Estrutura de Dados

Prof. Tiago Eugenio de Melo, MSc.
tiago@comunidadesol.org
www.tiagodemelo.info

Sumário

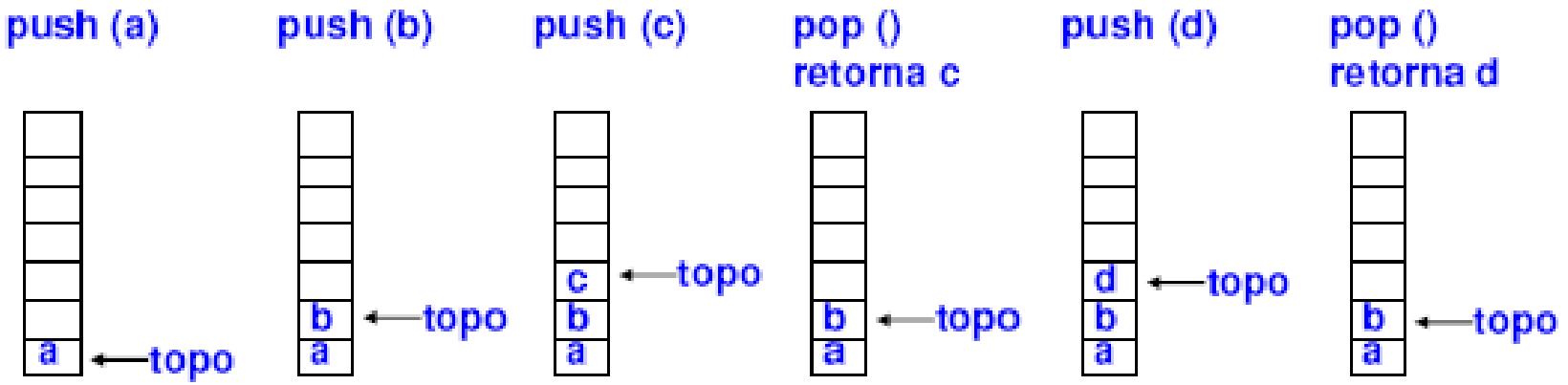
- Pilha
- Recursão
- Listas encadeadas
- Filas

Pilha

- O novo é elemento é inserido no topo e o acesso é apenas no topo.
- O primeiro elemento que sai é o último que entrou (LIFO – *last in last out).*
- Operações básicas:
  - empilhar (push) um novo elemento, inserindo-o no topo.
  - desempilhar (pop) um elemento, removendo-o do topo.

Pilha

- Operações na pilha



Pilha

- Implementações de pilha
  - usando vetor.
  - usando lista encadeada.

Interface do tipo pilha

- Interface do TAD Pilha: pilha.h
  - função pilha_cria
    - aloca dinamicamente a estrutura da pilha.
    - inicializa os seus campos e retorna o seu ponteiro.
  - funções pilha_push e pilha_pop
    - inserem e retiram, respectivamente, um elemento do topo da pilha.

Interface do tipo pilha

- Interface do TAD Pilha: pilha.h
  - função pilha_vazia
    - informa se a pilha está vazia.
  - função pilha_libera
    - destrói a pilha, liberando a memória usada pela estrutura.

Interface do tipo pilha

- Resumo

```
typedef struct pilha Pilha;

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);
```

tipo Pilha:
- definido na interface
- depende da implementação do struct pilha

Implementação da pilha com 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.

```
#define N  50     /* número máximo de elementos */

struct pilha {
   int n;         /* vet[n]:    primeira posição livre do vetor    */
   float vet[N];  /* vet[n-1]:  topo da pilha                      */
                  /* vet[0] a vet[N-1]:  posições ocupáveis        */
};
```

Implementação da pilha com vetor

- função pilha_cria
  - aloca dinamicamente um vetor.
  - inicializa a pilha como sendo vazia, isto é, com o número de elementos igual a zero.

tipo Pilha: definido na interface
struct pilha: determina a implementação

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

Implementação da pilha com vetor

- função pilha_push
  - insere um elemento na pilha.
  - usa a próxima posição livre do vetor, se houver.

```
void pilha_push (Pilha* p, float v)
{
  if (p->n == N) {          /* capacidade esgotada     */
    printf("Capacidade da pilha estourou.\n");
    exit(1);                /* aborta programa         */
  }
  /* insere elemento na próxima posição livre          */
  p->vet[p->n] = v;
  p->n++;                   /* equivalente a:  p->n = p->n + 1 */
}
```

Implementação da pilha com vetor

- função pilha_pop
  - retira o elemento do topo da pilha, retornando o seu valor.
  - verificar se a pilha está ou não vazia.

```
float pilha_pop (Pilha* p)
{ float v;
  if (pilha_vazia(p)) { printf("Pilha vazia.\n");
                        exit(1); }          /* aborta programa          */
  /* retira elemento do topo */
  v = p->vet[p->n-1];
  p->n--;
  return v;
}
```

Implementação da pilha com lista

- Implementação de pilha com lista
  - elementos da pilha armazenados na lista.
  - pilha representada por um ponteiro para o primeiro nó da lista.

```
/* nó da lista para armazenar valores reais */
struct lista {
   float info;
   struct lista* prox;
};
typedef struct lista Lista;

/* estrutura da pilha */
struct pilha {
   Lista* prim;                    /* aponta para o topo da pilha */
};
```

Implementação da pilha com lista

- função pilha_cria
  - aloca a estrutura da pilha.
  - inicializa a lista como sendo vazia.

```
Pilha* pilha_cria (void)
{
  Pilha* p = (Pilha*) malloc(sizeof(Pilha));
  p->prim = NULL;
  return p;
}
```

Implementação da pilha com lista

- função pilha_push
  - insere novo elemento n 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 da pilha com lista

- função pilha_pop
  - retira o elemento no 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;
}
```

Implementação da pilha com 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);
}
```

Uso de pilhas

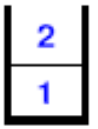
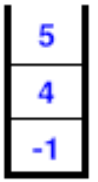
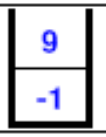
- Notação para expressões aritméticas
  - infixa
    - operador entre os operandos.
    - (1 – 2) * (4 + 5)
  - pós-fixa
    - operador após os operandos.
    - 1 2 – 4 5 + *
  - pré-fixa
    - operador antes dos operandos.
    - * - 1 2 + 4 5

Uso de pilhas

- A calculadora HP, por exemplo, utiliza a notação pós-fixa.
- A avaliação de expressões aritméticas pós-fixadas:
  - cada operando é empilhado numa pilha de valores.
  - quando se encontra um operador:
    - desempilha-se o número apropriado de operandos (dois para operandos binários e para operadores unários).
    - realiza-se a operação devida.
    - empilha-se o resultado.
- Exemplo: avaliação da expressão 1 2 – 4 5 + *

Uso de pilhas

- Exemplo

empilhe os valores 1 e 2	1 2 – 4 5 + *	
quando aparece o operador "–"	1 2 – 4 5 + *	
desempilhe 1 e 2		
empilhe -1, o resultado da operação (1 - 2)		
empilhe os valores 4 e 5	1 2 – 4 5 + *	
quando aparece o operador "+"	1 2 – 4 5 + *	
desempilhe 4 e 5		
empilhe 9, o resultado da operação (4+5)		
quando aparece o operador "*"	1 2 – 4 5 + *	
desempilhe -1 e 9		
empilhe -9, o resultado da operação (-1*9)		

Implementação da calculadora

- Interface da calculadora calc.h

```
typedef struct calc Calc;

/* funções exportadas */
Calc* calc_cria (char* f);
void calc_operando (Calc* c, float v);
void calc_operador (Calc* c, char op);
void calc_libera (Calc* c);

/* tipo representando a calculadora */
struct calc {
  char f[21];       /* formato para impressão (cadeia de caracteres) */
  Pilha* p;         /* pilha de operandos */
};
```

Implementação da calculadora

- função cria
  - recebe como entrada uma cadeia de caracteres com o formato que será utilizado pela calculadora para imprimir os valores.
  - cria uma calculadora inicialmente sem operandos na pilha.

```
Calc* calc_cria (char* formato)
{
 Calc* c = (Calc*) malloc(sizeof(Calc));
 strcpy(c->f,formato);        /* copia a cadeia de caracteres " formato "
                                 para a cadeia de caracteres " c->f "      */
 c->p = pilha_cria();         /* cria pilha vazia                          */
 return c;
}
```

Implementação da calculadora

- função operando
  - coloca no topo da pilha o valor passado como parâmetro.

```
void calc_operando (Calc* c, float v)
{
 /* empilha operando */
 pilha_push(c->p,v);

 /* imprime topo da pilha */
 printf(c->f,v);
}
```

Implementação da calculadora

- função operador
  - retira dois valores do topo da pilha (operadores são binários).
  - efetua a operação correspondente.
  - coloca o resultado no topo da pilha:
    - operações válidas: + - * /
    - se não existirem operandos na pilha, assume-se que são zero.

Implementação da calculadora

```
void calc_operador (Calc* c, char op)
{
 float v1, v2, v;
 if (pilha_vazia(c->p))          /* desempilha operandos */
    v2 = 0.0;
 else
    v2 = pilha_pop(c->p);
 if (pilha_vazia(c->p))
    v1 = 0.0;
 else
    v1 = pilha_pop(c->p);
 switch (op) {                   /* faz operação          */
    case '+':   v = v1+v2;       break;
    case '-':   v = v1-v2;       break;
    case '*':   v = v1*v2;       break;
    case '/':   v = v1/v2;       break;
 }
pilha_push(c->p,v);              /* empilha resultado     */
printf(c->f,v);                  /* imprime topo da pilha */
}
```

Implementação da calculadora

```
/* Programa para ler expressão e chamar funções da calculadora */

#include <stdio.h>
#include "calc.h"

int main (void)
{
 char c;
 float v;
 Calc* calc;

 /* cria calculadora com formato de duas casas decimais */
 calc = calc_cria ("%.2f\n");
```

Referências

- Estrutura de Dados (PUC-RJ)
  - http://www.inf.puc-rio.br/~inf1620

Recursão

- Muitos problemas têm a seguinte propriedade: cada instância do problema contém uma instância menor do mesmo problema.
- Dizemos que esses problemas têm estrutura recursiva.

Recursão

- Para resolver tais problemas podemos aplicar o seguinte método:
  - se a instância em questão é pequena, resolva-a diretamente (use força bruta se necessário);
  - senão, reduza-a a uma instância menor do mesmo problema, aplique o método à instância menor e volte à instância original.
- A aplicação desse método produz um algoritmo recursivo.

Recursão

- O que é recursão?
  - Quando se executa um procedimento qualquer e no decorrer de sua execução o computador encontra como sub-procedimento o próprio procedimento executado, diz-se que o mesmo é recursivo, ou seja, recursão é o ato de um procedimento executar a si mesmo.

Recursão

- Onde se aplica?
  - A recursão é usada em grande escala na área da computação científica, no desenvolvimento de programas do tipo árvores e seus vários estilos, e no estudo dos fractais, o que é de extrema curiosidade científica pois pouco se sabe sobre como construí-los e qual o mecanismo interno de sua construção.

Recursão

- Tipos de recursão
  - Recursão infinita
    - O procedimento recursivo é infinito quando não existe nenhuma condição que faça o procedimento ser interrompido.

Recursão

- Tipos de recursão
  - Recursão finita
    - A recursão finita é conhecida quando se encontra uma estrutura de condição de parada em algum momento durante uma execução.
    - A recursão ainda se apresenta em alguns outros casos. Quando a chamada recursiva aparece no final de um procedimento, dá-se o nome de recursão terminal. Este tipo é de compreensão consideravelmente fácil, pois existe uma maior probabilidade de se prever o que pode acontecer.

Recursão

- Tipos de recursão
  - Recursão finita
    - Recursão inicial é quando a chamada recursiva se encontra no início do procedimento e central quando se apresenta no meio do módulo. Estas últimas possuem um grau de complexidade bem maior do que a primeira, pois se torna quase que impossível prever o que irá acontecer no término do procedimento, sendo difícil rastreá-lo e entender como se processa cada etapa das chamadas recursivas dentro do mesmo.

Recursão

- Dificuldades da recursão
  - A recursão é tratada por estudantes e professores sob pontos de vista bastante diferentes no que diz respeito à facilidade e dificuldade de se entender este processo. Essas questões podem estar diretamente ligadas ao grau de conhecimento de cada um, visto que se uma pessoa possui uma base computacional relativamente boa, esta terá maior facilidade para entender o processo em si (a execução) e o funcionamento interno do computador para executar a chamada recursiva (conceito de pilha).

Recursão

- Exemplo de função para calcular fatorial (iterativo)

```
int fatorial(int num){
    int i, fat;
    fat = 1;
    for (i = 2; i <= num; i++){
        fat *= i;
    }
    return fat;
}
```

Recursão

- Exemplo de função para calcular fatorial (recursivo)

```
int fatorial(int num) {
   if ( num < 2 ) return 1;
     return (num * fatorial(num – 1));
}
```

  - A linha do if da função é o **critério de parada**, pois não chama a função fatorial novamente.
  - A última linha é a **chamada recursiva**, pois chama a função fatorial com parâmetro num-1.

Comparativo

- Verifique que a função fatorial ficou menor que a iterativa e não exigiu variáveis auxiliares.
- Isto é o que normalmente ocorre, porém a desvantagem da recursão é que na maioria dos casos ela é executada mais lentamente que a interação.

Recursão

- Exemplo: série de Fibonacci
  - A série de Fibonacci consiste de uma seqüência determinada pela seguinte regra:
    - Os primeiros dois números são 1.
    - Os demais são calculados por:

      $$fibo(i) = fibo(i-1) + fibo(i-2)$$

    - Seqüência: 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Recursão

- Exemplo de Fibonacci (iterativo)

```
int fibo( int posicao ) {
    int i, num1 = 1, num2 = 1;
    if ( posicao == 0 ) return num1;
    for ( i = 0; i < posicao; i++ ) {
        num2 = num2 + num1;
        num1 = num2 – num1;
    }
    return num 1;
}
```

Recursão

- Exemplo de Fibonacci (recursivo)

```
int fibo(int pos) {
    if (pos < 2) return 1;
    return (fibo(pos - 1) + fibo(pos - 2);
}
```

  - O critério de parada é a linha do if e a recursão está na última linha, mas neste caso chamando duas vezes a função fibo.

Recursão

- Exemplo
  - Considere o seguinte problema: Determinar o valor de um elemento máximo de um vetor v[0 .. n-1] .
  - É claro que o problema só faz sentido se o vetor não é vazio, ou seja, se n ≥ 1.

Recursão

- Exemplo
  - Solução iterativa do problema:

```
int maximo (int n, int v[ ]){
    int j, x;
    x = v[0];
    for (j = 1; j < n; j += 1)
        if (x < v[j]) x = v[j];
    return x;
}
```

- Exemplo
  - Solução recursiva do problema:

```
int maximo_r (int n, int v[])
{
   if (n == 1)
      return v[0];
   else {
      int x;
      x = maximo_r (n-1, v);
      if (x > v[n-1])
         return x;
      else
         return v[n-1];
   }
}
```

Torre de Hanói

- O problema das Torres de Hanói foi inicialmente proposta pelo matemático francês Edouard Lucas, em 1883.
- Lucas elaborou para seu "invento" uma lenda curiosa sobre uma torre muito grande, a Torre de Brama, que foi criada no início dos tempos, com três hastes contendo 64 discos concêntricos.

Torre de Hanói

- O criador do universo também gerou uma comunidade de monges cuja única atividade seria mover os discos da haste original ("A") para uma de destino ("C") e estabeleceu o mundo acabaria quando os monges terminassem sua tarefa.

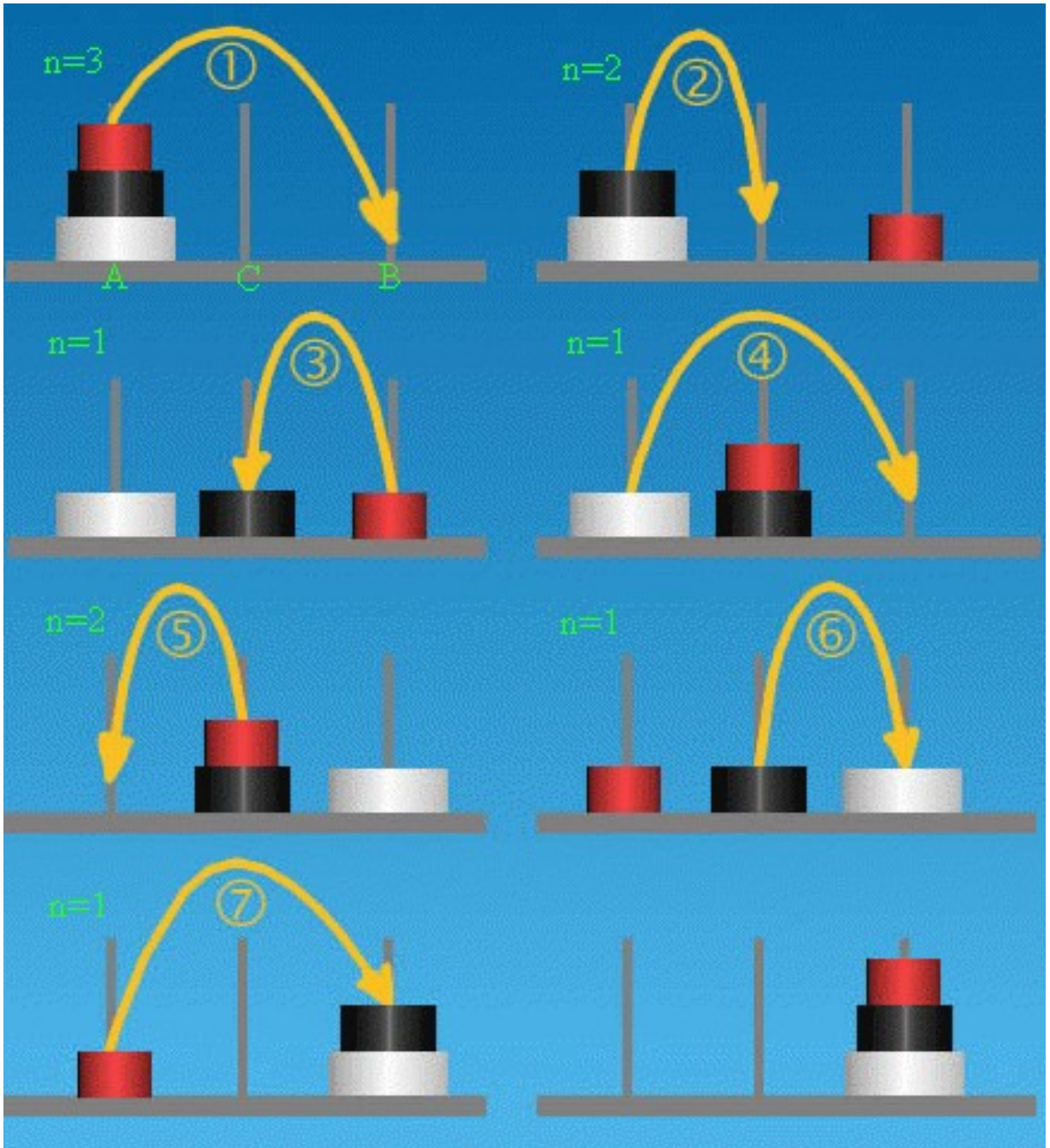
Torre de Hanói

- Porém, os monges deveriam respeitar três regras na sua tarefa:
  1. nunca colocar um disco maior sobre um disco menor;
  2. pode-se mover um único disco por vez;
  3. nunca colocar um disco noutro lugar que não numa das três hastes.
- Assim, sua tarefa é encontrar a regra de movimentação ótima (que atinja o objetivo com um número mínimo de movimentos) e com isso estimar quanto tempo ainda nos resta!!

Torre de Hanói

- Suponha que cada disco leve 1 segundo para ser movido. Tente encontrar uma fórmula que, dado "n" devolva o número mínimo de movimentos para "n" discos.

Torre de Hanói



Torre de Hanói

```
#include <stdio.h>

void hanoi(int n, char a, char b, char c)
{
/* mova n discos do pino a para o pino b usando
   o pino c como intermediario                    */
  if (n == 1)
    printf("mova disco %d de %c para %c\n", n, a, b);
  else
  {
    hanoi(n - 1, a, c, b);                          // H1
    printf("mova disco %d de %c para %c\n", n, a, b);
    hanoi(n - 1, c, b, a);                          // H2
  }
}
```

Torre de Hanói

```
int main(void)
{
  int numDiscos;
  scanf("%d", &numDiscos);
  hanoi(numDiscos, 'A', 'B', 'C');
  // pausa antes do fim
  fflush(stdin);
  getchar();
  return 0;
}
```

Exercícios

- A função abaixo promete encontrar o valor de um elemento máximo de v[0..n-1]. A função cumpre a promessa?

```
int maxi (int n, int v[]) {
   int j, m = v[0];
   for (j = 1; j < n; j++)
      if (v[j-1] < v[j]) m = v[j];
   return m;
}
```

Exercícios

- Qual o valor de X (4)?

```
int X (int n) {
    if (n == 1 || n == 2) return n;
    else return X (n-1) + n * X (n-2);
}
```

- Escreva uma função recursiva que calcule a soma dos dígitos de um inteiro positivo n. A soma dos dígitos de 132, por exemplo, é 6.

Referências

- Neto, Antônio José dos Santos. Explorando Recursão e Fractals. IV Congresso RIBIE, Brasília, 1998.
- http://www.ime.usp.br/~pf/algoritmos/aulas/recu.html
- http://www.unicamp.br/~hans/mc102/C/algoritmo/_hanoi.html
- http://www.ime.usp.br/~leo/imatica/programas/hanoi/index.html

Listas encadeadas

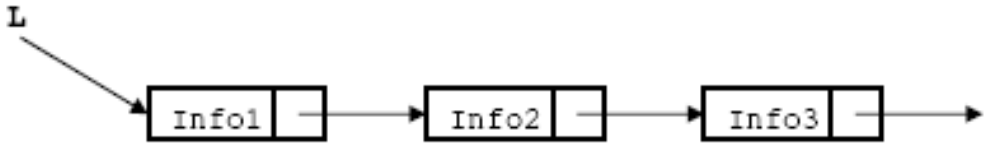
- Motivação
  - Vetores:
    - Ocupam espaços contíguos de memória.
    - Permitem acesso randômico aos elementos.
    - Devem ser dimensionados com um número máximo de elementos.

Listas encadeadas

- Estruturas de dados dinâmicas:
  - Aumentam ou diminuem de tamanho à medida que os elementos são inseridos ou removidos.
  - Exemplo:
    - Listas encadeadas.

Listas encadeadas

- Sequência encadeada de elementos, chamados nós da lista.
- Nó da lista é representado por dois campos:
  - A informação armazenada.
  - Ponteiro para o próximo elemento da lista.
- A lista é representada por um ponteiro para o primeiro nó.
- O ponteiro do último elemento aponta para NULL.



Listas encadeadas

- Exemplo:
  - Lista encadeada armazenando valores inteiros.
  - Estrutura *lista*
    - Estrutura dos nós da lista.
  - Tipo *Lista*
    - Tipos dos nós da lista.

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

lista é uma estrutura auto-referenciada, pois o campo *prox* é um ponteiro para uma próxima estrutura do mesmo tipo

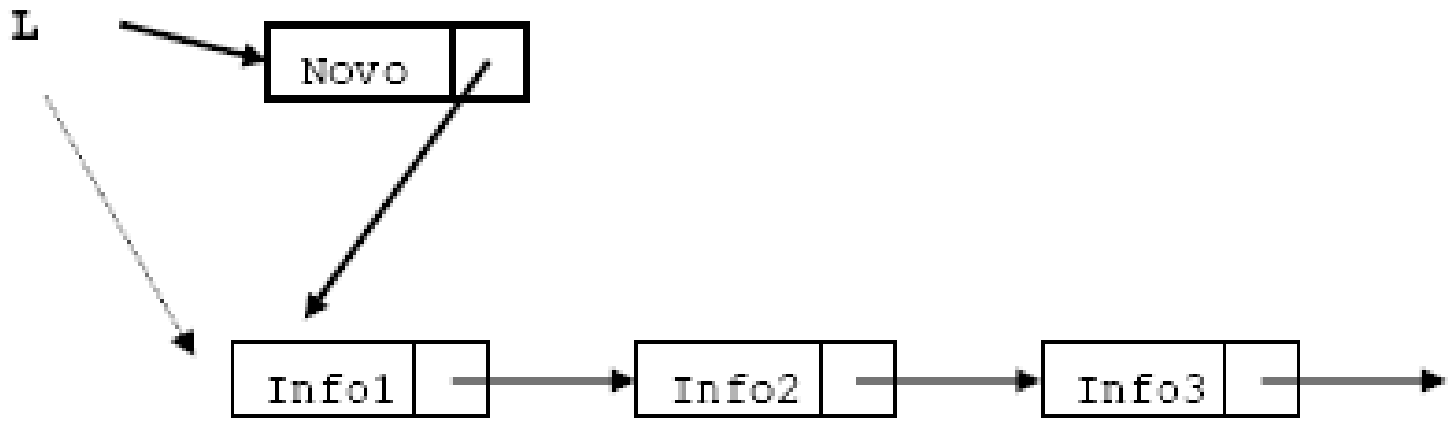
uma lista encadeada é representada pelo ponteiro para seu primeiro elemento, do tipo Lista*

- Exemplo – função de criação
  - Cria uma lista vazia, representada pelo ponteiro NULL.

```
/* função de criação: retorna uma lista vazia */
Lista* lst_cria (void)
{
   return NULL;
}
```

Listas encadeadas

- Exemplo – função de inserção
  - Aloca memória para armazenar o elemento.
  - Encadeia o elemento na lista existente.



- Implementação da função inserção

```
/* inserção no início: 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;
}
```

Listas encadeadas

- Exemplo – 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.

Listas encadeadas

- Exemplo – função para imprimir uma lista
  - Imprime os valores dos elementos armazenados.

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

variável auxiliar p:

- ponteiro, usado para armazenar o endereço de cada elemento
- dentro do loop, aponta para cada um dos elementos da lista

Listas encadeadas

- Exemplo – função para verificar se uma lista está vazia.
  - Retorna 1 se a lista estiver vazia ou 0 se não estiver vazia.

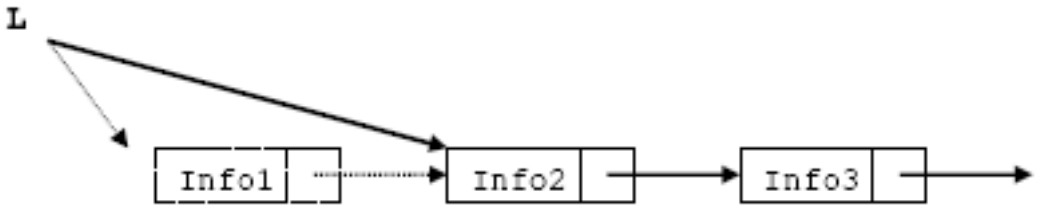
```
/* função vazia: retorna 1 se vazia ou 0 se não vazia */
int lst_vazia (Lista* l)
{
   return (l == NULL);
}
```

- Exemplo – função 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.

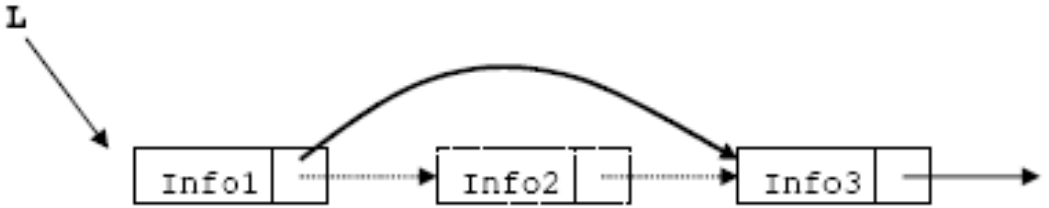
```
/* função busca: busca um elemento na lista */
Lista* busca (Lista* l, int v)
{ Lista* p;
  for (p=l; p!=NULL; p = p->prox) {
    if (p->info == v)
      return p;
  }
  return NULL;      /* não achou o elemento */
}
```

Listas encadeadas

- Exemplo – função para retirar um elemento da lista
  - 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.



Listas encadeadas

```
/* função 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 anterior */
  while (p != NULL && p->info != v)
  {  ant = p;
     p = p->prox; }
  /* verifica se achou elemento */
  if (p == NULL)
     return l;              /* não achou: retorna lista original */
  /* retira elemento */
  if (ant == NULL)
        { /* retira elemento do inicio */
          l = p->prox; }
  else { /* retira elemento do meio da lista */
          ant->prox = p->prox; }
  free(p);
  return l;
}
```

Listas encadeadas

- Exemplo – função para liberar a lista.
  - Destrói a lista, liberando todos os seus elementos alocados.

```
void lst_libera (Lista* l)
{
   Lista* p = l;
   while (p != NULL) {
      Lista* t = p->prox;      /* guarda referência p/ próx. elemento      */
      free(p);                 /* libera a memória apontada por p          */
      p = t;                   /* faz p apontar para o próximo             */
   }
}
```

Listas encadeadas

- TAD – Lista de inteiros

```
/* TAD: lista de inteiros */

typedef struct lista Lista;

Lista* lst_cria (void);
void lst_libera (Lista* l);

Lista* lst_insere (Lista* l, int i);
Lista* lst_retira (Lista* l, int v);

int lst_vazia (Lista* l);
Lista* lst_busca (Lista* l, int v);
void lst_imprime (Lista* l);
```

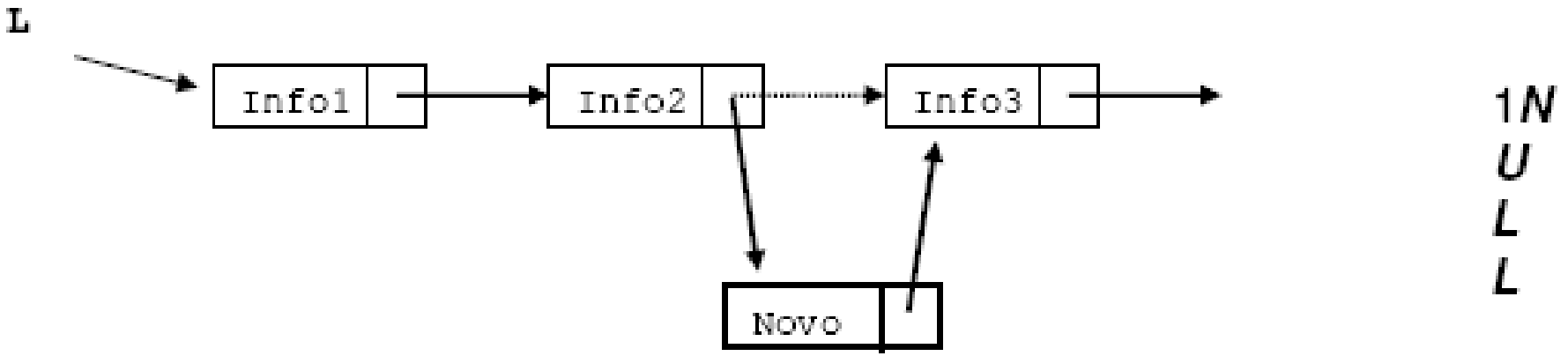
Listas encadeadas

programa que utiliza as funções de lista exportadas

```
#include <stdio.h>
#include "lista.h"
int main (void)
{
  Lista* l;                 /* declara uma lista não iniciada     */
  l = lst_cria();           /* inicia lista 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, 56);    /* insere na lista o elemento 56      */
  l = lst_insere(l, 78);    /* insere na lista o elemento 78      */
  lst_imprime(l);           /* imprimirá: 78 56 45 23             */
  l = lst_retira(l, 78);
  lst_imprime(l);           /* imprimirá: 56 45 23                */
  l = lst_retira(l, 45);
  lst_imprime(l);           /* imprimirá: 56 23                   */
  lst_libera(l);
  return 0;
}
```

Listas encadeadas

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

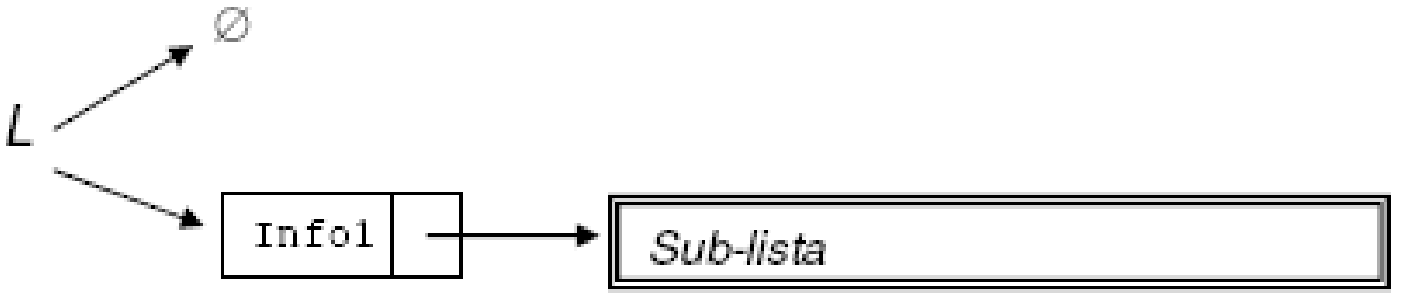


Listas encadeadas

```
/* função insere_ordenado: insere 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 posição de inserção */
  while (p != NULL && p->info < v)
  {  ant = p; p = p->prox; }
  /* cria novo elemento */
  novo = (Lista*) malloc(sizeof(Lista));
  novo->info = v;
  /* encadeia elemento */
  if (ant == NULL)
          { /* insere elemento no início          */
            novo->prox = l; l = novo; }
     else { /* insere elemento no meio da lista */
            novo->prox = ant->prox;
            ant->prox = novo; }
  return l;
}
```

Implementações recursivas

- Definição recursiva de lista
  - Uma lista é:
    - Uma lista vazia; ou
    - Um elemento seguido de uma (sub)lista.



Implementações recursivas

- Exemplo – função para imprimir uma lista.
  - Se a lista estiver vazia, não imprime nada.
  - Caso contrário,
    - Imprima a informação associada ao primeiro nó, dada por l->info.
    - Imprima a sub-lista, dada por l->prox, chamando recursivamente a função.

Implementações recursivas

```
/* Função imprime recursiva */
void lst_imprime_rec (Lista* l)
{
   if ( ! lst_vazia(l)) {
      /* imprime primeiro elemento     */
      printf("info: %d\n",l->info);
      /* imprime sub-lista             */
      lst_imprime_rec(l->prox);
   }
}
```

Implementações recursivas

- Exemplo – função para retirar um elemento da lista.
  - Retire o elemento, se ele for o primeiro da lista (sub-lista).
  - Caso contrário, chame a função recursivamente para retirar o elemento da sub-lista.

Implementações recursivas

```
/* 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;
}
```

é necessário re-atribuir o valor de l->prox na chamada recursiva, já que a função pode alterar o valor da sub-lista

Implementações recursivas

- Exemplo – função para testar a igualdade de duas listas.
- int lsg_igual (Lista* l1, Lista* l2);
  - Implementação não recursiva.
    - Percorre as duas listas usando dois ponteiros auxiliares.
      - Se as 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.

Implementações recursivas

```
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;
}
```

Implementações recursivas

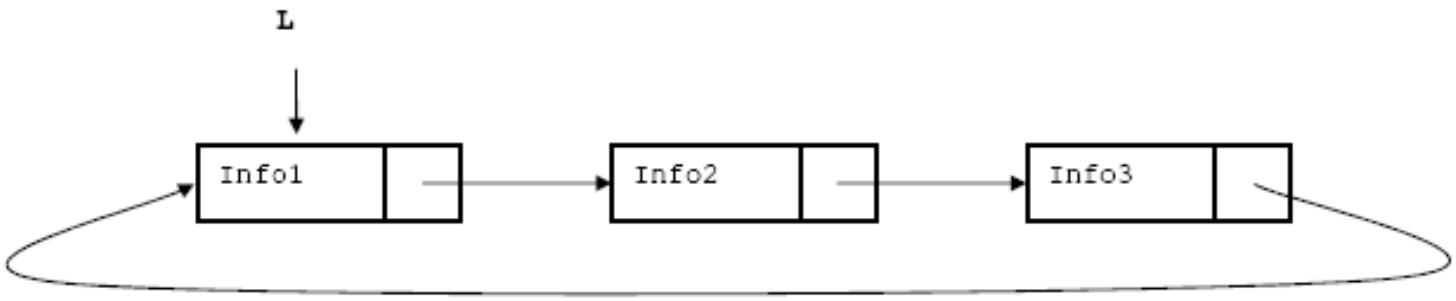
- Exemplo – função para testar a igualdade de duas listas.
  - Implementações recursivas:
    - Se as duas listas dadas são vazias, são iguais.
    - Se não forem ambas vazias, mas uma delas é vazia, são consideradas diferentes.
    - Se ambas não forem vazias, teste:
      - Se informações associadas aos primeiros nós são iguais; e
      - Se as sub-listas são iguais.

Implementações recursivas

```
int lst_igual (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 Circulares

- Lista circular:
  - O último elemento tem como próximo o primeiro elemento da lista, formando um ciclo.
  - A lista pode ser representada por um ponteiro para um elemento inicial qualquer da lista.



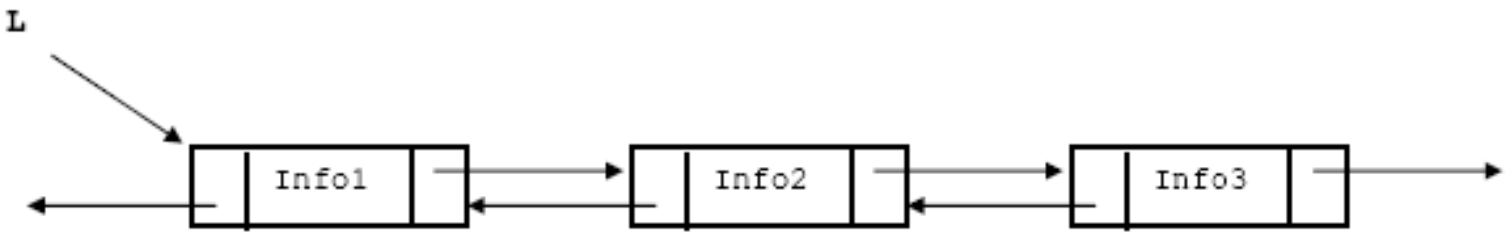
Listas Circulares

- Exemplo – função para imprimir uma lista circular
  - Visita todos os elementos a partir do ponteiro do elemento inicial até alcançar novamente esse mesmo elemento.
  - Se a lista é vazia, o ponteiro para um elemento inicial é NULL.

```
/* função imprime: imprime valores dos elementos */
void lcirc_imprime (Lista* l)
{
  Lista* p = l;       /* faz p apontar para o nó inicial */
  /* testa se lista não é vazia e então percorre com do-while */
  if (p) do {
    printf("%d\n", p->info);  /* imprime informação do nó */
    p = p->prox;              /* avança para o próximo nó */
  } while (p != l);
}
```

Listas Duplamente Encadeada

- 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

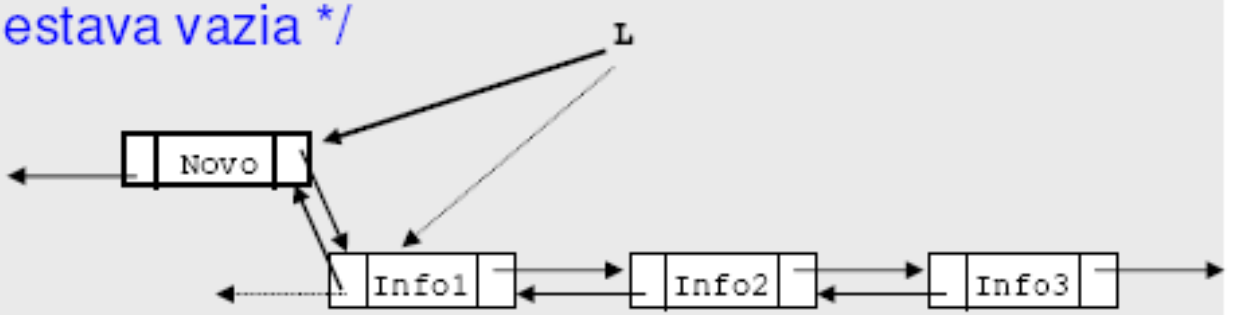
- Exemplo
  - Lista encadeada armazenando valores inteiros.
  - Estrutura lista2
    - Estrutura dos nós da lista.
  - Tipo Lista2
    - Tipos dos nós da lista.

```
struct lista2 {
   int info;
   struct lista2* ant;
   struct lista2* prox;
};
typedef struct lista2 Lista2;
```

Listas Duplamente Encadeadas

- Exemplo – 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;
}
```



Listas Duplamente Encadeadas

- Exemplo – 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).

Listas Duplamente Encadeadas

- Exemplo – função de busca

```
/* 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 */
}
```

Listas Duplamente Encadeadas

- Exemplo – 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.

Listas Duplamente Encadeadas

- Exemplo – função para retirar um elemento da lista
  - 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.

Listas Duplamente Encadeadas

```
/* função retira: remove elemento da lista */
Lista2* lst2_retira (Lista2* l, int v) {
  Lista2* p = busca(l,v);

  if (p == NULL)
    return l;         /* não achou o elemento: retorna lista inalterada */

  /* retira elemento do encadeamento */
  if (l == p)         /* testa se é o primeiro elemento */
    l = p->prox;
  else
    p->ant->prox = p->prox;

  if (p->prox != NULL)       /* testa se é o último elemento */
    p->prox->ant = p->ant;

  free(p);

  return l;
}
```

Listas de Tipos Estruturados

- Lista de tipo estruturado:
  - A informação associada a cada nó de uma lista encadeada pode ser mais complexa, sem alterar o encadeamento dos elementos.
  - As funções representadas para manipular listas de inteiros podem ser adaptadas para tratar listas de outros tipos.

Listas de Tipos Estruturados

- Lista de tipo estruturado:
  - O campo da informação pode ser representado por um ponteiro para uma estrutura, em lugar da estrutura em si.
  - Independente da informação armazenada na lista, a estrutura do nó é sempre composta de:
    - Um ponteiro para a informação; e
    - Um ponteiro para o próximo nó da lista.

Lista de Tipos Estruturados

- Exemplo – Lista de retângulos

```
struct retangulo {
   float b;
   float h;
};
typedef struct retangulo Retangulo;

struct lista {
   Retangulo info;
   struct lista *prox;
};
```

campo da informação representado por um ponteiro para uma estrutura, em lugar da estrutura em si

Lista de Tipos Estruturados

- Exemplo – função auxiliar para alocar um nó.

```
static Lista* aloca (float b, float h)
{
   Retangulo* r = (Retangulo*) malloc(sizeof(Retangulo));
   Lista* p = (Lista*) malloc(sizeof(Lista));
   r->b = b;
   r->h = h;
   p->info = r;
   p->prox = NULL;
   return p;
}
```

Para alocar um nó, são necessárias duas alocações dinâmicas:
uma para criar a estrutura do retângulo e outra para criar a estrutura do nó.

O valor da base associado a um nó p seria acessado por: p->info->b.

Lista de Tipos Estruturados

- Listas heterogêneas
  - A representação da informação por um ponteiro permite construir listas heterogêneas, isto é, listas em que as informações armazenadas diferem de nó para nó.

Lista de Tipos Estruturados

- Exemplo:
  - Lista de retângulos, triângulos ou círculos.
  - Áreas desses objetos são dadas por:

$$r = b * h \qquad t = \frac{b * h}{2} \qquad c = \pi\, r^2$$

Lista de Tipos Estruturados

```
struct retangulo {
   float b;
   float h;
};
typedef struct retangulo Retangulo;

struct triangulo {
   float b;
   float h;
};
typedef struct triangulo Triangulo;

struct circulo {
   float r;
};
typedef struct circulo Circulo;
```

Lista de Tipos Estruturados

- Exemplo
  - A lista é homegênea – todos os nós contêm os mesmos campos.
    - Um ponteiro para o próximo nó da lista.
    - Um ponteiro para a estrutura que contém a informação.
      - Deve ser do tipo genérico (ou seja, do tipo void*), pois pode apontar para um retângulo, um triângulo ou um círculo.
    - Um identificador indicando qual o objeto o nó armazena.
      - Consultando esse identificador, o ponteiro genérico pode ser convertido no ponteiro específico para o objeto e os campos do objeto podem ser acessados.

Lista de Tipos Estruturados

```
/* Definição dos tipos de objetos */
#define RET 0
#define TRI 1
#define CIR 2

/* Definição do nó da estrutura */
struct listahet {
   int       tipo;
   void      *info;
   struct    listahet *prox;
};
typedef struct listahet ListaHet;
```

Listas Duplamente Encadeadas

- Exemplo – função para a criação de um nó da lista.

```
/* Cria um nó com um retângulo */
ListaHet* cria_ret (float b, float h)
{
   Retangulo* r;
   ListaHet* p;
   /* aloca retângulo */
   r = (Retangulo*) malloc(sizeof(Retangulo));
   r->b = b; r->h = h;
   /* aloca nó */
   p = (ListaHet*) malloc(sizeof(ListaHet));
   p->tipo = RET;
   p->info = r;
   p->prox = NULL;
   return p;
}
```

a função para a criação de um nó possui três variações, uma para cada tipo de objeto

Listas Duplamente Encadeadas

- Exemplo – função para calcular a maior área.
  - Retorna a maior área entre os elementos da lista.
  - Para cada nó, de acordo com o tipo de objeto que armazena, chama uma função específica para o cálculo da área.

Listas Duplamente Encadeadas

```
/* função para cálculo da área de um retângulo */
static float ret_area (Retangulo* r)
{
   return r->b * r->h;
}

/* função para cálculo da área de um triângulo */
static float tri_area (Triangulo* t)
{
   return (t->b * t->h) / 2;
}

/* função para cálculo da área de um círculo */
static float cir_area (Circulo* c)
{
   return PI * c->r * c->r;
}
```

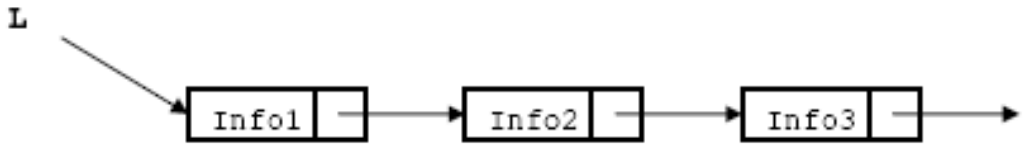
Listas Duplamente Encadeadas

```
/* função para cálculo da área do nó (versão 2) */
static float area (ListaHet* p)
{
   float a;
   switch (p->tipo) {
      case RET:
         a = ret_area(p->info);
      break;
      case TRI:
         a = tri_area(p->info);
      break;
      case CIR:
         a = cir_area(p->info);
      break;
   }
   return a;
}
```

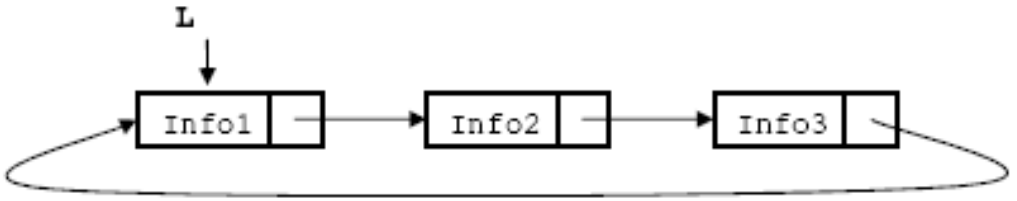
a conversão de ponteiro genérico para ponteiro específico ocorre quando uma das funções de cálculo da área é chamada:
passa-se um ponteiro genérico, que é atribuído a um ponteiro específico, através da conversão implícita de tipo

Resumo

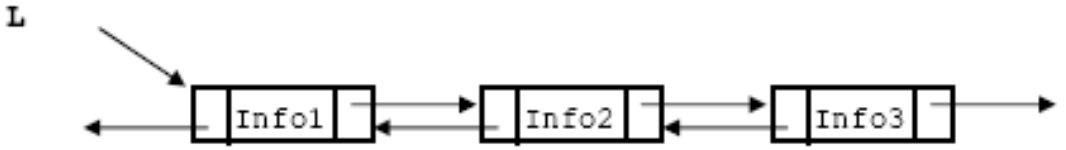
Listas encadeadas



Listas circulares

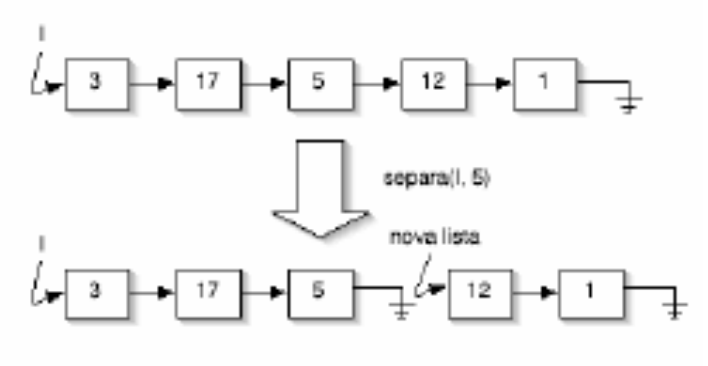


Listas duplamente encadeadas



Exercícios

2.6. Considerando listas de valores inteiros, implemente uma função que receba como parâmetro uma lista encadeada e um valor inteiro n e divida a lista em duas, de tal forma que a segunda lista comece no primeiro nó logo após a primeira ocorrência de n na lista original. A figura a seguir ilustra essa separação:



Essa função deve obedecer ao protótipo:

```
Lista* separa (Lista* l, int n);
```

A função deve retornar um ponteiro para a segunda sub-divisão da lista original, enquanto l deve continuar apontando para o primeiro elemento da primeira sub-divisão da lista.

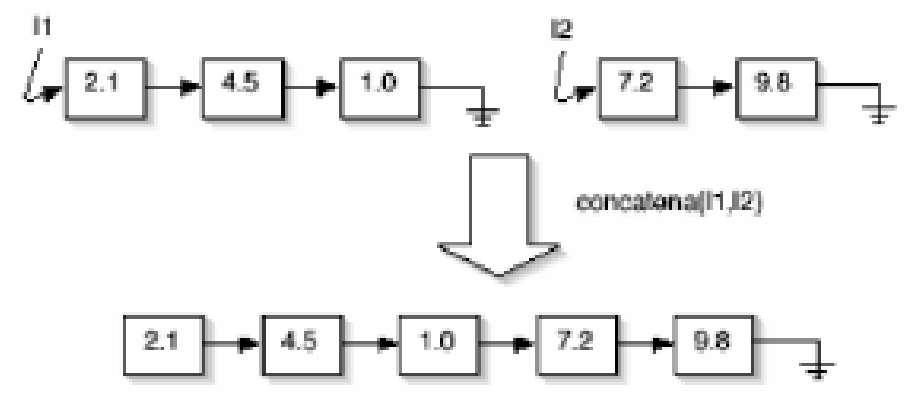
Exercícios

- Considere estruturas de listas encadeadas que armazenam valores reais. O tipo que representa um nó da lista é dado por:

```
struct lista {
   float info;
   struct lista* prox;
};
typedef struct lista Lista;
```

Exercícios

- Implemente uma função que, dadas duas listas encadeadas l1 e l2, concatene l2 no final da lista l1, conforme a figura abaixo:



- A função deve retornar a lista resultante da concatenação, obedecendo ao protótipo:
  - `Lista* concatena (Lista* l1, Lista* l2);`
  - Observe que l1 e/ou l2 podem ser vazias.

Exercícios

Considere a implementação de uma lista encadeada para armazenar números reais dada pelo tipo abaixo:

```
struct lista {
   float info;
   struct lista* prox;
};
typedef struct lista Lista;
```

Implemente uma função que, dados uma lista encadeada e um número inteiro não negativo `n`, remova da lista seus `n` primeiros nós e retorne a lista resultante. Caso `n` seja maior do que o comprimento da lista, todos os seus elementos devem ser removidos e o resultado da função deve ser uma lista vazia. Essa função deve obedecer o seguinte protótipo:

```
Lista* retira_prefixo (Lista* l, int n);
```

Referências bibliográficas.

- Estrutura de Dados (PUC-RJ)
  - http://www.inf.puc-rio.br/~inf1620

Filas

- Sumário
  - Introdução
  - Interface do tipo fila
  - Implementação de fila com vetor
  - Implementação de fila com lista
  - Fila dupla
  - Implementação de fila dupla com lista

Filas

- Um novo elemento é inserido no final da fila e um elemento é retirado do início da fila.
  - Fila = “o primeiro que entra é o primeiro que sai” (FIFO)
  - Pilha = “o último que entra é o primeiro que sai” (LIFO)

Interface do tipo fila

- Implementações
  - Usando um vetor.
  - Usando uma lista encadeada.
  - Simplificação:
    - Fila armazena valores reais.

Interface do tipo fila

- Interface do tipo abstrato Fila: fila.h
  - Função fila_cria
    - Aloca dinamicamente a estrutura da fila.
    - Inicializa seus campos e retorna seu ponteiro.
  - Função fila_insere e função fila_retira
    - Insere e retira, respectivamente, um valor real na fila.
  - Função fila_vazia
    - Informa se a fila está ou não vazia.
  - Função fila_libera
    - Destróia a fila, liberando toda a memória usada pela estrutura.

Interface do tipo fila

```
typedef struct fila Fila;

Fila* fila_cria (void);

void fila_insere (Fila* f, float v);

float fila_retira (Fila* f);

int fila_vazia (Fila* f);

void fila_libera (Fila* f);
```

tipo Fila:
- definido na interface
- depende da implementação do struct fila

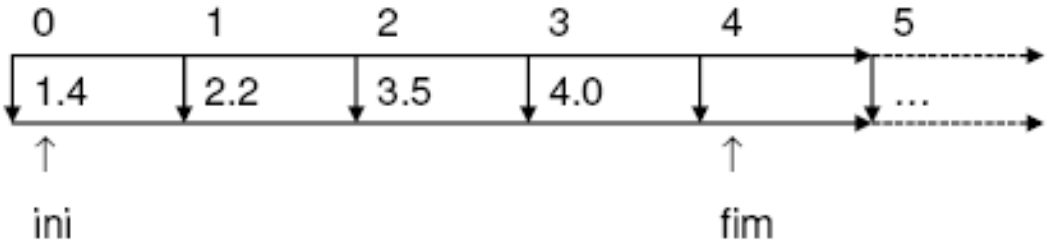
Implementação de fila com vetor

- Implementação de fila com vetor
  - Vetor (vet) armazena os elementos da fila.
  - Estruturas de fila:

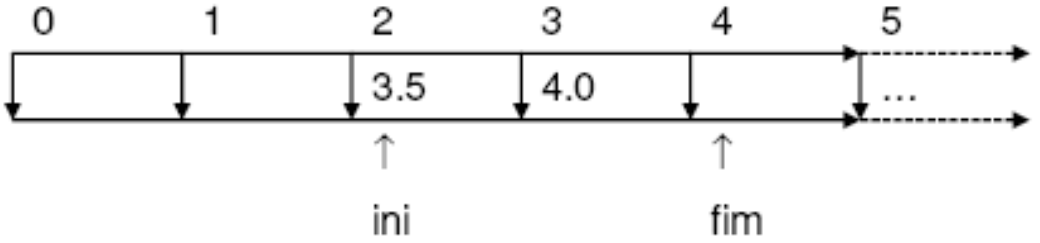
```
#define N  100      /* número máximo de elementos */

struct fila {
   int n;           /* número de elementos na fila */
   int ini;         /* posição do próximo elemento a ser retirado da fila */
   float vet[N];
};
```

Implementação de fila com vetor

- Implementação de fila com vetor
  - Processo de inserção e remoção com extremidades opostas da fila faz com que a fila ande no vetor.
    - Inserção dos elementos 1.4, 2.2, 3.5 e 4.0.



    - Remoção de dois elementos.



Implementação de fila com vetor

- Implementação de fila com vetor
  - Incremento das posições do vetor de forma “circular”:
    - Se o último elemento da fila ocupa a última posição do vetor, os novos elementos são inseridos a partir do início do vetor.
    - Exemplo:
      - Quatro elementos: 20.0, 20.8, 21.2 e 24.3.
      - Distribuídos dois no fim do vetor e dois no início.

Implementação de fila com vetor

- Implementação de fila com vetor
  - Incremento das posições do vetor de forma “circular”.
    - Usa o operador módulo “%”.
  - Parâmetros da fila:
    - n = número de elementos na fila.
    - ini = posição do próximo elemento a ser retirado da fila.
    - fim = posição onde será inserido o próximo elemento.

```
...
fim = (ini+n)%N
...
```

Implementação de fila com vetor

- Função fila_cria
  - Aloca dinamicamente um vetor.
  - Inicializa a fila como sendo vazia (número de elementos = 0).

tipo Fila: definido na interface
struct fila: determina a implementação

```
Fila* fila_cria (void)
{
  Fila* f = (Fila*) malloc(sizeof(Fila));
  f->n = 0;          /* inicializa fila como vazia          */
  f->ini = 0;        /* escolhe uma posição inicial         */
  return f;
}
```

Implementação de fila com vetor

- Função fila_insere
  - Insere um elemento no final da fila.
  - Usa a próxima posição livre do vetor, se houver.

```
void fila_insere (Fila* f, float v)
{  int fim;
   if (f->n == N) {   /* fila cheia: capacidade esgotada */
      printf("Capacidade da fila estourou.\n");
      exit(1);        /* aborta programa */
   }
   /* insere elemento na próxima posição livre */
   fim = (f->ini + f->n) % N;
   f->vet[fim] = v;
   f->n++;
}
```

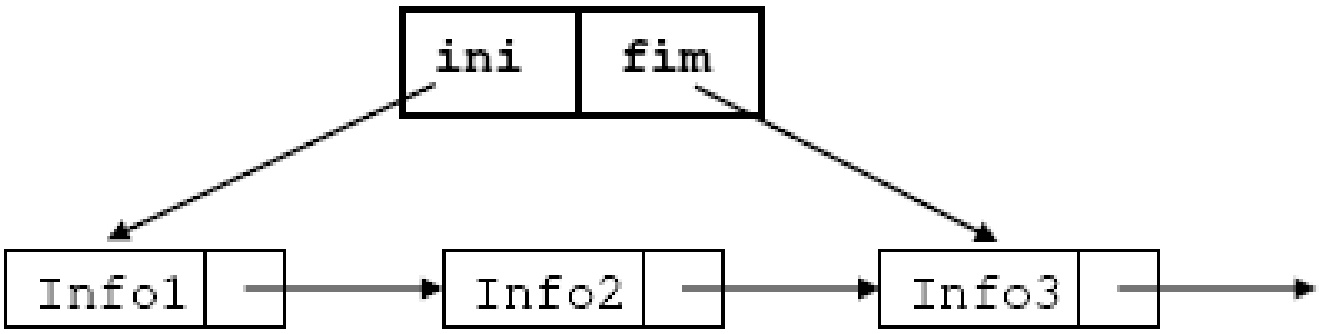
Implementação de fila com vetor

- Função fila_retira
  - Retira o elemento do início da fila, retornando o seu valor.
  - Verifica se a fila está ou não vazia.

```
float fila_retira (Fila* f)
{ float v;
   if (fila_vazia(f)) {
      printf("Fila vazia.\n");
      exit(1);        /* aborta programa */
   }
   /* retira elemento do início */
   v = f->vet[f->ini];
   f->ini = (f->ini + 1) % N;
   f->n--;
   return v;
}
```

Implementação de fila com lista

- Implementação de fila com lista
  - Elementos da fila armazenados na lista.
  - Usa dois ponteiros.
    - ini: aponta para o primeiro elemento da fila.
    - fim: aponta para o último elemento da fila.



Implementação de fila com lista

- Implementação de fila com lista
  - Elementos da fila armazenados na lista.
  - Fila representada por um ponteiro para o primeiro nó da lista.

```
/* nó da lista para armazenar valores reais */
struct lista {
   float info;
   struct lista* prox;
};
typedef struct lista Lista;

/* estrutura da fila */
struct fila {
   Lista* ini;
   Lista* fim;
};
```

Implementação de fila com lista

- Função fila_cria
  - Cria aloca a estrutura da fila.
  - Inicializa a lista como sendo vazia.

```
Fila* fila_cria (void)
{
  Fila* f = (Fila*) malloc(sizeof(Fila));
  f->ini = f->fim = NULL;
  return f;
}
```

Implementação de fila com lista

- Função fila_insere
  - Insere novo elemento n no final da lista.

```
void fila_insere (Fila* f, float v)
{
  Lista* n = (Lista*) malloc(sizeof(Lista));
  n->info = v;              /* armazena informação                */
  n->prox = NULL;           /* novo nó passa a ser o último       */
  if (f->fim != NULL)       /* verifica se lista não estava vazia */
    f->fim->prox = n;
  else                      /* fila estava vazia                  */
    f->ini = n;
  f->fim = n;               /* fila aponta para novo elemento     */
}
```

Implementação de fila com lista

- Função fila_retira
  - Retira o elemento do início da lista.

```
float fila_retira (Fila* f)
{  Lista* t;
   float v;
   if (fila_vazia(f)) {  printf("Fila vazia.\n");
                         exit(1); }          /* aborta programa          */
   t = f->ini;
   v = t->info;
   f->ini = t->prox;
   if (f->ini == NULL)                       /* verifica se fila ficou vazia */
      f->fim = NULL;
   free(t);
   return v;
}
```

Implementação de fila com lista

- Função file_libera
  - Libera a fila depois de liberar todos os elementos da lista.

```
void fila_libera (Fila* f)
{
   Lista* q = f->ini;
   while (q!=NULL) {
      Lista* t = q->prox;
      free(q);
      q = t;
   }
   free(f);
}
```

Fila dupla

- Fila dupla:
  - Fila na qual é possível:
    - Inserir novos elementos no início e no fim.
    - Retirar elementos de ambos os extremos.
  - Simula, dentro de uma mesma estrutura, duas filas, com os elementos em ordem inversa uma da outra.

Interface do tipo fila dupla

- Interface do tipo abstrato Fila2: fila2.h
  - Função fila2_cria
    - Aloca dinamicamente a estrutura da fila.
    - Inicializa seus campos e retorna seu ponteiro.
  - Função fila2_insere_fim e função fila2_retira_ini
    - Insere no fim e retira do início, respectivamente, um valor real na fila.
  - Função fila2_insere_ini e função fila2_retira_fim
    - Insere no início e retira do fim, respectivamente, um valor real na fila.

Interface do tipo fila dupla

- Interface do tipo abstrato Fila2: fila2.h
  - Função fila2_vazia
    - Informa se a fila está ou não vazia.
  - Função fila2_libera
    - Destrói a fila, liberando toda a memória usada pela estrutura.

Interface do tipo fila dupla

```
typedef struct fila2 Fila2;

Fila2* fila2_cria (void);

void fila2_insere_ini (Fila2* f, float v);

void fila2_insere_fim (Fila2* f, float v);

float fila2_retira_ini (Fila2* f);

float fila2_retira_fim (Fila2* f);

int fila2_vazia (Fila2* f);

void fila2_libera (Fila2* f);
```

Interface do tipo fila dupla com vetor

- Função fila2_insere_ini
  - Insere elemento no início da fila.
    - Índice do elemento que procedente ini é dado por (ini – 1 + N) % N.

```
void fila2_insere_ini (Fila* f, float v)
{  int prec;
   if (f->n == N) {   /* fila cheia: capacidade esgotada    */
      printf("Capacidade da fila estourou.\n");
      exit(1);          /* aborta programa                    */
   }
   /* insere elemento na posição precedente ao início  */
   prec = (f->ini - 1 + N) % N;          /* decremento circular */
   f->vet[prec] = v;
   f->ini = prec;     /* atualiza índice para início          */
   f->n++;
}
```

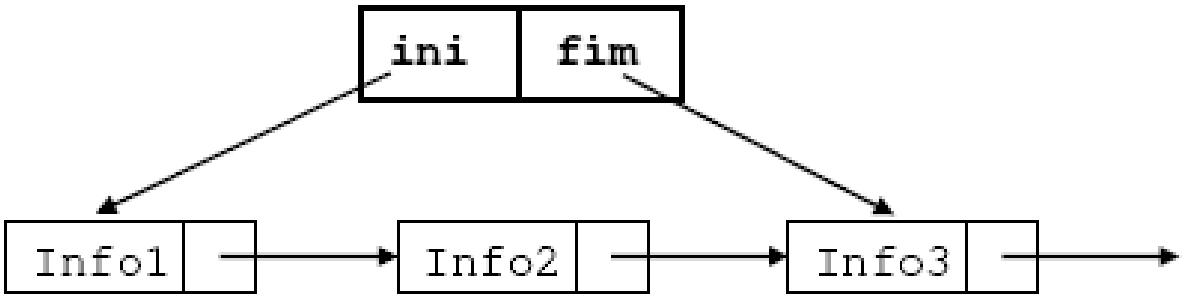
Interface do tipo fila dupla com vetor

- Função fila2_retira_final
  - Retira elemento do final da fila.
  - Índice do último elemento é dado por (ini + n - 1)%N

```
float fila_retira (Fila* f)
{ float v;
   if (fila_vazia(f)) {
      printf("Fila vazia.\n");
      exit(1);        /* aborta programa */
   }
   /* retira elemento do início */
   v = f->vet[f->ini];
   f->ini = (f->ini + 1) % N;
   f->n--;
   return v;
}
```

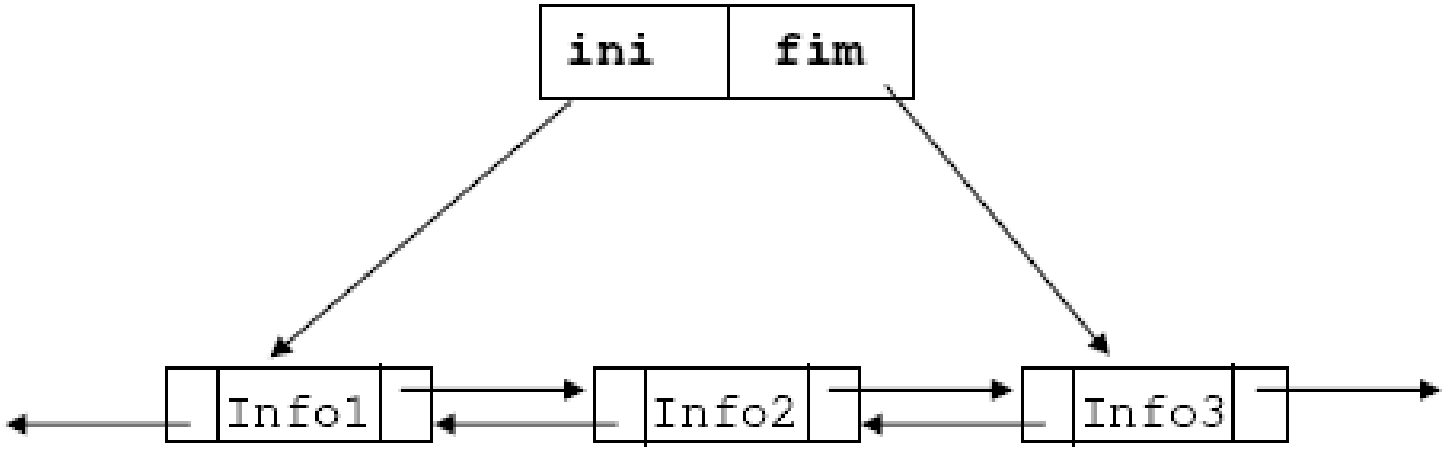
Implementação de fila dupla com lista duplamente encadeada

- Implementação de fila dupla com lista duplamente encadeada
  - Função para retirar do fim.
    - Não pode ser implementada de forma eficiente.
    - Dado o ponteiro para o último elemento da lista, não é possível acessar de forma eficiente o anterior, que passaria a ser o último elemento.



Implementação de fila dupla com lista duplamente encadeada

- Implementação de fila dupla com lista duplamente encadeada
  - dado o ponteiro de um nó, é possível acessar ambos os elementos adjacentes.
  - resolve o problema de acessar o elemento anterior ao último.



Implementação de fila dupla com lista duplamente encadeada

- Implementação de fila dupla com lista duplamente encadeada.

```
/* nó da lista para armazenar valores reais */
struct lista2 {
   float info;
   struct lista2* ant;
   struct lista2* prox;
};
typedef struct lista2 Lista2;

/* estrutura da fila */
struct fila2 {
   Lista2* ini;
   Lista2* fim;
};
```

Implementação de fila dupla com lista duplamente encadeada

- Função auxiliar: insere no início.
  - Insere novo elemento n no início da lista duplamente encadeada.

```
/* função auxiliar: insere no início */
static Lista2* ins2_ini (Lista2* ini, float v)
{
   Lista* p = (Lista2*) malloc(sizeof(Lista2));
   p->info = v;
   p->prox = ini;
   p->ant = NULL;
   if (ini != NULL)              /* verifica se lista não estava vazia */
      ini->ant = p;
   return p;
}
```

Implementação de fila dupla com lista duplamente encadeada

- Função auxiliar: insere no fim.
  - Insere novo elemento n no fim da lista duplamente encadeada.

```
/* função auxiliar: insere no fim */
static Lista2* ins2_fim (Lista2* fim, float v)
{
   Lista2* p = (Lista2*) malloc(sizeof(Lista2));
   p->info = v;
   p->prox = NULL;
   p->ant = fim;
   if (fim != NULL)              /* verifica se lista não estava vazia */
      fim->prox = p;
   return p;
}
```

Implementação de fila dupla com lista duplamente encadeada

- Função auxiliar: retira do início.
  - Retira elemento do início da lista duplamente encadeada.

```
/* função auxiliar: retira do início */
static Lista2* ret2_ini (Lista2* ini)
{
   Lista2* p = ini->prox;
   if (p != NULL)    /* verifica se lista não ficou vazia */
      p->ant = NULL;
   free(ini);
   return p;
}
```

Implementação de fila dupla com lista duplamente encadeada

- Função auxiliar: retira do fim.
  - Retira elemento do fim da lista duplamente encadeada.

```
/* função auxiliar: retira do início */
static Lista2* ret2_ini (Lista2* ini)
{
   Lista2* p = ini->prox;
   if (p != NULL)    /* verifica se lista não ficou vazia */
      p->ant = NULL;
   free(ini);
   return p;
}
```

Implementação de fila dupla com lista duplamente encadeada

- Funções fila2_insere_ini e fila2_insere_fim

```
void fila2_insere_ini (Fila2* f, float v) {
   f->ini = ins2_ini(f->ini,v);
   if (f->fim==NULL)            /* fila antes vazia? */
      f->fim = f->ini;
}


void fila2_insere_fim (Fila2* f, float v) {
   f->fim = ins2_fim(f->fim,v);
   if (f->ini==NULL)            /* fila antes vazia? */
      f->ini = f->fim;
}
```

Implementação de fila dupla com lista duplamente encadeada

- Função fila2_retira_ini

```
float fila2_retira_ini (Fila2* f) {
   float v;
   if (fila2_vazia(f)) {
      printf("Fila vazia.\n");
      exit(1);                    /* aborta programa */
   }
   v = f->ini->info;
   f->ini = ret2_ini(f->ini);
   if (f->ini == NULL)            /* fila ficou vazia? */
      f->fim = NULL;
   return v;
}
```

Implementação de fila dupla com lista duplamente encadeada

- Função fila2_retira_fim

```
float fila2_retira_fim (Fila2* f) {
   float v;
   if (vazia(f)) {
      printf("Fila vazia.\n");
      exit(1);                          /* aborta programa */
   }
   v = f->fim->info;
   f->fim = ret2_fim(f->fim);
   if (f->fim == NULL)                  /* fila ficou vazia? */
      f->ini = NULL;
   return v;
}
```

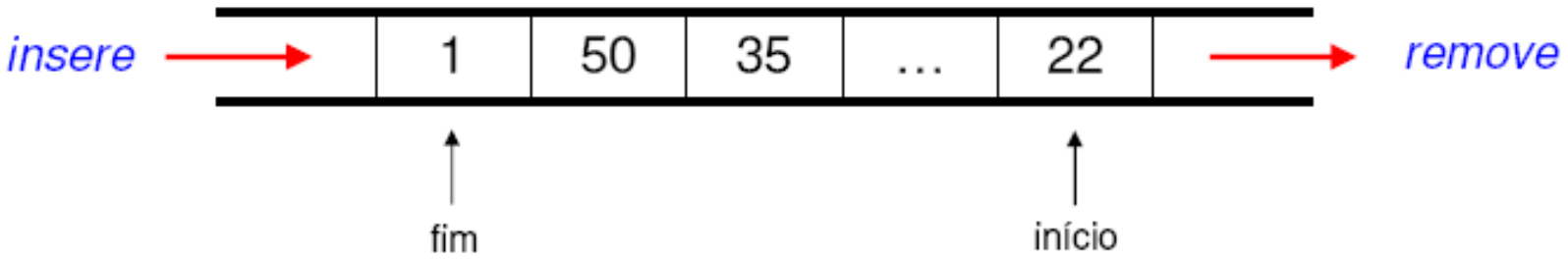
Implementação de fila dupla com lista duplamente encadeada

- Função fila_retira
  - Retira o elemento do início da lista.

```
float fila_retira (Fila* f)
{  Lista* t;
   float v;
   if (fila_vazia(f)) {  printf("Fila vazia.\n");
                         exit(1); }          /* aborta programa          */
   t = f->ini;
   v = t->info;
   f->ini = t->prox;
   if (f->ini == NULL)                       /* verifica se fila ficou vazia */
      f->fim = NULL;
   free(t);
   return v;
}
```

Resumo

- Fila
  - Insere: insere novo elemento no final da fila.
  - Remove: remove o elemento do início da fila.

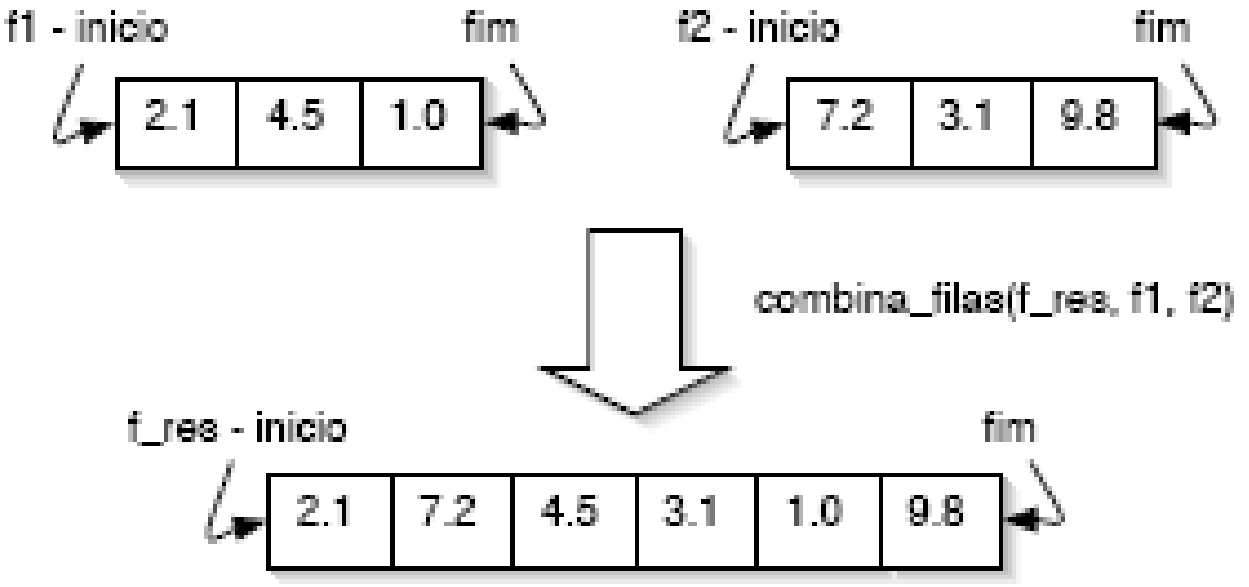


Exercícios

- Considere a existência de um tipo abstrato Fila de números de ponto flutuante, cuja interface está definida no arquivo `fila.h` da seguinte forma:
  - `typedef struct fila Fila;`
  - `Fila* cria(void);`
  - `Float retira (Fila* f);`
  - `int vazia (Fila* f);`
  - `void libera (Fila* f);`

Exercícios

- Sem conhecer a representação interna desse tipo abstrato Fila e usando apenas as funções declaradas no arquivo `fila.h`, implemente uma função que receba três filas, `f_res`, `f1` e `f2`, e transfira alternadamente os elementos de `f1` e `f2` para `f_res`, conforme ilustrado abaixo:



Exercícios

- Note que, ao final dessa função, as filas `f1` e `f2` vão estar vazias e a fila `f_res` vai conter todos os valores que estavam originalmente em `f1` e `f2` (inicialmente `f_res` pode ou não estar vazia). Essa função deve obedecer o protótipo.
  - `void combina_filas (Fila* f_res, Fila* f1, Fila* f2);`

Exercícios

- A **fila de prioridade** é uma estrutura na qual a classificação dos elementos determina o resultado de suas operações. Uma **fila de prioridade ascendente** é um conjunto de itens no qual podem ser inseridos itens arbitrariamente e a partir do qual apenas o menor item pode ser removido. Supondo uma fila de prioridade ascendente, a operação *pqinsere(apq, x)* insere o elemento x em apq e *pqmenorremove(apq)* remove o menor elemento e retorna o seu valor. Uma **fila de prioridade descendente** é idêntica à ascendente, exceto que o maior elemento é o que é removido. Neste caso, a operação de inserção continua a mesma, mas a operação de apagar muda (*pqmaiorremove(apq)*). Na fila de prioridade descendente, em uma operação de remoção, apenas o maior item é removido.

Exercícios

- A implementação de uma fila de prioridade não pode ser feita do mesmo modo que uma fila em geral, pois isso prejudicaria a ordenação do vetor. Veja só, se as operações envolvessem apenas inserção de elementos tudo ocorreria muito bem, mas, no caso de remoção, a situação é diferente. Se, em uma fila de prioridade ascendente, tivesse que se eliminar um elemento, inicialmente o menor item teria que ser localizado, o que implica acesso a todos os elementos da fila de prioridade. Outra situação que poderia ocorrer seria a de o item a ser eliminado não se localizar em uma das extremidades do vetor. Dessa forma, de acordo com a implementação a ser adotada, deveria ser tratada a forma como o vetor deveria se apresentar, de modo que não ficassem muitas lacunas no meio do vetor.
- Implemente uma rotina em C que faça as operações para um fila de prioridade ascendente (remoção, inserção etc).

Referências bibliográficas.

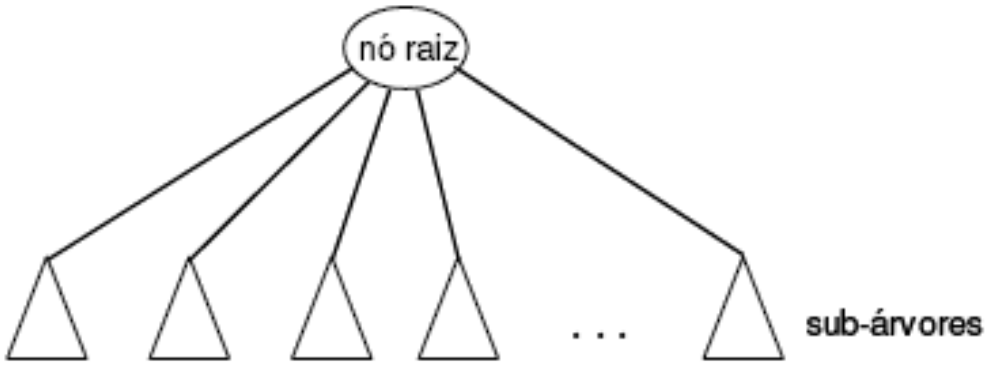
- Estrutura de Dados (PUC-RJ)
  - http://www.inf.puc-rio.br/~inf1620

Árvores Binárias

- Introdução
- Árvores binárias
  - Representação em C.
  - Ordens de percurso em árvores binárias.
  - Altura de uma árvore.
- Árvores com número variável de filhos
  - Representação em C.
  - Tipo abstrato de dado.
  - Altura da árvore.
  - Topologia binária.

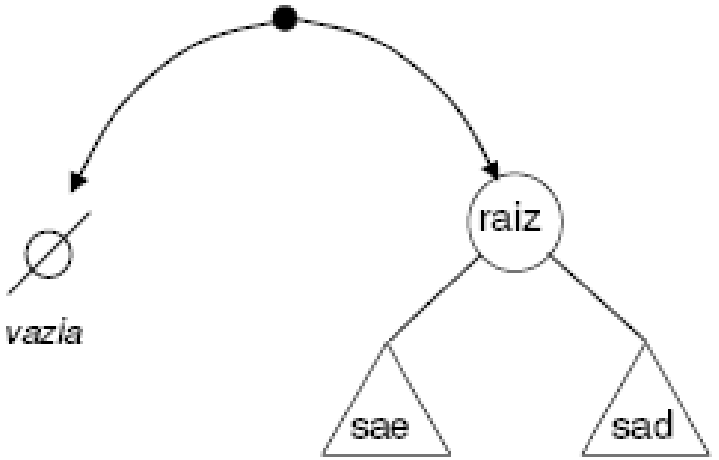
Árvores Binárias

- Árvore
  - Um conjunto de nós tal que:
    - Existe um nó r, denominado raiz, com zero ou mais sub-árvores, cujas raízes estão ligadas a r.
    - Os nós raízes destas sub-árvores são os filhos de r.
    - Os nós internos da árvore são os nós com filhos.
    - As folhas ou nós externos da árvore são os nós sem filhos.



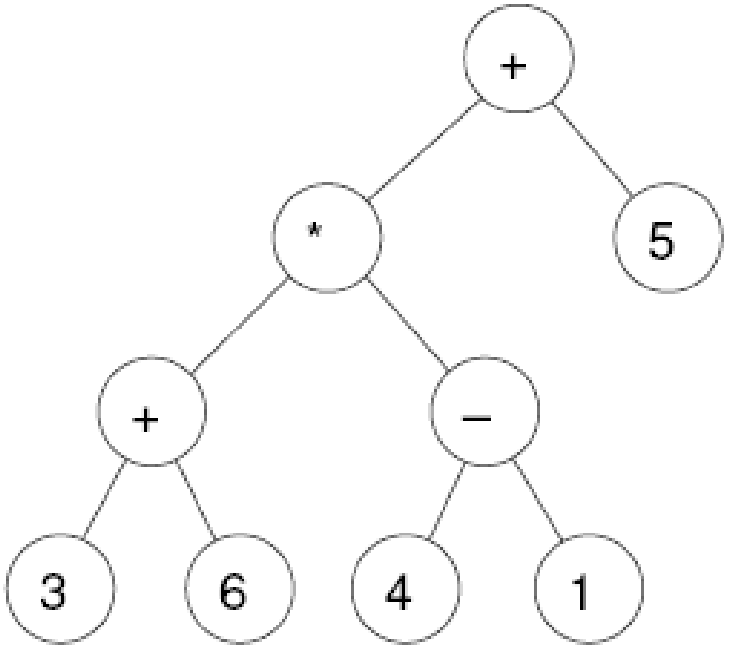
Árvores Binárias

- Árvore binária
  - Uma árvore em que cada nós tem zero, um ou dois filhos.
  - Uma árvore binária é:
    - Uma árvore vazia; ou
    - Um nós raiz com duas sub-árvores:
      - A sub-árvore da direita (SAD).
      - A sub-árvore da esquerda (SAE).



Árvores Binárias

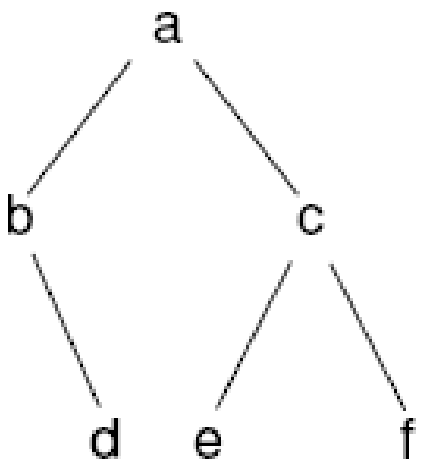
- Exemplo
  - Árvores binárias representando expressões aritméticas:
    - Nós folhas representando operandos.
    - Nós internos operadores.
    - Exemplo: (3 + 6) * (4 – 1) + 5



Árvores Binárias

- Notação contextual
  - A árvore principal é representada por < >
  - Árvores não vazias por <raiz sae sad>
  - Exemplo:

<a <b <> <d<><>> > <c <e<><>> <f<><>>> >



Árvores Binárias – Implementação em C

- Representação de uma árvore:
  - Através de um ponteiro para o nó raiz.
- Representação de um nó da árvore:
  - Estrutura em C contendo:
    - A informação propriamente dita (exemplo: um caractere).
    - Dois ponteiros para as sub-árvores, à esquerda e à direita.

```
struct arv {
   char info;
   struct arv* esq;
   struct arv* dir;
};
```

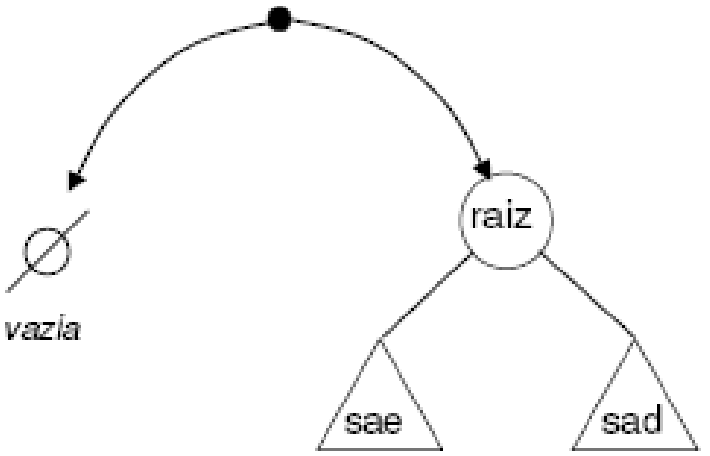
Árvores Binárias – Implementação em C

- Interface do tipo abstrato Árvore Binária:arv.h

```
typedef struct arv Arv;

Arv* arv_criavazia (void);
Arv* arv_cria (char c, Arv* e, Arv* d);
Arv* arv_libera (Arv* a);
int  arv_vazia (Arv* a);
int  arv_pertence (Arv* a, char c);
void arv_imprime (Arv* a);
```

Árvores Binárias – Implementação em C

- Implementação das funções:
  - Implementação recursiva, em geral.
  - Usa a definição recursiva da estrutura.
  - Uma árvore binária é:
    - Uma árvore vazia; ou
    - Um nó raiz com duas sub-árvores:
      - A sub-árvore da direita (sad).
      - A sub-árvore da esquerda (sae).



Árvores Binárias – Implementação em C

- Função arv_criavazia
  - Cria uma árvore vazia.

```
Arv* arv_criavazia (void)
{
   return NULL;
}
```

Árvores Binárias – Implementação em C

- Função arv_cria
  - Cria um nó raiz dadas a informação e as duas sub-árvores, a da esquerda e a da direita.
  - Retorna o endereço do nó raiz criado.

```
Arv* arv_cria (char c, Arv* sae, Arv* sad)
{
   Arv* p=(Arv*)malloc(sizeof(Arv));
   p->info = c;
   p->esq = sae;
   p->dir = sad;
   return p;
}
```

Árvores Binárias – Implementação em C

- arv_criavazia e arv_cria
  - As duas funções para a criação de árvores representam os dois casos da definição recursiva de árvore binária:
    - Uma árvore binária Arv* a;
      - É vazia a = arv_criavazia( );
      - É composta por uma raiz e duas sub-árvores a = arv_cria (c, sae, sad);

Árvores Binárias – Implementação em C

- Função arv_libera
  - Libera memória alocada pela estrutura da árvore.
    - As sub-árvores devem ser liberadas antes de se liberar o nó raiz.
    - Retorna uma árvore vazia, representada por NULL.

```
Arv* arv_libera (Arv* a){
   if (!arv_vazia(a)){
      arv_libera(a->esq);      /* libera sae */
      arv_libera(a->dir);      /* libera sad */
      free(a);                 /* libera raiz */
   }
   return NULL;
}
```

Árvores Binárias – Implementação em C

- Função arv_vazia
  - Indica se uma árvore é ou não vazia.

```
int arv_vazia (Arv* a)
{
   return a==NULL;
}
```

Árvores Binárias – Implementação em C

- Função arv_pertence
  - Verifica a ocorrência de um caractere c em um dos nós.
  - Retorna um valor booleano indicando a ocorrência ou não do caractere na árvore.

```
int arv_pertence (Arv* a, char c){
   if (arv_vazia(a))
      return 0;                    /* árvore vazia: não encontrou */
   else
      return a->info==c ||
             arv_pertence(a->esq,c) ||
             arv_pertence(a->dir,c);
}
```

Árvores Binárias – Implementação em C

- Função arv_imprime
  - Percorre recursivamente a árvore, visitando todos os nós e imprimindo sua informação.

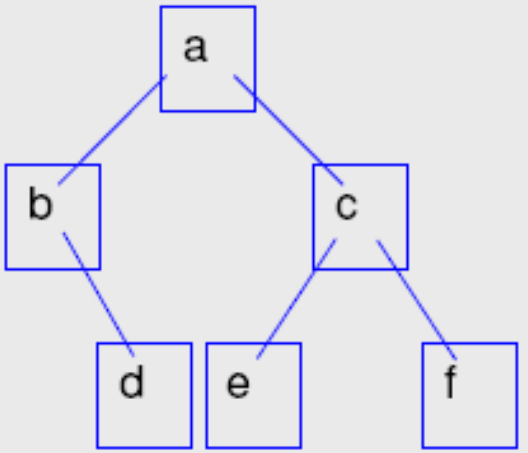
```
void arv_imprime (Arv* a)
{
   if (!arv_vazia(a)){
      printf("%c ", a->info);          /* mostra raiz */
      arv_imprime(a->esq);             /* mostra sae */
      arv_imprime(a->dir);             /* mostra sad */
   }
}
```

Árvores Binárias – Implementação em C

- Exemplo:

<a <b <> <d <><>> > <c <e <><> > <f <><> > > >

```
/* sub-árvore 'd' */
Arv* a1= arv_cria('d',arv_criavazia(),arv_criavazia());
/* sub-árvore 'b' */
Arv* a2= arv_cria('b',arv_criavazia(),a1);
/* sub-árvore 'e' */
Arv* a3= arv_cria('e',arv_criavazia(),arv_criavazia());
/* sub-árvore 'f' */
Arv* a4= arv_cria('f',arv_criavazia(),arv_criavazia());
/* sub-árvore 'c' */
Arv* a5= arv_cria('c',a3,a4);
/* árvore 'a' */
Arv* a = arv_cria('a',a2,a5 );
```

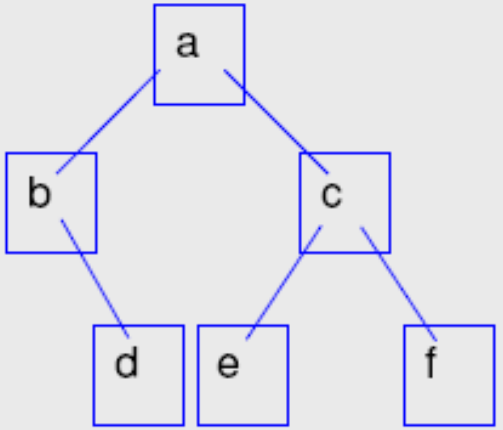


Árvores Binárias – Implementação em C

- Exemplo:

<a <b <> <d <><>> > <c <e <><> > <f <><> > > >

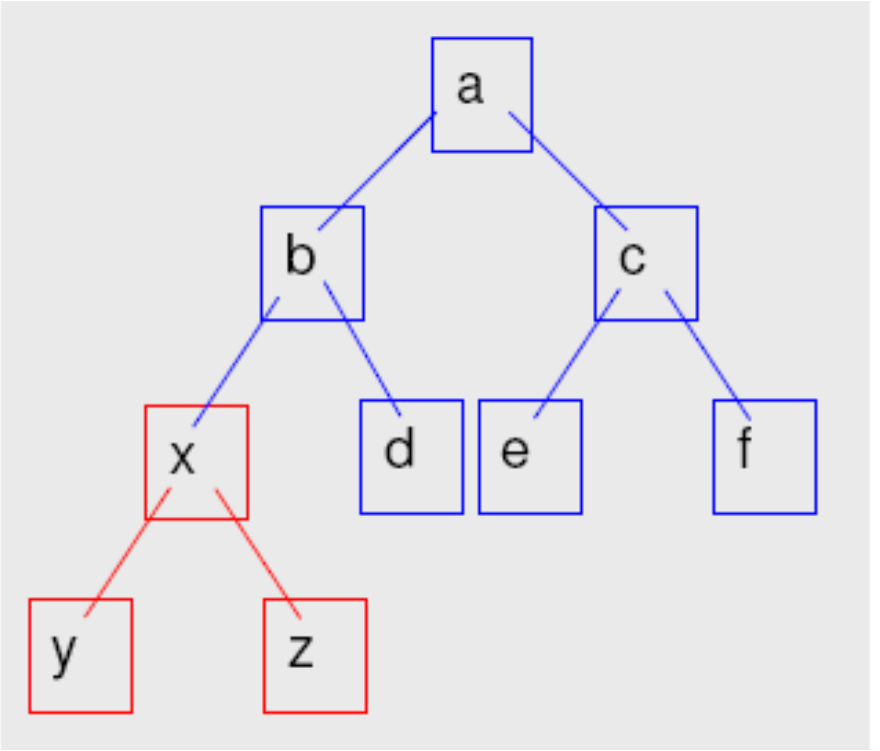
```
Arv* a = arv_cria('a',
           arv_cria('b',
               arv_criavazia(),
               arv_cria('d', arv_criavazia(), arv_criavazia())
           ),
           arv_cria('c',
               arv_cria('e', arv_criavazia(), arv_criavazia()),
               arv_cria('f', arv_criavazia(), arv_criavazia())
           )
         );
```



Árvores Binárias – Implementação em C

- Exemplo – acrescenta nós.

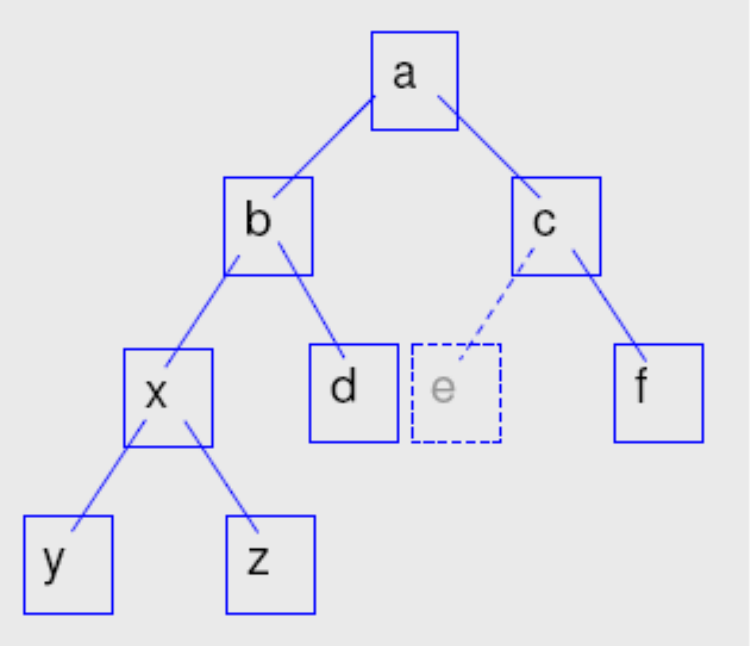
```
a->esq->esq =
   arv_cria('x',
            arv_cria('y',
                     arv_criavazia(),
                     arv_criavazia()),
            arv_cria('z',
                     arv_criavazia(),
                     arv_criavazia())
           );
```



Árvores Binárias – Implementação em C

- Exemplo – libera nós.

```
a->dir->esq =
            arv_libera(a->dir->esq);
```

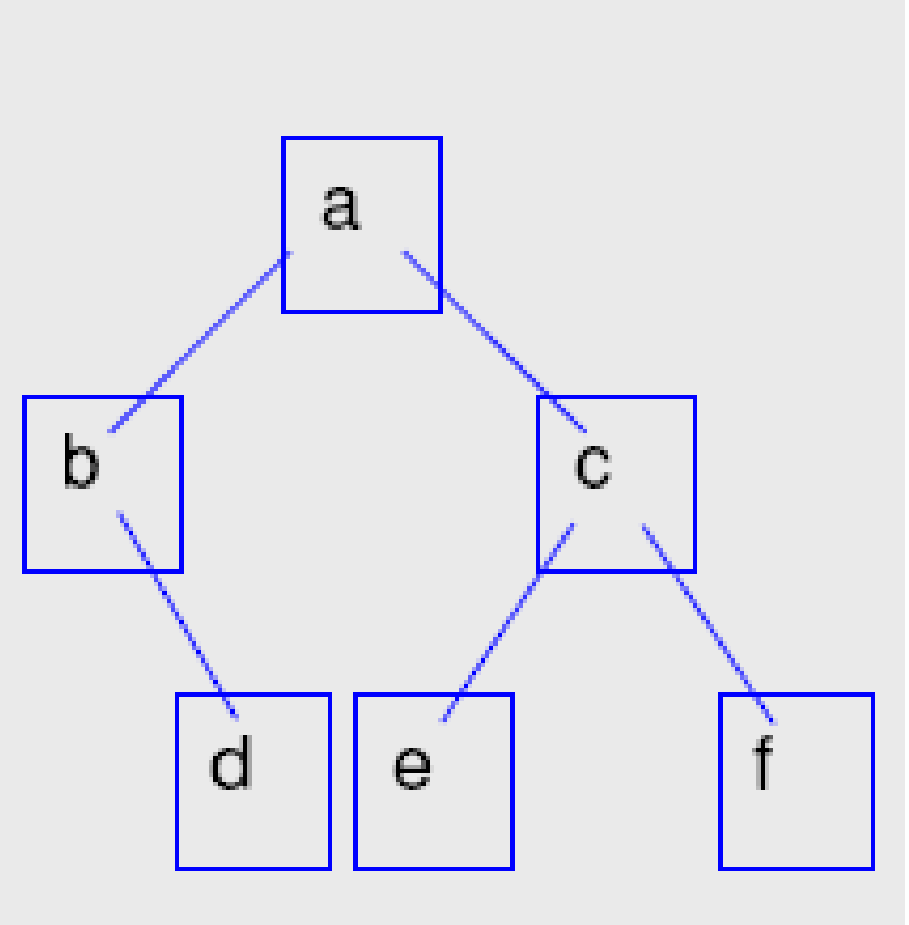


Árvores Binárias – Implementação em C

- Ordens de percurso:
  - Pré-ordem:
    - Visita a raiz, percorre sae e percorre sad.
  - Ordem simétrica:
    - Percorre sae, visita raiz e percorre sad.
  - Pós-ordem:
    - Percorre sae, percorre sad e visita raiz.

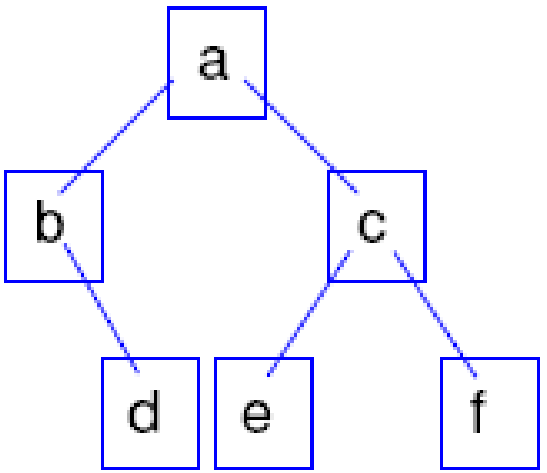
Árvores Binárias – Implementação em C

- Exemplo:
  - Pré-ordem:
    - Exemplo: a b d c e f.
  - Ordem simétrica:
    - Exemplo: b d a e c f.
  - Pós-ordem:
    - Exemplo: d b e f c a.



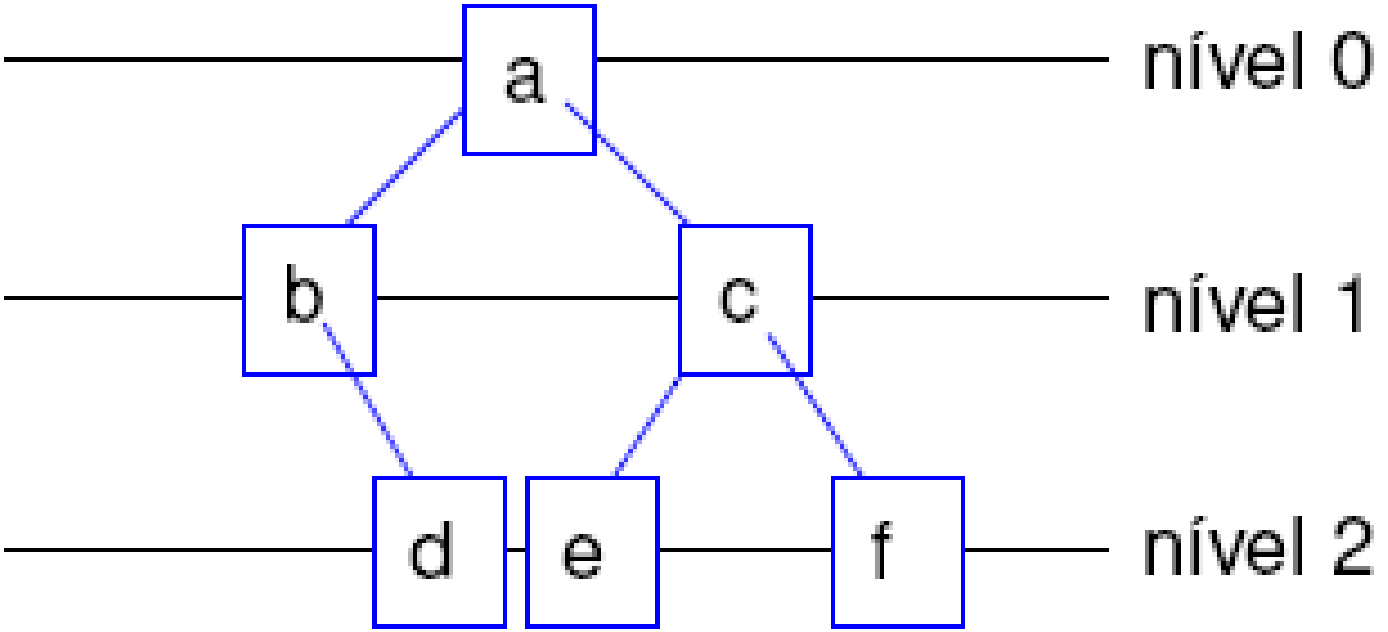
Árvores Binárias – altura

- Propriedade fundamental de árvores:
  - Só existe um caminho da raiz para qualquer nó.
- Altura de uma árvore:
  - Comprimento do caminho mais longo da raiz até uma das folhas.
    - A altura de uma árvore com um único nó raiz é zero.
    - A altura de uma árvore vazia é -1.
    - Exemplo: h = 2.



Árvores Binárias – altura

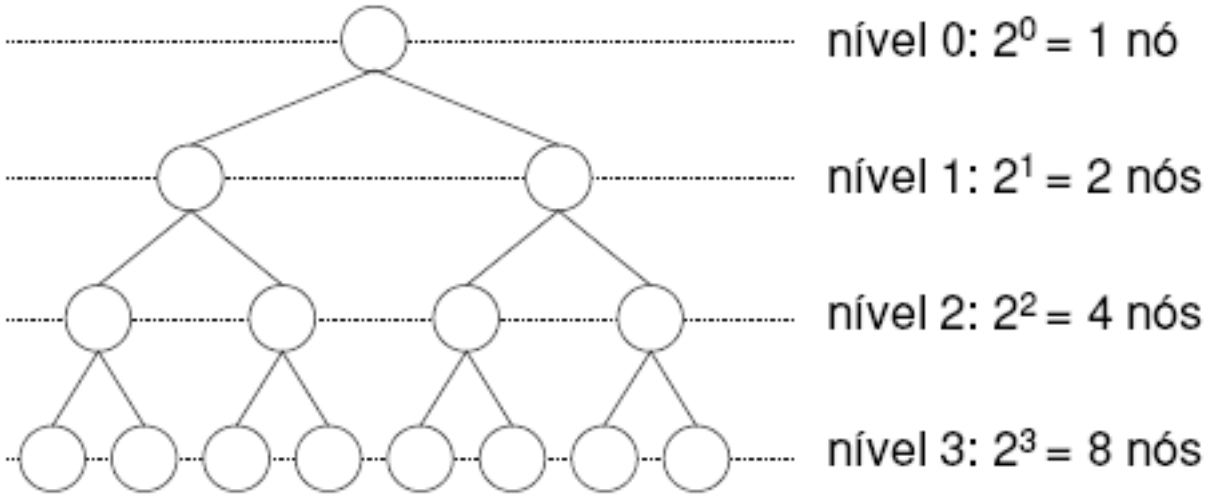
- Nível de um nó.
  - A raiz está no nível 0, seus filhos diretos no nível 1, ...
  - O último nível da árvore é a altura da árvore.



Árvores Binárias - altura

- Árvore cheia
  - Todos os seus nós internos têm duas sub-árvores associadas.
  - Número n de nós de uma árvore cheia de altura h:

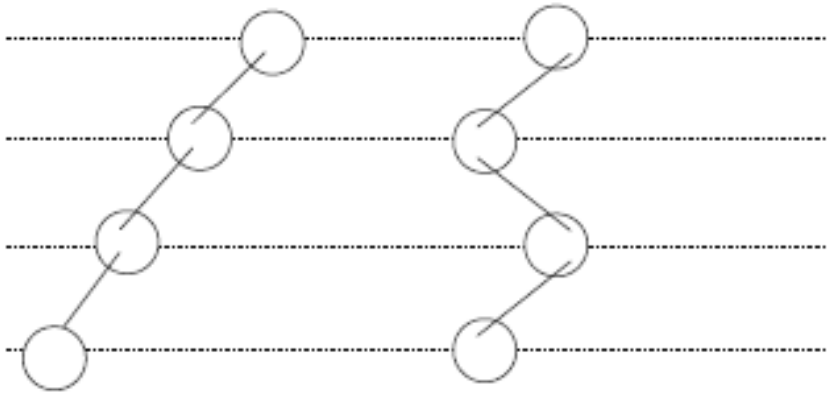
$$n = 2^{h+1} - 1$$



Árvores Binárias - altura

- Árvore degenerada
  - Todos os seus nós internos têm uma única sub-árvore associada.
  - Número n de nós de uma árvore degenerada de altura h:

$$n = h+1$$

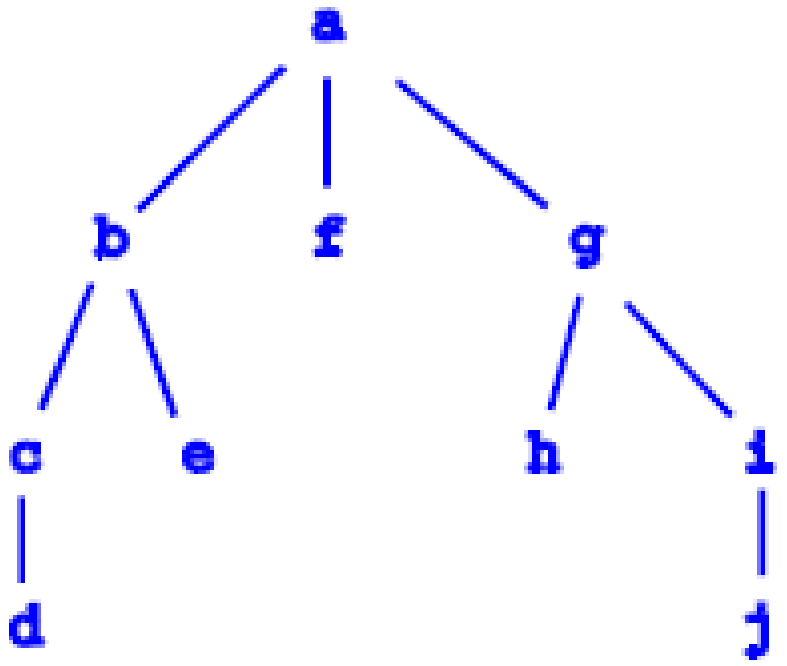


Árvores Binárias - altura

- Esforço computacional necessário para alcançar qualquer nó da árvore.
  - Proporcional à altura da árvore.
  - Altura de uma árvore binária com n nós.
    - Mínima → proporcional a log n (caso da árvore cheia).
    - Máxima → proporcional a n (caso a árvore degenerada).

Árvores com número variável de filhos

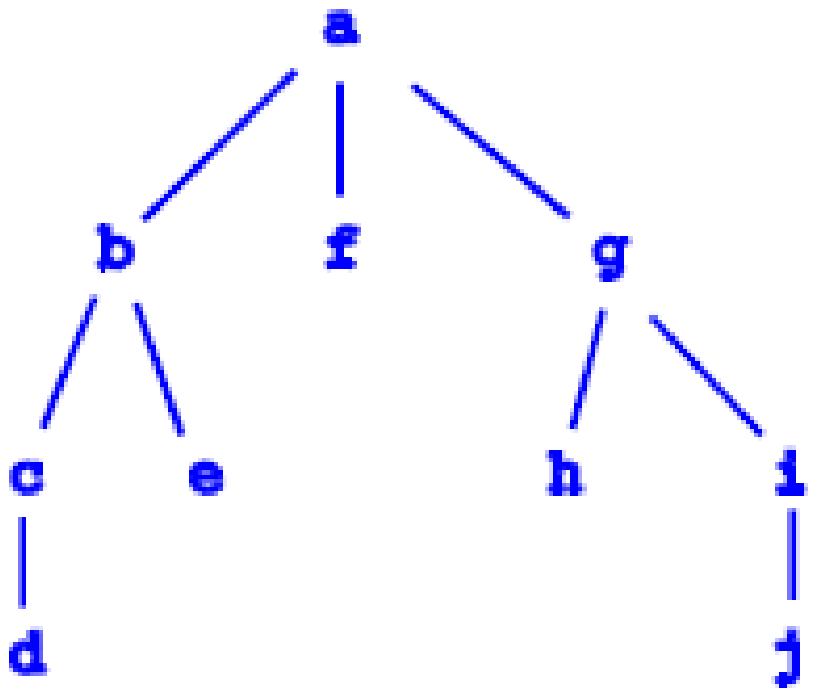
- Cada nó pode ter mais do que duas sub-árvores associadas.
- Sub-árvores de um nó dispostas em ordem:
  - Primeira sub-árvore (sa1), segunda sub-árvore (sa2), etc.



Árvores com número variável de filhos

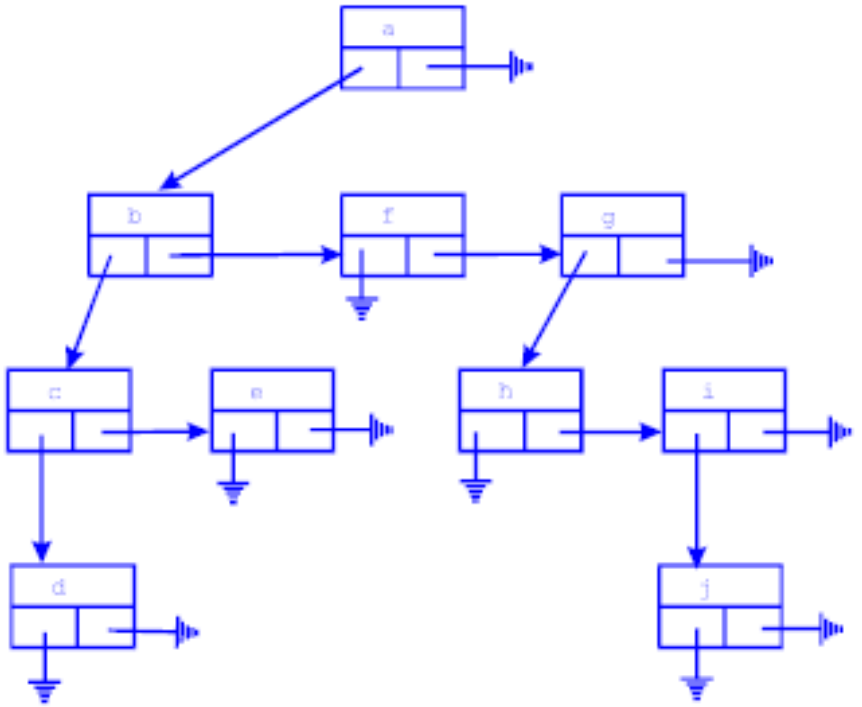
- Notação textual: <raiz sa1 sa2 ... san>
- Exemplo:

α = <a <b <c <d> > <e> > <f> <g <h> <i <j> > > >



Árvores com número variável de filhos – representação em C

- Representação de árvore com número variável de filhos:
  - Utiliza uma lista de filhos:
    - Um nó aponta apenas para seu primeiro (prim) filho.
    - Cada um de seus filhos aponta para o próximo (prox) irmão.



Árvores com número variável de filhos – representação em C

- Representação de um nó da árvore:
  - Estrutura em C contendo:
    - A informação propriamente dita (exemplo: um caractere).
    - Ponteiro para a primeira sub-árvore filha.
      - NULL se o nó for de uma folha.
    - Ponteiro para a próxima sub-árvore irmão.
      - NULL se for o último filho.

```
struct arvvar {
   char info;
   struct arvvar *prim;        /* ponteiro para eventual primeiro filho */
   struct arvvar *prox;        /* ponteiro para eventual irmão */
};

typedef struct arvvar ArvVar;
```

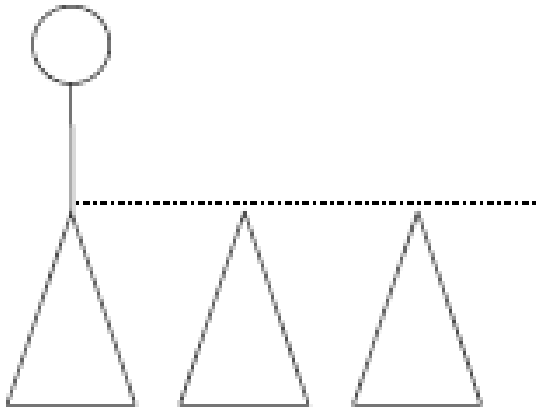
Árvores com número variável de filhos – representação em C

- Interface do tipo abstrato Árvore Variável: arrvar.h

```
typedef struct arvvar ArvVar;

ArvVar* arvv_cria (char c);
void    arvv_insere (ArvVar* a, ArvVar* sa);
void    arvv_imprime (ArvVar* a);
int     arvv_pertence (ArvVar* a, char c);
void    arvv_libera (ArvVar* a);
```

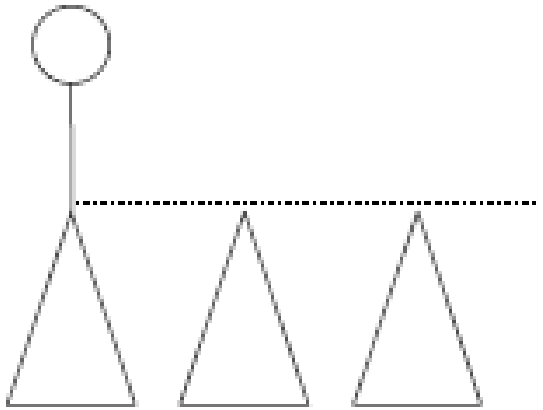
Árvores com número variável de filhos – representação em C

- Implementação das funções:
  - Implementação recursiva, em geral.
  - Usa a definição recursiva da estrutura.
  - Uma árvore é composta por:
    - Um nó raiz.
    - Zero ou mais sub-árvores.



Árvores com número variável de filhos – representação em C

- Implementação das funções (cont.):
  - Uma árvore não pode ser vazia.
  - Uma folha é identificada como um nó com zero sub-árvores.
    - Uma folha não é um nó com sub-árvores vazias, como nas árvores binárias.
  - Funções não consideram o caso de árvores vazias.



Árvores com número variável de filhos – representação em C

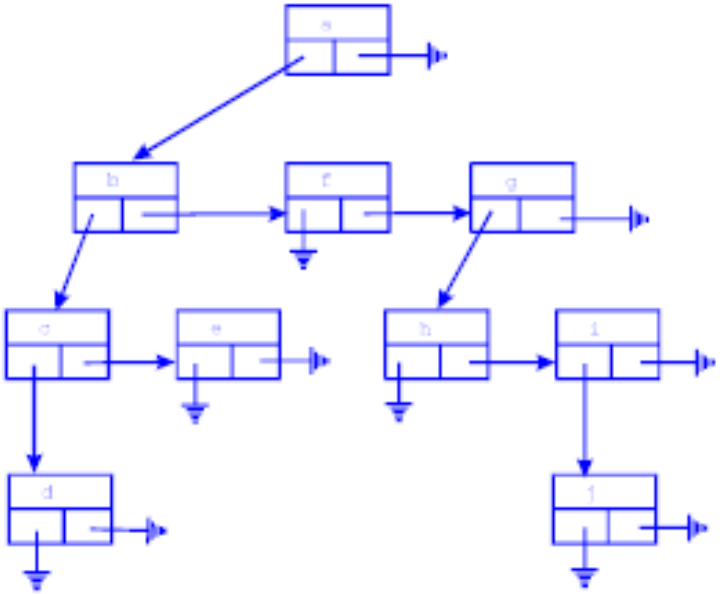
- Função arvv_cria
  - Cria uma folha.
    - Aloca o nó.
    - Inicializa os campos, atribuindo NULL aos campos prim e prox.

```
ArvVar* arvv_cria (char c)
{
   ArvVar *a =(ArvVar *) malloc(sizeof(ArvVar));
   a->info = c;
   a->prim = NULL;
   a->prox = NULL;
   return a;
}
```

Árvores com número variável de filhos – representação em C

- Função arvv_insere
  - Insere uma nova sub-árvore como filha de um dado, sempre no início da lista, por simplicidade.

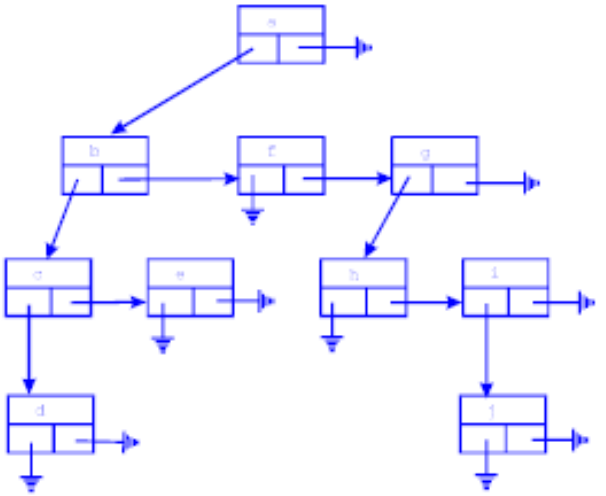
```
void arvv_insere (ArvVar* a, ArvVar* sa)
{
   sa->prox = a->prim;
   a->prim = sa;
}
```



Árvores com número variável de filhos – representação em C

- Função arvv_imprime
  - Imprime conteúdo dos nós em pré-ordem.

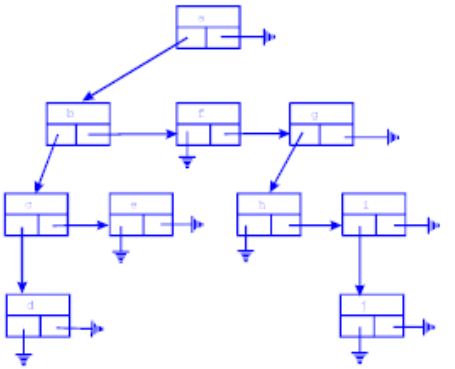
```
void arvv_imprime (ArvVar* a)
{
   ArvVar* p;
   printf("<%c\n",a->info);
   for (p=a->prim; p!=NULL; p=p->prox)
      arvv_imprime(p);   /* imprime filhas */
   printf(">");
}
```



Árvores com número variável de filhos – representação em C

- Função arvv_pertence
  - Verifica a ocorrência de uma dada informação na árvore.

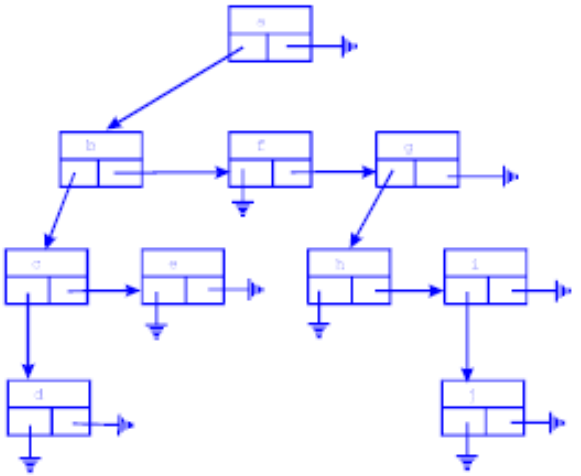
```
int arvv_pertence (ArvVar* a, char c)
{
   ArvVar* p;
   if (a->info==c)
      return 1;
   else {
      for (p=a->prim; p!=NULL; p=p->prox) {
         if (arvv_pertence(p,c))
            return 1;
      }
      return 0;
   }
```



Árvores com número variável de filhos – representação em C

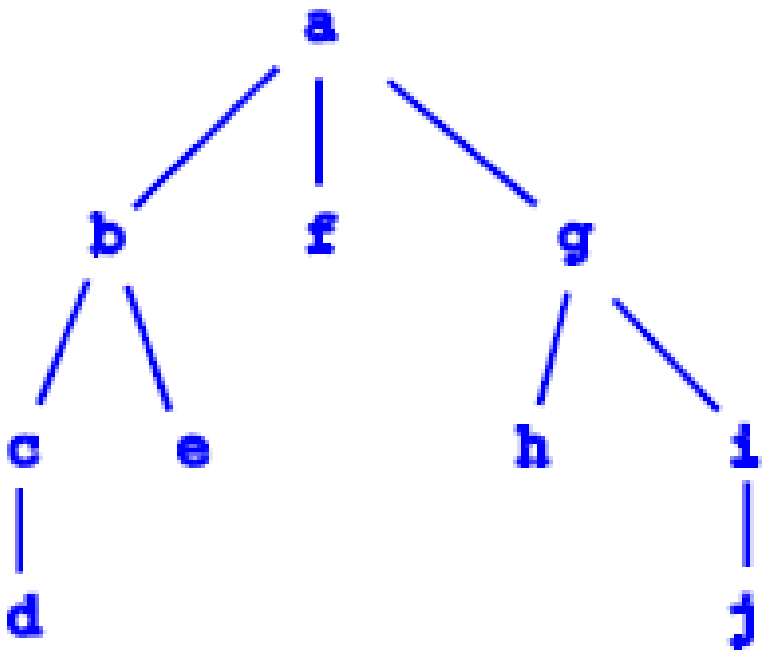
- Função arvv_libera
  - Libera a memória alocada pela árvore.
  - Libera as sub-árvores antes de liberar o espaço associado a um nó (libera em pós-ordem).

```
void arvv_libera (ArvVar* a)
{
   ArvVar* p = a->prim;
   while (p!=NULL) {
      ArvVar* t = p->prox;
      arvv_libera(p);
      p = t;
   }
   free(a);
}
```



Árvores com número variável de filhos – altura

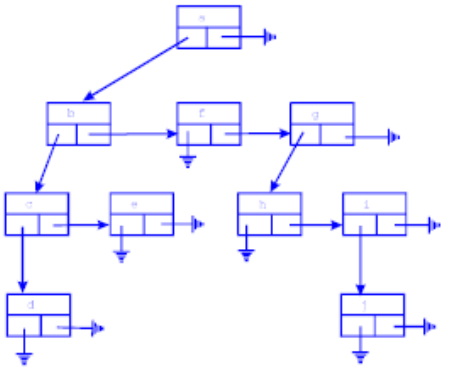
- Nível e altura.
  - Definidos de forma semelhante a árvores binárias.
  - Exemplo: h = 3.



Árvores com número variável de filhos – altura

- Função arvv_altura
  - Maior altura entre as sub-árvores, acrescido de uma unidade.
  - Caso o nó raiz não tenha filhos, a altura da árvore deve ser 0.

```
int arvv_altura (ArvVar* a)
{
   int hmax = -1;  /* -1 para arv. sem filhos */
   ArvVar* p;
   for (p=a->prim; p!=NULL; p=p->prox) {
      int h = arvv_altura(p);
      if (h > hmax)
          hmax = h;
   }
   return hmax + 1;
}
```



Árvores com número variável de filhos – topologia binária

- Topologia binária.
  - Representação de um nó de uma árvore variável é equivalente a representação de um nó da árvore binária.
  - Nó possui informação e dois ponteiros para sub-árvores.
    - Árvore binária:
      - Ponteiros para as sub-árvores à esquerda e à direita.
    - Árvore variável:
      - Ponteiros para a primeira sub-árvore filha e para a sub-árvore irmã.

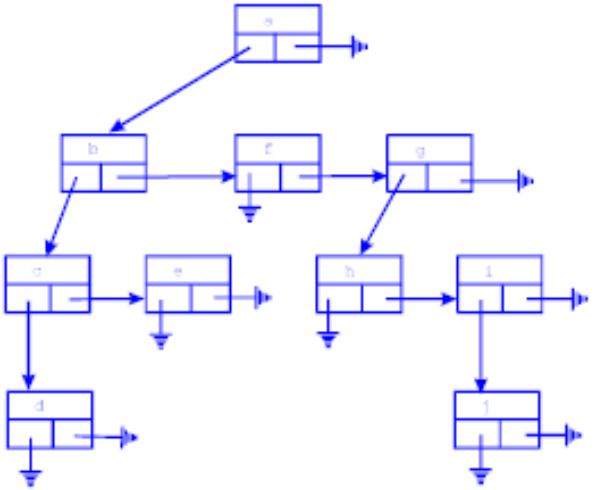
Árvores com número variável de filhos – topologia binária

- Topologia binária
  - Redefinição de árvore com número variável de filhos:
    - Árvore vazia, ou
    - Um nó raiz tendo duas sub-árvores, identificadas como a sub-árvore filha e a sub-árvore irmã.
  - Re-implementação das funções:
    - Pode se basear na nova definição.
    - O caso da árvore vazia agora deve ser considerado.

Árvores com número variável de filhos – topologia binária

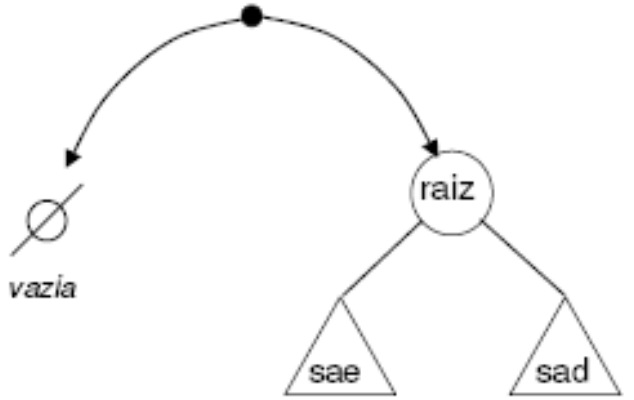
- Função para calcular altura:
  - A altura da sub-árvore filha acrescido de uma unidade e a altura da sub-árvore irmã.

```
static max2 (int a, int b)
{ return (a > b) ? a : b; }

int arvv_altura (ArvVar* a)
{  if (a==NULL)
      return -1;
   else
      return max2(1+arvv_altura(a->prim),
                  arvv_altura(a->prox));
}
```

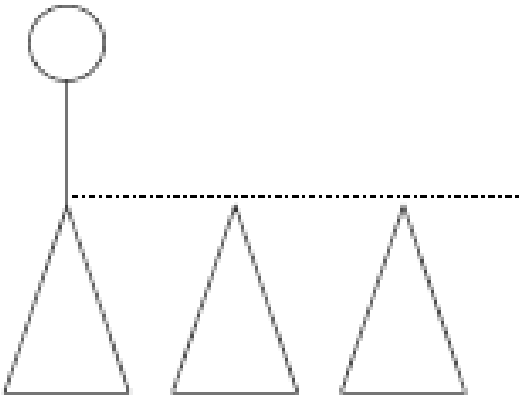


Resumo

- Árvore binária
  - Uma árvore vazia; ou
  - Um nó raiz com duas sub-árvores:
    - A sub-árvore da direita (sad).
    - A sub-árvore da esquerda (sed).



- Árvore com número variável de filhos
  - Um nó raiz.
  - Zero ou mais sub-árvores.



Exercícios

- Implemente uma função que retorne a quantidade de folhas de uma árvore binária. Essa função deve obedecer ao seguinte protótipo:

```
int folhas (Arv* a);
```

- Implemente uma função que compare se duas árvores binárias são iguais. Essa função deve obedecer ao seguinte protótipo:

```
Arv* igual (Arv* a, Arv* b);
```

Exercícios

- Implemente uma função que retorne a quantidade de folhas de uma árvore com número variável de filhos. Essa função deve obedecer ao protótipo:

```
int folhas (ArvVar* a);
```

Referências bibliográficas.

- Estrutura de Dados (PUC-RJ)
  - http://www.inf.puc-rio.br/~inf1620