a seguir



Alternativa(não implementar!)

- Como alternativa à função de inserção vista poderíamos fazer a função insere receber o endereço da variável que representa a lista
- Assim, dentro da própria função insere, poderíamos atualizar o valor que representa a lista na função principal
- Nesse caso, os parâmetros das funções seriam do tipo ponteiro de ponteiro para lista (Lista**), e seu conteúdo poderia ser acessado/atualizado de dentro da função por meio do operador conteúdo(*l).
- Ficaria assim:



Insere atualizando (não implementar!)

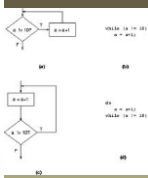
```
//insercao no inicio: atualiza valor da lista
void lst_insere(Lista** l, int i){
    Lista* novo = (Lista*)malloc(sizeof(Lista));
    novo->info = i;
    novo->prox = *l;
    *l = novo;
}
```

- Uma função cliente chamaria essa função do seguinte modo:

```
Lista* l = lst_cria();   //cria lista vazia
lst_insere(&l, 23);      //insere o elemento 23
```

Função insere - considerações

- A escolha de qual estratégia usar é uma questão de gosto (ou desgosto) do programador
- Mas deve-se ser consistente, adotando, se possível, sempre a mesma estratégia
- Para evitar as mais complicadas indireções múltiplas (ponteiros para ponteiros), sempre que possível optaremos pela primeira versão.
- O uso do valor de retorno parece a forma mais natural de se programar em C



Função que percorre os elementos de uma lista

- Para ilustrar uma função que percorre todos os elementos de uma lista, vamos implementar uma função que imprime os valores armazenados em uma lista:

```
//funcao imprime: imprime valores dos elementos
void lst_imprime(Lista* l){
    Lista* p;   //variavel auxiliar para percorrer a lista
    for(p = l; p !=NULL; p = p->prox)
        printf("info = %d\n", p->info);
}
```

- Para percorrer um vetor usamos uma variável inteira (que armazenava os índices)
- Em uma lista encadeada a variável deve ser um ponteiro, usada para armazenar o endereço de cada elemento.
- A variável **p** acima aponta para cada um dos elementos da lista, do primeiro ao último.

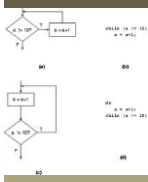
Função que verifica se a lista está vazia

- Pode ser útil implementar uma função que verifica se a lista está ou não vazia.
- Retorna 1 se a lista estiver vazia ou 0 se não estiver vazia

```
//funcao vazia: retorna 1 se vazia ou 0 se nao vazia
int lst_vazia(Lista* l){
    if (l == NULL)
        return 1;
    else
        return 0;
}
```

- Ou melhor:

```
//funcao vazia: retorna 1 se vazia ou 0 se nao vazia
int lst_vazia(Lista* l){
    return (l == NULL);
}
```



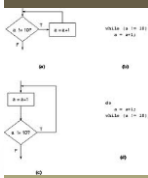
Função de Busca

- Recebe a informação referente ao elemento a pesquisar
- Retorna o ponteiro do nó da lista que representa o elemento, ou NULL, caso o elemento não seja encontrado na lista

```
//funcao busca: busca um elemento na lista
Lista* lst_busca(Lista* l, int v){
    Lista* p;
    for(p=l;p!=NULL;p=p->prox){
        if(p->info == v)
            return p;
    }
    return NULL;     //nao achou o elemento
}
```

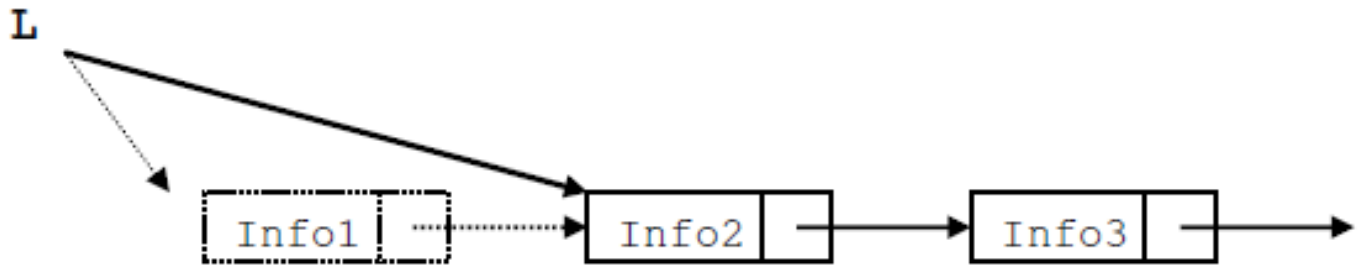
Função que retira um elemento

- A função tem como parâmetros de entrada, a lista e o valor do elemento que deseja ser retirado
- E deve atualizar o valor da lista, pois se o elemento removido for o primeiro da lista, o valor da lista deve ser atualizado
- A função é mais complexa. Se o elemento for o primeiro, deve se fazer o novo valor da lista passar a ser o ponteiro para o segundo elemento
- Daí, pode se liberar o espaço alocado para o elemento que se quer retirar
- Se o elemento estiver no meio da lista deve-se fazer um jogo com ponteiros, deve-se fazer o elemento anterior ao que vai ser removido passar a apontar para o seguinte e daí pode se liberar o que se quer retirar
- No segundo caso, precisamos do ponteiro para o elemento anterior a fim de acertar o encadeamento

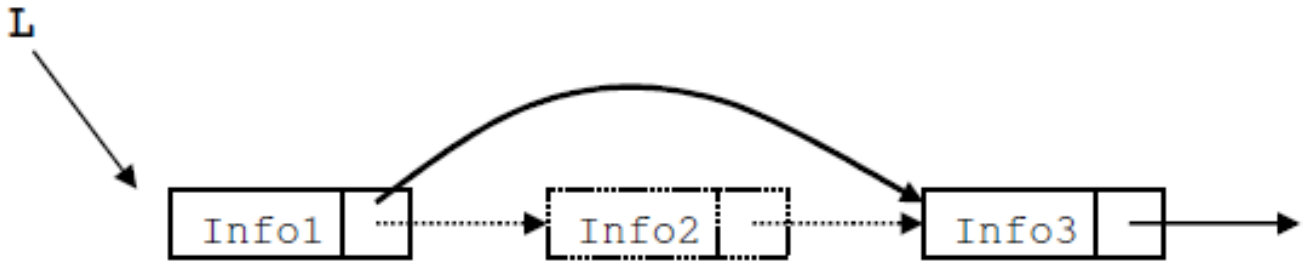


Função que retira um elemento

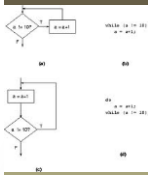
- Recebe como entrada a lista e o valor do elemento a retirar
- Atualiza o valor da lista, se o elemento removido for o primeiro



- Caso contrário, apenas remove o elemento da lista

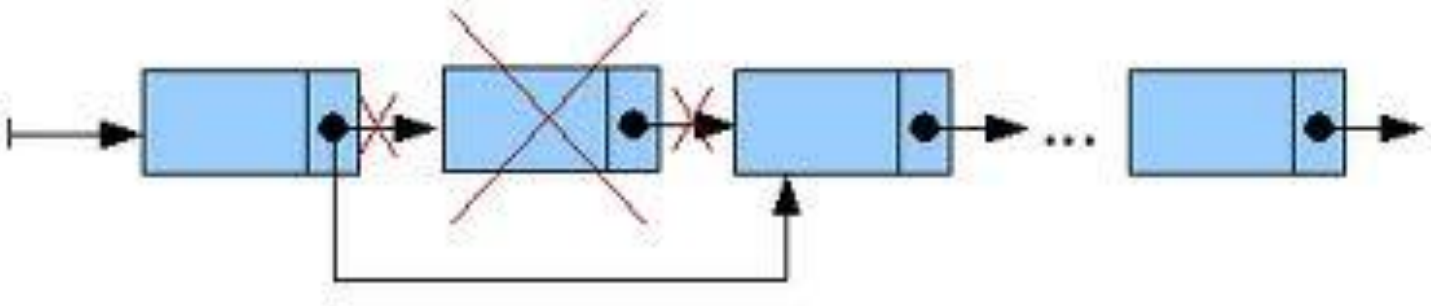


```c
//funcao retira: retira elemento da lista
Lista* lst_retira(Lista* l, int v){
    Lista* ant = NULL;  //ponteiro para elemento anterior
    Lista* p = l;       //ponteiro para percorrer a lista

    //procura elemento na lista, guardando o anterior
    while (p!=NULL && p->info != v){
        ant = p;
        p = p->prox;
    }

    //verifica se achou o elemento
    if(p == NULL)
        return l;   //nao achou: retorna lista original

    //retira o elemento
    if(ant == NULL){
        //retira o elemento do inicio
        l = p->prox;
    }
    else{
        //retira o elemento do meio da lista
        ant->prox = p->prox;
    }
    free(p);
    return l;
}
```



Função que retira um elemento

- O caso de excluir o último elemento recai no caso de excluir um elemento no meio da lista.

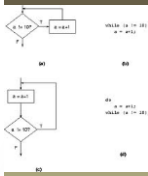


Função para liberar a lista

- Destrói a lista, liberando todos os elementos alocados

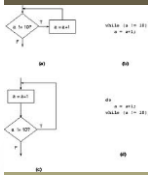
```
void lst_libera (Lista* l){
    Lista* p = l;
    while(p!=NULL){
        Lista* t = p->prox; //guarda referencia ao proximo elemento
        free(p);            //libera a memoria apontada por p
        p = t;              //faz p apontar para o proximo
    }
}
```

- Importante: devemos guardar referencia para o próximo elemento antes de liberar o atual (se liberássemos o elemento e depois tentássemos acessar o encadeamento, estaríamos acessando um espaço de memória que não estaria mais reservado para o uso)



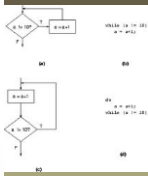
Exemplo de programa

```c
int main(void){
    Lista* l;                  //declara uma lista nao atualizada
    l = lst_cria();            //cria e inicializa lista como vazia
    l = lst_insere(l, 23);     //insere na lista o elemento 23
    l = lst_insere(l, 45);     //insere na lista o elemento 45
    l = lst_insere(l, 32);     //insere na lista o elemento 32
    l = lst_insere(l, 48);     //insere na lista o elemento 32
//  ...
    lst_imprime(l);
    l = lst_retira(l, 48);
    printf("\n");
    lst_imprime(l);
    lst_libera(l);
    return 0;
}
```



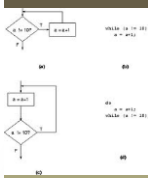
Exercício

- Alterar o programa anterior para permitir que o usuário continue inserindo dados na lista (exibir um menu para inserir elemento e imprimir a lista)
- Apresentar o MENU com as opções
- Alterar a lista para em vez de inteiros, trabalhar com números reais. (em um novo programa)
- PARA SEXTA-FEIRA(23/06)!
- Enviar para o e-mail: edkallenn.lima@uninorteac.edu.br ou para o edkevan@gmail.com com o título:
- [ED-2017-1-TRAB-LISTA1]NomeSobrenome



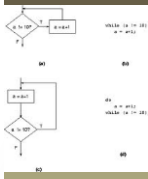
```c
/*
Função : Exemplo simples de listas encadeadas
Autor : Edkallenn - Data : 06/04/2012
Observações:                                    */
#include <stdio.h>
#include <stdlib.h>
#define QL printf("\n")
//definicao
struct lista{
    int info;
    struct lista* prox;
};
//sinonimo
typedef struct lista Lista;
//prototipos
Lista* lst_cria(void);
Lista* lst_insere(Lista* l, int i);
void lst_imprime(Lista* l);
int lst_vazia(Lista* l);
Lista* lst_busca(Lista* l, int v);
Lista* lst_retira(Lista* l, int v);
void lst_libera (Lista* l);
```

```c
main(){
    Lista *l_nova;
    l_nova = lst_cria();
    int opcao, num;

    do{
        system("cls");
        printf("Programa de Lista Encadeada"); QL;
        printf("\t OPCOES \t"); QL;
        printf("1 - Inserir elementos na lista"); QL;
        printf("2 - Exibir elementos"); QL;
        printf("3 - Remove elementos"); QL;
        printf("5 - Sair");QL; QL;
        do{
            scanf("%d", &opcao);
        }while ((opcao!=1)&&(opcao!=2)&&(opcao!=3)&&(opcao!=5));
        if(opcao==1){
            QL; printf("Digite um numero para inserir na lista: ");
            scanf("%d", &num);
            l_nova = lst_insere(l_nova, num);
        }
        if(opcao==2){
            QL; printf("Lista Encadeada digitada: ");QL;
            lst_imprime(l_nova);
            system("pause");
        }
        if(opcao==3){
            QL; printf("Digite um numero a ser removido da lista: ");
            scanf("%d", &num);
            l_nova = lst_retira(l_nova, num); lst_imprime(l_nova);
            system("pause");
        }
    }while(opcao!=5);
  lst_libera(l_nova);
  getchar();
}
```



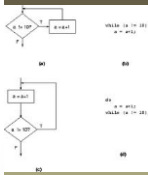
Vantagens da Lista encadeada (ligada)

- A inserção ou remoção de um elemento na lista não implica a mudança de lugar de outros elementos;
- Não é necessário definir, no momento da criação da lista, o número máximo de elementos que esta poderá ter.
- Ou seja, é possível alocar memória "dinamicamente", apenas para o número de nós necessários



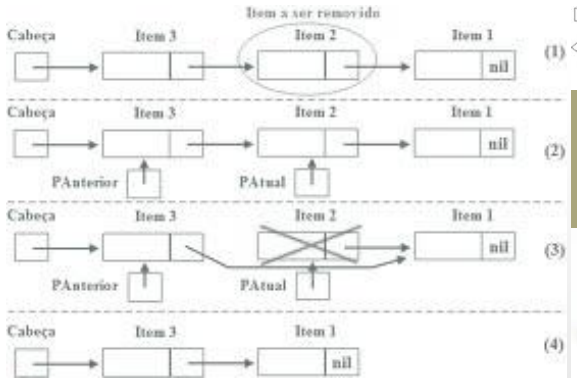
Desvantagens da Lista Encadeada

- A manipulação torna-se mais "perigosa" uma vez que, se o encadeamento (ligação) entre elementos da lista for mal feito, toda a lista pode ser perdida;
- Para **aceder** ao elemento na posição ***n*** da lista, deve-se percorrer os ***n-1*** anteriores.



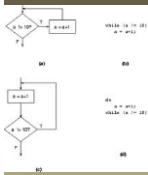
Manutenção da lista ordenada

- A função de inserção que vimos anteriormente armazena os elementos na lista na ordem inversa à ordem de inserção, pois um novo elemento é sempre inserido no início da lista
- Se quisermos manter os elementos na lista em determinada ordem, temos de encontrar primeiro a posição correta para inserir o novo elemento.



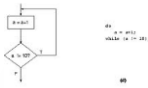
Lista ordenada

- Essa função não é eficiente, pois temos de percorrer a lista, elemento por elemento, para achar a posição de inserção.
- Se a ordem não for relevante, opta-se por fazer inserções no início, pois o custo computacional dessa operação independe do número de elementos na lista
- Entretanto, se desejarmos manter os elementos em ordem, cada novo elemento deve ser inserido na ordem correta.



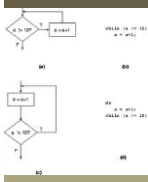
Lista ordenada

- Como exemplo, vamos considerar que queremos manter nossa lista de números inteiros em ordem crescente.
- A função tem a mesma assinatura da função anterior, mas percorre os elementos da lista a fim de encontrar a posição correta para o novo elemento.
- Temos de saber, portanto, inserir um elemento no meio da lista



Lista ordenada

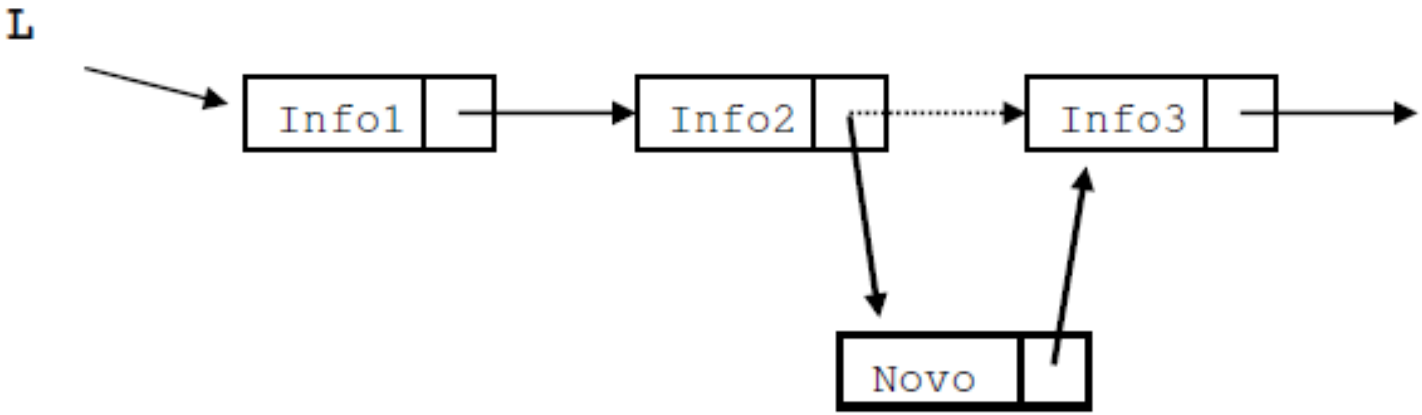
- Devemos localizar o elemento da lista que precederá o elemento novo a ser inserido.
- De posse do ponteiro para este elemento, podemos encadear o novo elemento da lista
- Ele apontará para o próximo elemento na lista, e o elemento precedente apontará para o novo



```c
/*funcao insere_ordenado: insere o elemento em ordem*/
Lista* lst_insere_ordenado(Lista* l, int v){
    Lista* novo;
    Lista* ant = NULL;  //ponteiro para elemento anterior
    Lista* p = l;       //ponteiro para percorrer a lista

    //procura posicao de insercao
    while(p != NULL && p->info < v){
        ant = p;
        p = p->prox;
    }

    //cria o novo elemento
    novo = (Lista*) malloc(sizeof(Lista));
    novo->info = v;

    //encadeia elemento
    if(ant == NULL){    //insere o elemento no inicio
        novo->prox = l;
        l = novo;
    }
    else{               //insere elemento no meio da lista
        novo->prox = ant->prox;
        ant->prox = novo;
    }
    return l;
}
```

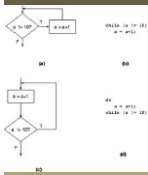
Lista ordenada

- Função de inserção percorre os elementos da lista até encontrar a posição correta para a inserção do novo



Exercício

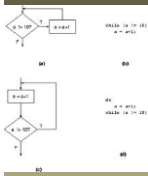
- Alterar o programa anterior para permitir a inserção em ordem crescente e decrescente.



```
175  Lista* lst_insere_ordenado_dec(Lista* l, int v){
176      Lista* novo;
177      Lista* ant = NULL;  //ponteiro para elemento anterior
178      Lista* p = l;       //ponteiro para percorrer a lista
179
180      //procura posicao de insercao
181      while(p != NULL && p->info > v){  //percorre naturalmente
182          ant = p;
183          p = p->prox;
184      }
185
186      //cria o novo elemento
187      novo = (Lista*) malloc(sizeof(Lista));
188      novo->info = v;
189
190      //encadeia elemento
191      if(ant == NULL){    //insere o elemento no inicio
192          novo->prox = l;
193          l = novo;
194      }
195      else{               //insere elemento no meio da lista
196          novo->prox = ant->prox;
197          ant->prox = novo;
198      }
199      return l;
200  }
201
```

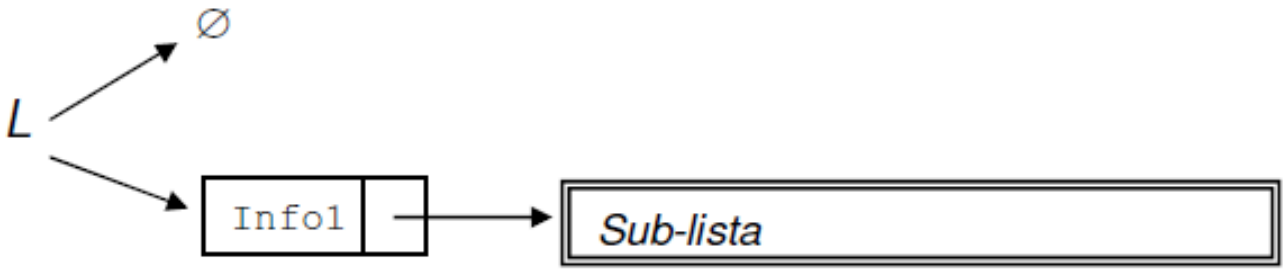
Implementações recursivas

- Uma lista pode ser definida de maneira recursiva
- Podemos dizer que uma lista encadeada é representada por:
  - uma lista vazia; ou
  - um elemento, seguido de uma (sub)lista
- Neste último caso, o segundo elemento da lista representa o primeiro elemento da sublista.



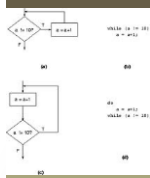
Lista recursiva

- Definição:
  - uma lista vazia; ou
  - um elemento seguido de uma (sub-)lista



Lista recursiva

- Com base nesta definição recursiva, pode-se implementar as funções de lista recursivamente.
- Por exemplo, a função para imprimir os elementos de uma lista
- Devemos seguir a definição recursiva para implementar a função
- Primeiro, deve-se verificar se a lista é vazia.
- Se for, nada a imprimir
- Caso contrário, a lista é composta pelo primeiro nó, dado por **`l`**, e por uma sublista, dada por **`l->prox`**
- Assim, imprime-se a informação associada ao primeiro nó, acessando **`l->info`** e imprimir as informações da sublista
- Para imprimir a sublista usa-se a própria função



Lista recursiva

```c
//Funcao imprime recursividade
void lst_imprime_rec(Lista* l){
    if(lst_vazia(l))
        return;
    else{
        //imprime o primeiro elemento
        printf("info = %d\n", l->info);
        //imprime a sublista
        lst_imprime_rec(l->prox);
    }
}
```

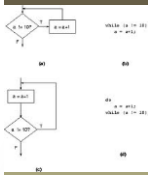
- É fácil perceber que se invertermos o teste ela pode ficar mais compacta

```c
//imprime recursiva
void lst_imprime_rec2 (Lista* l){
    if(!lst_vazia(l)){
        //imprime o primeiro elemento
        printf("info = %d\n", l->info);
        //imprime a sublista
        lst_imprime_rec2(l->prox);
    }
}
```

```c
main(){
    Lista *l_nova;
    l_nova = lst_cria();
    int opcao, num;

    do{
        system("cls");
        printf("Programa de Lista Encadeada"); QL;
        printf("\t OPCOES \t"); QL;
        printf("1 - Inserir elementos na lista"); QL;
        printf("2 - Exibir elementos"); QL;
        printf("3 - Remove elementos"); QL;
        printf("5 - Sair");QL; QL;
        do{
            scanf("%d", &opcao);
        }while ((opcao!=1)&&(opcao!=2)&&(opcao!=3)&&(opcao!=5));
        if(opcao==1){
            QL; printf("Digite um numero para inserir na lista: ");
            scanf("%d", &num);
            l_nova = lst_insere_ordenado(l_nova, num);
        }
        if(opcao==2){
            QL; printf("Lista Encadeada digitada: ");QL;
            lst_imprime_rec(l_nova);
            system("pause");
        }
        if(opcao==3){
            QL; printf("Digite um numero a ser removido da lista: ");
            scanf("%d", &num);
            l_nova = lst_retira(l_nova, num); QL; lst_imprime_rec2(l_nova);
            system("pause");
        }
    }while(opcao!=5);
  lst_libera(l_nova);
  getchar();
}
```

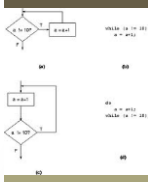
Importante

- NÃO é recomendado tentar seguir, passo a passo, a execução de uma implementação recursiva e sim ENTENDÊ-LA com base apenas na definição recursiva do objeto em questão – no caso, a lista encadeada
- É fácil alterar esse código para obter a impressão dos elementos da lista em ordem inversa: basta inverter a ordem das chamadas às funções **`printf`** e **`imprime_rec`**



Função retira recursiva

- A função para retirar o elemento da lista também pode ser escrita de forma recursiva.
- Nesse caso só retira-se o elemento da lista se ele for o primeiro da lista (ou da sublista).
- Se o elemento não for o primeiro, chama-se a função recursivamente para retirar o elemento da sublista



Retira Recursiva

```c
/* Função retira recursiva */
Lista* lst_retira_rec (Lista* l, int v){
    if (!lst_vazia(l)) {
    /* verifica se elemento a ser retirado é o primeiro */
        if (l->info == v) {
            Lista* t = l; /* temporário para poder liberar */
            l = l->prox;
            free(t);
        }
        else {
            /* retira de sub-lista */
            l->prox = lst_retira_rec(l->prox,v);
        }
    }
    return l;
}
```

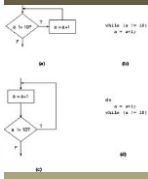
- Salienta-se apenas a necessidade de reatribuir o valor de **`l->prox`** na chamada recursiva, já que a função pode alterar o valor da sub-lista

```c
main(){
    Lista *l_nova;
    l_nova = lst_cria();
    int opcao, num;

    do{
        system("cls");
        printf("Programa de Lista Encadeada"); QL;
        printf("\t OPCOES \t"); QL;
        printf("1 - Inserir elementos na lista"); QL;
        printf("2 - Exibir elementos"); QL;
        printf("3 - Remove elementos"); QL;
        printf("5 - Sair");QL; QL;
        do{
            scanf("%d", &opcao);
        }while ((opcao!=1)&&(opcao!=2)&&(opcao!=3)&&(opcao!=5));
        if(opcao==1){
            QL; printf("Digite um numero para inserir na lista: ");
            scanf("%d", &num);
            l_nova = lst_insere_ordenado(l_nova, num);
        }
        if(opcao==2){
            QL; printf("Lista Encadeada digitada: ");QL;
            lst_imprime_rec(l_nova);
            system("pause");
        }
        if(opcao==3){
            QL; printf("Digite um numero a ser removido da lista: ");
            scanf("%d", &num);
            l_nova = lst_retira_rec(l_nova, num); QL; lst_imprime_rec2(l_nova);
            system("pause");
        }
    }while(opcao!=5);
  lst_libera(l_nova);
  getchar();
}
```

Função liberar recursiva

- A função para liberar uma lista também pode ser reescrita recursivamente, de forma muito simples.
- Aqui, se a lista não for vazia, libera-se primeiro a sublista e depois libera-se a lista

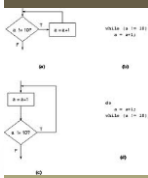
```
void lst_libera_rec(Lista* l){
    if(!lst_vazia(l)){
        lst_libera_rec(l->prox);
        free(l);
    }
}
```



```
27  main(){
28      Lista *l_nova;
29      l_nova = lst_cria();
30      int opcao, num;
31
32      do{
33          system("cls");
34          printf("Programa de Lista Encadeada"); QL;
35          printf("\t OPCOES \t"); QL;
36          printf("1 - Inserir elementos na lista"); QL;
37          printf("2 - Exibir elementos"); QL;
38          printf("3 - Remove elementos"); QL;
39          printf("5 - Sair");QL; QL;
40          do{
41              scanf("%d", &opcao);
42          }while ((opcao!=1)&&(opcao!=2)&&(opcao!=3)&&(opcao!=5));
43          if(opcao==1){
44              QL; printf("Digite um numero para inserir na lista: ");
45              scanf("%d", &num);
46              l_nova = lst_insere_ordenado(l_nova, num);
47          }
48          if(opcao==2){
49              QL; printf("Lista Encadeada digitada: ");QL;
50              lst_imprime_rec(l_nova);
51              system("pause");
52          }
53          if(opcao==3){
54              QL; printf("Digite um numero a ser removido da lista: ");
55              scanf("%d", &num);
56              l_nova = lst_retira_rec(l_nova, num); QL; lst_imprime_rec2(l_nova);
57              system("pause");
58          }
59      }while(opcao!=5);
60      lst_libera_rec(l_nova);
61      getchar();
62  }
```

Função para comparar listas

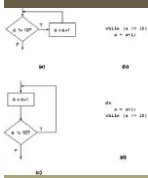
- A utilização de implementações recursivas para listas encadeadas é uma questão de opção.
- Em geral, a implementação não recursiva é mais eficiente do ponto de vista do esforço computacional dispensado, pois minimiza o número de chamadas de funções
- No entanto, algumas implementações podem ficar mais simples se feitas de forma recursiva



Comparar Listas

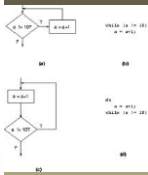
- Como exemplo disto, consideremos a implementação de uma função para testar se duas listas dadas são iguais
- Duas listas são iguais se têm a mesma sequência de elementos, naturalmente, com o mesmo número de elementos.
- O protótipo dessa função é dado por:

```
int lst_igual(Lista* l1, Lista* l2);
```



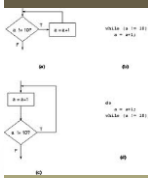
Comparar – não recursiva

- Percorre as duas listas, usando dois ponteiros auxiliares:
  - se duas informações forem diferentes, as listas são diferentes
- Ao terminar uma das listas (ou as duas):
  - se os dois ponteiros auxiliares são NULL, as duas listas têm o mesmo número de elementos e são iguais



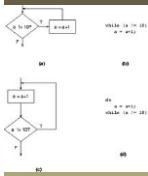
Compara Listas – não recursiva

```
int lst_igual (Lista* l1, Lista* l2){
    Lista* p1; /* ponteiro para percorrer l1 */
    Lista* p2; /* ponteiro para percorrer l2 */
    for (p1=l1, p2=l2; p1 != NULL && p2 != NULL; p1 = p1->prox, p2 = p2->prox)
    {
        if (p1->info != p2->info)
            return 0;
    }
    return p1==p2;
}
```



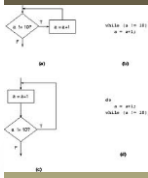
Comparar listas - recursiva

- Uma implementação recursiva deve ser pensada com base na definição recursiva de lista.
- Primeiramente testa-se os casos base, nos quais as listas podem ser vazias
- Assim verifica-se se as duas listas dadas são vazias. Se forem, são iguais, logicamente.
- Se não forem, deve-se verificar se uma delas é vazia
- Se for, conclui-se que trata-se de listas diferentes.
- Se ambas não forem vazias testa-se a igualdade entre as informações associadas aos primeiros nós das listas e verifica-se a igualdade das sublistas



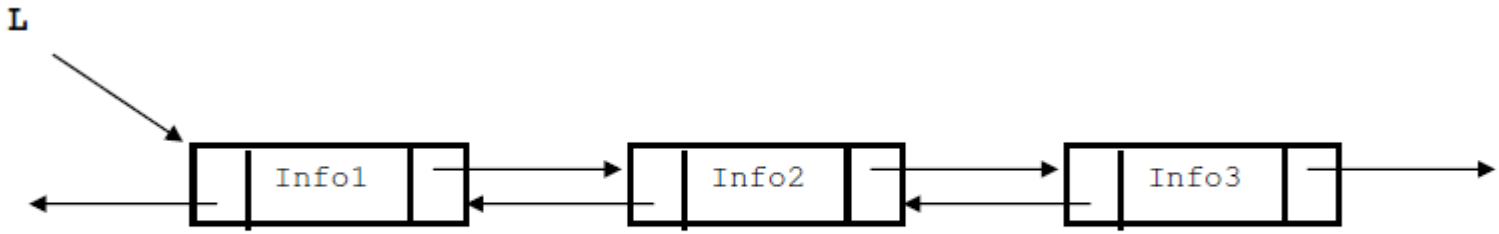
Compara recursiva

```
//compara recursiva
int lst_igual_rec (Lista* l1, Lista* l2){
    if (l1 == NULL && l2 == NULL)
        return 1;
    else if (l1 == NULL || l2 == NULL)
            return 0;
        else
            return l1->info == l2->info && lst_igual(l1->prox, l2->prox);
}
```



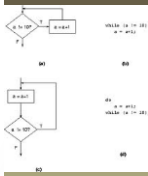
Listas Duplamente Encadeadas

- Definição de Lista Duplamente Encadeada:
  - cada elemento tem um ponteiro para o próximo elemento e um ponteiro para o elemento anterior
  - dado um elemento, é possível acessar o próximo e o anterior
  - dado um ponteiro para o último elemento da lista, é possível percorrer a lista em ordem inversa



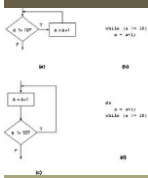
Listas Duplamente encadeadas

- Exemplo:
  - lista encadeada armazenando valores inteiros
  - estrutura **`lista2`**
    - estrutura dos nós da lista
  - tipo **`Lista2`**
    - tipo dos nós da lista



Estrutura – definição e tipo

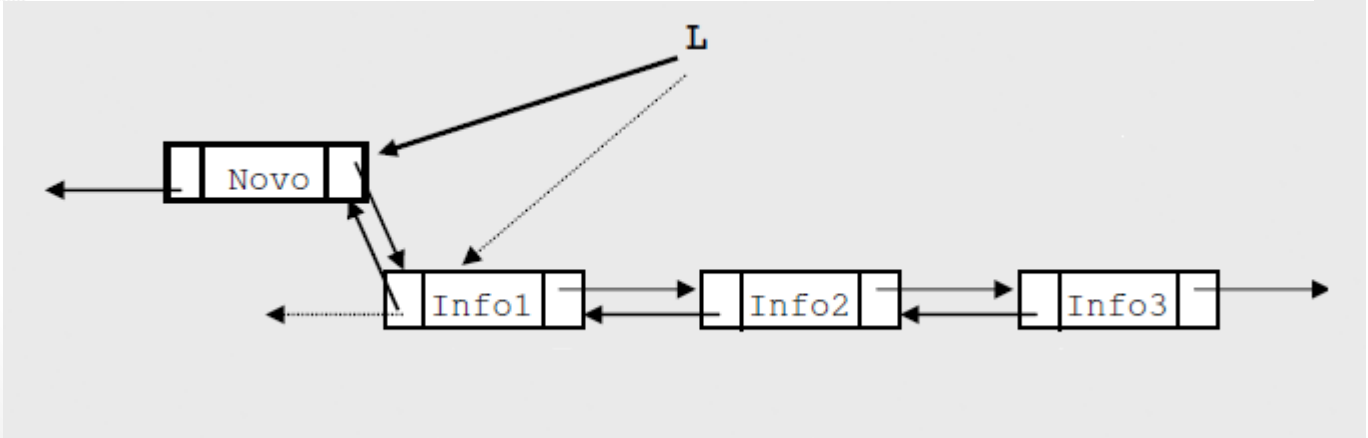
```
1  /* Função : Listas duplamente encadeadas
2  / Autor : Edkallenn
3  / Data : 06/04/2012
4  / Observações:
5  */
6  #include <stdio.h>
7  #include <stdlib.h>
8  #define QL printf("\n")
9
10 //definicao
11 struct lista2 {
12     int info;
13     struct lista2* ant;
14     struct lista2* prox;
15 };
16 //sinônimo
17 typedef struct lista2 Lista2;
```



Listas duplamente encadeadas

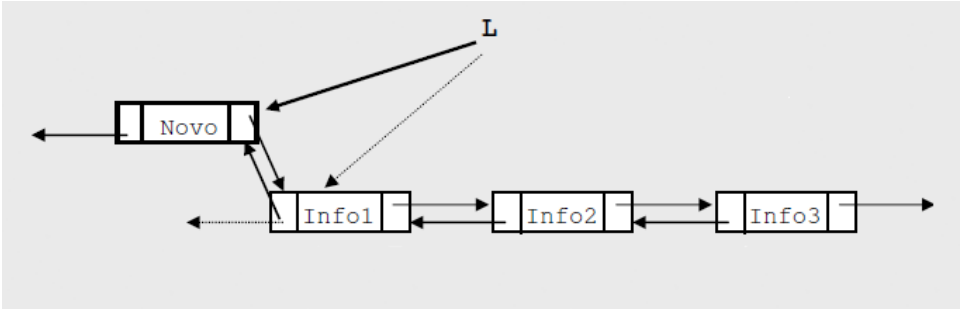
- Função de inserção (no início da lista)

```
/* inserção no início: retorna a lista atualizada */
Lista2* lst2_insere (Lista2* l, int v){
    Lista2* novo = (Lista2*) malloc(sizeof(Lista2));
    novo->info = v;
    novo->prox = l;
    novo->ant = NULL;
    /* verifica se lista não estava vazia */
    if (l != NULL)
        l->ant = novo;
    return novo;
}
```



Função de Inserção

- Ela insere no início da lista. Após a alocação do novo elemento ela acerta o encadeamento
- O novo elemento é encadeado no início da lista
- Assim, ele tem como próximo elemento o antigo primeiro elemento da lista e como anterior o valor NULL
- Depois a função testa se a lista não era vazia, pois nesse caso, o elemento anterior do então primeiro elemento passa a ser o novo elemento
- De qualquer modo, o novo elemento passa a ser o primeiro da lista e deve ser retornado como valor da lista atualizada



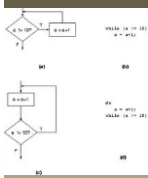
Função de busca

- Recebe a informação referente ao elemento a pesquisar
- Retorna o ponteiro do nó da lista que representa o elemento, ou NULL, caso o elemento não seja encontrado na lista
- Implementação idêntica à lista encadeada (simples)

```
/* função busca: busca um elemento na lista */
Lista2* lst2_busca (Lista2* l, int v){
    Lista2* p;
    for (p=l; p!=NULL; p=p->prox)
        if (p->info == v)
            return p;
    return NULL; /* não achou o elemento */
}
```

Função para retirar um elemento da lista

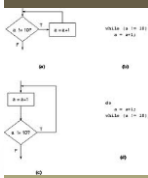
- p aponta para o elemento a retirar
- se p aponta para um elemento no meio da lista:
  - o anterior passa a apontar para o próximo:
    `p->ant->prox = p->prox;`
  - o próximo passa a apontar para o anterior:
    `p->prox->ant = p->a;`
- se p aponta para o último elemento:
  - não é possível escrever
    `p->prox->ant`, pois `p->prox` é `NULL`
- se p aponta para o primeiro elemento:
  - não é possível escrever `p->ant->prox`, pois `p->ant` é `NULL`
  - é necessário atualizar o valor da lista, pois o primeiro elemento será removido



```
49  /* função retira: remove elemento da lista */
50  Lista2* lst2_retira (Lista2* l, int v) {
51
52      Lista2* p = lst2_busca(l,v);
53
54      if (p == NULL)
55          return l; /* não achou o elemento: retorna lista inalterada */
56
57      /* retira elemento do encadeamento */
58      if (l == p) /* testa se é o primeiro elemento */
59          l = p->prox;
60      else
61          p->ant->prox = p->prox;
62
63      if (p->prox != NULL) /* testa se é o último elemento */
64      p->prox->ant = p->ant;
65
66      free(p);
67
68      return l;
69  }
```

Protótipos

```
19  //protótipos
20  Lista2* lst2_cria(void);
21  Lista2* lst2_insere (Lista2* l, int v);
22  Lista2* lst2_busca (Lista2* l, int v);
23  Lista2* lst2_retira (Lista2* l, int v);
24  void lst2_imprime(Lista2* l);
25  int lst2_vazia(Lista2* l);
26  void lst2_libera (Lista2* l);
27
```



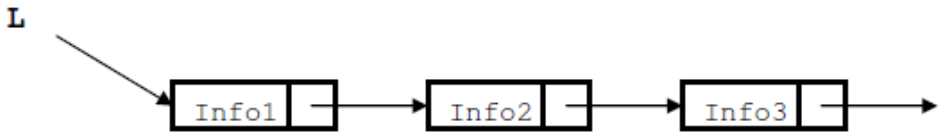
```c

//Funcao de Criacao - retorna uma lista vazia
Lista2* lst2_cria(void){
    return NULL;
}

//funcao imprime: imprime valores dos elementos
void lst2_imprime(Lista2* l){
    Lista2* p;   //variavel auxiliar para percorrer a lista
    for(p = l; p !=NULL; p = p->prox)
        printf("info = %d\n", p->info);
}

//funcao vazia: retorna 1 se vazia ou 0 se nao vazia
int lst2_vazia(Lista2* l){
    return (l == NULL);
}

void lst2_libera (Lista2* l){
    Lista2* p = l;
    while(p!=NULL){
        Lista2* t = p->prox; //guarda referencia ao proximo elemento
        free(p);             //libera a memoria apontada por p
        p = t;               //faz p apontar para o proximo
    }
}
```

Programa para testas as listas encadeadas

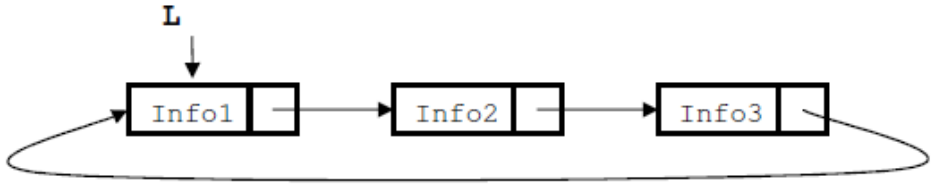
```
 96  int main(void){
 97      Lista2* lista2;                 //declara uma lista nao atualizada
 98      lista2 = lst2_cria();           //cria e inicializa lista como vazia
 99      lista2 = lst2_insere(lista2, 23);  //insere na lista o elemento 23
100      lista2 = lst2_insere(lista2, 45);  //insere na lista o elemento 45
101      lista2 = lst2_insere(lista2, 32);  //insere na lista o elemento 32
102      lista2 = lst2_insere(lista2, 48);  //insere na lista o elemento 32
103
104      lst2_imprime(lista2);
105      QL;
106      lista2 = lst2_retira(lista2, 32);
107      QL;
108      lst2_imprime(lista2);
109      lst2_libera(lista2);
110      return 0;
111  }
```

Resumo

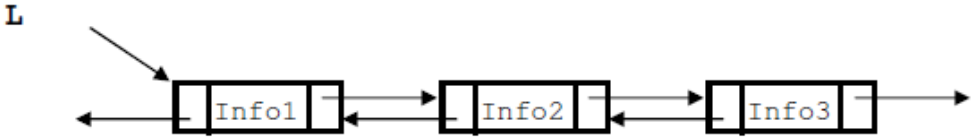
Listas encadeadas



Listas circulares

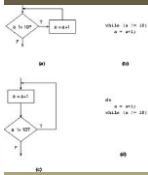


Listas duplamente encadeadas

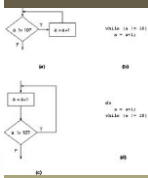


Lista encadeada

- Exemplo de pedidos simples.

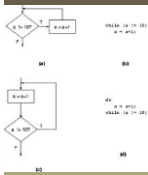


```c
/* Função : Exemplo de Lista - programa pedidos simples
/ Autor : Edkallenn
/ Data : 06/04/2012
/ Observações:
*/
#include <stdio.h>
#include <stdlib.h>
/* caso haja problemas com touppeer e getch - retira coment
#include <ctype.h>
#include <conio.h>
*/
typedef struct lista
{
  int cli;
  float val;
  struct lista *prox;
} lista;

lista * plista;

int cliente;
float valor;

char opcao_menu()
{
  system("cls");
  printf(" (I)ncluir pedido\n");
  printf(" (E)xcluir pedido\n");
  printf(" (L)istar pedidos\n");
  printf(" (F)im\n");
  printf("> ");
  return (toupper(getche()));
}
```



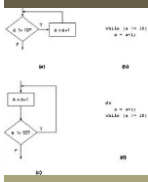
Ler pedido

```
34
35    void ler_pedido()
36    {
37       printf("\nNumero do cliente ..... ");
38       scanf("%d",&cliente);
39       printf("Valor do pedido ...... ");
40       scanf("%f",&valor);
41    }
42
```

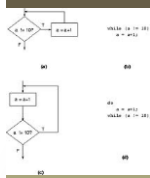


```c
void incluir_pedido()
{
  lista *p,*q,*r;

  ler_pedido();

  p = (lista *)calloc(1,sizeof(lista));

  p->cli = cliente;
  p->val = valor;
  p->prox = NULL;

  /* Incluir pedido na lista */
  if (plista == NULL)
    plista = p;
  else
  {
    r = NULL;
    q = plista;
    while ((q != NULL) && (q->val > valor))
    {
      r = q;
      q = q->prox;
    }
    if (r == NULL)
      plista = p;
    else
      r->prox = p;
    p->prox = q;
  }
}
```

```
74
75  void excluir_pedido()
76  {
77    lista *p,*q;
78
79    ler_pedido();
80
81    q = NULL;
82    p = plista;
83    while ((p != NULL) && ((p->cli != cliente) || (p->val != valor)))
84    {
85      q = p;
86      p = p->prox;
87    }
88    if (p == NULL)
89      printf("Esse pedido nao existe!\n");
90    else
91    {
92      if (q != NULL)
93        q->prox = p->prox;
94      else
95        plista = p->prox;
96      free(p);
97      printf("Pedido cancelado!\n");
98    }
99  }
100
```

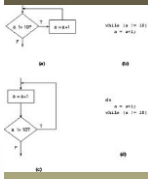


```c
void listar_pedidos()
{
  lista *p;

  p = plista;
  printf("\n  Lista de Pedidos\n");
  printf("--------+-----------\n");
  printf("CLIENTE |   VALOR   \n");
  printf("--------+-----------\n");
  while (p != NULL)
  {
    printf("  %04d  | %10.2f\n",p->cli,p->val);
    p = p->prox;
  }
}

int main(int args, char * arg[])
{
  char op;

  plista = NULL;

  do
  {
    op = opcao_menu();
    switch (op)
    {
      case 'I': incluir_pedido(); break;
      case 'E': excluir_pedido(); break;
      case 'L': listar_pedidos(); break;
    }
    printf("\n");
    system("pause");
  }
  while (op != 'F');
  return 0;
}
```

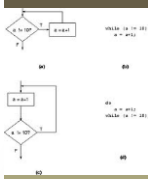


Lista duplamente encadeada

- Problema: Deseja-se escrever um programa capaz de somar números inteiros arbitrariamente grandes. Cada inteiro deve ser representado como uma lista encadeada de seus dígitos.

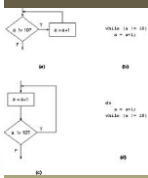


```c
/* Função : Soma de inteiros usando lista encadeada
/ Autor : Edkallenn
/ Data : 06/04/2012
/ Observações:
*/
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <conio.h>

typedef struct lista
{
  int dig;
  struct lista *prox;
  struct lista *prev;
} lista;

lista *ini[3],*fim[3];

char opcao_menu()
{
  system("cls");
  printf(" (L)er numeros\n");
  printf(" (M)ostrar soma\n");
  printf(" (F)im\n");
  printf("> ");
  return (toupper(getche()));
}

```

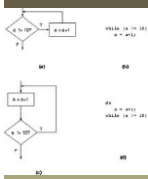


```
30  void zerar_numeros()
31  {
32    int i;
33
34    for (i = 0; i < 3; i++)
35    {
36      ini[i] = NULL;
37      fim[i] = NULL;
38    }
39  }
40
41  void incluir_inicio(int index, int digito)
42  {
43    lista *p;
44
45    p = (lista *)calloc(1,sizeof(lista));
46
47    p->dig = digito;
48    p->prox = NULL;
49    p->prev = NULL;
50
51    if (ini[index] == NULL)
52      fim[index] = p;
53    else
54    {
55      ini[index]->prev = p;
56      p->prox = ini[index];
57    }
58    ini[index] = p;
59  }
60
```

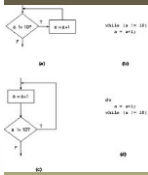
```c
void incluir_final(int index, int digito)
{
  lista *p;

  p = (lista *)calloc(1,sizeof(lista));

  p->dig = digito;
  p->prox = NULL;
  p->prev = NULL;

  if (fim[index] == NULL)
    ini[index] = p;
  else
  {
    fim[index]->prox = p;
    p->prev = fim[index];
  }
  fim[index] = p;
}

void digitos_numero(int n)
{
  int d;

  printf("\nNum%d = ",n+1);
  while (1)
  {
    scanf("%d",&d);
    if (d == -1) break;
    incluir_final(n,d);
  }
}
```



```
 93
 94  void ler_numeros()
 95  {
 96    zerar_numeros();
 97    printf("\nDigitos separados por espacos (-1, para terminar)\n");
 98    digitos_numero(0);
 99    digitos_numero(1);
100  }
101
102  void mostra_numero(char *s, lista *p)
103  {
104    printf("\n%s = ",s);
105    while (p != NULL)
106    {
107      printf("%d",p->dig);
108      p = p->prox;
109    }
110  }
111
```



```c
void somar_numeros()
{
  lista *p1,*p2;
  int soma,vai1;

  ini[2] = fim[2] = NULL;

  p1 = fim[0];
  p2 = fim[1];

  vai1 = 0;
  while ((p1 != NULL) && (p2 != NULL))
  {
    soma = vai1 + (p1->dig + p2->dig);
    vai1 = ((soma >= 10)? 1 : 0);
    soma = soma % 10;
    incluir_inicio(2,soma);
    p1 = p1->prev;
    p2 = p2->prev;
  }
  if (p1 == NULL)
    while (p2 != NULL)
    {
      soma = vai1 + p2->dig;
      vai1 = ((soma >= 10)? 1 : 0);
      soma = soma % 10;
      incluir_inicio(2,soma);
      p2 = p2->prev;
    }
  if (p2 == NULL)
    while (p1 != NULL)
    {
      soma = vai1 + p1->dig;
      vai1 = ((soma >= 10)? 1 : 0);
      soma = soma % 10;
      incluir_inicio(2,soma);
      p1 = p1->prev;
    }
}
```



```
151
152   void mostrar_soma()
153   {
154     mostra_numero("Num1", ini[0]);
155     mostra_numero("Num2", ini[1]);
156     somar_numeros();
157     mostra_numero("Soma", ini[2]);
158   }
159
160   int main(int args, char * arg[])
161   {
162     char op;
163
164     zerar_numeros();
165
166     do
167     {
168       op = opcao_menu();
169       switch (op)
170       {
171         case 'L': ler_numeros(); break;
172         case 'M': mostrar_soma(); break;
173       }
174       printf("\n");
175       system("pause");
176     }
177     while (op != 'F');
178     return 0;
179   }
```

Análise do programa

```
typedef struct lista
{
  int dig;
  struct lista *prox;
  struct lista *prev;
} lista;
```

A estrutura de dados para a representação de um número é uma lista com **2 ponteiros**: próxima célula (**prox**) e célula anterior (**prev**).

```
void mostra_numero(char *s, lista *p)
{
  printf("\n%s = ",s);
  while (p != NULL)
  {
    printf("%d",p->dig);
    p = p->prox;
  }
}
```

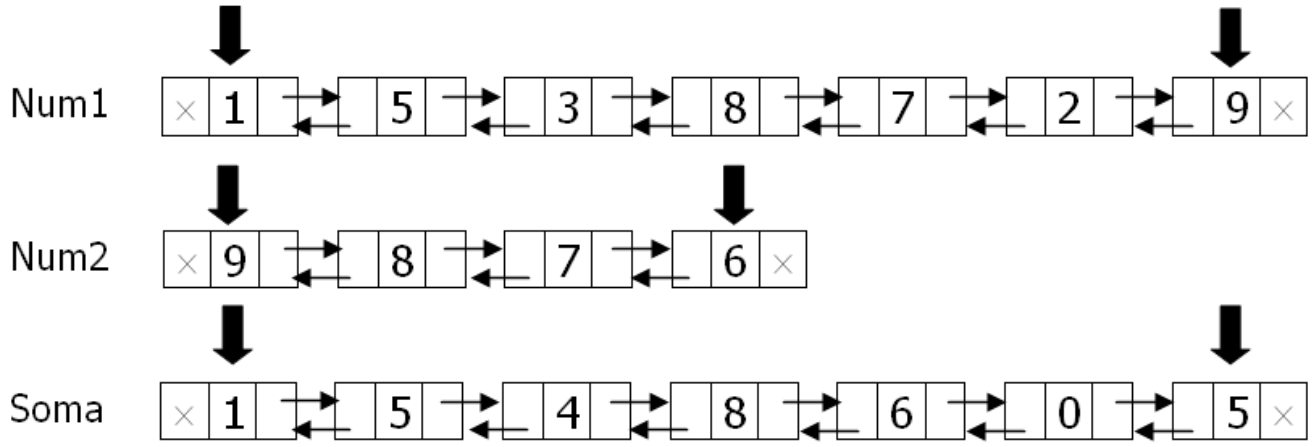
Para mostrar um número é melhor percorrer a lista pelo ponteiro **prox**.

```
void somar_numeros()
{
  lista *p1,*p2;
  int soma,vai1;

  ini[2] = fim[2] = NULL;

  p1 = fim[0];
  p2 = fim[1];

  vai1 = 0;
  while ((p1 != NULL) && (p2 != NULL))
  {
    soma = vai1 + (p1->dig + p2->dig);
    vai1 = ((soma >= 10)? 1 : 0);
    soma = soma % 10;
    incluir_inicio(2,soma);
    p1 = p1->prev;
    p2 = p2->prev;
  }
  ...
}
```

Para somar os números é melhor percorrer as listas pelo ponteiro **prev**.

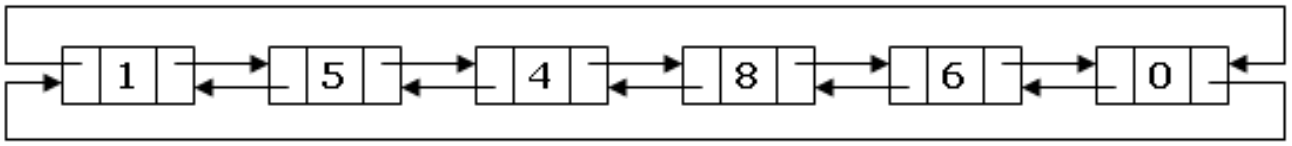
Análise

- As listas duplamente encadeadas são interessantes quando é necessário percorrer a lista em ambos os sentidos.
- No caso dos inteiros arbitrariamente grandes, para a leitura dos dígitos é interessante percorrer a lista de dígitos do início para o fim, mas para a soma, é interessante percorrer a lista de dígitos do fim para o início.



Análise

- As operações de inserção e de exclusão também podem ser implementadas facilmente em listas duplamente encadeadas ordenadas.
- Nestas listas é preciso usar ponteiros para indicar o início e o final da lista.
- Um caso especial de lista duplamente encadeada é a lista circular.
- Numa lista circular, o ponteiro prox do "último" elemento aponta para o "primeiro" elemento da lista e o ponteiro prev do "primeiro" elemento aponta para o "último" elemento da lista.



Pilhas

- O que pensamos quando ouvimos falar em pilhas na TI?
- Isso?

Na verdade:

- Essa é a

Pilhas

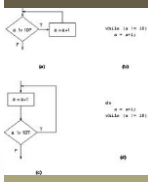
- Uma pilha é uma das várias estruturas de dados dinâmicas que admitem remoção de elementos e inserção de novos elementos. Mais especificamente, uma **pilha** (= *stack*) é uma estrutura sujeita à seguinte regra de operação: sempre que houver uma remoção, o elemento removido é o que está na estrutura há menos tempo.

Pilhas

- Em outras palavras, o primeiro objeto a ser inserido na pilha é o último a ser removido. Essa política é conhecida pela sigla LIFO (= *Last-In-First-Out*).

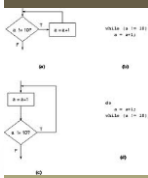
Pilhas - comentários

- Uma das estruturas de dados mais simples é a pilha.
- Possivelmente por essa razão, é a estrutura de dados mais utilizada em programação, sendo inclusive implementada diretamente pelo *hardware* da maioria das máquinas modernas.
- A ideia fundamental da pilha é que todo o acesso a seus elementos é feito através do seu topo. Assim, quando um elemento novo é introduzido na pilha, passa a ser o elemento do topo, e o único elemento que pode ser removido da pilha é o do topo.



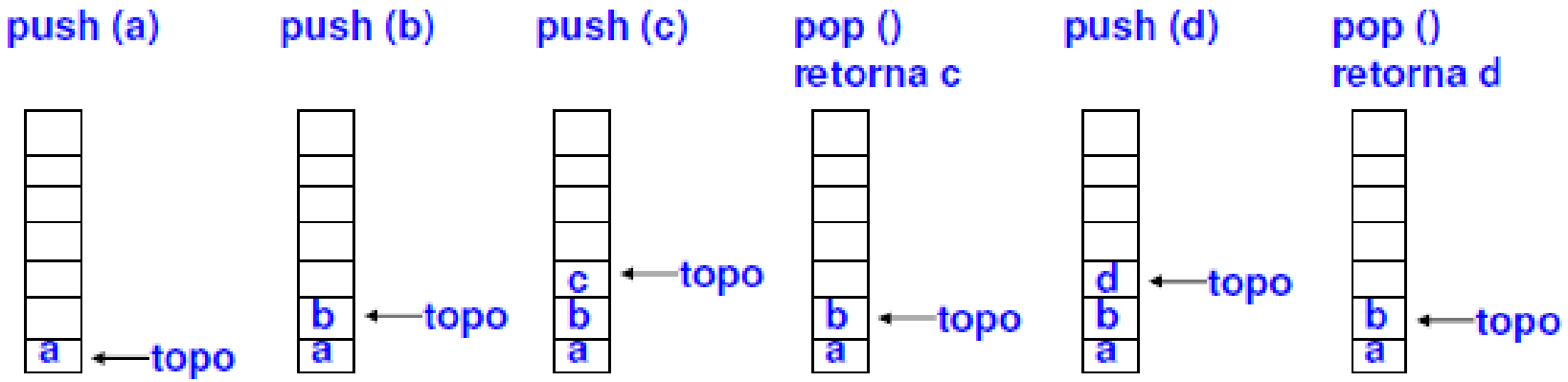
Analogia

- Para entendermos o funcionamento de uma estrutura de pilha, podemos fazer uma analogia com uma pilha de pratos.
- Se quisermos adicionar um prato na pilha, o colocamos no topo.
- Para pegar um prato da pilha, retiramos o do topo. Assim, temos que retirar o prato do topo para ter acesso ao próximo prato.
- A estrutura de pilha funciona de maneira análoga. Cada novo elemento é inserido no topo e só temos acesso ao elemento do topo da pilha.



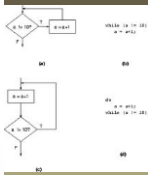
Operações básicas

- Existem duas operações básicas que devem ser implementadas numa estrutura de pilha:
- A operação para **empilhar** um novo elemento, inserindo-o no topo, e a operação para **desempilhar** um elemento, removendo-o do topo.
- É comum nos referirmos a essas duas operações pelos termos em inglês *push* (empilhar) e *pop* (desempilhar).



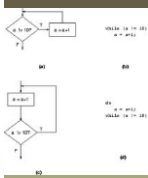
Exemplo

- O exemplo de utilização de pilha mais próximo é a própria pilha de execução da linguagem C.
- As variáveis locais das funções são dispostas numa pilha e uma função só tem acesso às variáveis que estão no topo (não é possível acessar as variáveis da função locais às outras funções).



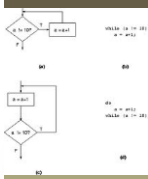
Interface (TAD) Pilha

- Vamos considerar uma interface do tipo pilha usando duas estratégias: vetor e lista encadeada
- Para simplificar, vamos considerar uma pilha que armazena valores reais
- As operações serão as seguintes:
  - Criar uma pilha vazia
  - Inserir um elemento no topo (push)
  - Remover um elemento do topo
  - Verificar se a pilha está vazia
  - Liberar a estrutura da pilha

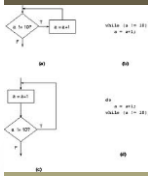


Interface do tipo pilha

- Interface do tipo abstrato Pilha*: pilha.h*
  - função pilha_cria
    - aloca dinamicamente a estrutura da pilha
    - inicializa seus campos e retorna seu ponteiro
  - funções pilha_push e pilha_pop
    - inserem e retiram, respectivamente, um valor real na pilha
  - função pilha_vazia
    - informa se a pilha está ou não vazia
  - função pilha_libera
    - destrói a pilha, liberando toda a memória usada pela estrutura.

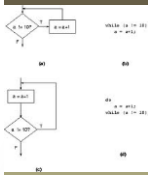


```c
#include <stdio.h>
#include <stdlib.h>

typedef struct pilha Pilha;

//prototipos
Pilha* pilha_cria (void);
void pilha_push (Pilha* p, float v);
float pilha_pop (Pilha* p);
int pilha_vazia (Pilha* p);
void pilha_libera (Pilha* p);
```



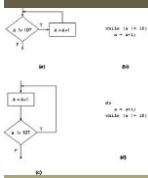
Implementações

- usando um vetor
- usando uma lista encadeada
- simplificação:
  - pilha armazena valores reais



Pilha como vetor

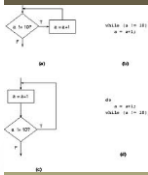
- A implementação com vetor é bastante simples.
- Devemos ter um vetor (vet) para armazenar os elementos da pilha.
- Os elementos inseridos ocupam as primeiras posições do vetor.
- Desta forma, se temos n elementos armazenados na pilha, o elemento vet[n-1] representa o elemento do topo.
- A estrutura que representa o tipo pilha deve, portanto, ser composta pelo vetor e pelo número de elementos armazenados.



```c
/* Função : Pilhas
/ Autor : Edkallenn
/ Data : 06/04/2012
/ Observações:
*/
#include <stdio.h>
#include <stdlib.h>
#define QL printf("\n")

#define MAX 50 /* número máximo de elementos */
struct pilha {
    int n;          /* vet[n]: primeira posição livre do vetor */
    float vet[MAX];   /* vet[n-1]: topo da pilha */
                    /* vet[0] a vet[MAX-1]: posições ocupáveis */
};

typedef struct pilha Pilha;
```

Pilha como vetor

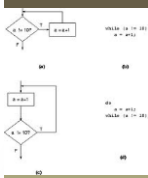
- Implementação de pilha com vetor
- vetor (vet) armazena os elementos da pilha
- elementos inseridos ocupam as primeiras posições do vetor
  - elemento vet[n-1] representa o elemento do topo



Função para criar a pilha

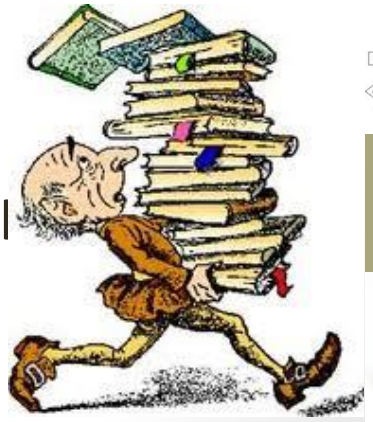
- A função para criar a pilha
  - Aloca dinamicamente essa estrutura e inicializa a pilha como sendo vazia, isto é, com o número de elementos igual a zero.

```
25  Pilha* pilha_cria (void)
26  {
27      Pilha* p = (Pilha*) malloc(sizeof(Pilha));
28      p->n = 0; /* inicializa com zero elementos */
29      return p;
30  }
```



Inserir elemento na pilha

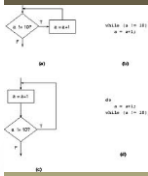
- Para inserir um elemento na pilha, usamos a próxima posição livre do vetor.
- Devemos ainda assegurar que exista espaço para a inserção do novo elemento, tendo em vista que trata-se de um vetor com dimensão fixa.
- Portanto a função função **push**:
  - insere um elemento na pilha
  - usa a próxima posição livre do vetor, se houver



Insere

- Push

```
31
32  void pilha_push (Pilha* p, float v)
33  {
34      if (p->n == MAX) { /* capacidade esgotada */
35          printf("Capacidade da pilha estourou.\n");
36          exit(1); /* aborta programa */
37      }
38      /* insere elemento na próxima posição livre */
39      p->vet[p->n] = v;
40      p->n++; /* equivalente a: p->n = p->n + 1 */
41  }
```



Retira (pop)

- A função **pop** retira o elemento do topo da pilha, fornecendo seu valor como retorno.
- Podemos também verificar se a pilha está ou não vazia.

```
43  float pilha_pop (Pilha* p)
44  {
45      float v;
46      if (pilha_vazia(p)) {
47          printf("Pilha vazia.\n");
48          exit(1);
49      } /* aborta programa */
50      /* retira elemento do topo */
51      v = p->vet[p->n-1];
52      p->n--;
53      return v;
54  }
```

Pilha vazia?

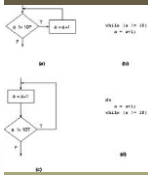
- A função que verifica se a pilha está vazia pode ser dada por:

```
56  int pilha_vazia (Pilha* p)
57  {
58      return (p->n == 0);
59  }
60
```

Libera pilha

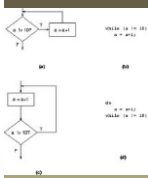
- Finalmente, a função para liberar a memória alocada pela pilha pode ser:

```
61    void pilha_libera (Pilha* p)
62    {
63        free(p);
64    }
65
```



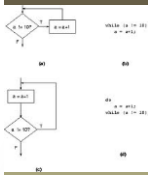
Programa teste

```
73  main(){
74
75      Pilha* nova_pilha;
76
77      nova_pilha = pilha_cria();
78      pilha_push(nova_pilha, 25);
79      pilha_push(nova_pilha, 15);
80      pilha_push(nova_pilha, 40);
81      pilha_push(nova_pilha, 60);
82      pilha_push(nova_pilha, 13.56);
83      printf("Pilha Original: "); QL;
84      pilha_imprime(nova_pilha);  QL;
85      pilha_pop(nova_pilha);
86      printf("Pilha Original apos um pop(): "); QL;
87      pilha_imprime(nova_pilha);
88      pilha_push(nova_pilha, 75);
89      printf("Pilha Original apos um push(): "); QL;
90      pilha_imprime(nova_pilha);
91      pilha_pop(nova_pilha);
92      printf("Pilha Original apos um pop(): "); QL;
93      pilha_imprime(nova_pilha);
94
95      pilha_libera(nova_pilha);
96
97  }
98
```



Exercício

- Apresentar um menu para inserir (push), retirar(pop) elementos na pilha (exibir a pilha a cada operação.



Implementação de pilha como lista linear

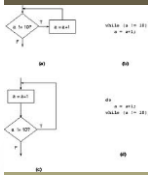
- Elementos da pilha armazenados na lista
- Pilha representada como um ponteiro para o primeiro nó da lista

```
10  /* nó da lista para armazenar valores reais */
11  struct lista {
12      float info;
13      struct lista* prox;
14  };
15  typedef struct lista Lista;
16
17  /* estrutura da pilha */
18  struct pilha {
19      Lista* prim;     /* aponta para o topo da pilha */
20  };
21  typedef struct pilha Pilha;
```

Implementação de pilha como lista

- Função pilha_cria
  - Cria – aloca a estrutura da pilha
  - Inicializa a lista como sendo vazia

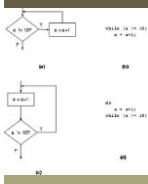
```
30  Pilha* pilha_cria (void)
31  {
32      Pilha* p = (Pilha*) malloc(sizeof(Pilha));
33      p->prim = NULL;
34      return p;
35  }
```



Implementação de pilha como lista

- Função pilha_push
  - Insere novo elemento v no início da lista

```
void pilha_push (Pilha* p, float v)
{
    Lista* n = (Lista*) malloc(sizeof(Lista));
    n->info = v;
    n->prox = p->prim;
    p->prim = n;
}
```



Implementação de pilha como lista

- Função pilha_pop
  - Retira o elemento do início da lista

```
float pilha_pop (Pilha* p){
    Lista* t;
    float v;
    if (pilha_vazia(p)){
        printf("Pilha vazia.\n");
        exit(1);     /* aborta programa */
    }
    t = p->prim;
    v = t->info;
    p->prim = t->prox;
    free(t);
    return v;
}
```

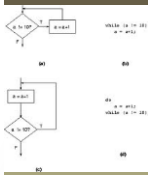
Pilha como lista

- Função pilha_libera
  - Libera a pilha depois de liberar todos os elementos da lista

```
void pilha_libera (Pilha* p){
    Lista* q = p->prim;
    while (q!=NULL){
        Lista* t = q->prox;
        free(q);
        q = t;
    }
    free(p);
}
```

Pilha Vazia?

```
69  int pilha_vazia (Pilha* p)
70  {
71      return (p->prim==NULL);
72  }
73
```

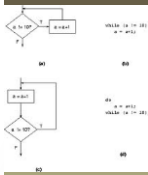


Para Testar

```
82  main(){
83
84      Pilha* nova_pilha;
85
86      nova_pilha = pilha_cria();
87      pilha_push(nova_pilha, 25);
88      pilha_push(nova_pilha, 15);
89      pilha_push(nova_pilha, 40);
90      pilha_push(nova_pilha, 60);
91      pilha_push(nova_pilha, 13.56);
92      printf("Pilha Original: "); QL;
93      pilha_imprime(nova_pilha);  QL;
94      pilha_pop(nova_pilha);
95      printf("Pilha Original apos um pop(): "); QL;
96      pilha_imprime(nova_pilha);
97      pilha_push(nova_pilha, 75);
98      printf("Pilha Original apos um push(): "); QL;
99      pilha_imprime(nova_pilha);
100     pilha_pop(nova_pilha);
101     printf("Pilha Original apos um pop(): "); QL;
102     pilha_imprime(nova_pilha);
103
104     pilha_libera(nova_pilha);
105
106 }
107
```

Exercício

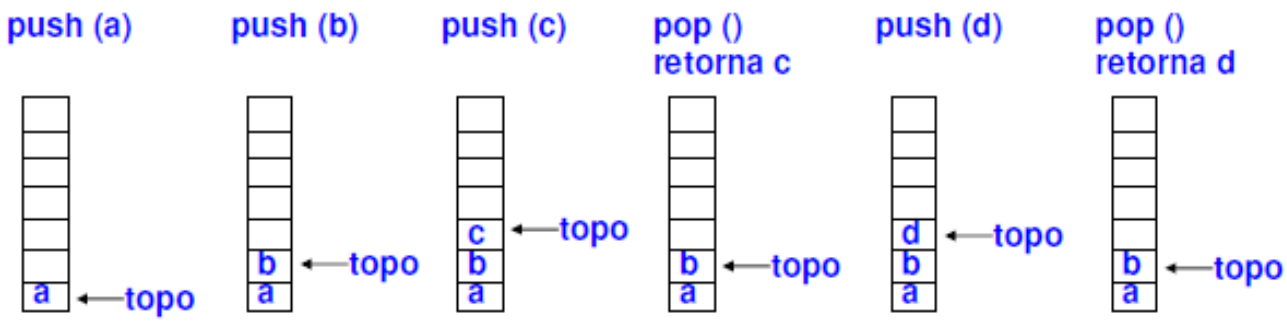
- Apresentar um menu para inserir (push), retirar(pop) elementos na pilha (exibir a pilha a cada operação.



Resumo

Pilha

top	retorna o topo da pilha
push	*insere* novo elemento no topo da pilha
pop	*remove o* elemento do topo da pilha



VER A LISTA DE EXERCÍCIOS QUE ESTARÁ DISPONÍVEL NO BLOG E NO DROPBOX.