Estrutura de Dados

Claudio Cesar de Sá, Alessandro Ferreira Leite, Lucas Hermman Negri, Gilmário Barbosa

Departamento de Ciência da Computação
Centro de Ciências e Tecnológias
Universidade do Estado de Santa Catarina

24 de julho de 2017

Sumário (1)

Sumário (2)

Sumário (3)

Agradecimentos

Vários autores e colaboradores ...

- Ao Google Images ...

Disciplina

Estrutura de Dados – EDA001

- ***Turma:***
- ***Professor:*** Claudio Cesar de Sá
  - `claudio.sa@udesc.br`
  - Sala 13 Bloco F
- ***Carga horária:*** 72 horas-aula • Teóricas: 36 • Práticas: 36
- ***Curso:*** BCC
- ***Requisitos:*** LPG, Linux, LMA,
- ***Período:*** 2º semestre de 2017
- ***Horários:***
  - 3ª 15h20 (2 aulas) - F-205 – aula expositiva
  - 5ª 15h20 (2 aulas) - F-205 – lab

Ementa

Ementa

Representação e manipulação de tipos abstratos de dados. Estruturas lineares. Introdução a estruturas hierárquicas. Métodos de classificação. Análise de eficiência. Aplicações.

Objetivos (1)

- ***Geral:***

Objetivos (2)

- ***Específicos:***
  - Descrever o histórico e quadro atual da Inteligência Artificial – Moderna.
  - Compreender a noção de Teoria de Problemas, computabilidade e complexidade na ótica de IA e IAD.
  - Diferencia IAD (orientação a divisão de problemas) versus SMA (orientação a coordenação de agentes)
  - Conhecer diferentes arquiteturas de agentes
  - Modelar problemas computacionais através de aplicação de agentes.
  - Descrever o processo de tomada de decisão e aprendizagem computacional baseado em Estrutura de Dados.
  - Conceber, projetar e construir sistemas computacionais capazes de aplicar sistemas multiagentes como técnica de resolução.

Conteúdo programático

Ferramentas

- Linguagem C
- Codeblock
- Linux

Metodologia e avaliação (1)

Metodologia:

As aulas serão expositivas e práticas. A cada novo assunto tratado, exemplos são demonstrados utilizando ferramentas computacionais adequadas para consolidar os conceitos tratados.

Metodologia e avaliação (2)

Avaliação

- Tres provas – $\approx$ 90%
  - P_1: xx/set
  - P_2: yy/out
  - P_2: zz/nov

  (provão: todo conteúdo)
- Exercícios de laboratório – $\approx$ %
- Presença e participação
- Média para aprovação: 6,0 (seis)
  Nota maior ou igual a 6,0, repito a mesma no Exame Final. Caso contrário, regras da UDESC

Dinâmica de Aula

- Teoria na 3a. feira
- Prática na 5a. feira
- E/ou 50% do tempo em teoria, 50% implementações

Bibliografia (1)

Básica:

-
- https://github.com/claudiosa/CCS/tree/master/estrutura_dados_EDA

Complementar:

-

Capítulo xxxxx – Pilha

Pontos fundamentais a serem cobertos:

1. Contexto e motivação
2. Definição
3. Implementações
4. Exercícios

Introdução

- Uma das estruturas de dados mais simples.
- É a estrutura de dados mais utilizada em programação.
- É uma metáfora emprestada do mundo real, que a computação utiliza para resolver muitos problemas de forma simplificada.

Definição

Definição

Um conjunto ordenado de itens no qual novos itens podem ser inseridos e a partir do qual podem ser eliminados em uma extremidade denominada topo da pilha.

Definição

Definição

Um conjunto ordenado de itens no qual novos itens podem ser inseridos e a partir do qual podem ser eliminados em uma extremidade denominada topo da pilha.

Definição

Uma seqüência de objetos, todos do mesmo tipo, sujeita às seguintes regras de comportamento:

1. Sempre que solicitado a remoção de um elemento, o elemento removido é o último da seqüência.
2. Sempre que solicitado a inserção de um novo elemento, o objeto é inserido no fim da seqüência (topo).

Pilha

- Uma pilha é um objeto dinâmico, constantemente mutável, onde elementos são inseridos e removidos.
- Em uma pilha, cada novo elemento é inserido no topo.
- Os elementos da pilha só podem ser retirado na ordem inversa à ordem em que foram inseridos
  - O primeiro que sai é o último que entrou.
  - Por essa razão, uma pilha é dita uma estrutura do tipo: LIFO(*last-in, first* ou UEPS último a entrar é o primeiro a sair.)

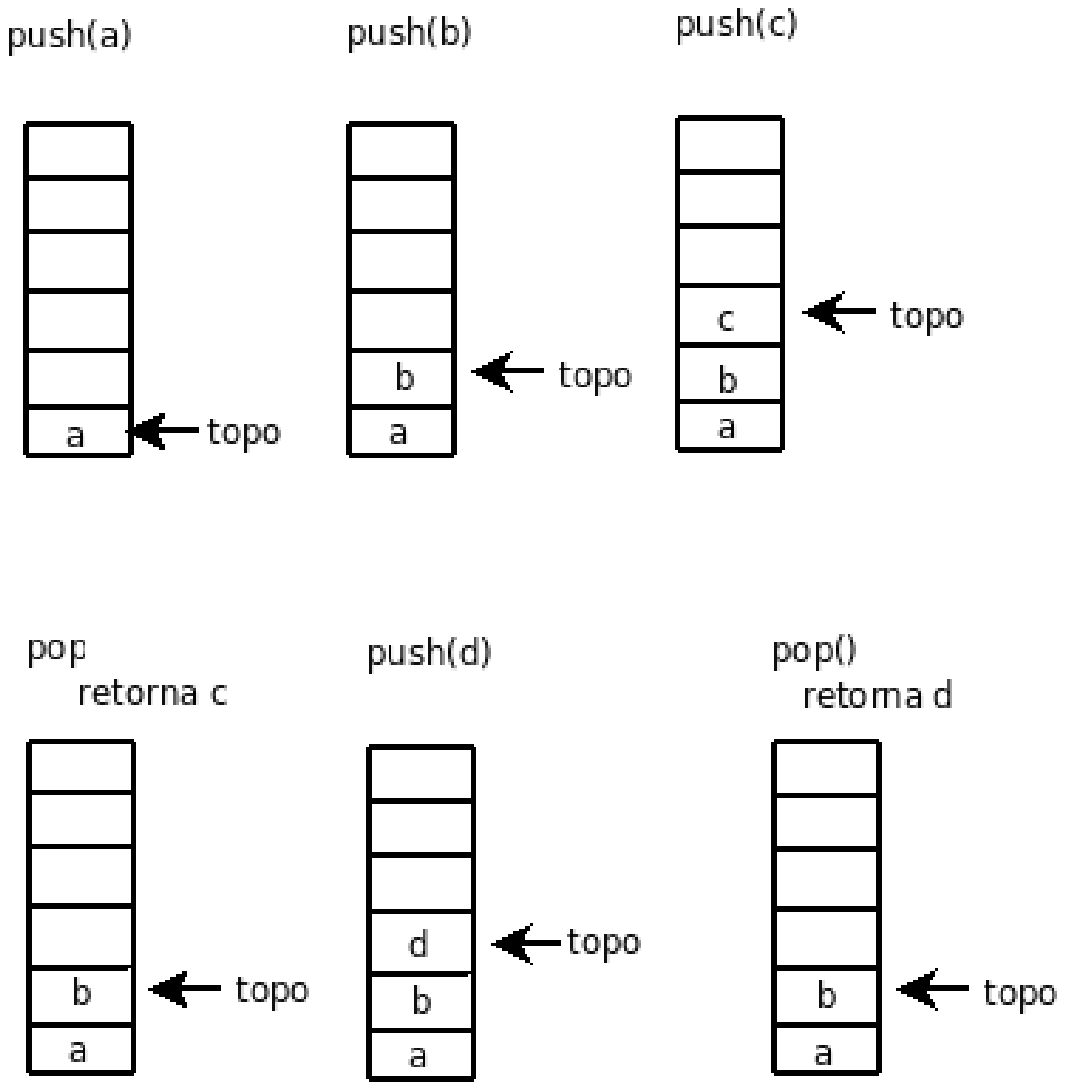
Operações básicas

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

Operação	**Descrição**
push(p, e)	empilha o elemento *e*, inserindo-o no topo da pilha *p*.
pop(p)	desempilha o elemento do topo da pilha *p*.

Tabela: Operações básicas da estrutura de dados pilha.

Exemplo


push(a)
a
topo
push(b)
b
a
topo
push(c)
c
b
a
topo
pop
retorna c
b
a
topo
push(d)
d
b
a
topo
pop()
retoma d
b
a
topo

Operações auxiliares

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

Operação	**Descrição**
create	cria uma pilha vazia.
empty(p)	determina se uma pilha p está ou não vazia.
free(p)	libera o espaço ocupado na memória pela pilha p.

Tabela: Operações auxiliares da estrutura de dados pilha.

Interface do Tipo Pilha

```c
/* Definicao da estrutura */
typedef struct pilha Pilha;
/*Aloca dinamicamente a estrutura pilha, inicializando
 *seus campos e retorna seu ponteiro.*/
Pilha* create(void);

/*Insere o elemento e na pilha p.*/
void push(Pilha *p, int e);

/*Retira e retorna o elemento do topo da pilha p*/
int pop(Pilha *p);

/*Informa se a pilha p esta ou nao vazia.*/
int empty(Pilha *p);
```

Implementação de Pilha com Vetor

- Normalmente as aplicações que precisam de uma estrutura pilha, é comum saber de antemão o número máximo de elementos que precisam estar armazenados simultaneamente na pilha.
- Essa estrutura de pilha tem um limite conhecido.
- Os elementos são armazenados em um vetor.
- Essa implementação é mais simples.
- Os elementos inseridos ocupam as primeiras posições do vetor.

Implementação de Pilha com Vetor

- Seja p uma pilha armazenada em um vetor VET de N elementos:
  1. O elemento vet[topo] representa o elemento do topo.
  2. A parte ocupada pela pilha é vet[0 .. topo - 1].
  3. A pilha está vazia se topo = -1.
  4. Cheia se topo = N - 1.
  5. Para desempilhar um elemento da pilha, não vazia, basta

$$x = vet[topo - -]$$

.

  6. Para empilhar um elemento na pilha, em uma pilha não cheia, basta

$$vet[t + +] = e$$

.

Implementação de Pilha com Vetor

```c
#define N 20 /* numero maximo de elementos*/
#include <stdio.h>
#include "pilha.h"

/*Define a estrutura da pilha*/
struct pilha{
  int topo; /* indica o topo da pilha */
  int elementos[N]; /* elementos da pilha*/
};

Pilha* create(void){
  Pilha* p = (Pilha*) malloc(sizeof(Pilha));
  p->topo = -1; /* inicializa a pilha com 0 elementos */
  return p;
}
```

Implementação de Pilha com Vetor

- Empilha um elemento na pilha

```
void push(Pilha *p, int e){
     if (p->topo == N - 1){ /* capacidade esgotada */
         printf("A pilha esta cheia");
         exit(1);
     }
     /* insere o elemento na proxima posicao livre */
     p->elementos[++p->topo] = e;
}
```

Implementação de Pilha com Vetor

- Desempilha um elemento da pilha

```
int pop(Pilha *p)
{
      int e;
      if (empty(p)){
         printf("Pilha vazia.\n");
         exit(1);
      }

      /* retira o elemento do topo */
      e = p->elementos[p->topo--];
      return e;
}
```

Implementação de Pilha com Vetor

```
/**
 * Verifica se a pilha p esta vazia
 */
int empty(Pilha *p)
{
    return (p->t == -1);
}
```

Exemplo de uso

- Na área computacional existem diversas aplicações de pilhas.
- Alguns exemplos são: caminhamento em árvores, chamadas de sub-rotinas por um compilador ou pelo sistema operacional, inversão de uma lista, avaliar expressões, entre outras.
- Uma das aplicações clássicas é a conversão e a avaliação de expressões algébricas. Um exemplo, é o funcionamento das calculadoras da HP, que trabalham com expressões pós-fixadas.

Capítulo xxxxx – Filas

Pontos fundamentais a serem cobertos:

1. Contexto e motivação
2. Definição
3. Implementações
4. Exercícios

Introdução

- Assim como a estrutura de dados Pilha, Fila é outra estrutura de dados bastante utilizada em computação.
- Um exemplo é a implementação de uma fila de impressão.
- Se uma impressora é compartilhada por várias máquinas, normalmente adota-se uma estratégia para determinar a ordem de impressão dos documentos.
- A maneira mais simples é tratar todas as requisições com a mesma prioridade e imprimir os documentos na ordem em que foram submetidos - o primeiro submetido é o primeiro a ser impresso.

Fila

Definição

Um conjunto ordenado de itens a partir do qual podem-se eliminar itens numa extremidade (chamada de início da fila) e no qual podem-se inserir itens na outra extremidade (chamada final da fila).

Representação

- Os nós de uma fila são armazenados em endereços contínuos.
- A Figura 1 ilustra uma fila com três elementos.

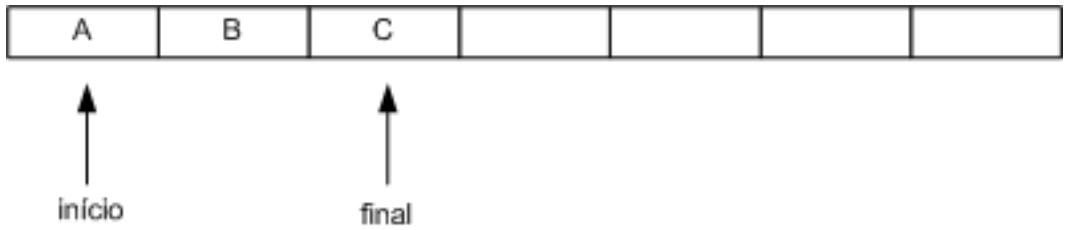


Figura: Exemplo de representação de fila.

- Após a retirada de um elemento (*primeiro*) temos:

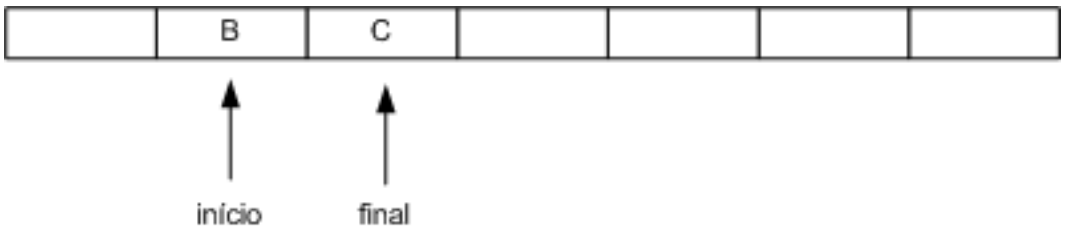


Figura: Representação de uma fila após a remoção do elemento "A".

Representação

- Após a inclusão de dois elementos temos:

Figura: Representação de uma fila após a inclusão de dois elementos “D” e “E”.

- Como podemos observar, a operação de inclusão e retirada de um item da fila incorre na mudança do endereço do ponteiro que informa onde é o início e o término da fila.

Representação

- Em uma fila, o primeiro elemento inserido é o primeiro a ser removido.
- Por essa razão, uma fila é chamada **fifo(*first-in first-out*)** – primeiro que entra é o primeiro a sair – ao contrário de uma pilha que é lifo (*last-in, first-out*)
- Para exemplificar a implementação em C, vamos considerar que o conteúdo armazenado na fila é do tipo inteiro.
- A estrutura de fila possui a seguinte representação:

```
struct fila{
  int elemento[N];
  int ini, n;
}
typedef struct fila Fila;
```

- Trata-se de uma estrutura heterogênea constituída de membros distintos entre si. Os membros são as variáveis *ini* e *fim*, que serve para armazenar respectivamente, o início e o fim da fila e o vetor *elemento* de inteiros que armazena os itens da fila.

Operações Primitivas

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

Operação	**Descrição**
criar()	aloca dinamicamente a estrutura da fila.
insere(f,e)	adiciona um novo elemento *(e)*, no final da fila *f*.
retira(f)	remove o elemento do início da fila *f*.

Tabela: Operações básicas da estrutura de dados fila.

Operações auxiliares

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

Operação	**Descrição**
vazia(f)	informa se a fila está ou não vazia.
libera(f)	destrói a estrutura, e assim libera toda a memória alocada.

Tabela: Operações auxiliares da estrutura de dados fila.

Interface do Tipo Fila

```c
typedef struct fila Fila;
/* Aloca dinamicamente a estrutura Fila, inicializando seus
 * campos e retorna seu ponteiro. A fila depois de criada
 * estarah vazia.*/
Fila* criar(void);

/* Insere o elemento e no final da fila f, desde que,
 * a fila nao esteja cheia.*/
void insere(Fila* f, int e);

/* Retira o elemento do inicio da fila, e fornece o
 * valor do elemento retirado como retorno, desde que a fila
 * nao esteja vazia*/
int retira(Fila* f);

/*Verifica se a fila f estah vazia*/
int vazia(Fila* f);

/*Libera a memoria alocada pela fila f*/
void libera(Fila* f);
```

Implementação de Fila com Vetor

- Assim como nos casos da pilha e lista, a implementação de fila será feita usando um vetor para armazenar os elementos.
- Isso implica, que devemos fixar o número máximo de elementos na fila.
- O processo de inserção e remoção em extremidades opostas fará a fila "andar" no vetor.
- Por exemplo, se inserirmos os elementos 8, 7, 4, 3 e depois retiramos dois elementos, a fila não estará mais nas posições iniciais do vetor.

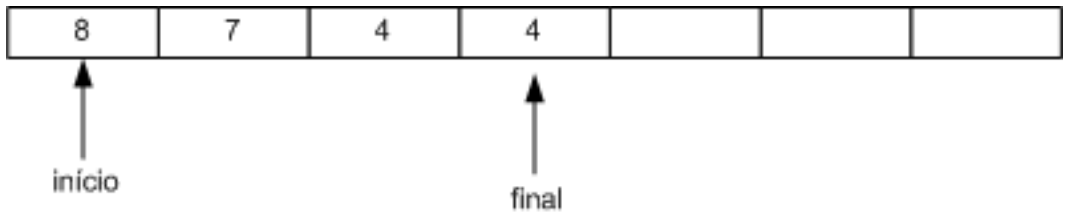


Figura: Fila após inserção de quatro elementos.

Implementação de Fila com Vetor

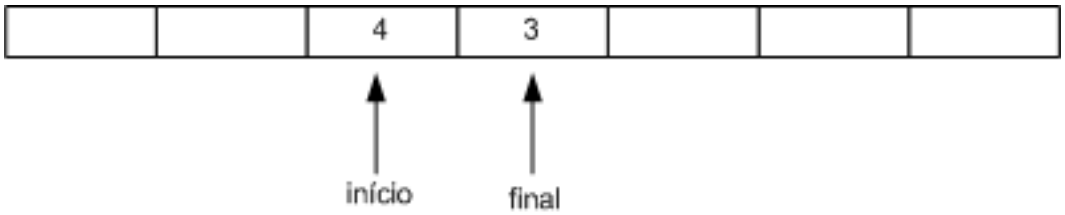


Figura: Fila após retirar dois elementos.

- Com essa estratégia, é fácil observar que, em um dado instante, a parte ocupada pelo vetor pode chegar a última posição.
- Uma solução seria ao remover um elemento da fila, deslocar a fila inteira no sentido do início do vetor.
- Entretanto, essa método é bastante ineficiente, pois cada retirada implica em deslocar cada elemento restante da fila. Se uma fila tiver 500 ou 1000 elementos, evidentemente esse seria um preço muito alto a pagar.

Implementação de Fila com Vetor

- Para reaproveitar as primeira posições do vetor sem implementar uma "re-arrumação" dos elementos, podemos incrementar as posições do vetor de forma "circular".
- Para essa implementação, os índices do vetor são incrementados de maneira que seus valores progridam "circularmente".
- Dessa forma, se temos 100 posições no vetor, os índices assumem os seguintes valores:

$$0, 1, 2, 3, \cdots, 98, 99, 0, 1, 2, 3, \cdots, 98, 99, \cdots$$

Função de Criação

- A função que cria uma fila, deve criar e retornar o ponteiro de uma fila vazia;
- A função deve informar onde é o início da fila, ou seja, fazer $f->ini = 0$, como podemos ver no código abaixo.
- A complexidade de tempo para criar a fila é constante, ou seja, $O(1)$.

```
/* Aloca dinamicamente a estrutura Fila, inicializando seus
 * campos e retorna seu ponteiro. A fila depois de criada
 * estarah vazia.
 */
Fila* criar(void)
{
    Fila* f = malloc(sizeof(Fila));
    f->n = 0;
    f->ini = 0;
    return f;
}
```

Função de Inserção

- Para inserir um elemento na fila, usamos a próxima posição livre do vetor, indicada por n.
- Devemos assegurar que há espaço para inserção do novo elemento no vetor, haja vista se tratar de um vetor com capacidade limitada.
- A complexidade de tempo para inserir um elemento na fila é constante, ou seja, $O(1)$.

```
/* Insere o elemento e no final da fila f.*/
void insere(Fila* f, int e)
{
   int fim;
   if (f->n == N){
       printf("Fila cheia!\n");
   }else{
       fim = (f->ini + f->n) % N;
       f->elementos[fim] = e;
       f->n++;
   }
}
```

Função de Remoção

- A função para retirar o elemento do início da fila fornece o valor do elemento retirado como retorno.
- Para remover um elemento, devemos verificar se a fila está ou não vazia.
- A complexidade de tempo para remover um elemento da fila é constante, ou seja, $O(1)$.

```
int retira(Fila* f)
{
  int e;
  if (vazia(f))
    printf("Fila vazia!\n");
  else{
    e = f->elementos[f->ini];
    f->ini = (f->ini + 1) % N;
    f->n--;
  }
  return e;
}
```

Exemplo de Uso da Fila

```c
#define N 10
#include <stdio.h>
#include "fila.h"

int main(void)
{
    Fila* f = criar();

    int i;
    for (i = 0; i < N; i++)
      insere(f,i * 2);

    printf("\nElementos removidos: ");

    for (i = 0; i < N/2; i++)
     printf("%d ", retira(f));

    system("pause");
}
```

Referências

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

Capítulo xxxxx – Listas

Pontos fundamentais a serem cobertos:

1.
2.
3.

Introdução

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

Definição

Definição

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

Representação

- Os nós de uma lista são armazenados em endereços contínuos.
- A relação de ordem é representada pelo fato de que se o endereço do nó x_i é conhecido, então o endereço do nó x_{i+1} também pode ser determinado.
- A Figura 6 apresenta a representação de uma lista linear de n nós, com endereços representados por k

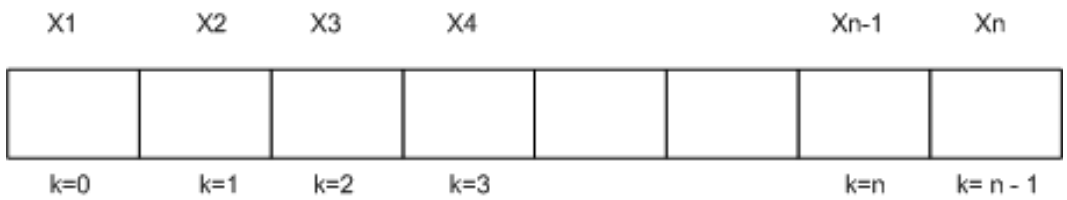


Figura: Exemplo de representação de lista.

Representação

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

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

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

Representação

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

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

Operações Primitivas

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

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

Tabela: Operações básicas da estrutura de dados lista.

Operações auxiliares

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

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

Tabela: Operações auxiliares da estrutura de dados lista.

Interface do Tipo Lista

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

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

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

/* Pesquisa na lista l o elemento e. A funcao retorna
 * o endereco(indice) do elemento se ele pertencer a lista
 * ou -1 caso contrario.*/
int pesquisar(Lista* l, int e);

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

Implementação da Lista

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

Função de Criação

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

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

Função de Inserção

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

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

Função de Remoção

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

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

Função de Pesquisa

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

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

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

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

Função de Impressão

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

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

Exemplo de Uso da Lista

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

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

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

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

Referências

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

Capítulo xxxxx – Recursão

Pontos fundamentais a serem cobertos:

1.
2.
3.

Recursão

- Um objeto é dito recursivo se ele consistir parcialmente ou for definido em termos de si próprio. Recursões não são encontradas apenas em matemática mas também no dia a dia.
- Recursão é uma técnica particularmente poderosa em definições matemáticas. Alguns exemplos: números naturais, estrutura de árvore e certas funções:
  1. Números naturais:
     - 1.1 0 é um número natural.
     - 1.2 O sucessor de um número natural é um número natural.
  2. Estruturas de árvores
     - 2.1 0 é uma árvore (chamada árvore vazia).
     - 2.2 Se t_1 e t_2 são árvores, então a estrutura que consiste de um nó com dois ramos t_1 *e* t_2 também é uma arvore.
  3. A função fatorial $n!$
     - 3.1 $0! = 1$
     - 3.2 $n > 0, n! = n * (n - 1)$

Recursão

- Se uma função f possuir uma referência explícita a si próprio, então a função é dita *diretamente recursiva*. Se f contiver uma referência a outra função g, que por sua vez contém uma referência direta ou indireta a f, então f é dita *indiretamente recursiva*.
- Em termos matemáticos, a recursão é uma técnica que através de substituições sucessivas reduz o problema a ser resolvido a um caso de solução mais simples (Dividir para conquistar).

Recursão

Exemplo

```
/**
 * Calcula a soma dos numeros inteiros
 * existentes entre in e n inclusive.
 */
int somatorio(int in, int n){
 int s = in;
 if (s < n)
 {
   return s + somatorio(s + 1, n);
 }
 return s;
}

public static void main(String args)
{
   print(somatorio(1, 100));
}
```

Recursão

1. Há dois requisitos-chave para garantir que a recursão tenha sucesso:
   1.1 Toda chamada recursiva tem de simplificar os cálculos de alguma maneira.
   1.2 Tem de haver casos especiais para tratar os cálculos mais simples diretamente.
2. Muitas recursões podem ser calculadas com laços. Entretanto, as soluções iterativas para problemas recursivos podem ser mais complexas.
3. Por exemplo, a permutação de uma palavra.

Recursão

- A permutação é um exemplo de recursão que seria difícil de programar utilizando laços simples.
- Uma permutação de uma palavra é simplesmente um rearranjo das letras. Por exemplo, a palavra "eat" tem seis permutações ($n!$, onde n é o número de letras que formam a palavra).
- Como gerar essas permutações?
- Simples, primeiro, gere todas as permutações que iniciam com a letra "e", depois as que iniciam com a letra "a" e finalmente as que iniciam com a letra "t".
- Mas, como gerar as permutações que iniciam com a letra "e"?
- Gere as permutações da sub-palavra "at". Porém, esse é o mesmo problema, mas com uma entrada mais simples, ou seja, uma palavra menor.
- Logo, podemos usar a recursão nesse caso.

Recursão

Como pensar recursivo

1. Combine várias maneiras de simplificar as entradas.
2. Combine as soluções de entradas mais simples para uma solução do problema original.
3. Encontre soluções para as entradas mais simples.
4. Implemente a solução combinando os casos simples e o passo de redução.

Eficiência da Recursão

1. A recursão pode ser uma ferramenta poderosa para implementar algoritmos complexos.
2. No entanto, a recursão pode levar a algoritmos que tem um desempenho fraco.
3. Vejamos quando a recursão é benéfica e quando é ineficiente.

Eficiência da Recursão

1. Considere a sequência de Fibonacci, uma sequência de números inteiros definidos pela equação:

$$f_1 = 1$$

$$f_2 = 1$$

$$f_n = f_{n-1} + f_{n-2}$$

2. Exemplo: $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \cdots$.
3. Vejamos uma implementação recursiva que calcule qualquer valor de n.

Eficiência da Recursão

```
int fibonacci(int n) {
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1) + fibonacci(n - 2);
}

void main(void) {
  int i;
  for (i = 1; i <= n; i++) {
    int f = fibonacci(i);
    printf("%d", f);
  }
}
}
```

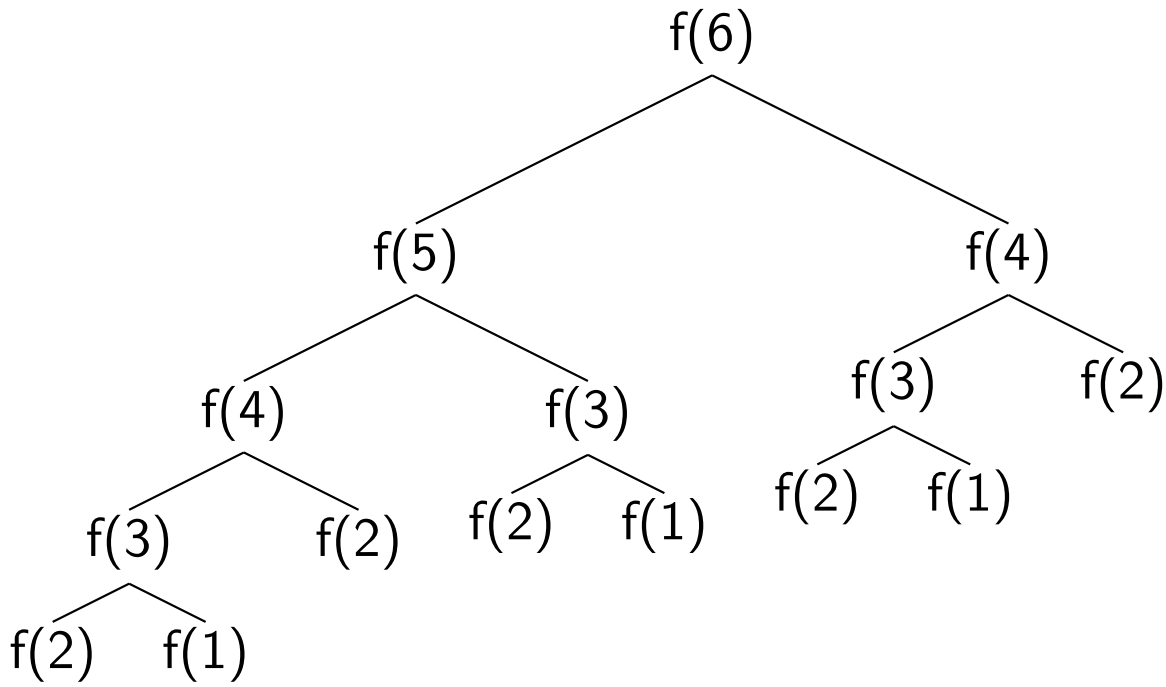
Eficiência da Recursão

1. Ao executarmos o programa de teste podemos notar que as primeiras chamadas à função **fibonacci** são bem rápidas. No entanto, para valores maiores, o programa pausa um tempo considerável entre as saídas.
2. Inicialmente isso não faz sentido, uma vez que podemos calcular de forma rápida com auxílio de uma calculadora esses números, de modo que para o computador não deveria demorar tanto em hipótese alguma.
3. Para descobrir o problema, vamos inserir mensagens de monitoração das funções e verificar a execução para $n = 6$.

Eficiência da Recursão

Início fibonacci n = 6
Início fibonacci n = 5
Início fibonacci n = 4
Início fibonacci n = 3
Início fibonacci n = 2
Término fibonacci n = 2, retorno = 1
Início fibonacci n = 1
Término fibonacci n = 1, retorno = 1
Término fibonacci n = 3, retorno = 2
Início fibonacci n = 2
Término fibonacci n = 2, retorno = 1
Término fibonacci n = 4, retorno = 3
Início fibonacci n = 3
Início fibonacci n = 2
Término fibonacci n = 2, retorno = 1
Início fibonacci n = 1
Término fibonacci n = 1, retorno = 1
Término fibonacci n = 3, retorno = 2
Término fibonacci n = 5, retorno = 5
Início fibonacci n = 4
Início fibonacci n = 3
Início fibonacci n = 2
Término fibonacci n = 2, retorno = 1
Início fibonacci n = 1
Término fibonacci n = 1, retorno = 1
Término fibonacci n = 3, retorno = 2
Início fibonacci n = 2
Término fibonacci n = 2, retorno = 1
Término fibonacci n = 4, retorno = 3
Término fibonacci n = 6, retorno = 8
Fibonacci(6) = 8

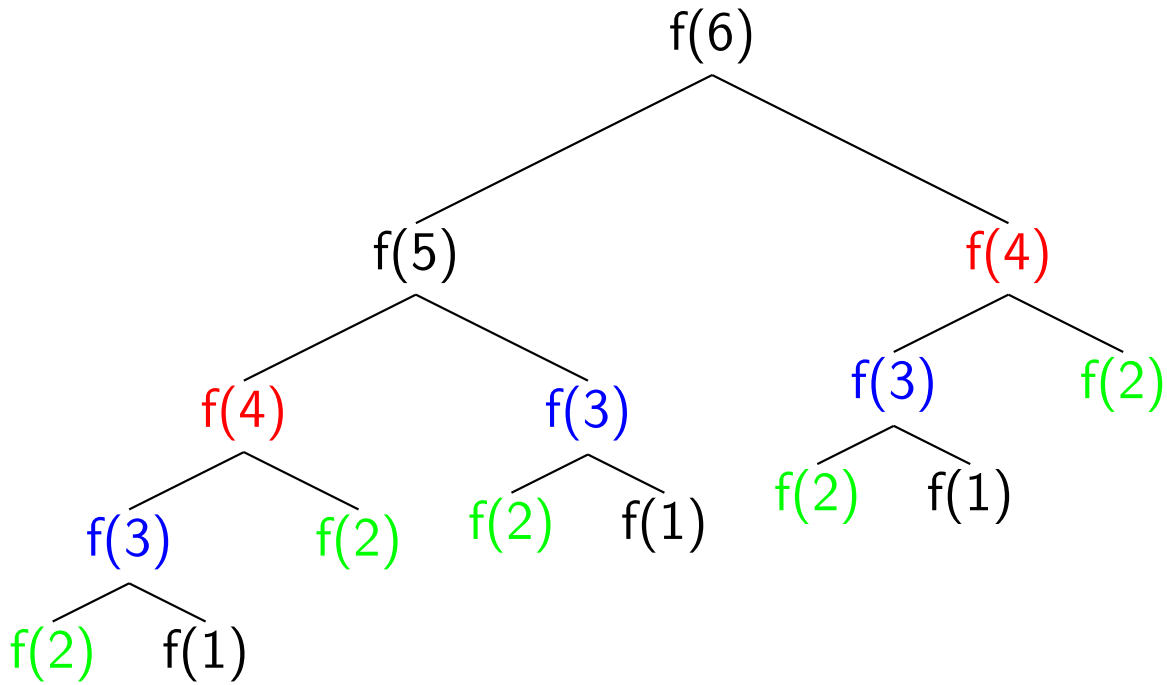
Eficiência da Recursão



Padrão de chamada de função/método recursivo *fibonacci*.

Eficiência da Recursão

1. Analisando o rastro de execução do programa fica claro porque o método leva tanto tempo.
2. Ele calcula os mesmos valores repetidas vezes.
3. Pelo exemplo, o calculo de **fibonacci(6)** chama **fibonacci(4)** duas vezes, **fibonacci(3)** três vezes, **fibonacci(2)** cinco vezes, e **fibonacci(1)** três vezes.
4. Diferente do cálculo que faríamos manualmente.



Em resumo...

Eficiência da Recursão

As vezes acontece de uma solução recursiva ser executada muito mais lentamente do que sua equivalente iterativa. Entretanto, na maioria dos casos, a solução recursiva é apenas levemente mais lenta.

Em muitos casos, uma solução recursiva é mais fácil de entender e implementar corretamente do que uma solução iterativa.

Capítulo xxxxx – Árvores

Pontos fundamentais a serem cobertos:

1.
2.
3.

Definição

- Uma árvore é uma estrutura hierárquica composta por nós e ligações entre eles
- Pode ser vista como um grafo acíclico
- Cada nó possui somente um pai e zero ou mais filhos

Estrutura

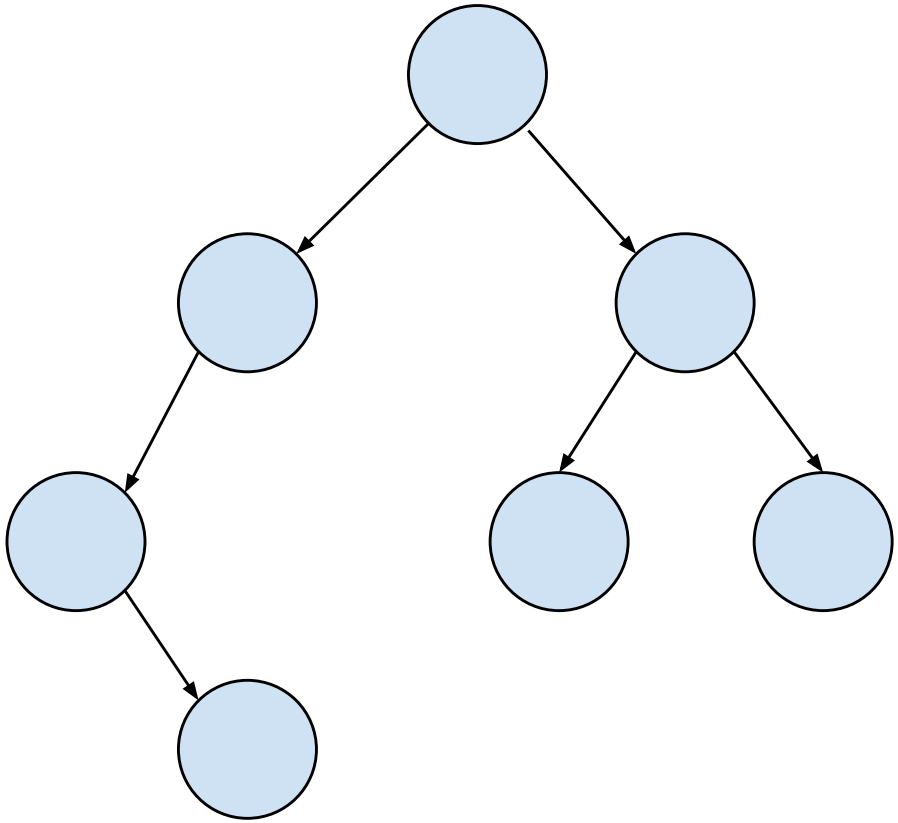


Figura: Exemplo de uma árvore

Árvore Binária de Busca - Definição

Árvore onde cada nó possui até 2 filhos. O filho da esquerda só pode conter chaves menores do que a do pai, enquanto que o filho da direita só comporta chaves maiores do que a do pai.

Árvore Binária de Busca

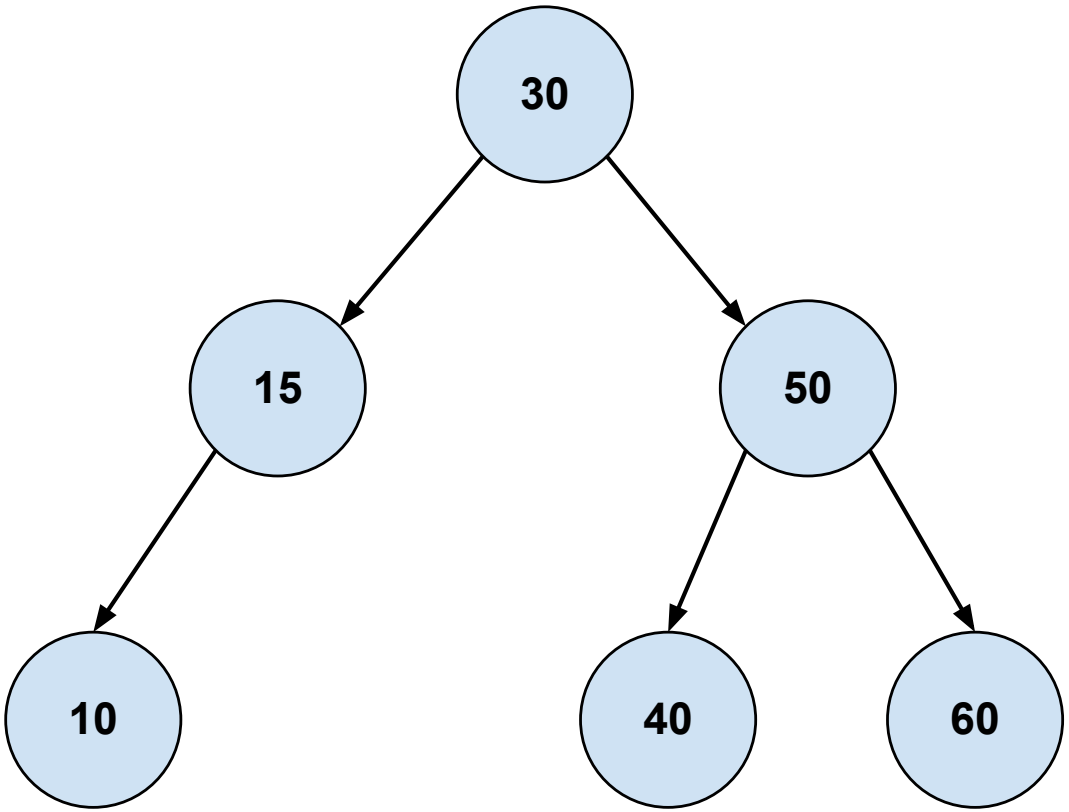
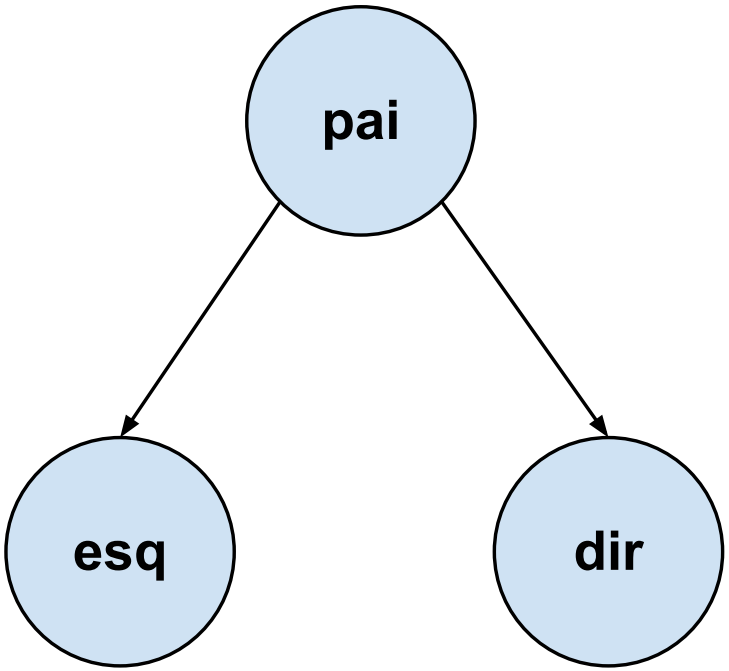


Figura: Exemplo de árvore binária de busca

Árvore Binária de Busca




```
struct ArvoreBinaria {
    struct ArvoreBinaria* esq;
    struct ArvoreBinaria* dir;

   // contém a chave e os dados satélite
    TipoItem valor;
};
```

Figura: Estrutura básica / nó

Operações Básicas

Operações Básicas

- Inserção
- Busca
- Remoção

Usos Comuns

- Dicionários / vetores associativos
- Filas de prioridades

Complexidade Computacional

Quando a árvore está balanceada todas as três operações podem ser implementadas com complexidade computacional igual a $O(\log n)$.

No pior caso (desbalanceamento) estas operações possuem complexidade $O(n)$ [**?**].

Árvore Binária de Busca - Inserção

```
INSERÇÃO(ARVORE, ITEM) {
    SE ITEM->CHAVE = ARVORE->CHAVE
        ARVORE->ITEM = ITEM
        return

    SE ITEM->CHAVE < ARVORE->CHAVE
        SE ARVORE->ESQ = NULO ENTÃO
            ARVORE->ESQ = ARVORE(ITEM)
        SENÃO
            INSERÇÃO(ARVORE->ESQ, ITEM)
    SENÃO
        SE ARVORE->DIR = NULO ENTÃO
            ARVORE->DIR = ARVORE(ITEM)
        SENÃO
            INSERÇÃO(ARVORE->DIR, ITEM)
}
```

Árvore Binária de Busca - Inserção

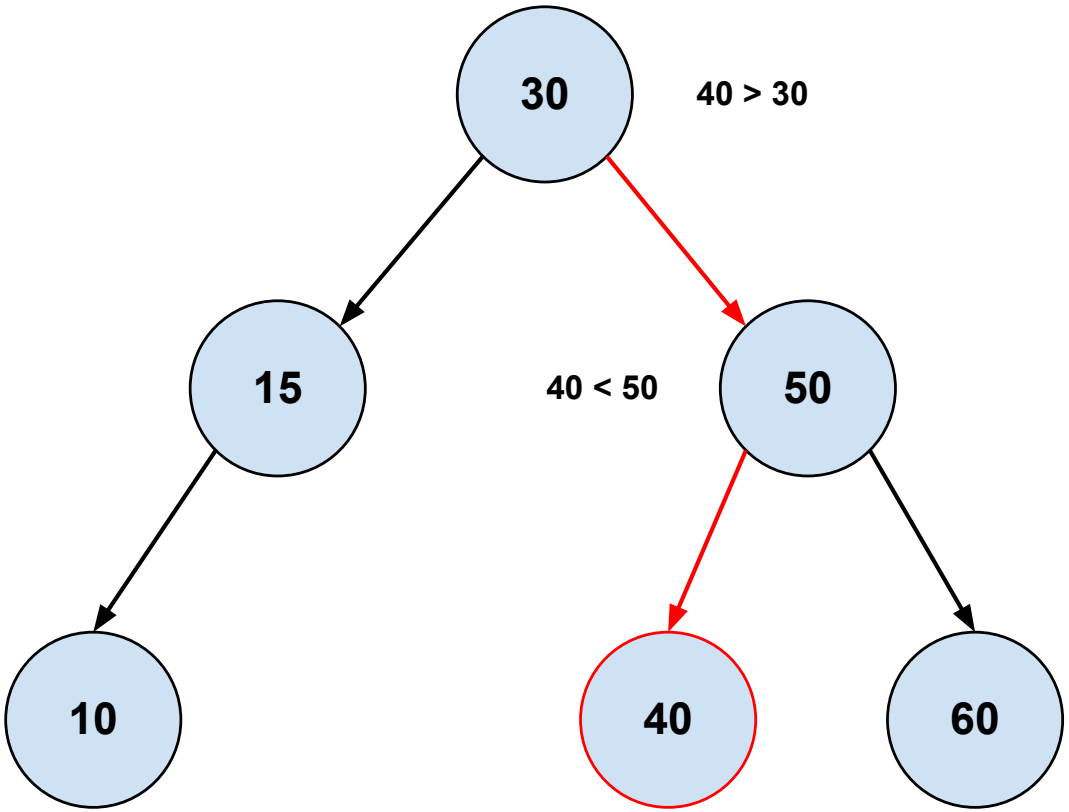


Figura: Exemplo de inserção da chave 40

Árvore Binária de Busca - Inserção

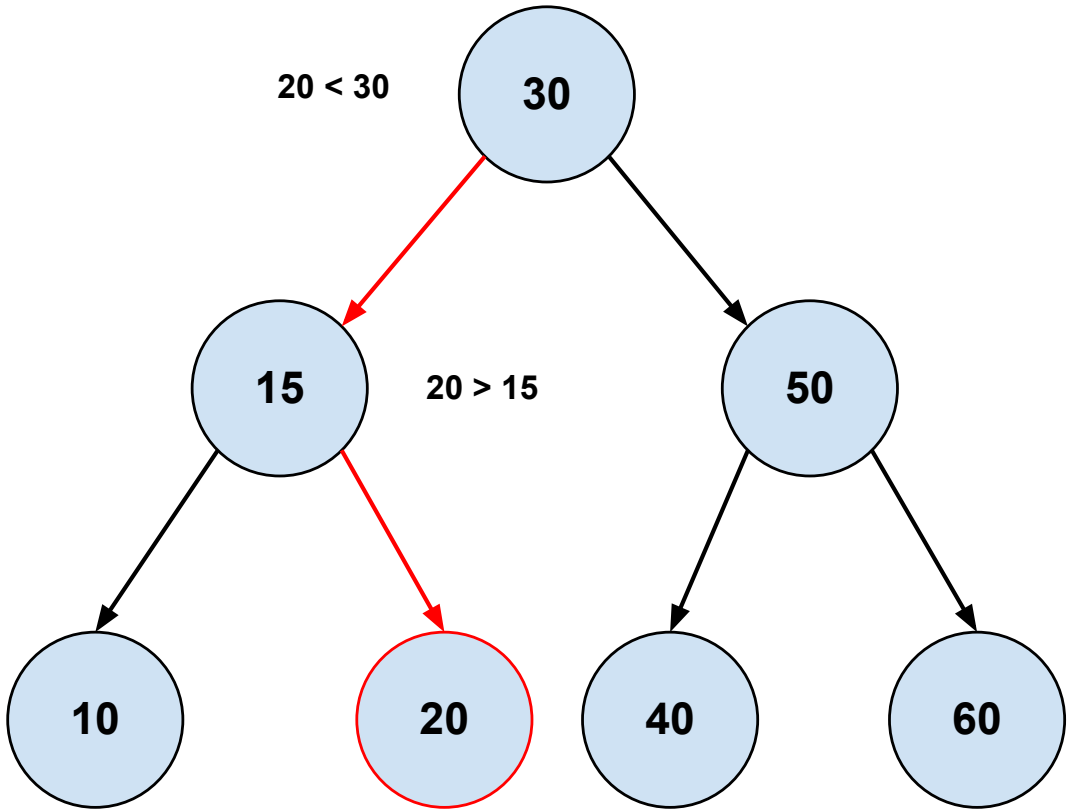


Figura: Exemplo de inserção da chave 20

Árvore Binária de Busca - Busca

```
BUSCA(ARVORE, CHAVE) {
    SE ARVORE = NULO
        return NULO

    SE ARVORE->CHAVE = CHAVE
        return ARVORE

    SE CHAVE < ARVORE->CHAVE
        return BUSCA(ARVORE->ESQ, CHAVE)
    SENÃO
        return BUSCA(ARVORE->DIR, CHAVE)
}
```

Árvore Binária de Busca - Remoção

A remoção de um nó se enquadra em um dos seguintes casos:

1. Remoção de um nó folha (nenhum filho)
2. Remoção de um nó com somente um filho
3. Remoção de um nó com dois filhos

O tratamento de cada caso foi apresentado em sala de aula.

Balanceamento

Uma árvore binária de busca balanceada garante operações de busca, inserção e remoção com complexidade $O(\log n)$, onde n é o número de nós, o que a torna atrativa para diversas aplicações.

Determinadas sequências de inserções ou remoções podem fazer com que uma ABB fique desbalanceada, tornando suas operações $O(n)$.

Cálculo da Altura

```
ALTURA(ARVORE) {
    SE ARVORE = NULO
        return -1

    A1 = ALTURA(ARVORE->DIR)
    A2 = ALTURA(ARVORE->ESQ)

    return maior(A1, A2) + 1
}
```

Cálculo da Altura

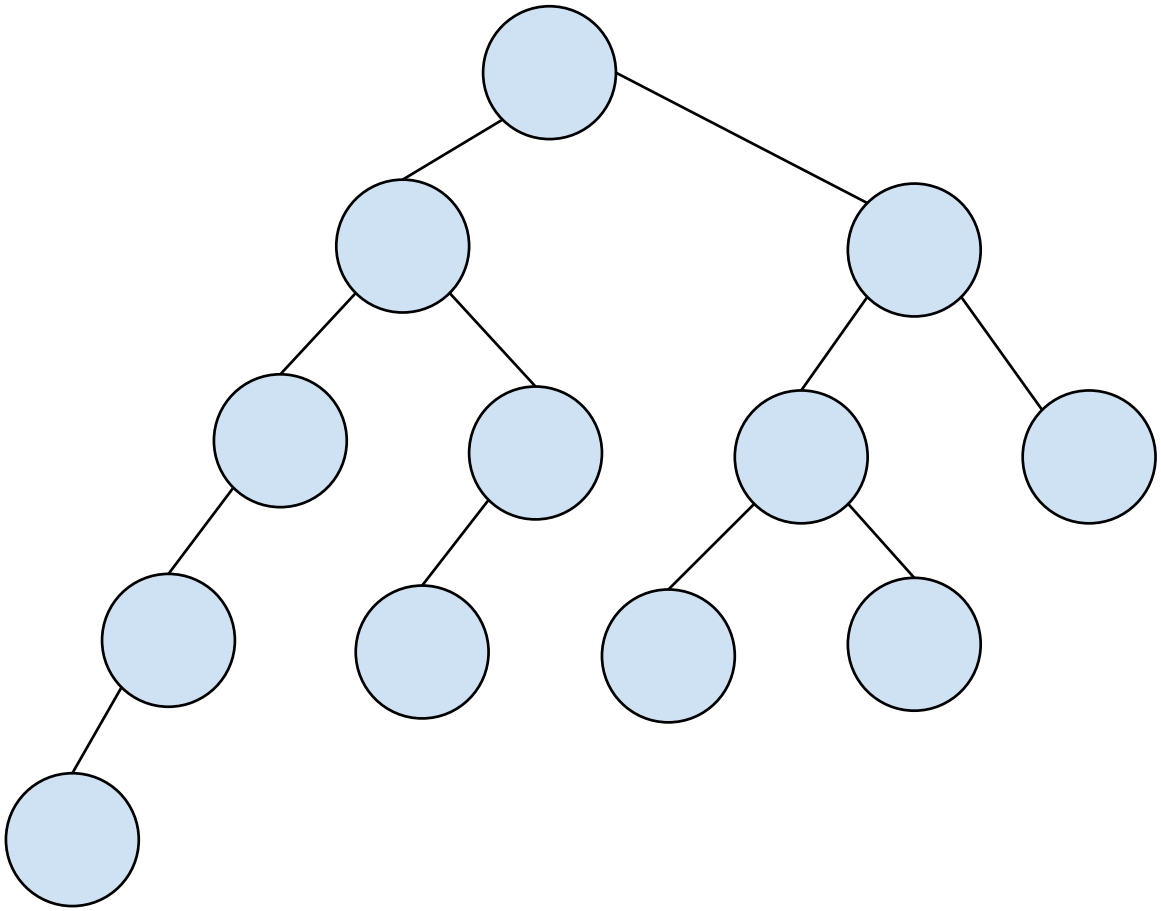


Figura: Exercício: determine a altura de cada subárvore.

Cálculo da Altura

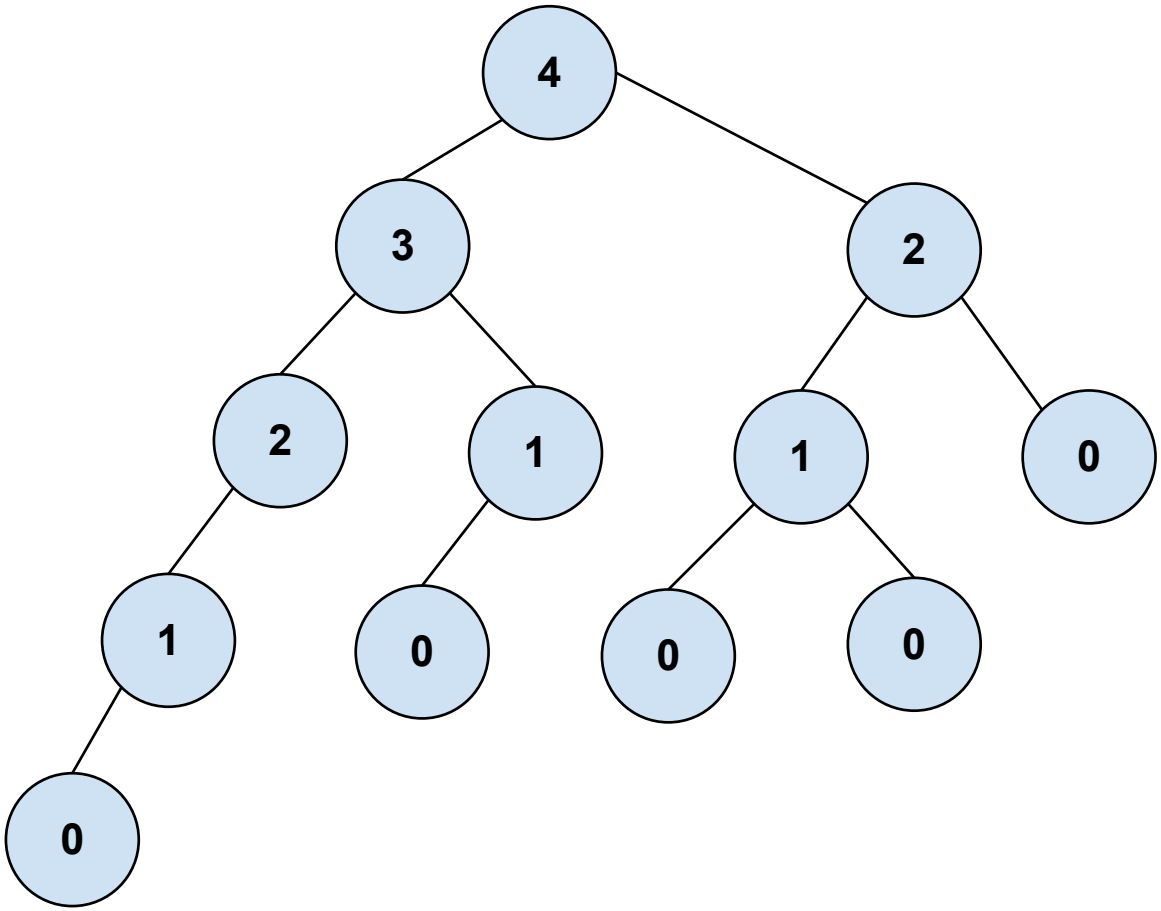


Figura: Resposta do exercício.

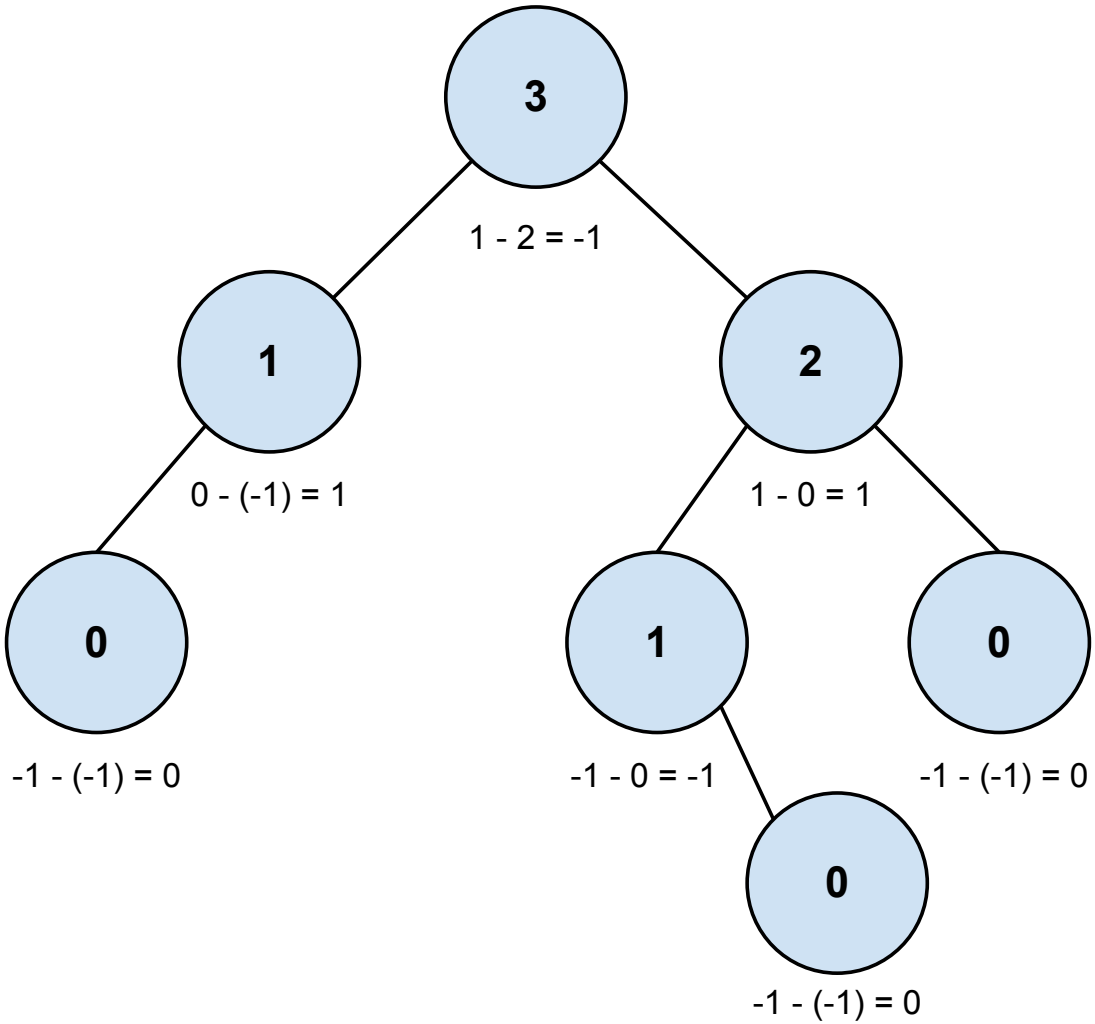
Cálculo do Fator de Balanceamento

```
FB(ARVORE) {
    A1 = ALTURA(ARVORE->ESQ)
    A2 = ALTURA(ARVORE->DIR)
    return A1 - A2
}
```

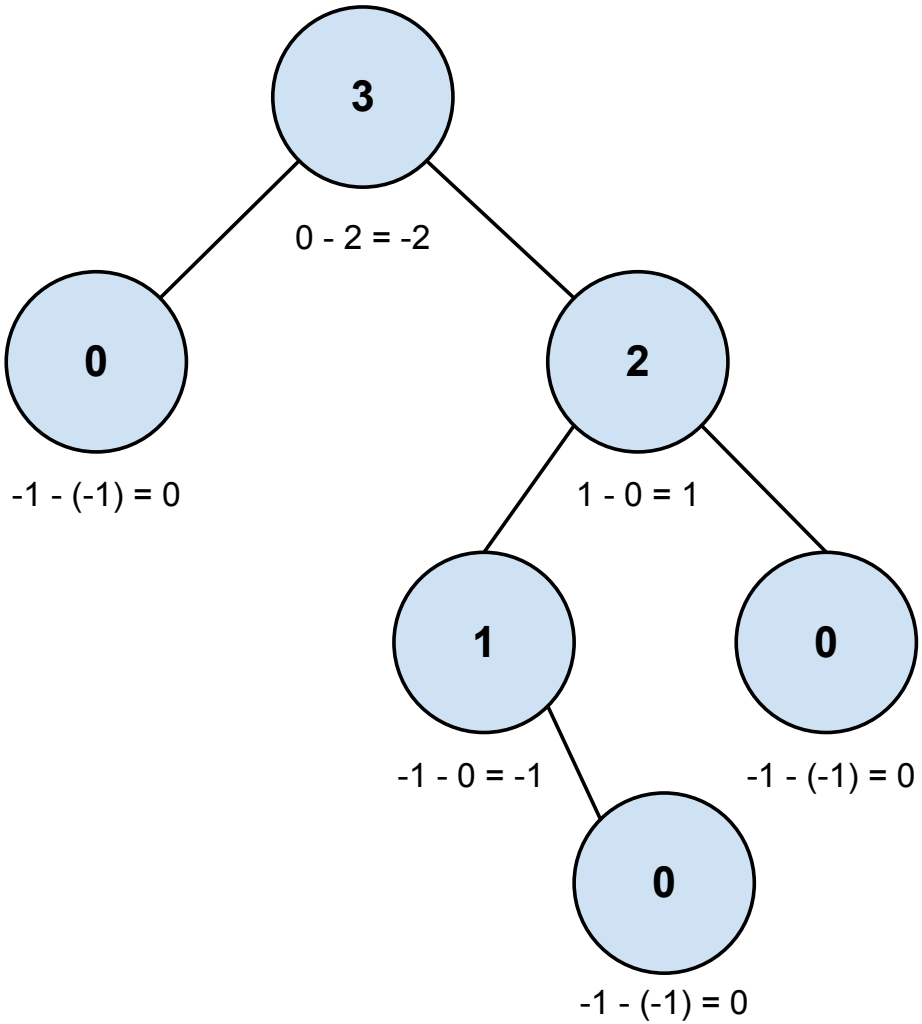
Balanceamento

- Uma ABB está balanceada quando cada nó possui um FB igual a -1, 0 ou 1
- Uma inserção ou remoção pode tornar uma árvore desbalanceada, necessitando de rotações para o seu balanceamento

Exemplo de ABB Balanceada



Exemplo de ABB Desbalanceada

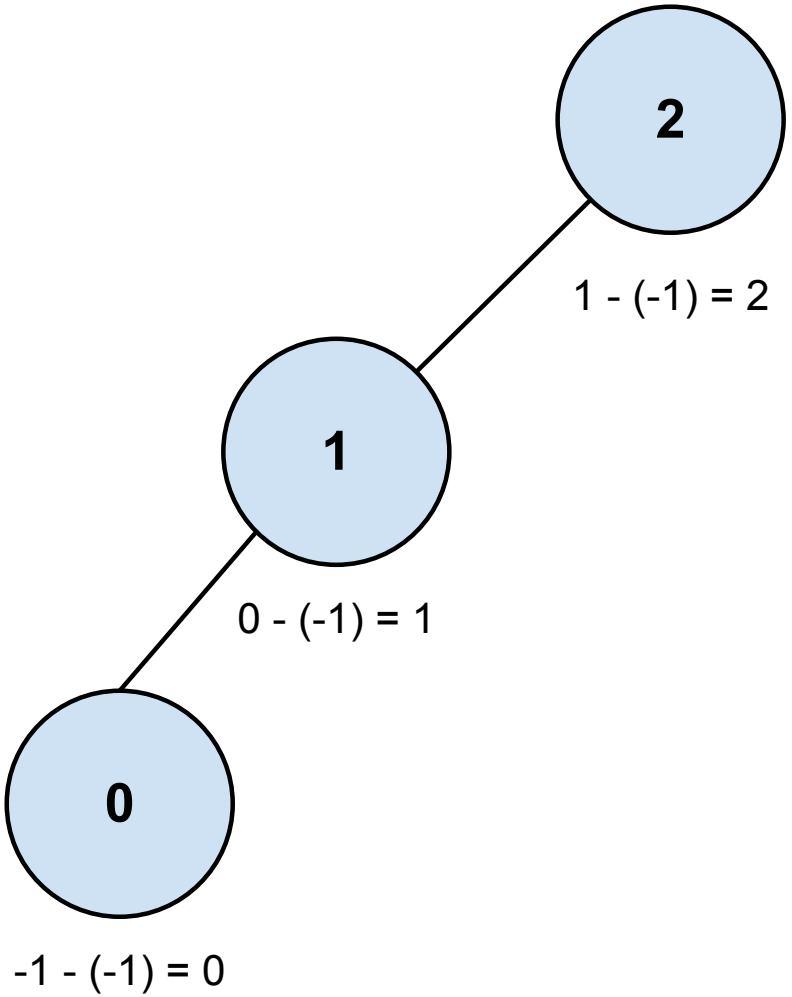


Operação de rotação

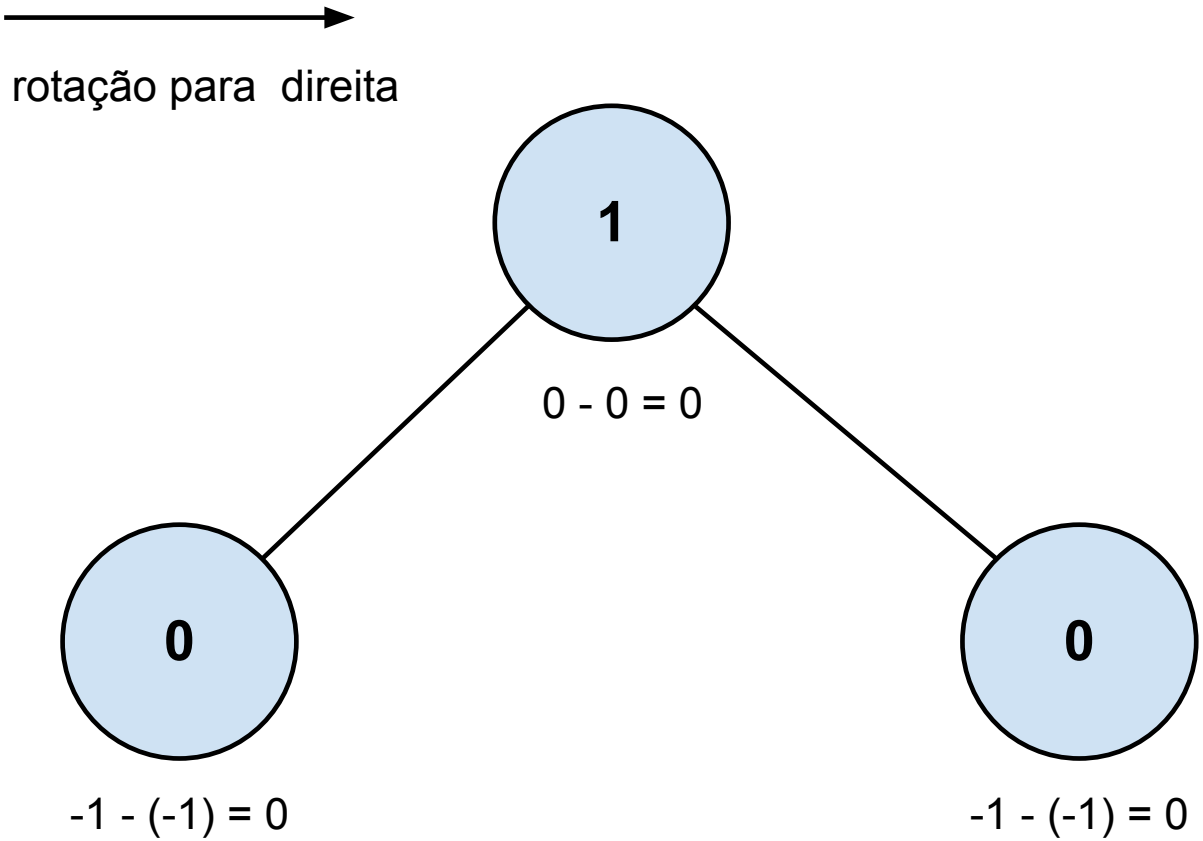
```
ROTACAO_DIREITA(RAIZ) {
    PIVO      = RAIZ->ESQ
    RAIZ->ESQ = PIVO->DIR
    PIVO->DIR = RAIZ
    RAIZ      = PIVO
}

ROTACAO_ESQUERDA(RAIZ) {
    PIVO      = RAIZ->DIR
    RAIZ->DIR = PIVO->ESQ
    PIVO->ESQ = RAIZ
    RAIZ      = PIVO
}
```

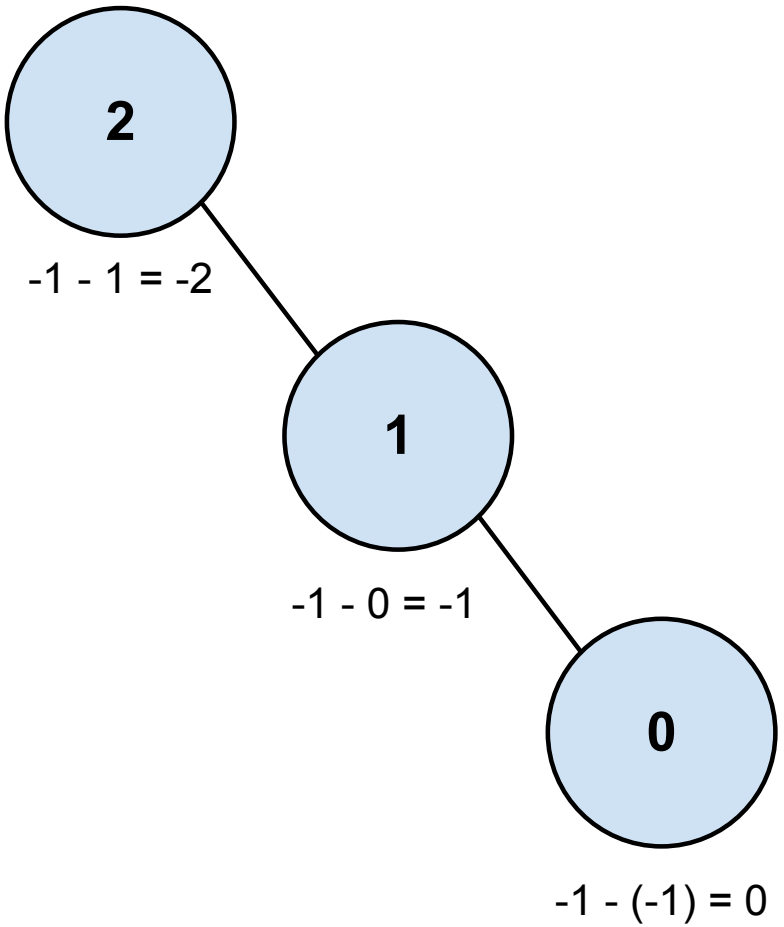
Rotação para Direita



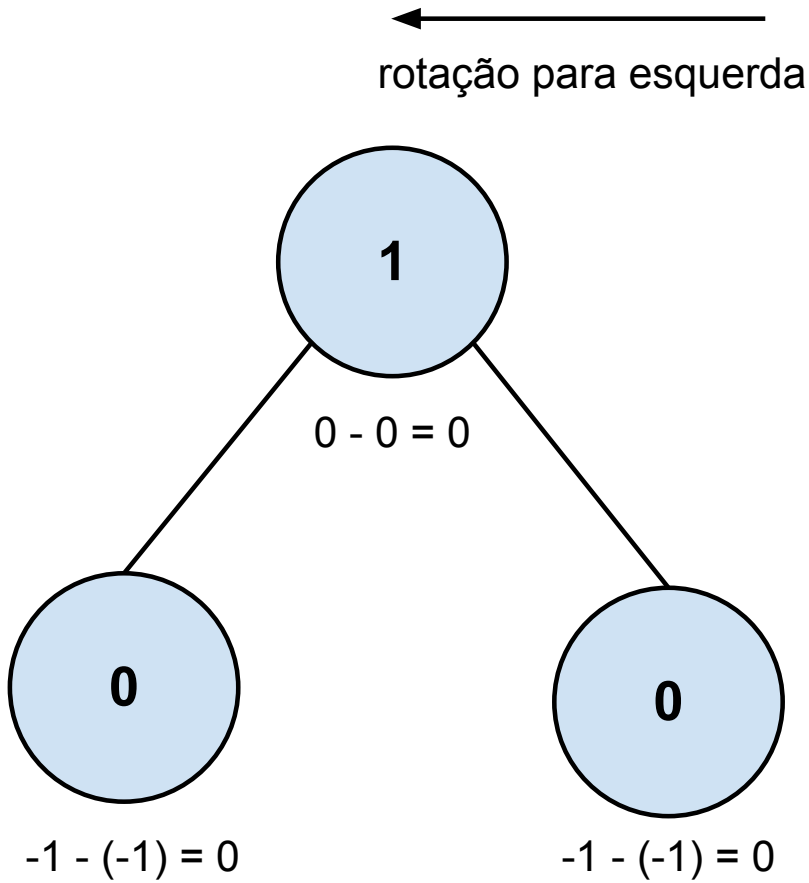
Rotação para Direita



Rotação para Esquerda



Rotação para Esquerda



Árvores AVL

- **AVL** desenvolvida por G. M. **A**delson-**V**elskii and E. M. **L**andis
- Garante o balanceamento da árvore ao realizar rotações após cada inserção ou remoção na ABB

Balanceamento - Inserção

```
BALANCEAMENTO(RAIZ) {
    SE FB(RAIZ) = -2 ENTÃO
        SE FB(RAIZ->DIR) = -1 ENTÃO
            ROTACAO_ESQUERDA(RAIZ)
        SENÃO
            ROTACAO_DIREITA(RAIZ->DIR)
            ROTACAO_ESQUERDA(RAIZ)
    SENÃO SE FB(RAIZ) = 2 ENTÃO
        SE FB(RAIZ->ESQ) = 1 ENTÃO
            ROTACAO_DIREITA(RAIZ)
        SENÃO
            ROTACAO_ESQUERDA(RAIZ->DIR)
            ROTACAO_DIREITA(RAIZ)
}
```

Balanceamento - Inserção

- Para que a árvore tenha um bom desempenho, é essencial que o balanceamento seja calculado eficientemente, isto é, sem a necessidade de percorrer toda a árvore após cada modificação
- Manter a árvore estritamente balanceada após cada modificação tem seu preço (desempenho). Árvores AVL são utilizadas normalmente onde o número de consultas é muito maior do que o número de inserções e remoções e quando a localidade de informação não é importante

Árvore de Espalhamento

- Reestrutura a árvore em cada operação de inserção, busca ou remoção por meio de operações de rotação
- Nome original: *splay tree* [**?**]. Não confundir com a Árvore N-Ária de Espalhamento (ANE) criada por professores da UDESC

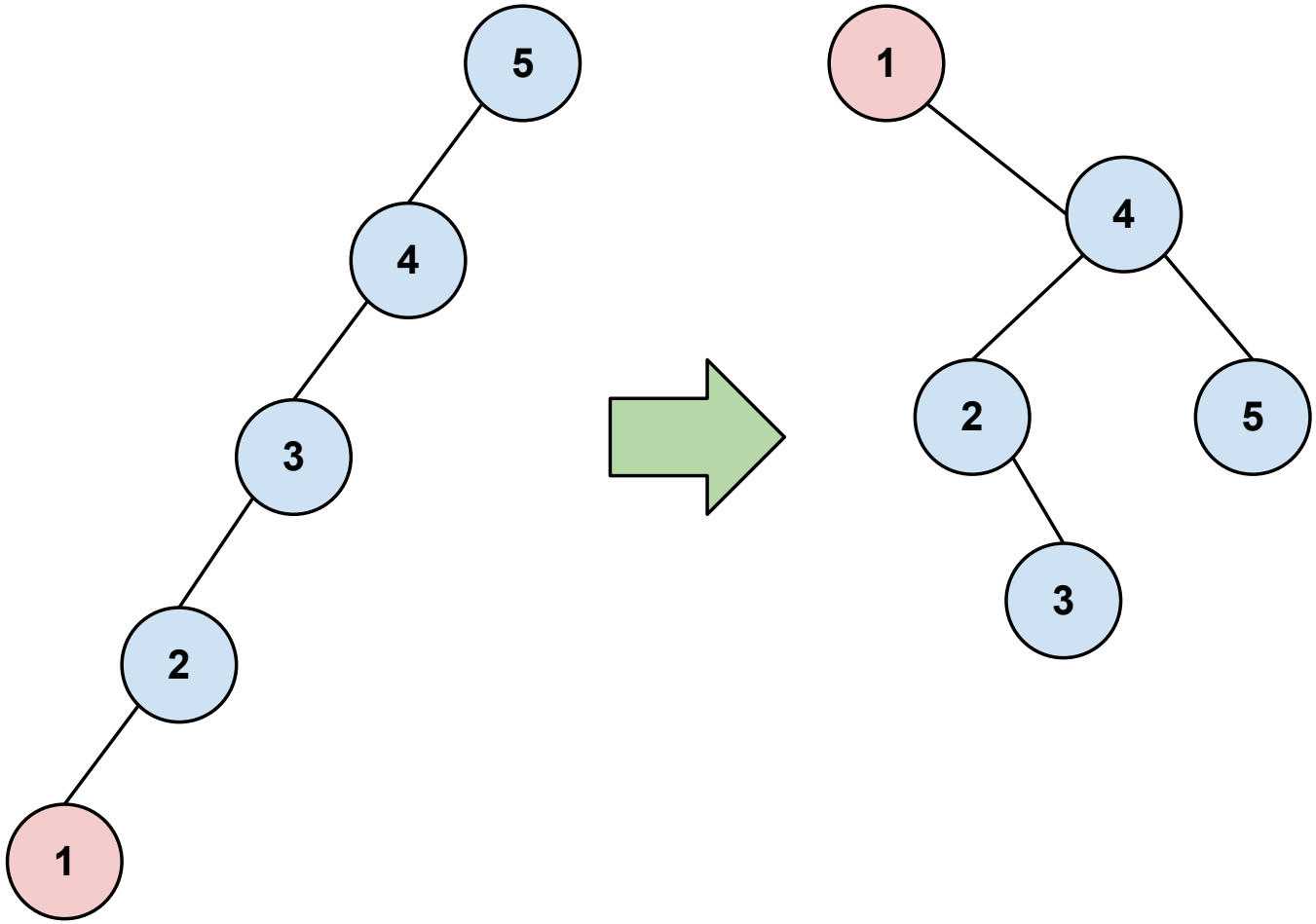
Árvore de Espalhamento

- Evita a repetição de casos ruins [O(n)] devido ao seu rebalanceamento natural
- Não realiza o cálculo de fatores de balanceamento, simplificando sua implementação
- Pior caso para uma operação se mantém O(n), mas, ao considerar uma cadeia de operações, *garante* uma complexidade amortizada de O(logn) para suas operações básicas

Árvore de Espalhamento

- Se baseia na operação de espalhamento, que utiliza rotações para mover uma determinada chave até a raiz
- A sua complexidade O(log n) em uma análise amortizada é garantida pelas rotações efetuadas, o que a difere do uso simples de heurísticas como o *mover para a raíz*

Exemplo - Espalhamento pela chave 1



Operações Básicas

Espalhamento Move a chave desejada para a raiz por uma sequência bem definida de operações de rotação

Busca Busca uma chave na árvore

Inserção Insere uma nova chave na árvore

Remoção Remove uma chave da árvore

Operações Básicas

- Uma árvore de espalhamento é uma árvore binária de busca válida, logo operações como os percursos (pré-em-pós) são idênticas as operações em uma ABB
- As operações de inserção, busca e remoção podem ser definidas com base na operação de espalhamento

Árvore de Espalhamento - Busca

```
BUSCA(RAIZ, CHAVE) {
    return ESPALHAMENTO(RAIZ, CHAVE)
}
```

Árvore de Espalhamento - Inserção

```
INSERE(RAIZ, CHAVE) {
    INSERE_ABB(RAIZ, CHAVE)
    return ESPALHAMENTO(RAIZ, CHAVE)
}
```

Árvore de Espalhamento - Remoção

```
REMOVE(RAIZ, CHAVE) {
    RAIZ = ESPALHAMENTO(RAIZ, CHAVE)

    SE RAIZ->DIR ENTÃO
        AUX = ESPALHAMENTO(RAIZ->DIR, CHAVE)
        AUX->ESQ = RAIZ->ESQ
    SENÃO
        AUX = RAIZ->ESQ

    return AUX
}
```

Estratégias de Espalhamento

Duas estratégias:

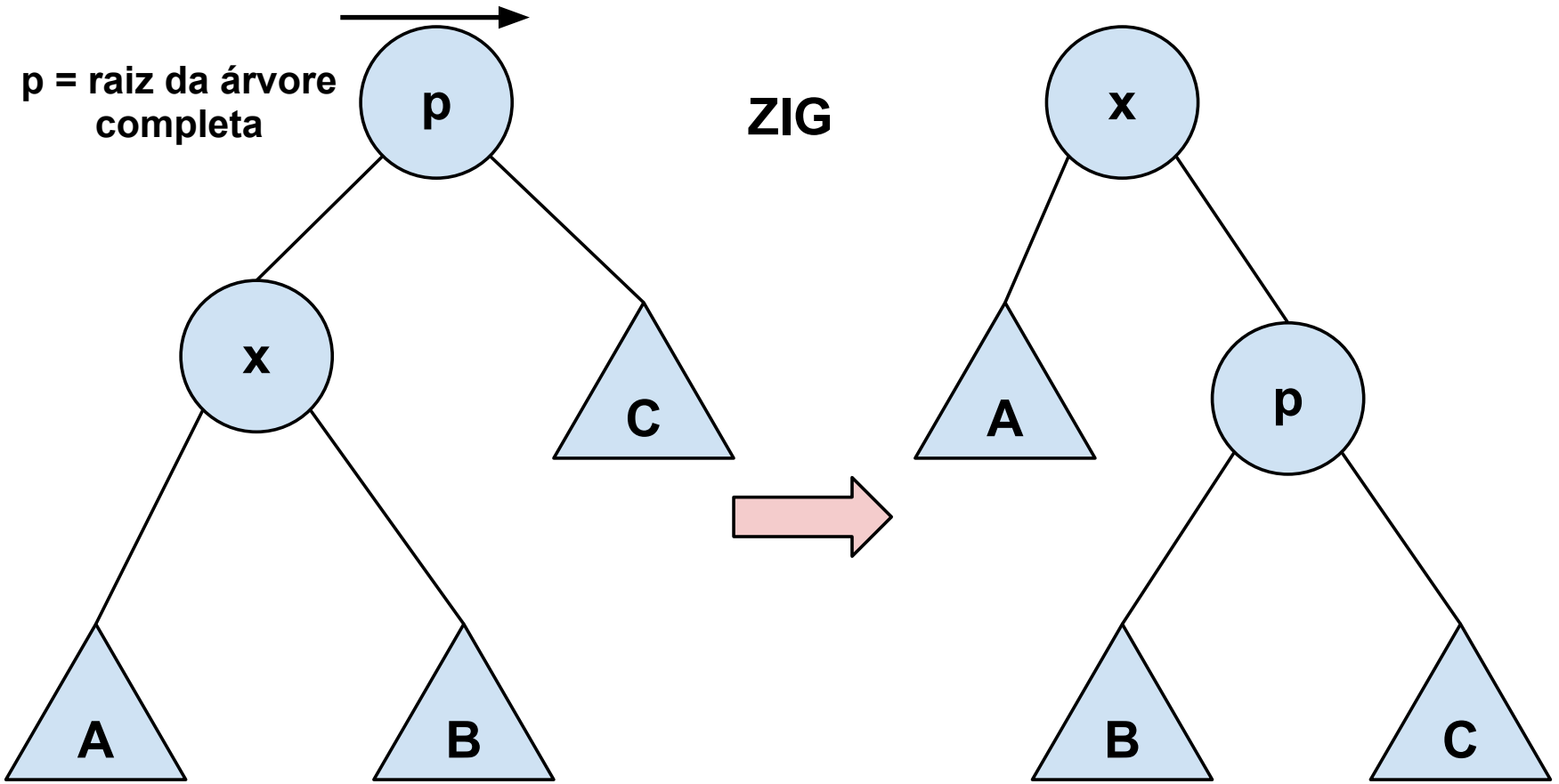
Bottom-Up Parte do nó acessado e o movimenta para a raiz da árvore por meio de rotações

Top-Down Parte do nó raiz, rotacionando e *removendo do caminho* os nós entre a raiz e o nó desejado, armazenando-os em duas árvores auxiliares, remontando a árvore completa na sua etapa final.

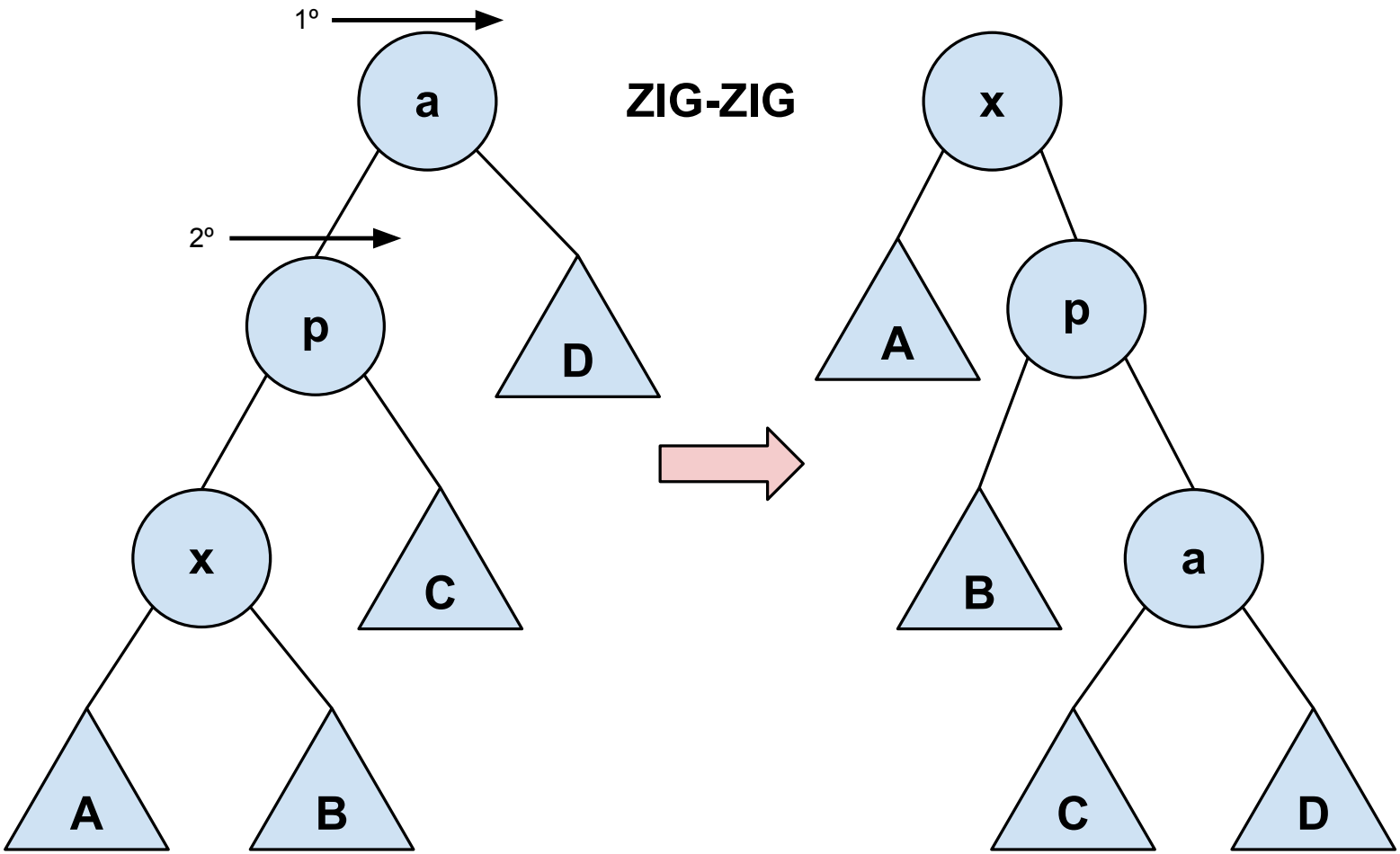
Espalhamento Bottom-Up

- Na estratégia Bottom-Up, a operação de espalhamento realiza rotações subindo gradativamente de níveis, a partir da chave desejada
- Enquanto a chave não estiver na raiz, deve-se verificar qual o caso aplicável (ZIG, ZIG-ZIG ou ZIG-ZAG) e realizar as rotações necessárias

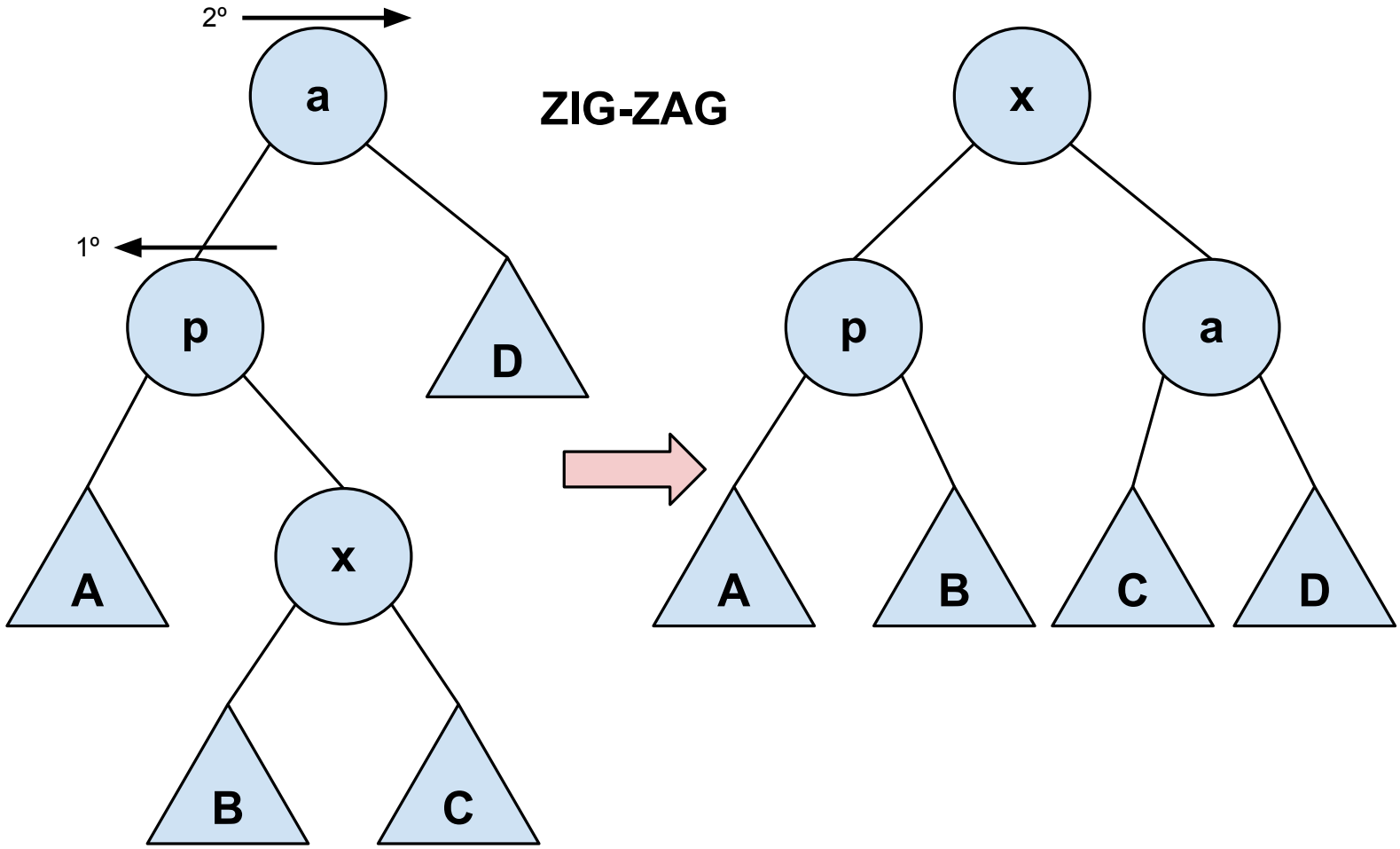
Caso 1: ZIG



Caso 2: ZIG-ZIG



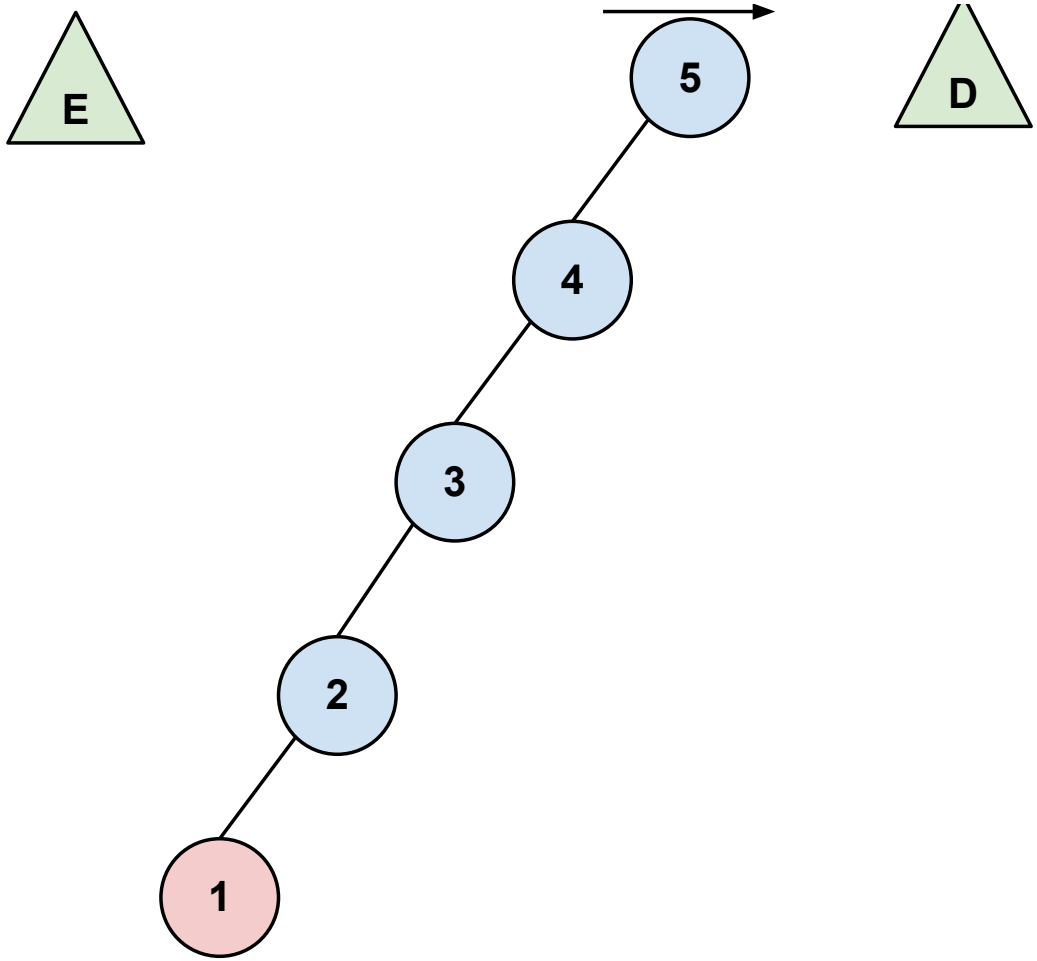
Caso 3: ZIG-ZAG



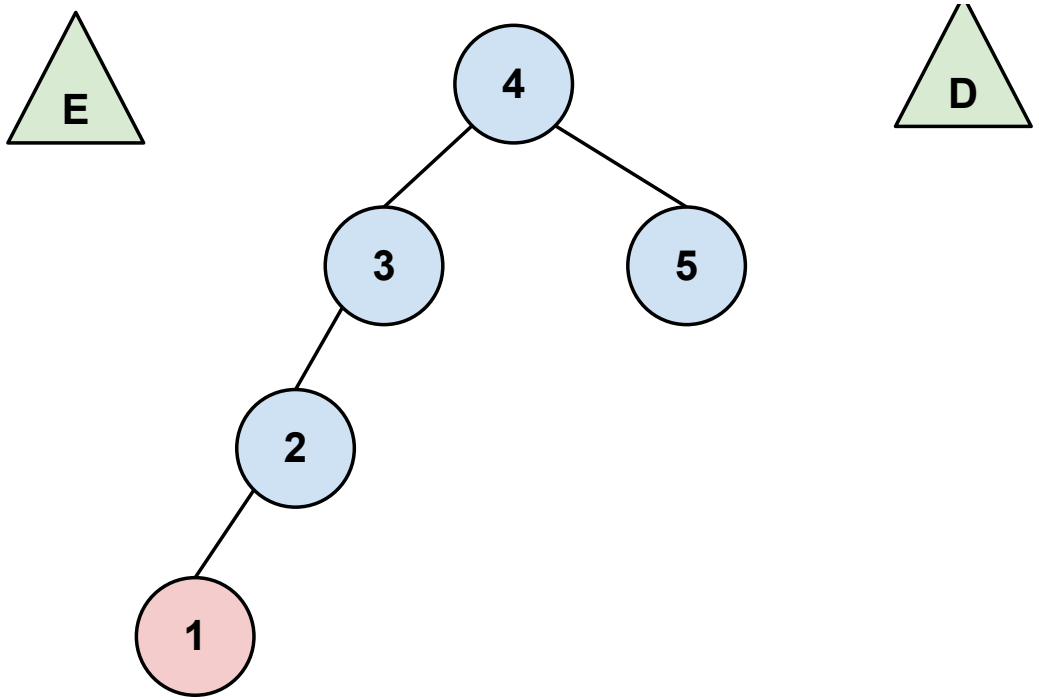
Espalhamento Top-Down

- Na estratégia Top-Down as chaves que estão no caminho da chave desejada para a raiz são rotacionadas e removidas para árvores auxiliares seguindo uma sequência de operações bem definidas
- Quando a chave desejada chega até a raiz, a árvore é remontada pelo retorno das chaves removidas

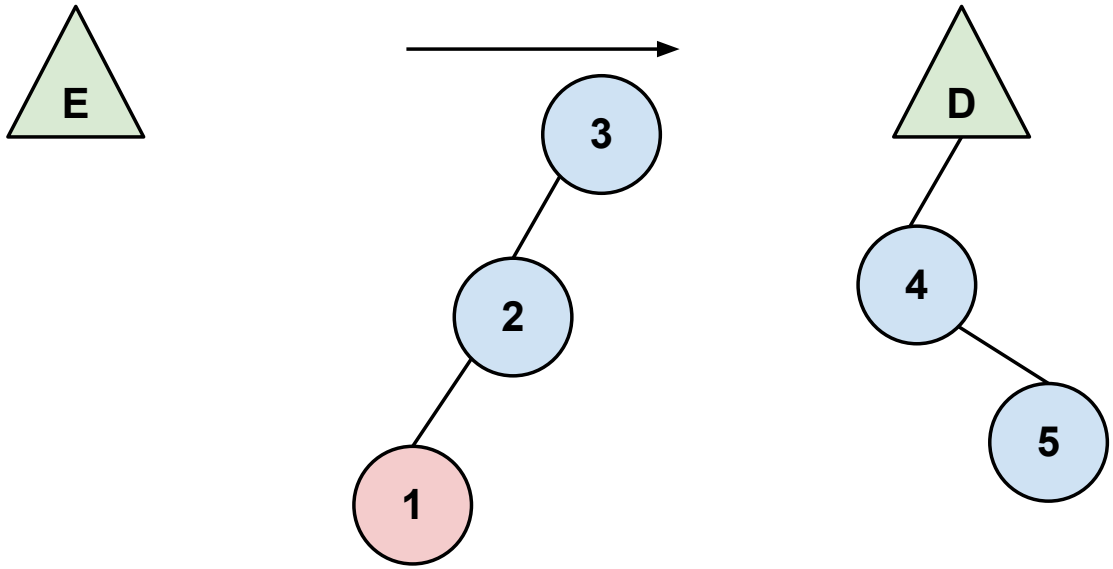
Exemplo: Top-Down 1/6



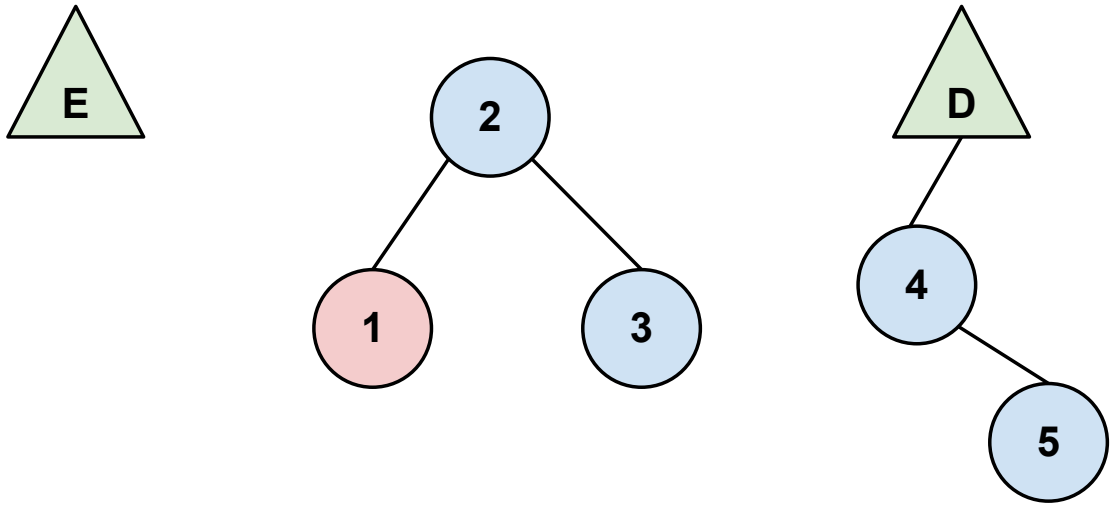
Exemplo: Top-Down 2/6



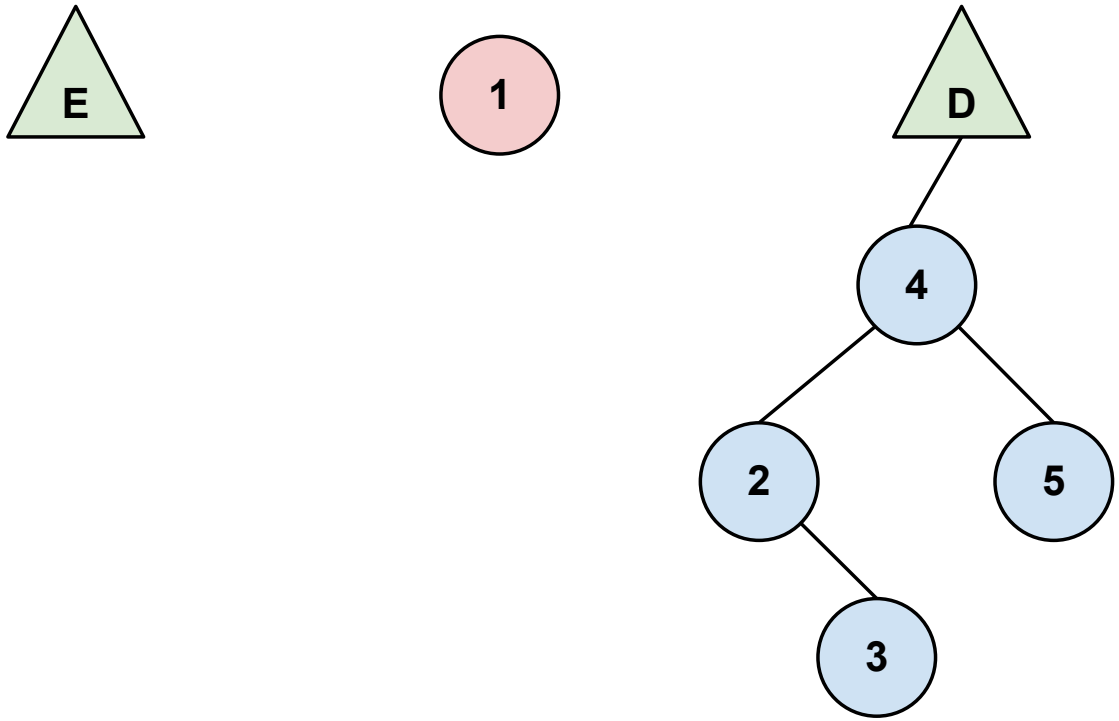
Exemplo: Top-Down 3/6



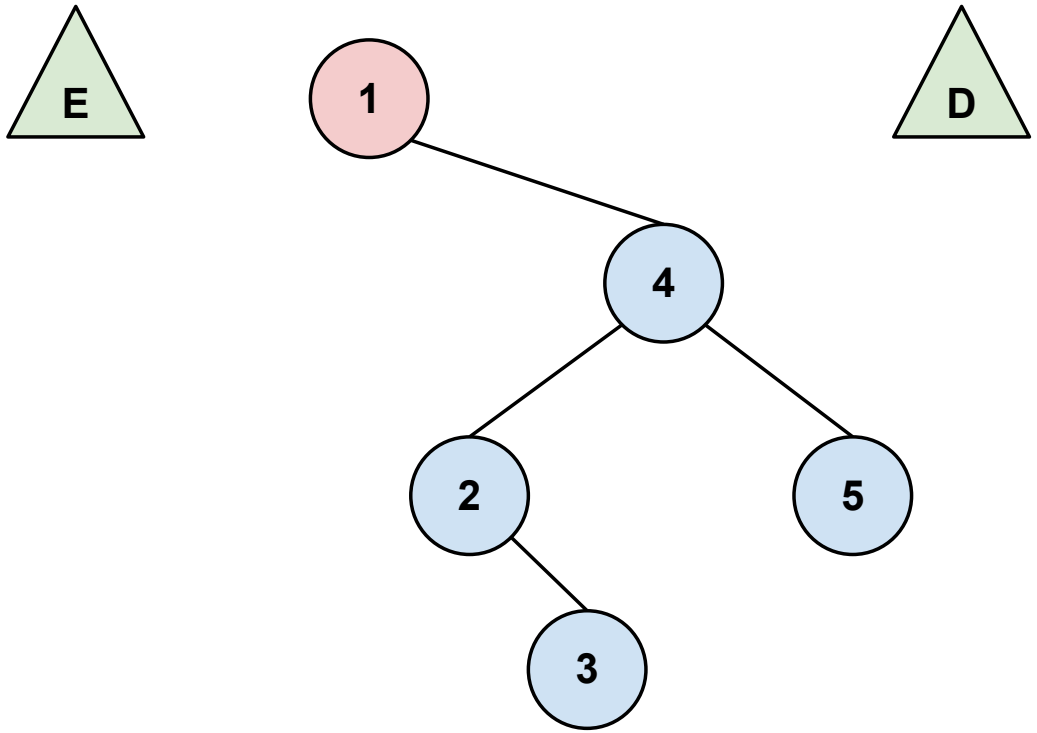
Exemplo: Top-Down 4/6



Exemplo: Top-Down 5/6



Exemplo: Top-Down 6/6



Capítulo xxxxx – Tabelas Hash

Pontos fundamentais a serem cobertos:

1.
2.
3.

Definição

Uma tabela hash é uma estrutura utilizada no mapeamento de chaves para seus respectivos valores. Por exemplo, um dicionário é uma estrutura que mapeia (relaciona) palavras aos seus significados.

Operações Básicas

Uma tabela hash atua como uma estrutura de dicionário ou vetor associativo, e suporta as seguintes operações básicas [**?**]:

- Inserção
- Busca
- Remoção

Sob hipóteses razoáveis (veremos adiante), todas as três operações podem ser implementadas com complexidade computacional próxima de $O(1)$.

Tabelas de endereço direto

- Utilizável quando o universo de chaves é suficientemente pequeno e representado por inteiros
- Para uma caso simplificado sem colisões de chaves, equivale ao uso de vetores, onde cada posição do vetor corresponde ao espaço na tabela para a entrada de chave igual à posição

Exemplo de tabela de endereço direto

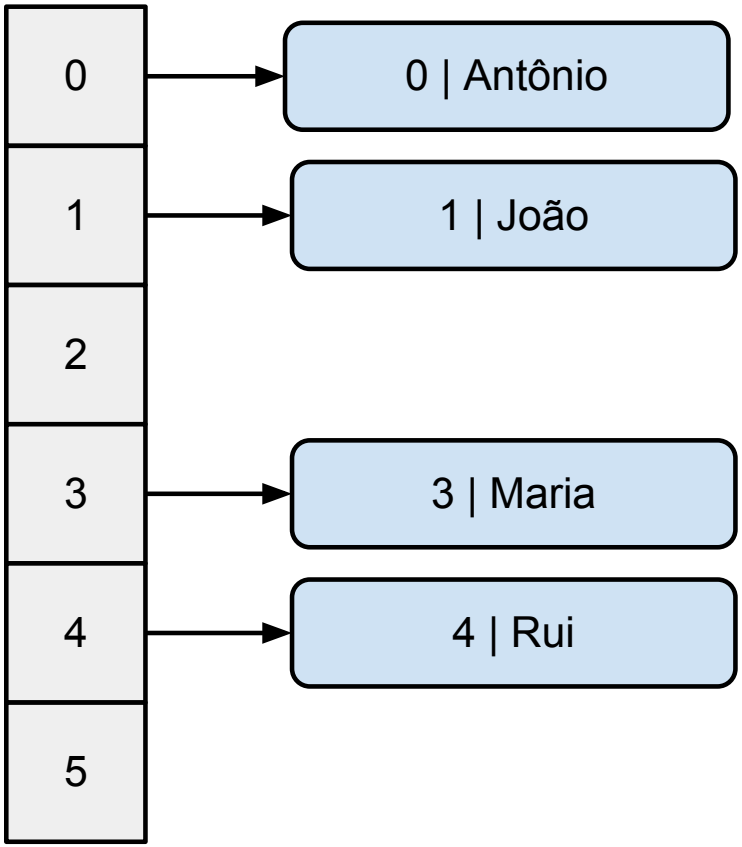


Figura: Tabela de endereço direto

Tabela de endereço direto - Inserção

```
INSERÇÃO(TABELA, DADO) {
    TABELA[ DADO->CHAVE ] = DADO
}
```

Tabela de endereço direto - Busca

```
BUSCA(TABELA, CHAVE) {
    return TABELA[ CHAVE ]
}
```

Tabela de endereço direto - Remoção

```
REMOÇÃO(TABELA, CHAVE) {
    TABELA[ CHAVE ] = NULO
}
```

Tabelas hash

No endereçamento direto teremos problemas nos seguintes casos:

- O universo (a faixa) de chaves é muito grande
- Os dados que deverão ser armazenados não possuem chaves numéricas

A solução está no uso de uma função de hash que faça o mapeamento de uma chave para um endereço válido de uma tabela.

Exemplo de tabela hash

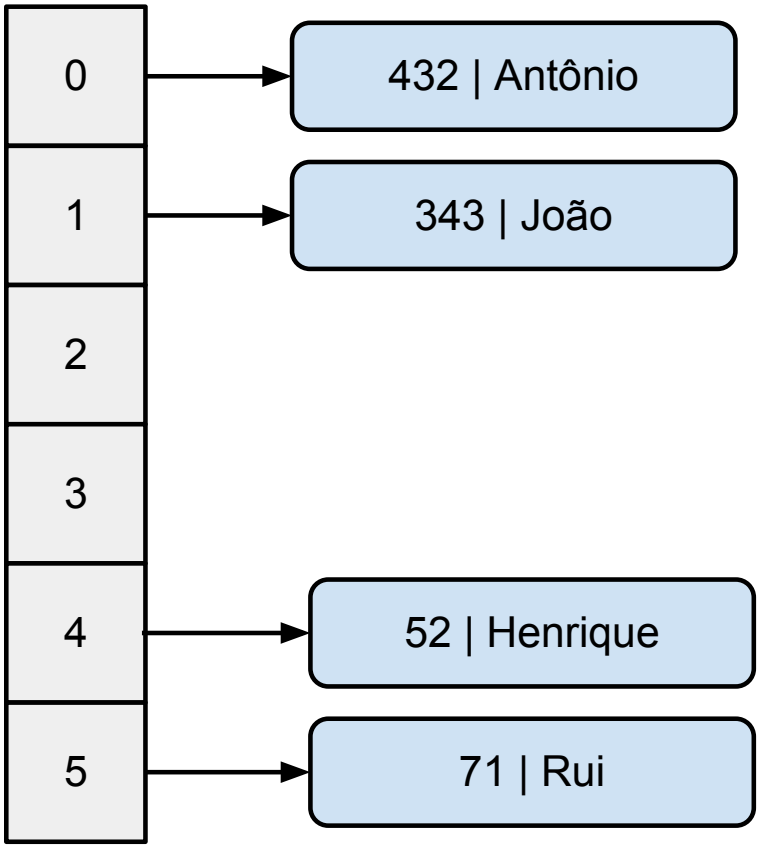


Figura: Tabela hash

Tabela hash - Inserção

```
INSERÇÃO(TABELA, DADO) {
    ENDEREÇO = HASH( DADO->CHAVE )
    TABELA[ ENDEREÇO ] = DADO
}
```

Tabela hash - Busca

```
BUSCA(TABELA, CHAVE) {
    ENDEREÇO = HASH( CHAVE )
    return TABELA[ ENDEREÇO ]
}
```

Tabela hash - Remoção

```
REMOÇÃO(TABELA, CHAVE) {
    ENDEREÇO = HASH( CHAVE )
    TABELA[ ENDEREÇO ] = NULO
}
```

Funções de hash

Necessidade: mapeamento da chave para um endereço.

Muitas vezes os dados que serão armazenados não possuem chaves numéricas (ou sua faixa é muito grande) e é necessário mapear um dado de outro tipo (como uma cadeia de caracteres) para um endereço.

Funções de hash

Necessidade: boa distribuição de endereços.

Na maioria das situações práticas, salvo aquelas nas quais os dados são conhecidos com antecedência e é possível realizar um hashing perfeito, uma função de hash irá retornar o mesmo endereço para diferentes dados em algum momento, gerando uma colisão.

Construção de uma boa função de hash

Logo, a função de hash utilizada deve:

- Mapear a chave para um endereço válido
- Ter uma boa distribuição de forma a minimizar as colisões
- Ser eficiente

Hash de chaves inteiras

Exemplos de funções de hash para chaves inteiras [**?**]:

- Método da divisão
- Método da multiplicação

Aqui definimos M como o número de endereços na tabela.

Funções de hash - método da divisão

```
HASH(CHAVE, M) {
    return CHAVE mod M
}
```

Método da divisão

Características:

- Método simples e rápido de ser computado
- Pode ter uma distribuição ruim dependendo do valor de M[1]

[1] ex.: M não deve ser uma potência de 2

Funções de hash - método da multiplicação

```
HASH(CHAVE, M) {
    AUX = M * ( (CHAVE * TAXA) mod 1)
    return floor( AUX )
}
```

Método da multiplicação

Características:

- A distribuição não é dependente do valor de M
- Funciona com qualquer valor de TAXA, mas a literatura sugere um valor próximo a

$$(\sqrt{5} - 1)/2 = 0,6180339887...$$

Para cadeias de caracteres

- Somatório (ruim)
- djb2

Funções de hash - somatório

```
HASH(CHAVE, M) {
    AUX = 0

    PARA CADA CARACTERE C EM CHAVE {
        AUX = AUX + C
    }

    return AUX mod M
}
```

Funções de hash - djb2

```
HASH(CHAVE, M) {
    AUX = 5381

    PARA CADA CARACTERE C EM CHAVE {
        AUX = AUX * 33 + C
    }

    return AUX mod M
}
```

Resolução de Colisões

- Salvo situações especiais, chaves diferentes serão mapeadas para a mesma posição, causando assim uma *colisão*
- Boas funções de *hash* diminuem o número de colisões, mas em uma situação normal elas irão ocorrer, logo a tabela deve tratar colisões
- Dois métodos comuns: por encadeamento e por endereçamento aberto

Resolução de Colisões por Encadeamento

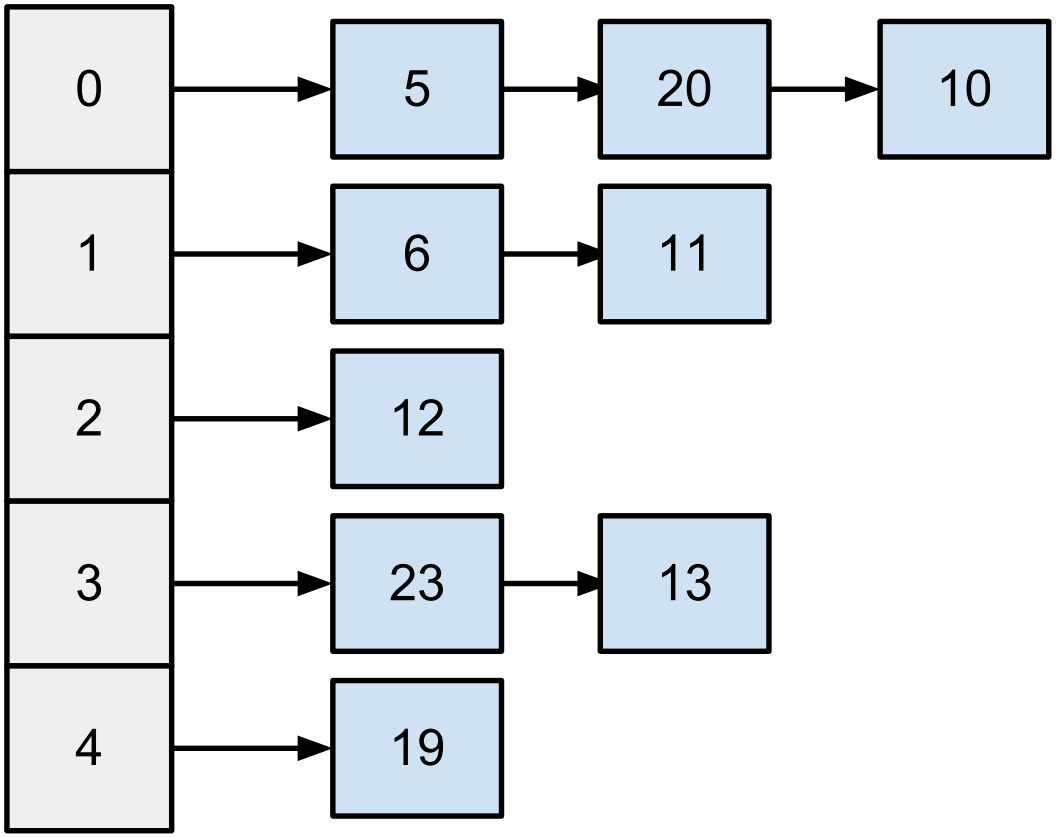


Figura: Resolução de colisões por encadeamento (end = chave mod 5)

Tabela hash (encadeamento) - Inserção

```
INSERÇÃO(TABELA, DADO) {
    ENDEREÇO = HASH( DADO->CHAVE )
    INSERE_LISTA( TABELA[ ENDEREÇO ], DADO )
}
```

Tabela hash (encadeamento) - Busca

```
BUSCA(TABELA, CHAVE) {
    ENDEREÇO = HASH( CHAVE )
    return BUSCA_LISTA( TABELA[ ENDEREÇO ], CHAVE )
}
```

Tabela hash (encadeamento) - Remoção

```
REMOÇÃO(TABELA, CHAVE) {
    ENDEREÇO = HASH( CHAVE )
    REMOVE_LISTA( TABELA[ ENDEREÇO ], CHAVE )
}
```

Endereçamento aberto

- Todos os elementos são armazenados na própria tabela
- Quando há uma colisão, escolhe-se uma nova posição para o novo dado
- A tabela pode ficar cheia, necessitando de redimensionamento
- As operações utilizam o processo de sondagem para encontrar a posição de um elemento

Endereçamento aberto

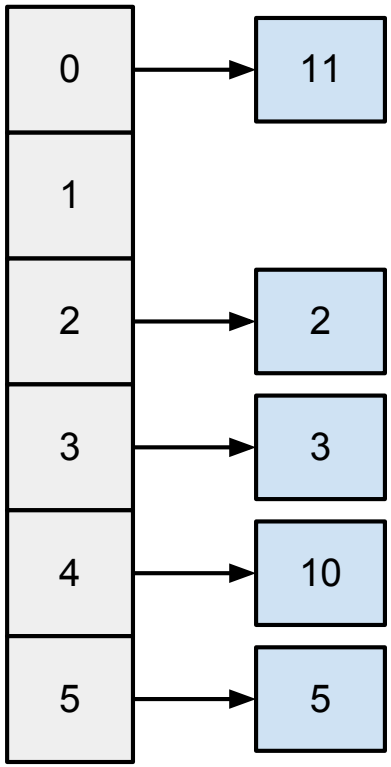


Figura: Endereçamento aberto, onde hash(chave) = chave % M

Tabela hash (endereçamento aberto) - Inserção

```
INSERÇÃO(TABELA, DADO) {
    I = 0

    FAÇA
        ENDEREÇO = HASH'( DADO->CHAVE, I )

        SE TABELA[ ENDEREÇO ] ESTÁ VAZIA
            TABELA[ ENDEREÇO ] = DADO
            return
        SENÃO
            I = I + 1
    ATÉ QUE I = M

    ERRO("TABELA CHEIA")
}
```

Tabela hash (endereçamento aberto) - Busca

```
BUSCA(TABELA, CHAVE) {
    I = 0

    FAÇA
        ENDEREÇO = HASH'( CHAVE, I )

        SE TABELA[ ENDEREÇO ] = CHAVE
            return TABELA[ ENDEREÇO ]
        SENÃO
            I = I + 1
    ATÉ QUE TABELA[ENDEREÇO] = VAZIO OU I = M

    return NULO
}
```

Tabela hash (endereçamento aberto) - Remoção

```
REMOÇÃO(TABELA, CHAVE) {
    I = 0

    FAÇA
        ENDEREÇO = HASH'( CHAVE, I )

        SE TABELA[ ENDEREÇO ] = CHAVE
            TABELA[ ENDEREÇO ] = REMOVIDO
            return TRUE
        SENÃO
            I = I + 1
    ATÉ QUE TABELA[ENDEREÇO] = VAZIO OU I = M

    return FALSE
}
```

Sondagem linear

$$hash'(chave, i) = (hash(chave) + i)\%M \tag{1}$$

Sondagem quadrática

$$hash'(chave, i) = (hash(chave) + c_1 i + c_2 i^2)\%M \quad (2)$$