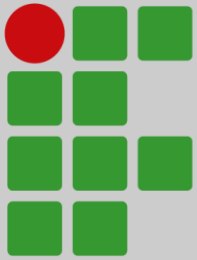


INSTITUTO FEDERAL

Campus Charqueadas

# Árvores Binárias

Prof. Silvana Teodoro

(silvanateo@gmail.com)

# Árvores Binárias

# Árvores Binárias
# **Implementação em C**

- **Representação: ponteiro para o nó raiz**
- **Representação de um nó na árvore:**
  - Estrutura em C contendo
  - A informação propriamente dita (exemplo: um caracere, ou inteiro)
  - Dois ponteiros para as sub-árvores, à esquerda e à direita

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

# Árvores Binárias
# **Implementação em C (arv.c)**

```
typedef struct arv Arv;
//Cria uma árvore vazia
Arv* arv_criavazia (void);
//cria uma árvore com a informação do nó raiz c, e
//com subárvore esquerda e e subárvore direita d
Arv* arv_cria (char c, Arv* e, Arv* d);
//libera o espaço de memória ocupado pela árvore a
Arv* arv_libera (Arv* a);
//retorna true se a árvore estiver vazia e false
//caso contrário
int arv_vazia (Arv* a);
//indica a ocorrência (1) ou não (0) do caracter c
int arv_pertence (Arv* a, char c);
//imprime as informações dos nós da árvore
void arv_imprime (Arv* a);
```

# Árvores Binárias
# **Implementação em C (arv.c)**

- **Função arv_criavazia**
  - cria uma árvore vazia

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

# Árvores Binárias
# **Implementação em C (arv.c)**

- **Função arv_cria**
  - cria um nó raiz dadas a informação e as duas sub-árvores, a da esquerda e a da direita
  - retorna o endereço do nó raiz criado

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

# Árvores Binárias
# **Implementação em C (arv.c)**

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

# Árvores Binárias
# **Implementação em C (arv.c)**

- **Função arv_vazia**
  - indica se uma árvore é ou não vazia

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

# Árvores Binárias
# **Implementação em C (arv.c)**

- **Função arv_libera**
  - libera memória alocada pela estrutura da árvore
  - as sub-árvores devem ser liberadas antes de se liberar o nó raiz
  - retorna uma árvore vazia, representada por NULL

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

# Árvores Binárias
# **Implementação em C (arv.c)**

- **Função arv_pertence**
  - verifica a ocorrência de um caractere c em um dos nós
  - retorna um valor booleano (1 ou 0) indicando a ocorrência ou não do caractere na árvore

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

# Árvores Binárias
# **Implementação em C (arv.c)**

- **Função arv_imprime**
  - percorre recursivamente a árvore, visitando todos os nós e imprimindo sua informação

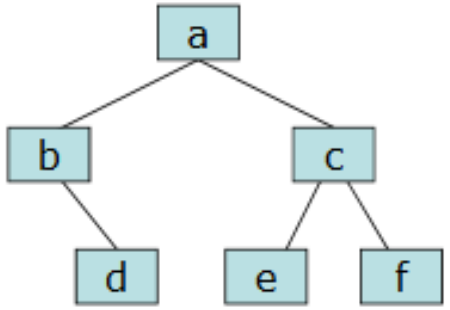
```
void arv_imprime (Arv* a){
  if (!arv_vazia(a)){
     printf("%c ", a->info); /* mostra raiz */
     arv_imprime(a->esq); /* mostra sae */
     arv_imprime(a->dir); /* mostra sad */
  }
}
```

# Árvores Binárias
# **Implementação em C (arv.c)**

- **Criar a árvore:**

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

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

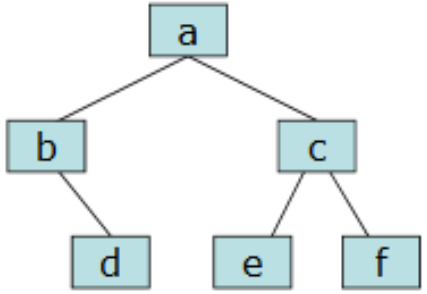


# Árvores Binárias
# **Implementação em C (arv.c)**

- **Criar a árvore:**

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

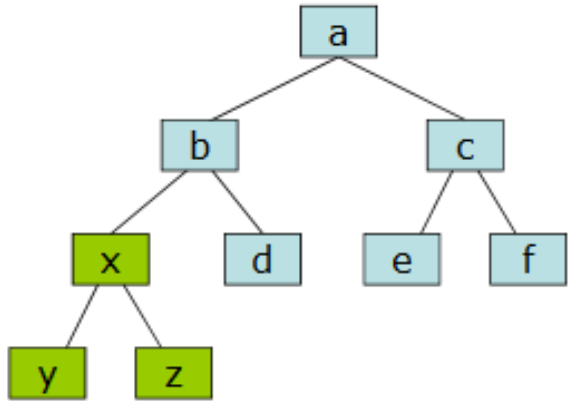
```
Arv* a = arv_cria('a',
           arv_cria('b',
               arv_criavazia(),
               arv_cria('d', arv_criavazia(), arv_criavazia())
           ),
           arv_cria('c',
              arv_cria('e', arv_criavazia(), arv_criavazia()),
              arv_cria('f', arv_criavazia(), arv_criavazia())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



# Árvores Binárias
# **Implementação em C (arv.c)**

- **Acrescenta nós x, y e z**

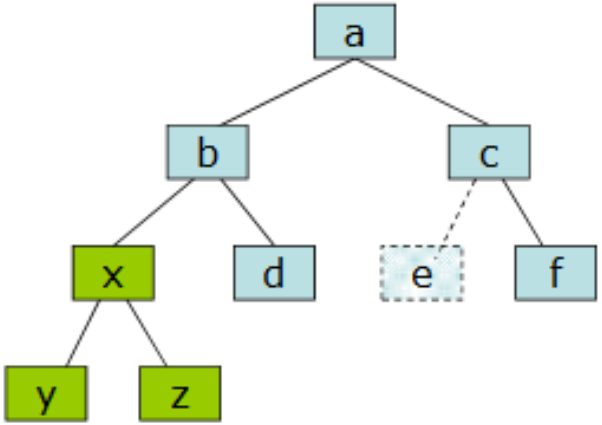
```
a->esq->esq =
  arv_cria('x',
      arv_cria('y',
            arv_criavazia(),
            arv_criavazia()),
      arv_cria('z',
            arv_criavazia(),
            arv_criavazia())
  );
```



# Árvores Binárias
# **Implementação em C (arv.c)**

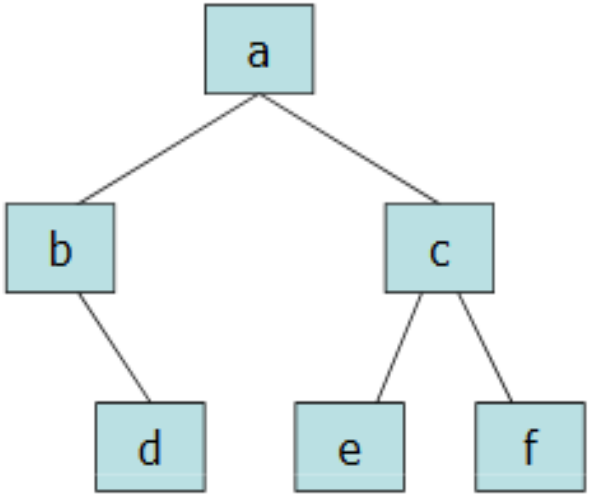
- **Libera nós**

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



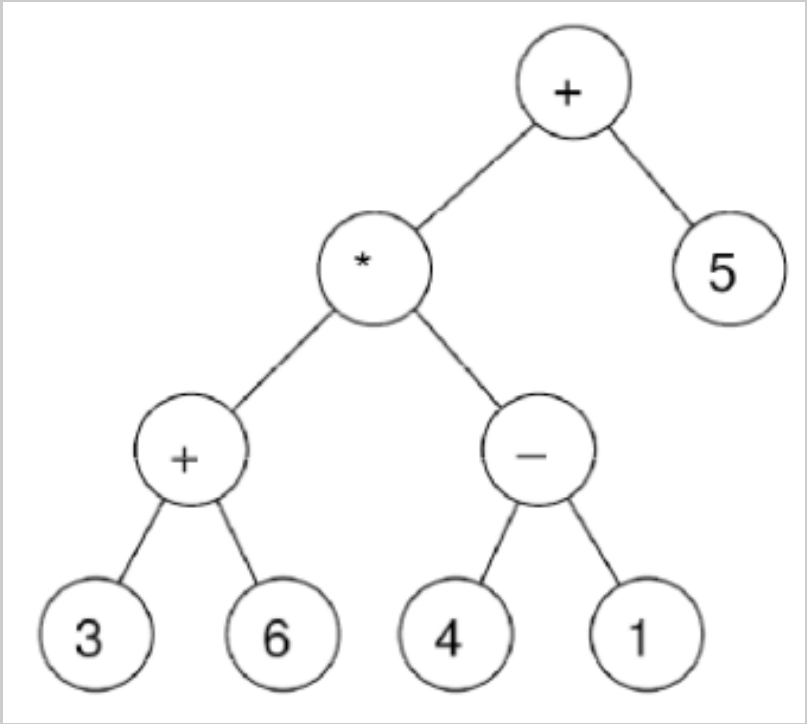
# Árvores Binárias
# **Ordem de Percurso**

- *Pré-ordem*:
  - trata *raiz*, percorre *sae*, percorre *sad*
  - exemplo: a b d c e f
- *Ordem simétrica (ou In-Ordem)*:
  - percorre *sae*, trata *raiz*, percorre *sad*
  - exemplo: b d a e c f
- *Pós-ordem*:
  - percorre *sae*, percorre *sad*, trata *raiz*
  - exemplo: d b e f c a



# Árvores Binárias
# **Ordem de Percurso**

- **Pré-ordem**
  - +*+36-415
- **In-ordem**
  - 3+6*4-1+5
- **Pós-ordem**
  - 36+41-*5+



# Árvores Binárias
# **Ordem de Percurso**

- **Pré-ordem**

```
void arv_preordem (Arv* a)
{
    if (!arv_vazia(a))
    {
        processa(a); // por exemplo imprime
        arv_preordem(a->esq);
        arv_preordem(a->dir);
    }
}
```

# Árvores Binárias
# **Ordem de Percurso**

- **In-ordem**

```
void arv_inordem (Arv* a)
{
    if (!arv_vazia(a))
    {
        arv_inordem (a->esq);
        processa (a); // por exemplo imprime
        arv_inordem (a->dir);
    }
}
```

# Árvores Binárias
# **Ordem de Percurso**

- **Pós-ordem**

```
void arv_posordem (Arv* a)
{
    if (!arv_vazia(a))
    {
        arv_posordem (a->esq);
        arv_posordem (a->dir);
        processa (a); // por exemplo imprime
    }
}
```

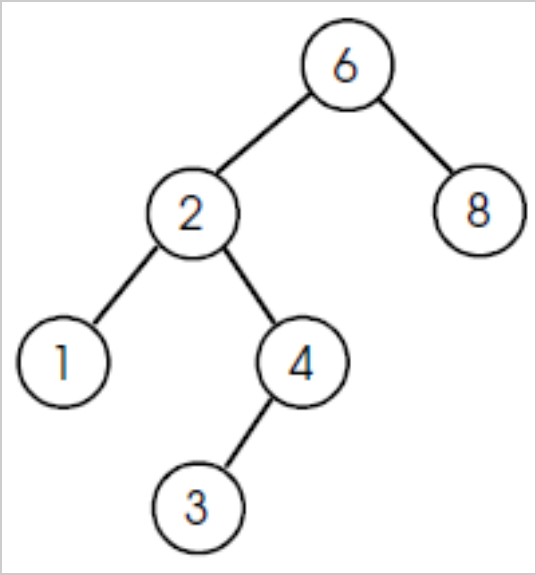
# Árvore Binária de Busca

# Busca linear x binária

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

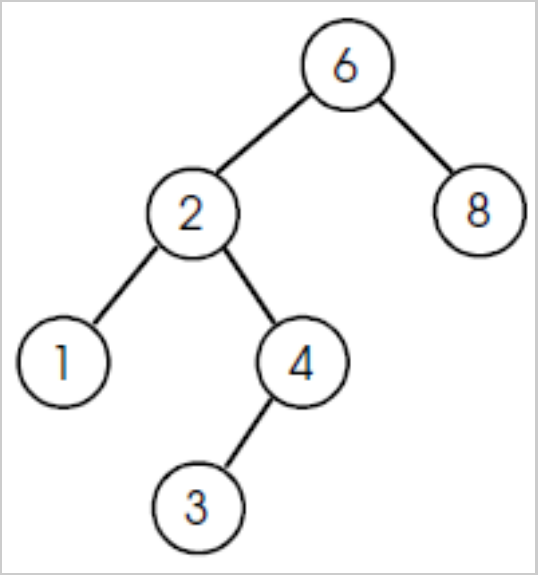
# Árvore binária de busca

- **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)*
  - o valor associado à **raiz** é sempre **menor** 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 **crescente**



# Árvore binária de busca

- **Pesquisa (busca) 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.**

```
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** (nó único)
    - 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_insere (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_insere(a->esq,v);
  else  /* v > a->info */
    a->dir = abb_insere(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

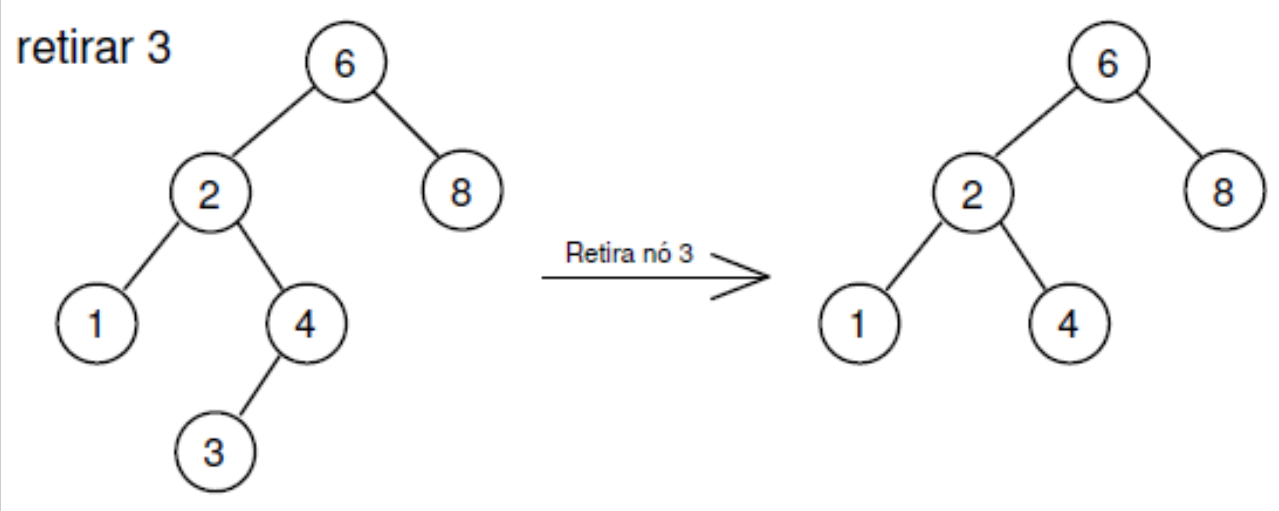
- **Operação de remoção:**
  - recebe um valor **v** a ser removido
  - retorna a **eventual nova raiz** da árvore
  - para **remover v**, faça:
    - se a **árvore for vazia**
      - **nada** tem que ser feito (retorna NULL)
    - se a **árvore não for vazia**
      - **compare** o valor armazenado 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**

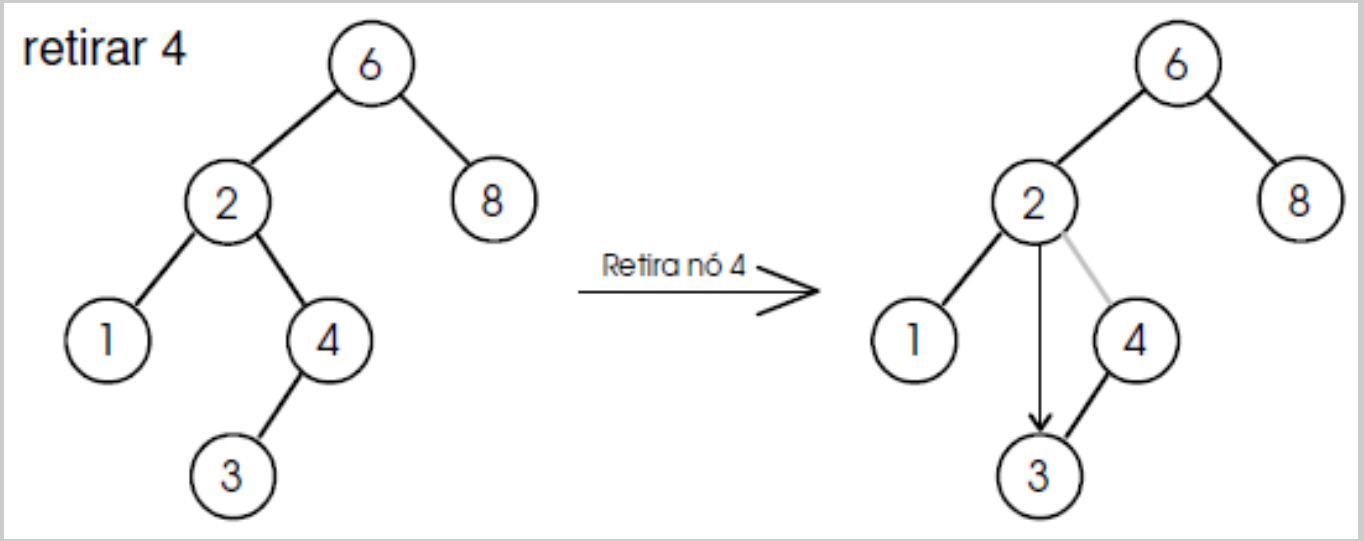
# Árvore binária de busca

- **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**



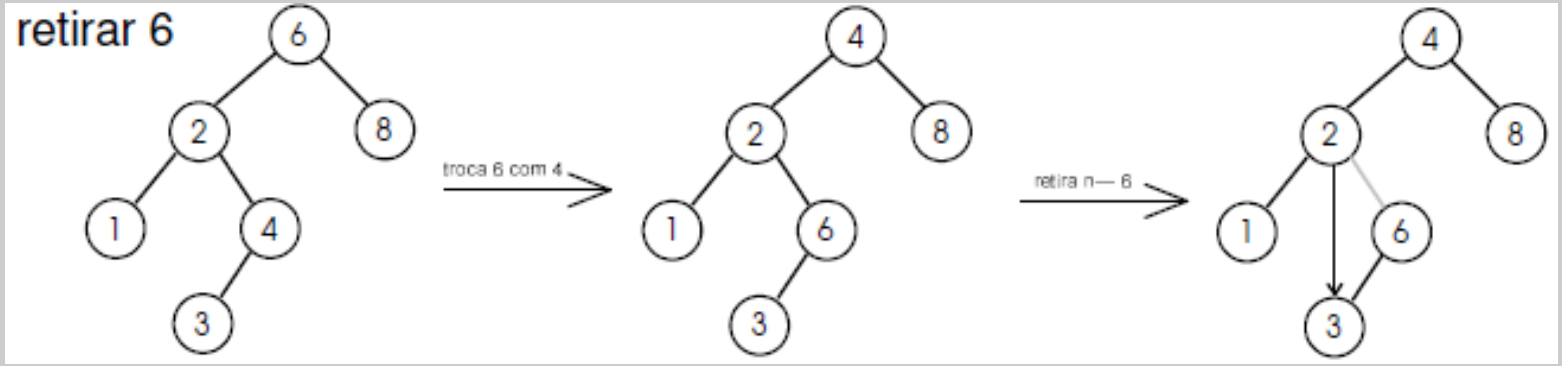
# Árvore binária de busca

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



# 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 à esquerda** (no caso, o filho da direita nunca existe)



# Árvore binária de busca

```
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);
    }
```

# Árvore binária de busca

```
    /* 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;
}
```

# Atividade

1) Implemente as funções de busca, inserção e remoção de árvores binárias de busca (apresentadas anteriormente), criando também as árvores vistas em aula.

# Leitura Complementar

- Aaron M. Tenenbaum, Yedidyah Langsam, Moshe J. Augenstein. **Estruturas de Dados Usando C.** Makron Books/Pearson Education, 1995.