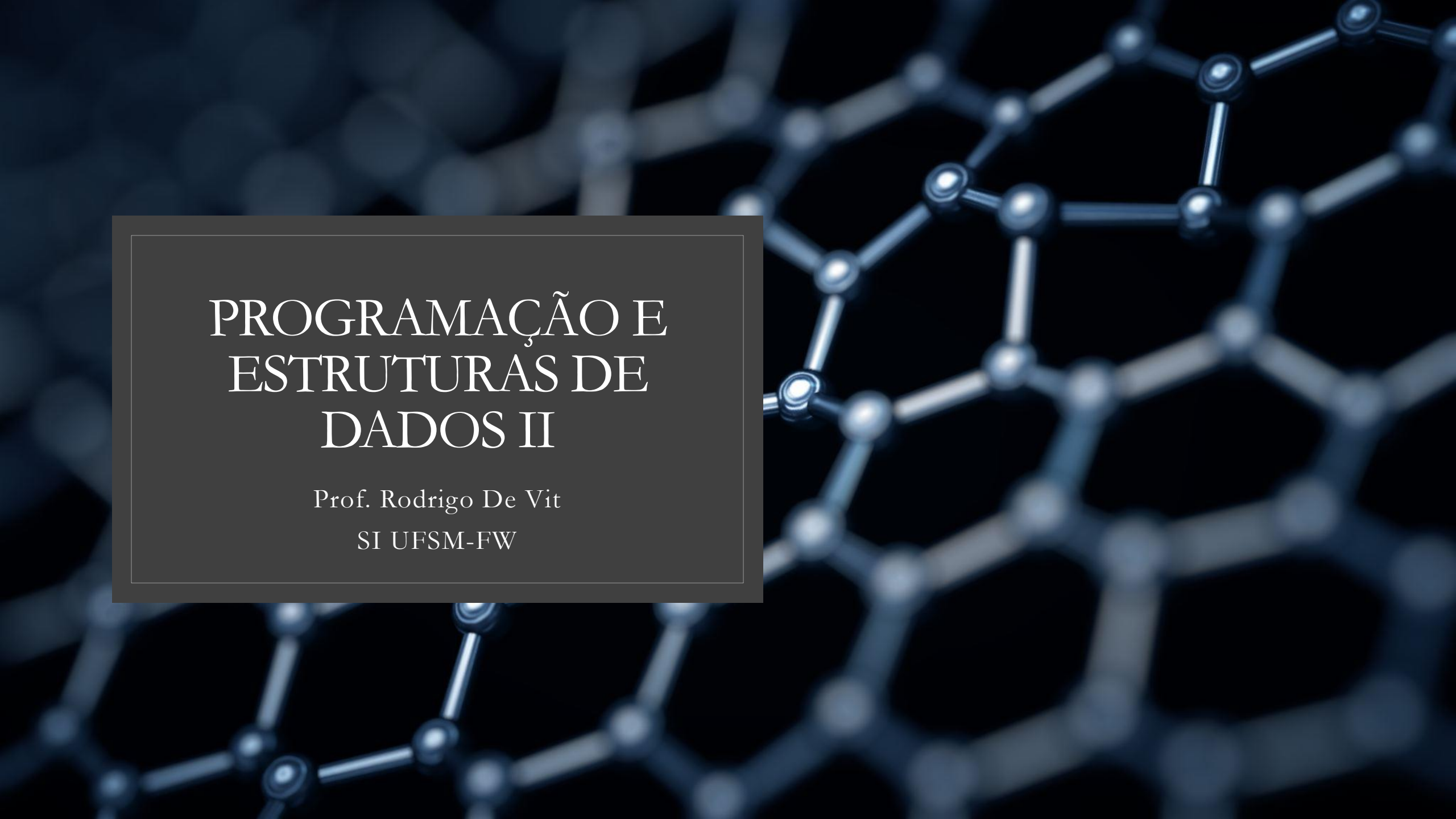


# PROGRAMAÇÃO E ESTRUTURAS DE DADOS II

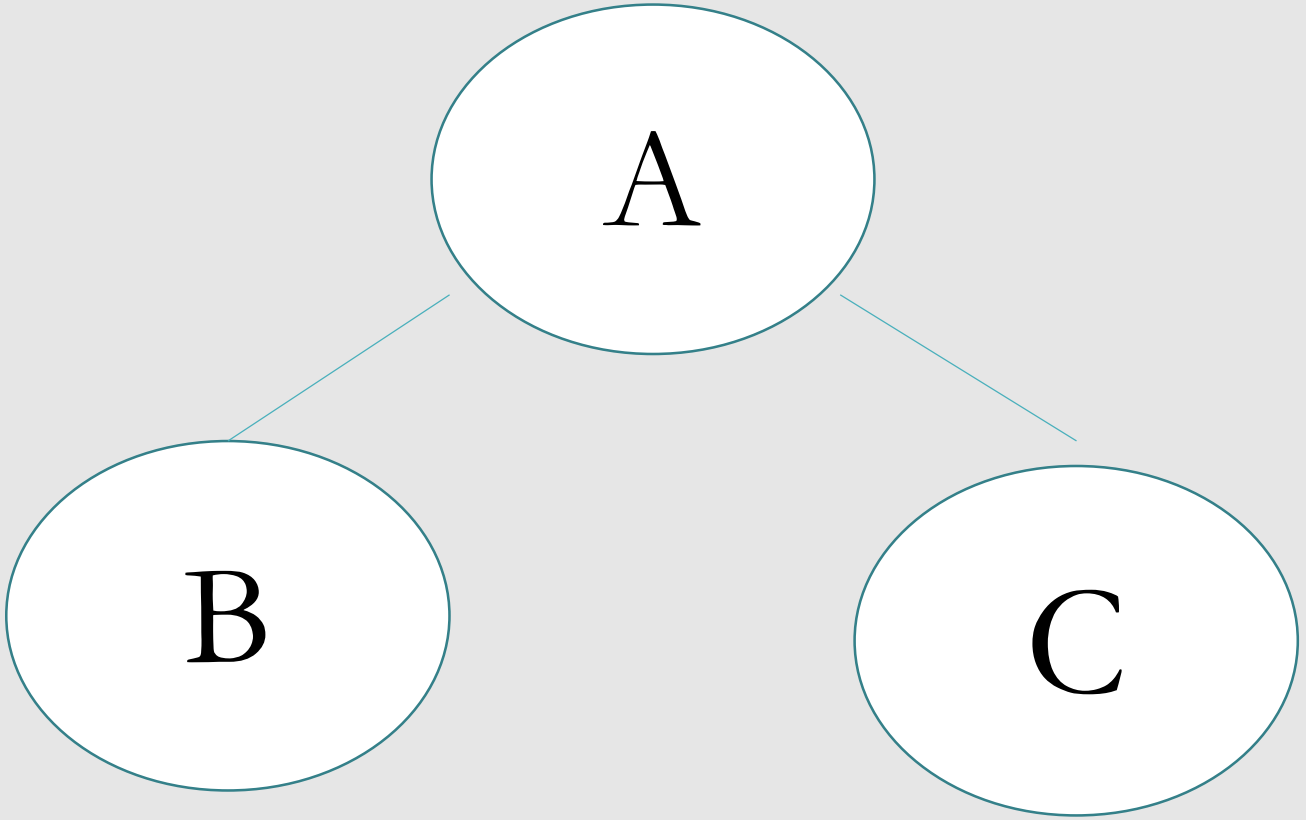
Prof. Rodrigo De Vit

SI UFSM-FW

# Revisão - Representação em C


- Árvore binária em C.
- Enxerto e poda em árvores. (vide `main()`)

# Revisão - Ordens de percurso em árvores binárias




Pré: ABC
Central: BAC
Pós: BCA

# Revisão - Representação em C Árvores Genéricas

```
struct arvgen {
        char info;
        struct arvgen *prim;
        struct arvgen *prox;
};
```

- Uma solução que leva a um aproveitamento melhor do espaço utiliza uma "lista de filhos": um nó aponta apenas para seu primeiro (prim) filho, e cada um de seus filhos, exceto o último, aponta para o próximo (prox) irmão.
- TAD:

cria: cria um nó folha, dada a informação a ser armazenada;

insere: insere uma nova sub-árvore como filha de um dado nó;

imprime: percorre todos os nós e imprime suas informações;

busca: verifica a ocorrência de um determinado valor num dos nós da árvore;

libera: libera toda a memória alocada pela árvore.

ArvGen.c

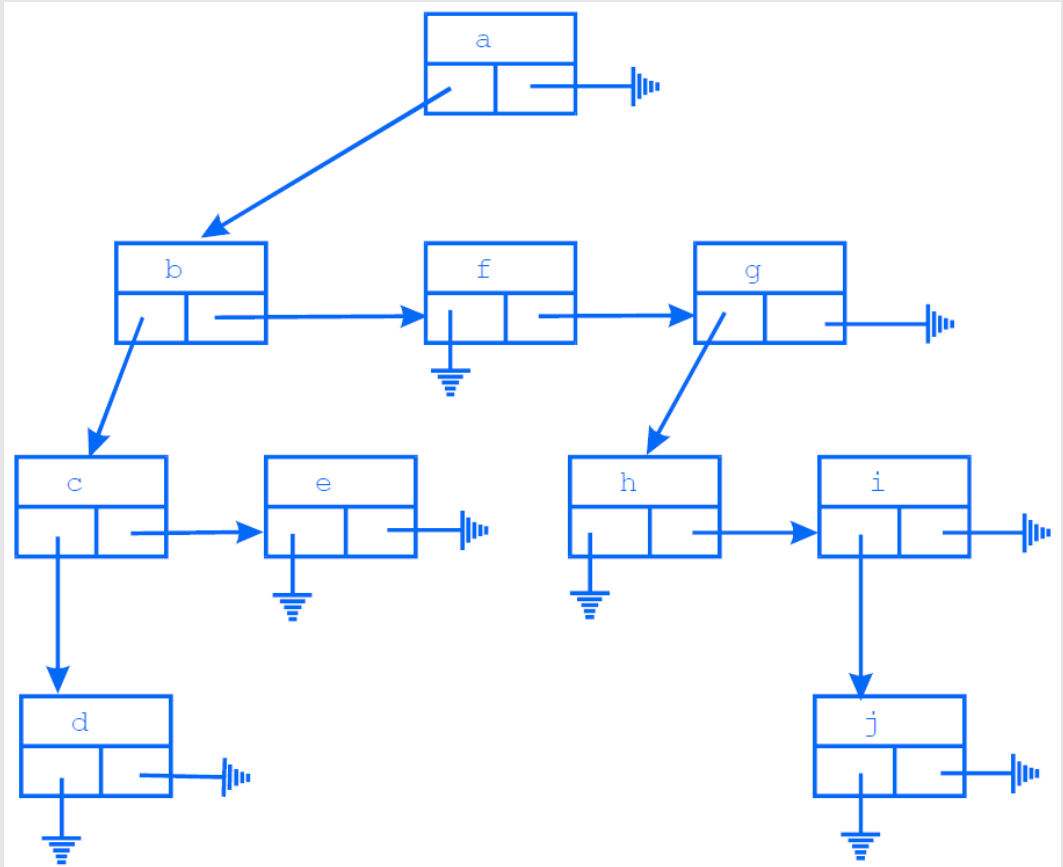
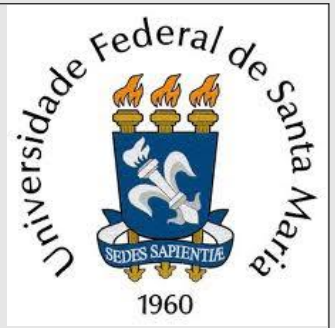



Figura 13.9: Exemplo usando "lista de filhos".

# Busca Binária




- **No caso dos elementos do vetor estarem em ordem,** podemos aplicar um algoritmo mais eficiente para realizarmos a busca. Trata-se do algoritmo de ***busca binária*****. A ideia do algoritmo é testar o elemento que buscamos com o valor do elemento armazenado no meio do vetor.** Se o elemento que buscamos **for menor que o elemento do meio,** sabemos que, se o elemento estiver presente no vetor, **ele estará na primeira parte do vetor; se for maior, estará na segunda parte do vetor; se for igual, achamos o elemento no vetor.** Se concluirmos que o elemento está numa das partes do vetor, **repetimos o procedimento considerando apenas a parte que restou: comparamos o elemento que buscamos com o elemento armazenado no meio dessa parte. Este procedimento é continuamente repetido, subdividindo a parte de interesse, até encontrarmos o elemento ou chegarmos a uma parte do vetor com tamanho zero.**

```
int busca_bin (int n, int* vet, int elem)
{
   /* no inicio consideramos todo o vetor */
   int ini = 0;
   int fim = n-1;
   int meio;
   /* enquanto a parte restante for maior que zero */
   while (ini <= fim) {
   meio = (ini + fim) / 2;
   if (elem < vet[meio])
      fim = meio - 1; /* ajusta posicão final */
   else if (elem > vet[meio])
      ini = meio + 1; /* ajusta posicão inicial */
   else
      return meio; /* elemento encontrado */
}
   /* não encontrou: restou parte de tamanho zero */
   return -1;
}
```

# bsearch() e qsort(), de <stdlib.h>


```
/* bsearch example */
#include <stdio.h>      /* printf */
#include <stdlib.h>     /* qsort, bsearch, NULL */

int compareints (const void * a, const void * b)
{
  return ( *(int*)a - *(int*)b );
}

int values[] = { 50, 20, 60, 40, 10, 30 };
```

```
int main ()
{
  int * pItem;
  int key = 40;
  qsort (values, 6, sizeof (int), compareints);
  pItem = (int*) bsearch (&key, values, 6, sizeof (int), compareints);
  if (pItem!=NULL)
    printf ("%d is in the array.\n",*pItem);
  else
    printf ("%d is not in the array.\n",key);
  return 0;
}
```

# Árvore binária de busca


- O **algoritmo de busca binária** apresenta bom desempenho computacional e **deve ser usado quando temos os dados ordenados armazenados num vetor**. No entanto, se precisarmos **inserir e remover elementos da estrutura, e ao mesmo tempo dar suporte a eficientes funções de busca, a estrutura de vetor (e, consequentemente, o uso do algoritmo de busca binária) não se torna adequada.** Para **inserirmos um novo elemento num vetor ordenado, temos que rearrumar os elementos no vetor, para abrir espaço para a inserção do novo elemento.** Situação análoga ocorre quando removemos um elemento do vetor. **Precisamos portanto de uma estrutura dinâmica que dê suporte a operações de busca.**
- Deve-se verificar a relação entre o número de nós de uma árvore binária e sua altura. **A cada nível, o número (potencial) de nós vai dobrando**.
- Assim, dizemos que uma árvore binária de altura *h* pode ter no máximo *O(2h)* nós, ou, pelo outro lado, que uma árvore binária com *n* nós pode ter uma altura mínima de *O(log n)*. **Essa relação entre o número de nós e a altura mínima da árvore é importante porque se as condições forem favoráveis, podemos alcançar qualquer um dos *n* nós de uma árvore binária a partir da raiz em, no máximo, *O(log n)* passos.** Se tivéssemos os *n* nós **em uma lista linear, o número máximo de passos seria *O(n)*, e, para os valores de *n* encontrados na prática, *log n* é muito menor do que *n*.**

# Árvore binária de busca

- As árvores binárias que serão consideradas nesta seção têm uma propriedade fundamental: **o valor associado à raiz é sempre maior que o valor associado a qualquer nó da sub-árvore à esquerda (*sae*), e é sempre menor que o valor associado a qualquer nó da sub-árvore à direita (*sad*).** Essa propriedade garante que, **quando a árvore é percorrida em ordem simétrica (*sae - raiz - sad*), os valores são encontrados em ordem crescente.**

- Vide abp.c.

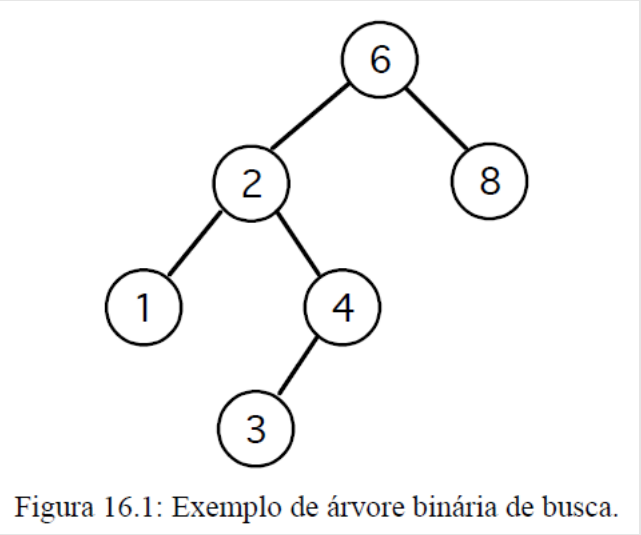


Figura 16.1: Exemplo de árvore binária de busca.

# Árvore binária de busca - remoção

- Essa operação é um pouco mais complexa que a de inserção. Existem três situações possíveis. **A primeira, e mais simples, é quando se deseja retirar um elemento que é folha da árvore** (isto é, um elemento que não tem filhos). Neste caso, basta retirar o elemento da árvore e atualizar o pai, pois seu filho não existe mais.
- **A segunda situação, ainda simples, acontece quando o nó a ser retirado possui um único filho.** Para **retirar esse elemento é necessário antes acertar o ponteiro do pai, "pulando" o nó: o único neto passa a ser filho direto.**

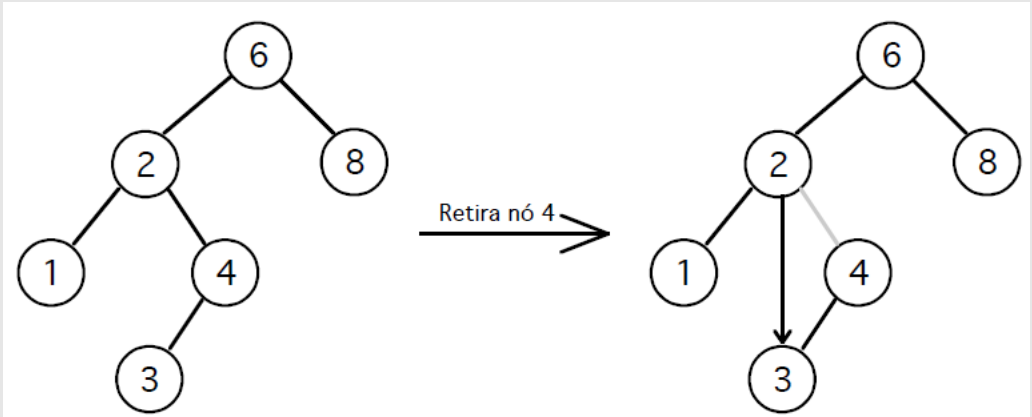
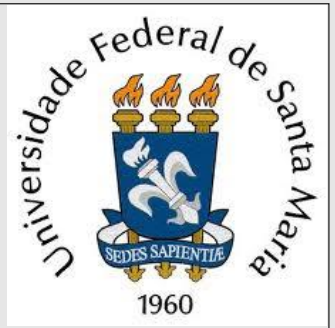


Figura 16.2: Retirada de um elemento com um único filho.

# Árvore binária de busca - remoção




- **O caso complicado ocorre quando o nó a ser retirado tem dois filhos.** Para poder retirar esse nó da árvore, devemos proceder da seguinte forma:
- **a)** encontramos o elemento que precede o elemento a ser retirado na ordenação. Isto equivale a encontrar o elemento mais à direita da sub-árvore à esquerda;
- **b)** trocamos a informação do nó a ser retirado com a informação do nó encontrado;
- **c)** retiramos o nó encontrado (que agora contém a informação do nó que se deseja retirar). Observa-se que retirar o nó mais à direita é trivial, pois esse é um nó folha ou um nó com um único filho (no caso, o filho da direita nunca existe).
- **O procedimento descrito acima deve ser seguido para não haver violação da ordenação da árvore. Observamos que, análogo ao que foi feito com o nó mais à direita da sub-árvore à esquerda, pode ser feito com o nó mais à esquerda da sub-árvore à direita (que é o nó que segue o nó a ser retirado na ordenação).**

# Árvore binária de busca - remoção

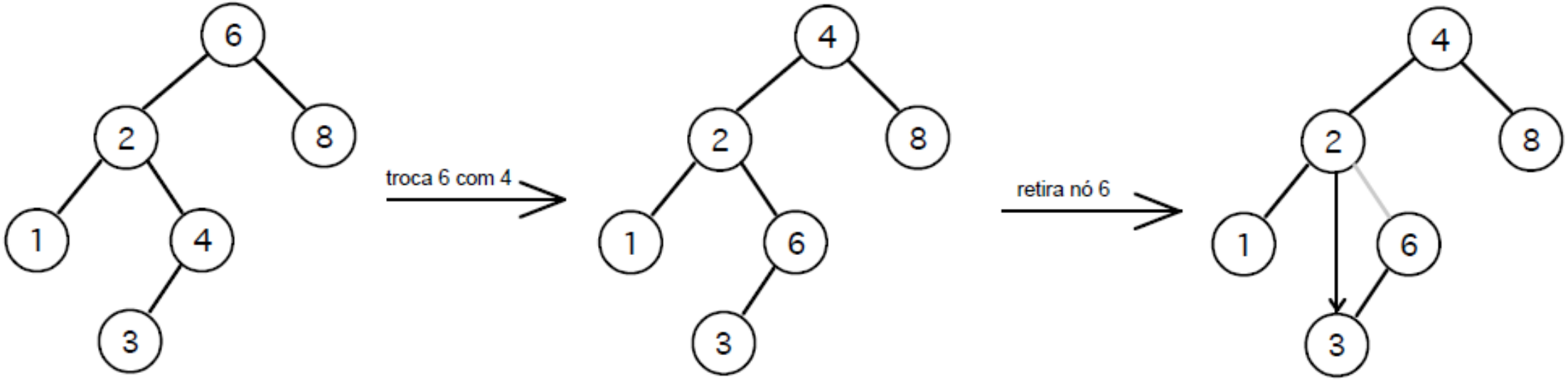



Figura 16.3: Exemplo da operação para retirar o elemento com informação igual a 6.