

# Estruturas de Dados

## Módulo 17 - Busca

PONTIFÍCIA UNIVERSIDADE CATÓLICA  
DO RIO DE JANEIRO



# Referências

Waldemar Celes, Renato Cerqueira, José Lucas Rangel,  
*Introdução a Estruturas de Dados*, Editora Campus  
(2004)

Capítulo 17 – Busca

# Tópicos

- Busca em vetor
  - Busca linear
  - Busca binária
  - Algoritmo genérico
- Árvore binária de busca
  - Apresentação
  - Operações em árvores binárias de busca
  - Árvores balanceadas

# Busca em Vetor

- Busca em vetor:
  - entrada: vetor `vet` com `n` elementos  
elemento `elem`
  - saída: `n` se o elemento `elem` ocorre em `vet[n]`  
`-1` se o elemento não se encontra no vetor

# Busca em Vetor

- Busca Linear:
  - percorra o vetor `vet`, elemento a elemento verificando se `elem` é igual a um dos elementos de `vet`

```
int busca (int n, int* vet, int elem)
{
    int i;

    for (i=0; i<n; i++) {
        if (elem == vet[i])
            return i;                /* elemento encontrado */
    }

    /* elemento não encontrado após percorrer todo o vetor */
    return -1;
}
```

# Busca em Vetor

- Análise da Busca Linear:
  - pior caso:
    - $n$  comparações, onde  $n$  representa o número de elementos do vetor
      - desempenho computacional varia linearmente em relação ao tamanho do problema (algoritmo de busca *linear*)
    - complexidade:  $O(n)$
  - caso médio:
    - $n/2$  comparações
      - desempenho computacional continua variando linearmente em relação ao tamanho do problema
    - complexidade:  $O(n)$

# Busca em Vetor

- Busca Linear (vetor em ordem crescente):

```
int busca_ord (int n, int* vet, int elem)
{
    int i;

    for (i=0; i<n; i++) {
        if (elem == vet[i])
            return i; /* elemento encontrado */
        else if (elem < vet[i])
            return -1; /* interrompe busca */
    }

    /* percorreu todo o vetor e não encontrou elemento */
    return -1;
}
```

0	1	1	1	2	3	4	5	7	8
---	---	---	---	---	---	---	---	---	---



6

# Busca em Vetor

- Análise de busca linear (vetor ordenado):
  - caso o elemento procurado não pertença ao vetor, a busca linear com vetor ordenado apresenta um desempenho ligeiramente superior à busca linear
  - pior caso:
    - algoritmo continua sendo linear
    - complexidade:  $O(n)$

# Busca em Vetor

- Busca binária:
  - entrada: vetor vet com n elementos, ordenado  
elemento elem
  - saída: n se o elemento elem ocorre em vet[n]  
-1 se o elemento não se encontra no vetor
  - procedimento:
    - compare elem com o elemento do meio de vet
    - se elem for menor, pesquise na primeira metade do vetor
    - se elem for maior, pesquise na segunda parte do vetor
    - se for igual, retorne a posição
    - continue o procedimento, subdividindo a parte de interesse, até encontrar o elemento ou chegar a uma parte do vetor com tamanho 0

```

int busca_bin (int n, int* vet, int elem)
{
    /* no início 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 posição final */
        else if (elem > vet[meio])
            ini = meio + 1;      /* ajusta posição inicial */
        else
            return meio;        /* elemento encontrado */
    }

    /* não encontrou: restou parte de tamanho zero */
    return -1;
}

```

# Busca em Vetor

- Análise de busca binária
  - pior caso:  $O(\log n)$ 
    - elemento não ocorre no vetor
    - 2 comparações são realizadas a cada ciclo
    - a cada repetição, a parte considerada na busca é dividida à metade
    - logo, no pior caso, são necessárias  $\log n$  repetições

Repetição	Tamanho do problema
<i>1</i>	<i>n</i>
<i>2</i>	<i>n/2</i>
<i>3</i>	<i>n/4</i>
...	...
<i>log n</i>	<i>1</i>

# Busca em Vetor

- Busca binária – implementação recursiva:
  - dois casos a tratar:
    - busca deve continuar na primeira metade do vetor:
      - chamada recursiva com parâmetros:
        - » o número de elementos da primeira parte restante
        - » o mesmo ponteiro para o primeiro elemento (pois a primeira parte tem o mesmo primeiro elemento do que o vetor como um todo)
    - busca deve continuar apenas na segunda parte do vetor:
      - chamada recursiva com parâmetros:
        - » número de elementos restantes
        - » ponteiro para o primeiro elemento dessa segunda parte
      - valor retornado deve ser corrigido

```

int busca_bin_rec (int n, int* vet, int elem)
{
    /* testa condição de contorno: parte com tamanho zero */
    if (n <= 0)
        return -1;
    else {
        /* deve buscar o elemento do meio */
        int meio = n / 2;

        if (elem < vet[meio])
            return busca_bin_rec(meio,vet,elem);
        else if (elem > vet[meio])
        {
            int r = busca_bin_rec(n-1-meio, &vet[meio+1],elem);
            if (r<0) return -1;
            else    return meio+1+r;        /* correção do valor retornado */
        }
        else
            return meio;                    /* elemento encontrado */
    }
}

```

# Busca em Vetor

- Busca binária genérica da biblioteca padrão:
  - disponibilizada via a biblioteca [\*stdlib.h\*](#)
  - independe do tipo de dado armazenado no vetor
  - implementação segue os princípios discutidos na implementação do algoritmo de busca binária em vetor

# Busca em Vetor

- Protótipo da busca binária da biblioteca padrão:

```
void* bsearch (void* info, void *v, int n, int tam,  
              int (*cmp)(const void*, const void*));
```

retorno: se elemento for encontrado, o endereço do elemento no vetor  
caso contrário, NULL

**info:** ponteiro para a chave de busca (dado que se deseja buscar no vetor)

**v:** ponteiro para o primeiro elemento do vetor  
(vetor ordenado segundo critério definido pela função de comparação)

**n:** número de elementos do vetor

**tam:** tamanho, em bytes, de cada elemento do vetor

**cmp:** ponteiro para a função de comparação

**const:** modificador de tipo para garantir que a função não modificará os valores dos elementos (devem ser tratados como constantes)

# Busca em Vetor

- Função de comparação:

```
int nome (const void*, const void*);
```

- definida pelo cliente
- recebe dois ponteiros genéricos (do tipo `void*`)
  - apontam para os dois elementos a comparar
  - modificador de tipo `const` garante que a função não modificará os valores dos elementos (devem ser tratados como constantes)
- deve retornar `-1`, `0`, ou `1`, se o primeiro elemento for menor, igual, ou maior que o segundo, respectivamente, de acordo com o critério de ordenação adotado

# Busca em Vetor

- Exemplo 1:
  - vetor de valores inteiros em ordem crescente

# Busca em Vetor

- Função de comparação para int:
  - recebe dois ponteiros para `int`

```
/* função de comparação de inteiros */
static int comp_int (const void* p1, const void* p2)
{
    /* converte ponteiros genéricos para ponteiros de int */
    int *info = (int*)p1;
    int *elem = (int*)p2;
    /* dados os ponteiros de int, faz a comparação */
    if (*info < *elem) return -1;
    else if (*info > *elem) return 1;
    else return 0;
}
```

```

/* Ilustra uso do algoritmo bsearch para vetor de int */
#include <stdio.h>
#include <stdlib.h>

/* função de comparação de inteiros - ver transparência anterior */
static int comp_int (const void* p1, const void* p2)
{...}

/* programa que faz a busca em um vetor */
int main (void)
{
    int v[8] = {12,25,33,37,48,57,86,92};
    int e = 57;      /* informação que se deseja buscar */
    int* p;
    p = (int*)bsearch(&e,v,8,sizeof(int),comp_int);
    if (p == NULL)
        printf("Elemento nao encontrado.\n");
    else
        printf("Elemento encontrado no indice: %d\n", p-v);
    return 0;
}

```

# Busca em Vetor

- Exemplo 2:
  - vetor de ponteiros para a estrutura aluno
  - chave de busca dada pelo nome do aluno

```
/* estrutura representando um aluno*/  
struct aluno {  
    char nome[81];           /* chave de ordenação */  
    char mat[8];  
    char turma;  
    char email[41];  
};  
typedef struct aluno Aluno;  
  
Aluno* vet[N];              /* vetor de ponteiros para Aluno */
```

# Busca em Vetor

- Função de comparação
  - recebe dois ponteiros que referenciam tipos distintos:
    - um ponteiro para uma cadeia de caracteres
    - um ponteiro para um elemento do vetor (no caso será um ponteiro para ponteiro de aluno, ou seja, um Aluno\*\*)

```
/* Função de comparação: char* e Aluno** */  
static int comp_alunos (const void* p1, const void* p2)  
/* converte ponteiros genéricos para ponteiros específicos */  
char* s = (char*)p1;  
Aluno **pa = (Aluno**)p2;  
/* faz a comparação */  
return strcmp(s,(*pa)->nome);  
}
```

# Árvore Binária de Busca

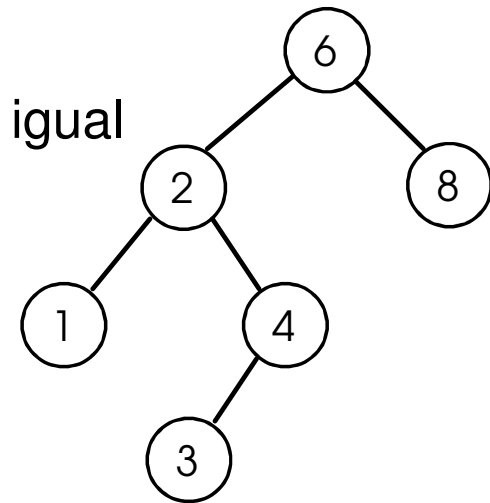
- Busca binária em vetor:
  - dados armazenados em vetor, de forma ordenada
  - bom desempenho computacional para pesquisa
  - inadequado quando inserções e remoções são freqüentes
    - exige re-arrumar o vetor para abrir espaço uma inserção
    - exige re-arrumar o vetor após uma remoção

# Árvore Binária de Busca

- Árvores binárias:
  - árvore binária balanceada:
    - os nós internos têm todos, ou quase todos, 2 filhos
    - qualquer nó pode ser alcançado a partir da raiz em  $O(\log n)$  passos
  - árvore binária degenerada:
    - todos os nós têm apenas 1 filho, com exceção da (única) folha
    - qualquer nó pode ser alcançado a partir da raiz em  $O(n)$  passos

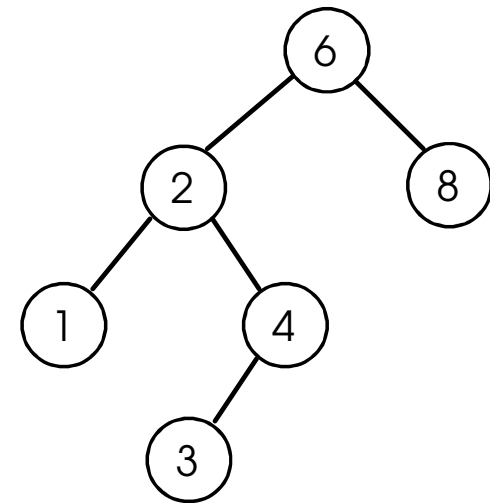
# Árvore Binária de Busca

- Árvores binárias de busca:
  - o valor associado à raiz é sempre maior que o valor associado a qualquer nó da sub-árvore à esquerda (*sae*) e
  - o valor associado à raiz é sempre menor ou igual (para permitir repetições) que o valor associado a qualquer nó da sub-árvore à direita (*sad*)
  - quando a árvore é percorrida em ordem simétrica (*sae - raiz - sad*), os valores são encontrados em ordem não decrescente



# Árvore Binária de Busca

- Pesquisa em árvores binárias de busca:
  - compare o valor dado com o valor associado à raiz
  - se for igual, o valor foi encontrado
  - se for menor, a busca continua na sae
  - se for maior, a busca continua na sad



# Árvore Binária de Busca

- Tipo árvore binária:
  - árvore é representada pelo ponteiro para o nó raiz

```
struct arv {  
    int info;  
    struct arv* esq;  
    struct arv* dir;  
};  
  
typedef struct arv Arv;
```

# Árvore Binária de Busca

- Operação de criação:
  - árvore vazia representada por NULL

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

# Árvore Binária de Busca

- Operação de impressão:
  - imprime os valores da árvore em ordem crescente, percorrendo os nós em ordem simétrica

```
void abb_imprime (Arv* a)
{
    if (a != NULL) {
        abb_imprime(a->esq);
        printf("%d\n",a->info);
        abb_imprime(a->dir);
    }
}
```

# Árvore Binária de Busca

- Operação de busca
  - explora a propriedade de ordenação da árvore
  - possui desempenho computacional proporcional à altura ( $O(\log n)$  para o caso de árvore balanceada)

```
Arv* abb_busca (Arv* r, int v)
{
    if (r == NULL) return NULL;
    else if (r->info > v) return abb_busca (r->esq, v);
    else if (r->info < v) return abb_busca (r->dir, v);
    else return r;
}
```

# Árvore Binária de Busca

- Operação de inserção
  - recebe um valor  $v$  a ser inserido
  - retorna o eventual novo nó raiz da (sub-)árvore
  - para adicionar  $v$  na posição correta, faça:
    - se a (sub-)árvore for vazia
      - crie uma árvore cuja raiz contém  $v$
    - se a (sub-)árvore não for vazia
      - compare  $v$  com o valor na raiz
      - insira  $v$  na sae ou na sad, conforme o resultado da comparação

# Árvore Binária de Busca

```
Arv* abb_inserere (Arv* a, int v)
{
    if (a==NULL) {
        a = (Arv*)malloc(sizeof(Arv));
        a->info = v;
        a->esq = a->dir = NULL;
    }
    else if (v < a->info)
        a->esq = abb_inserere(a->esq,v);
    else /* v < a->info */
        a->dir = abb_inserere(a->dir,v);
    return a;
}
```

é necessário atualizar os ponteiros para as sub-árvores à esquerda ou à direita quando da chamada recursiva da função, pois a função de inserção pode alterar o valor do ponteiro para a raiz da (sub-)árvore.

# Árvore Binária de Busca

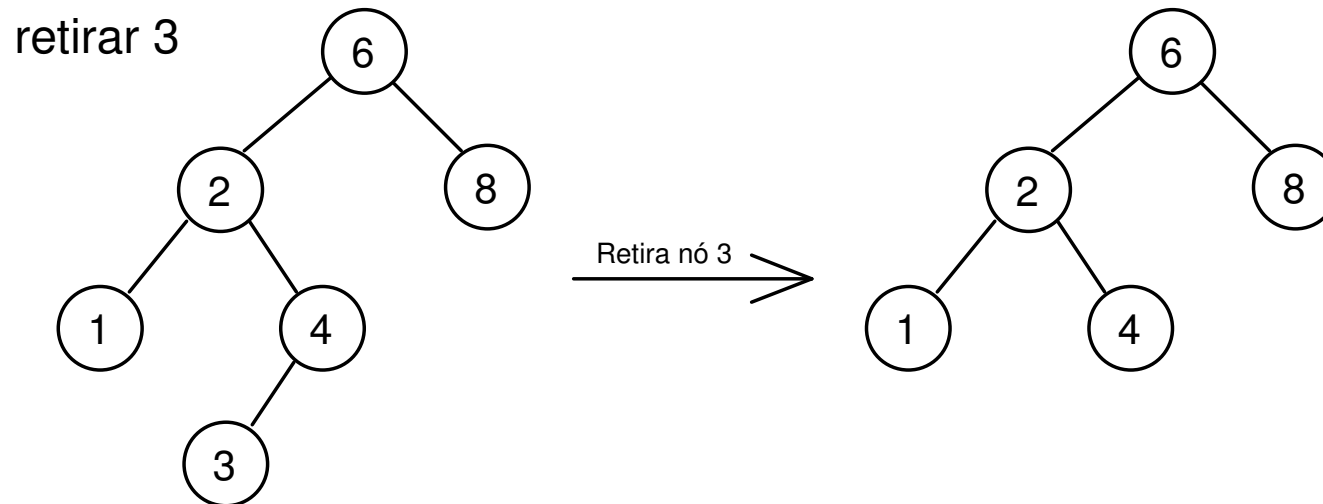
- Operação de remoção:
  - recebe um valor  $v$  a ser inserido
  - retorna a eventual nova raiz da árvore
  - para remover  $v$ , faça:
    - se a árvore for vazia
      - nada tem que ser feito
    - se a árvore não for vazia
      - compare o valor armazenado no nó raiz com  $v$
      - se for maior que  $v$ , retire o elemento da sub-árvore à esquerda
      - se for menor do que  $v$ , retire o elemento da sub-árvore à direita
      - se for igual a  $v$ , retire a raiz da árvore

# Árvore Binária de Busca

- Operação de remoção (cont.):
  - para retirar a raiz da árvore, há 3 casos:
    - caso 1: a raiz que é folha
    - caso 2: a raiz a ser retirada possui um único filho
    - caso 3: a raiz a ser retirada tem dois filhos

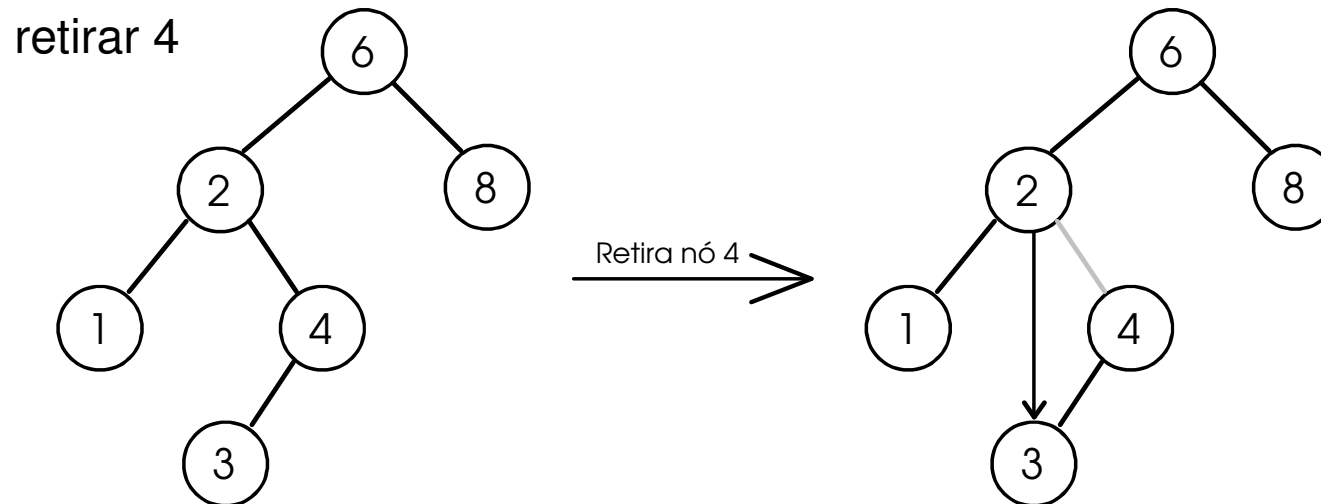
# Árvore Binária de Busca

- Caso 1: a raiz da sub-árvore é folha da árvore original
  - libere a memória alocada pela raiz
  - retorne a raiz atualizada, que passa a ser NULL



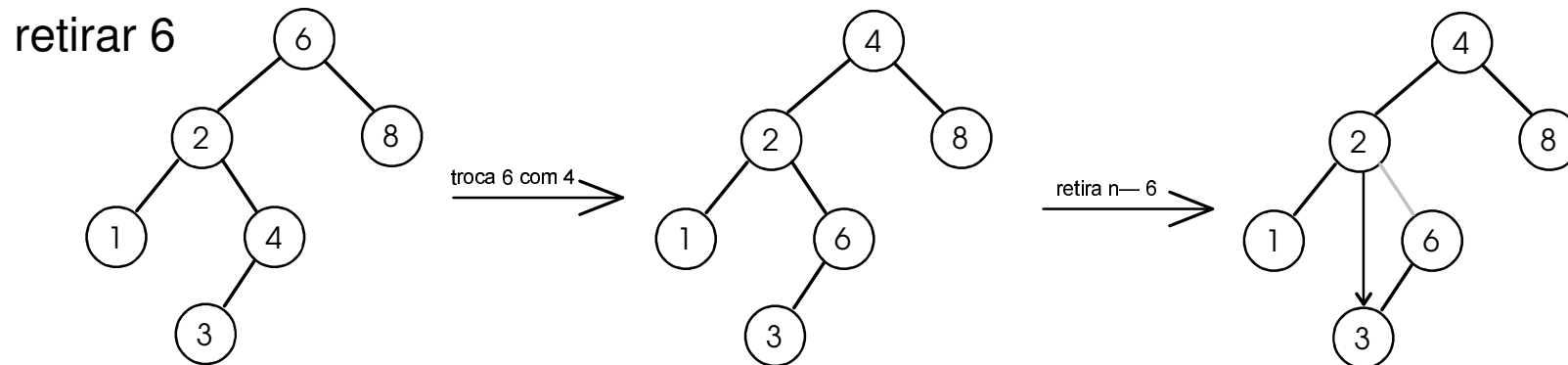
# Árvore Binária de Busca

- Caso 2: a raiz a ser retirada possui um único filho
  - libere a memória alocada pela raiz
  - a raiz da árvore passa a ser o único filho da raiz



# Árvore Binária de Busca

- Caso 3: a raiz a ser retirada tem dois filhos
  - encontre o nó N que precede a raiz na ordenação (o elemento mais à direita da sub-árvore à esquerda)
  - troque o dado da raiz com o dado de N
  - retire N da sub-árvore à esquerda (que agora contém o dado da raiz que se deseja retirar)
    - retirar o nó N mais à direita é trivial, pois N é um nó folha ou N é um nó com um único filho (no caso, o filho da direita nunca existe)



```

Arv* abb_retira (Arv* r, int v)
{
    if (r == NULL)
        return NULL;
    else if (r->info > v)
        r->esq = abb_retira(r->esq, v);
    else if (r->info < v)
        r->dir = abb_retira(r->dir, v);
    else {          /* achou o nó a remover */
        /* nó sem filhos */
        if (r->esq == NULL && r->dir == NULL) {
            free (r);
            r = NULL;
        }
        /* nó só tem filho à direita */
        else if (r->esq == NULL) {
            Arv* t = r;
            r = r->dir;
            free (t);
        }
    }
}

```

```

/* só tem filho à esquerda */
else if (r->dir == NULL) {
    Arv* t = r;
    r = r->esq;
    free (t);
}
/* nó tem os dois filhos */
else {
    Arv* f = r->esq;
    while (f->dir != NULL) {
        f = f->dir;
    }
    r->info = f->info;      /* troca as informações */
    f->info = v;
    r->esq = abb_retira(r->esq,v);
}
}
return r;
}

```

# Árvore Binária de Busca

- Algoritmo para balancear árvores binárias de busca:
  - reorganize (balanceie) a árvore com relação à raiz:
    - suponha que a *sae* possui  $m$  nós e que a *sad* possui  $m$  nós
    - se  $n > m+2$ ,
      - mova o valor da raiz para a *sae*, onde ele se tornará o maior valor
      - mova o menor elemento da *sad* para a raiz
    - se se  $n \leq m+2$ , proceda de forma semelhante
    - repita o processo até que a diferença entre os números de elementos das duas sub-árvores seja menor ou igual a 1
  - continue o processo recursivamente com o balanceamento das sub-árvores da raiz
  - (a remoção do menor (ou maior) elemento de uma árvore é mais simples do que a remoção de um elemento qualquer)

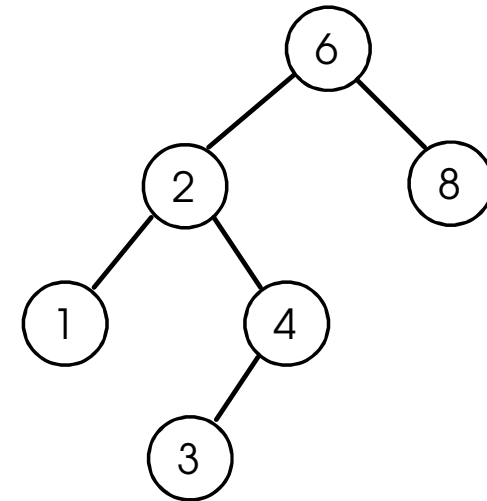
# Resumo

- Busca linear em vetor:
  - percorra o vetor, elemento a elemento, verificando se o elemento de interesse é igual a um dos elementos do vetor
- Busca binária:
  - compare elem com o elemento do meio de vet
  - se elem for menor, pesquise na primeira metade do vetor
  - se elem for maior, pesquise na segunda parte do vetor
  - se for igual, retorne a posição
  - continue o procedimento, subdividindo a parte de interesse, até encontrar o elemento ou chegar a uma parte do vetor com tamanho 0
- Pesquisa binária da biblioteca padrão:
  - disponibilizado via [stdlib.h](#), com protótipo

```
void* bsearch (void* info, void *v, int n, int tam,  
              int (*cmp)(const void*, const void*));
```

# Resumo

- Árvores binárias de busca:
  - o valor associado à raiz é maior que o valor associado a qualquer nó da sub-árvore à esquerda (sae) e
  - o valor associado à raiz é menor ou igual que o valor associado a qualquer nó da sub-árvore à direita (sad)



- Operação de busca
  - explora a propriedade de ordenação da árvore
  - possui desempenho computacional proporcional à altura ( $O(\log n)$  para o caso de árvore balanceada)