Estrutura de Dados

Claudio Cesar de Sá

claudio.sa@udesc.br

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

19 de julho de 2017

Sumário (1)

Agradecimentos

Vários autores e colaboradores ...

- Alessandro Ferreira Leite – ???
- Lucas Hermman Negri – IFMS
- Gilmário – UDESC
- Ao Google Images ...

Capítulo xxxxx – Estrutura de Dados Linear: Pilha

Pontos fundamentais a serem cobertos:

1.
2.
3.

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

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

```
#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 está 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

```c
/**
 * 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 – 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

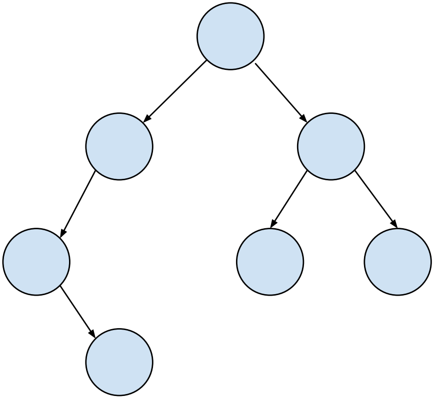


Figura: Exemplo de uma árvore

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

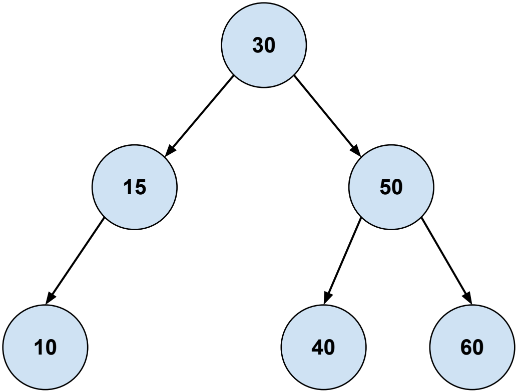
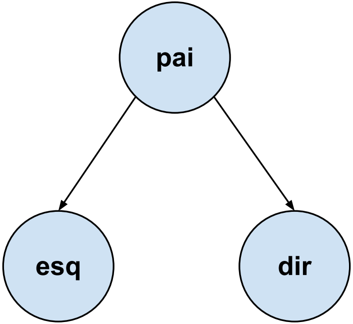


Figura: Exemplo de árvore binária de busca

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

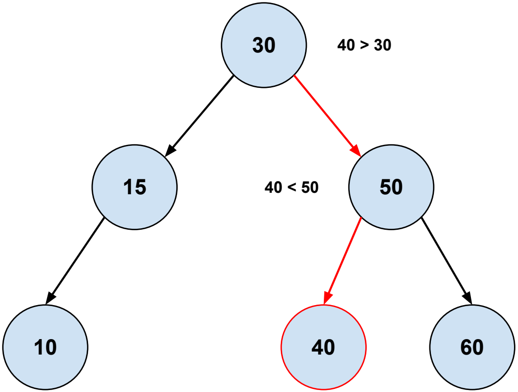


Figura: Exemplo de inserção da chave 40

Árvore Binária de Busca - Inserção

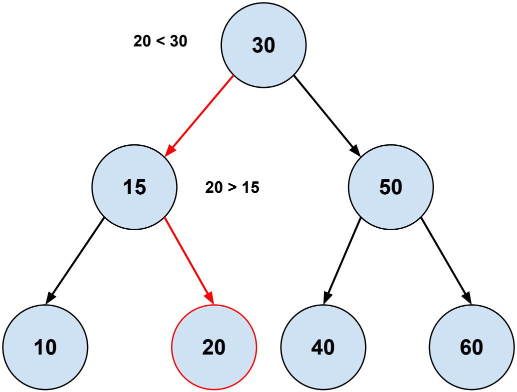


Figura: Exemplo de inserção da chave 20

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

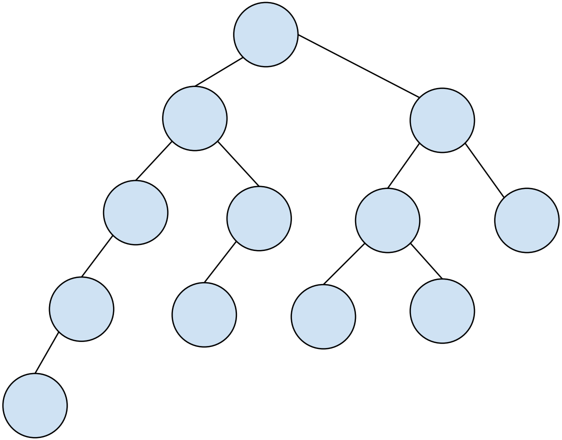


Figura: Exercício: determine a altura de cada subárvore.

Cálculo da Altura

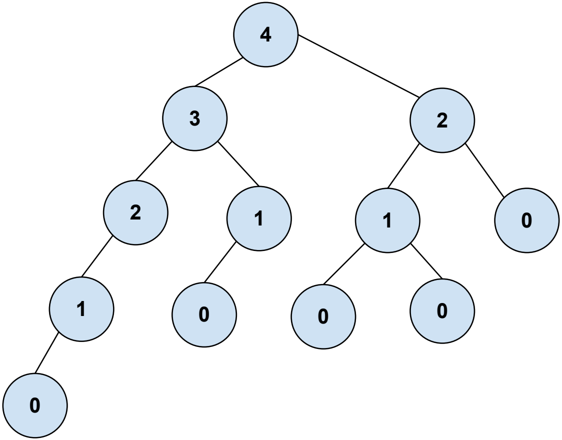


Figura: Resposta do exercício.

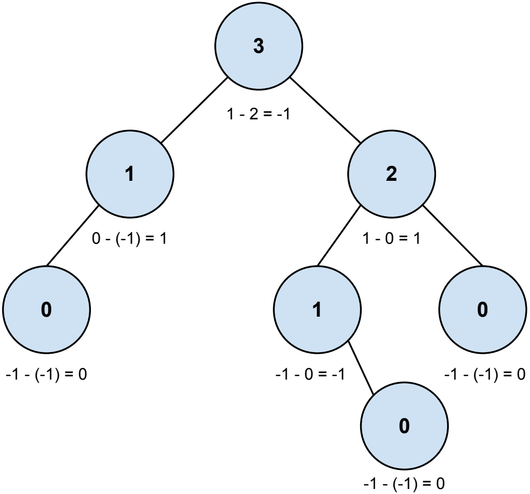
Cálculo do Fator de Balanceamento

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

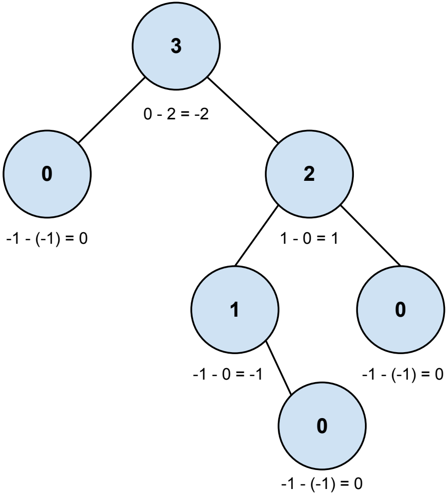
Balanceamento

- 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

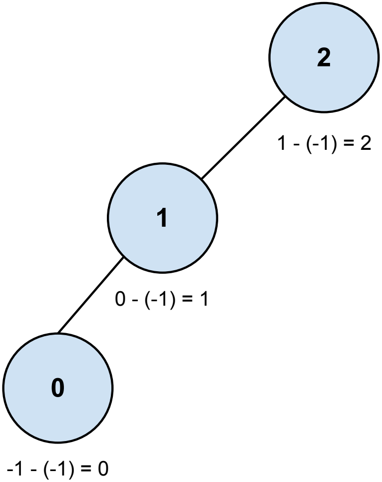


Operação de rotação

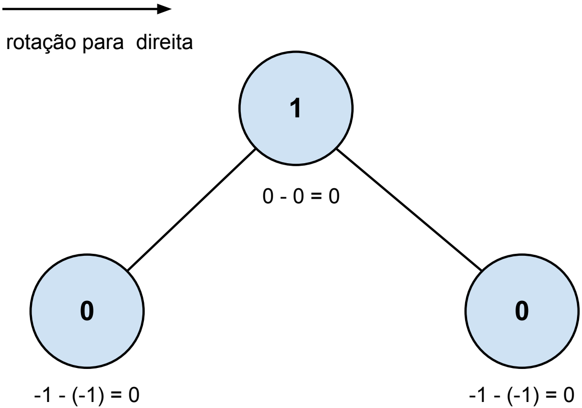
```
ROTACAO_DIREITA(RAIZ) {
    PIVO      = RAIZ->ESQ
    RAIZ->ESQ = PIVO->DIR
    PIVO->DIR = RAIZ
    RAIZ      = PIVO
}

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

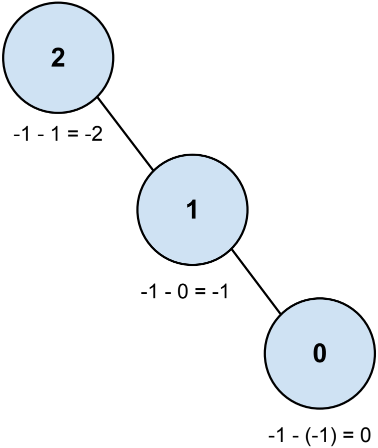
Rotação para Direita



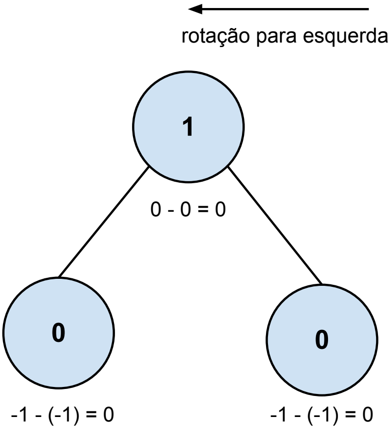
Rotação para Direita



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

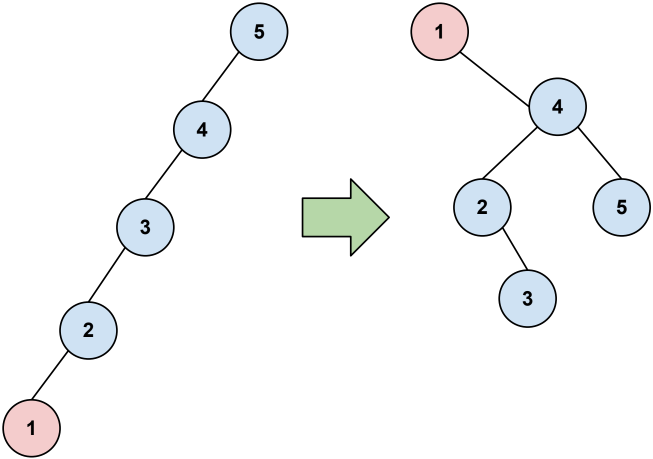
Árvore de Espalhamento

- 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

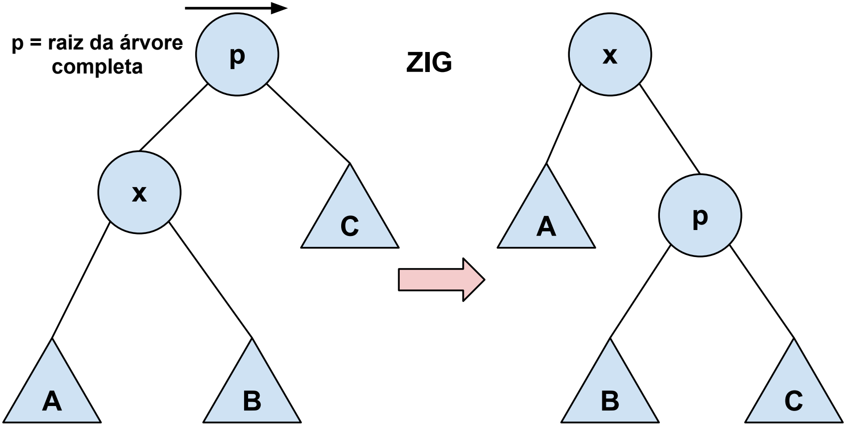
Duas estratégias:

- **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.

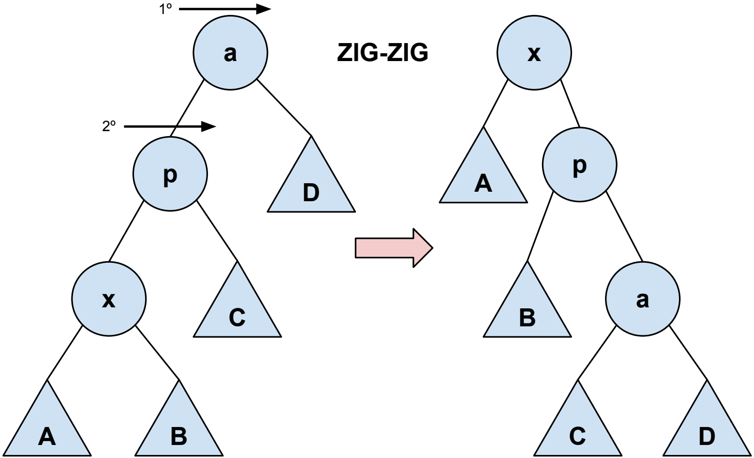
Espalhamento Bottom-Up

- 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

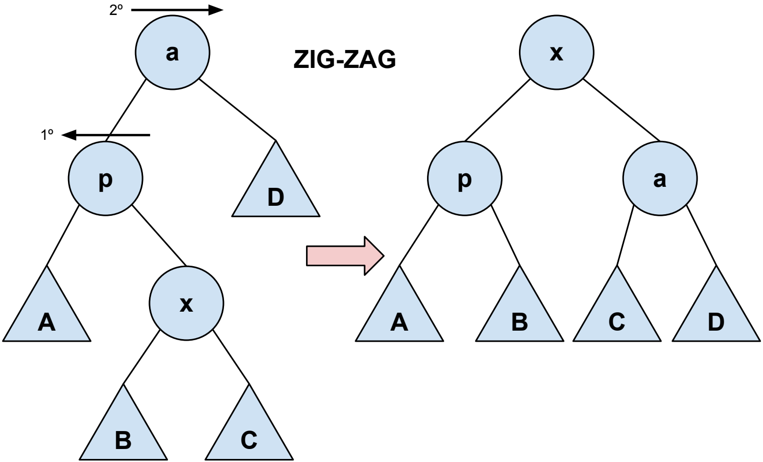
Caso 1: ZIG



Caso 2: ZIG-ZIG



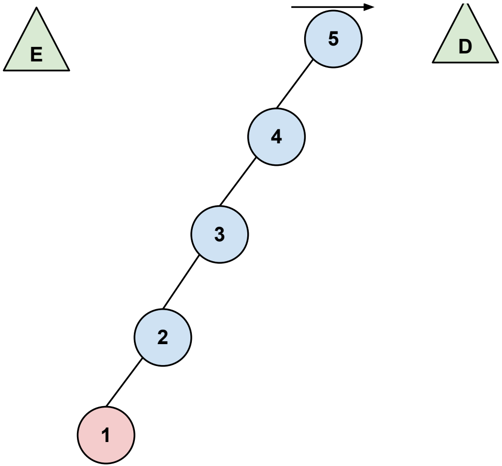
Caso 3: ZIG-ZAG



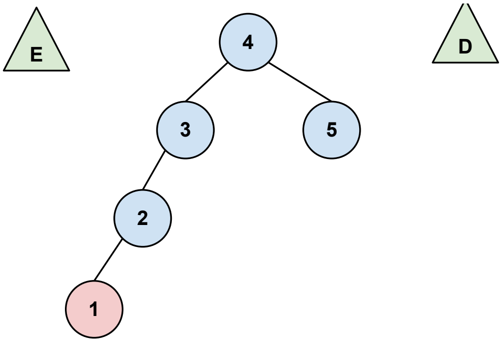
Espalhamento Top-Down

- 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

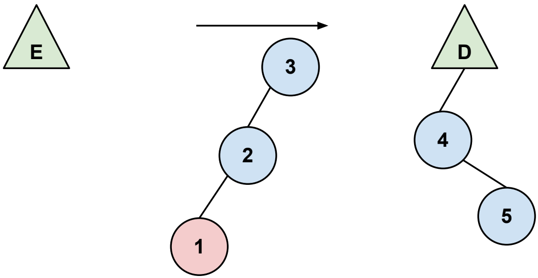
Exemplo: Top-Down 1/6



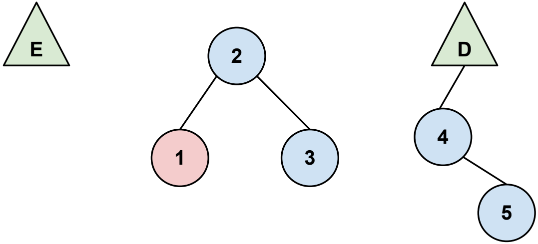
Exemplo: Top-Down 2/6



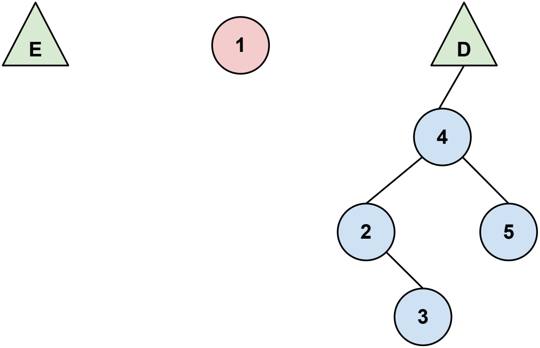
Exemplo: Top-Down 3/6



Exemplo: Top-Down 4/6



Exemplo: Top-Down 5/6



Exemplo: Top-Down 6/6