MC-202
Árvores Binárias de Busca

Lehilton Pedrosa
lehilton@ic.unicamp.br

Universidade Estadual de Campinas

Segundo semestre de 2024

Tempo para Inserção, Remoção e Busca

Usando Listas Duplamente Ligadas:

- Podemos inserir e remover em $\mathrm{O}(1)$
- Mas buscar demora $\mathrm{O}(n)$

Se usarmos vetores não-ordenados:

- Podemos inserir e remover em $\mathrm{O}(1)$
  - insira no final
  - para remover, troque com o último e remova o último
- Mas buscar demora $\mathrm{O}(n)$

Se usarmos vetores ordenados:

- Podemos buscar em $\mathrm{O}(\lg n)$
- Mas inserir e remover leva $\mathrm{O}(n)$

Veremos árvores binárias de busca

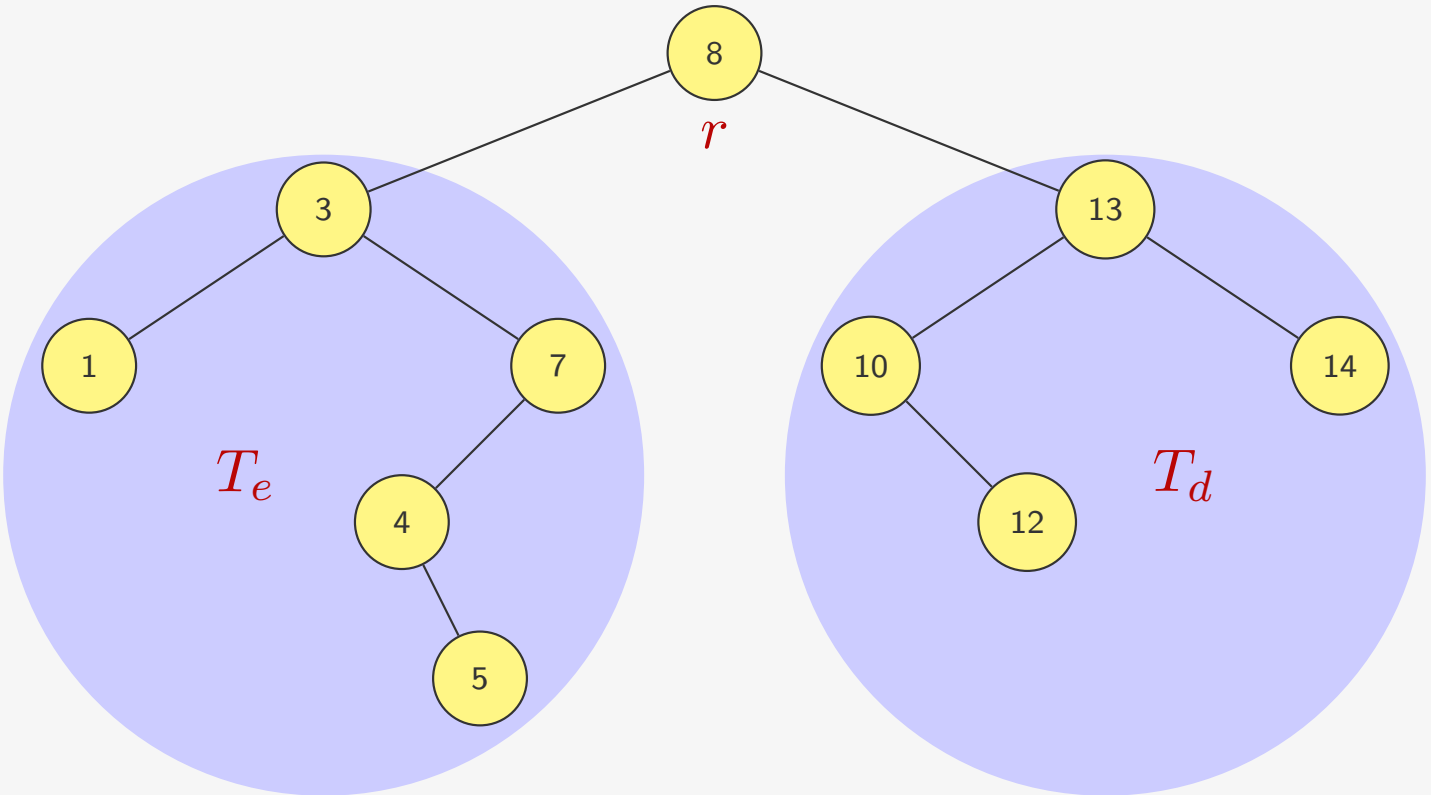
- primeiro uma versão simples, depois uma sofisticada
- versão sofisticada: três operações levam $\mathrm{O}(\lg n)$

Árvore Binária de Busca

Uma Árvore Binária de Busca (ABB) é uma árvore binária em que cada nó contém um elemento de um conjunto ordenável

Cada nó r, com subárvores esquerda T_e e direita T_d satisfaz a seguinte propriedade:

1. $e < r$ para todo elemento $e \in T_e$
2. $d > r$ para todo elemento $d \in T_d$



TAD — Árvores de Busca Binária

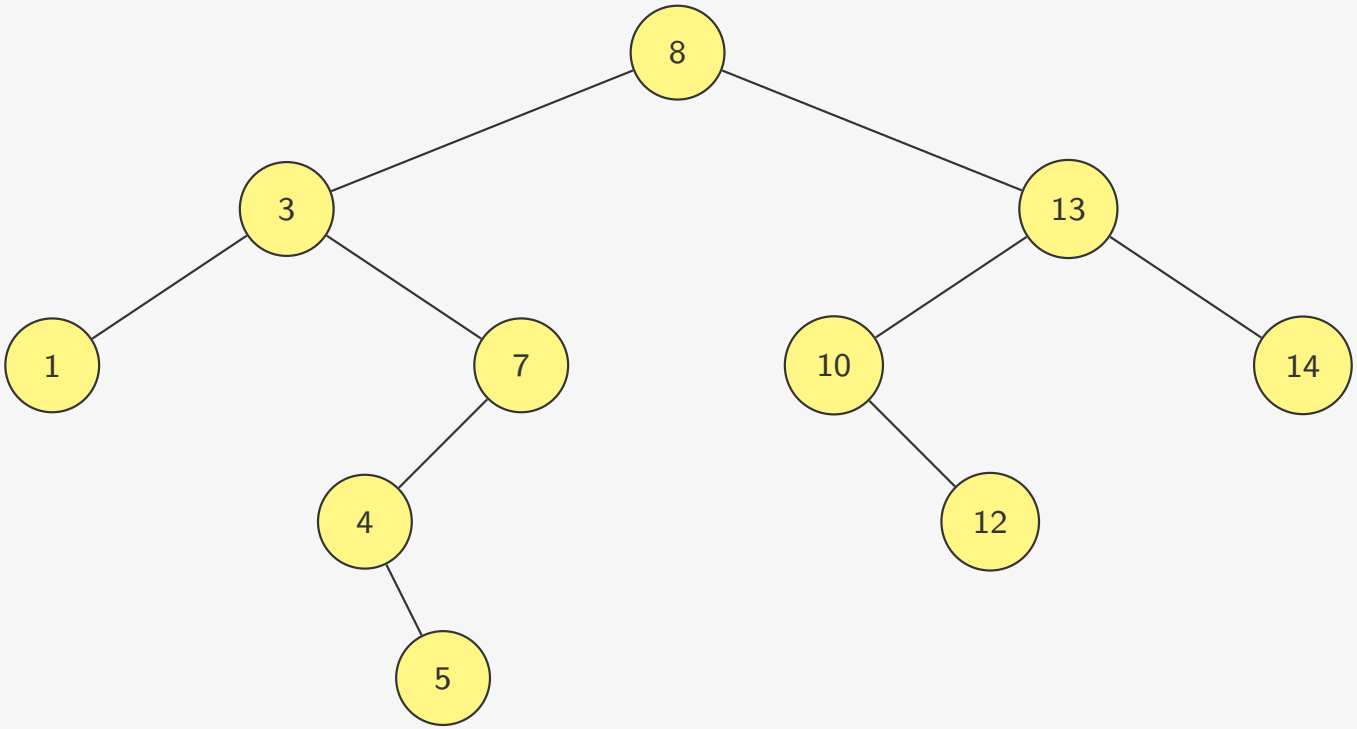
```
1  typedef struct no *p_no;
2
3  struct no {
4      int chave;
5      p_no esq, dir, pai; /* pai é opcional, usado para
6                           * buscar sucessor e antecessor */
7  };
8
9  p_no criar_arvore();
10
11 void destruir_arvore(p_no raiz);
12
13 p_no inserir(p_no raiz, int chave);
14
15 p_no remover(p_no raiz, int chave);
16
17 p_no buscar(p_no raiz, int chave);
18
19 p_no minimo(p_no raiz);
20
21 p_no maximo(p_no raiz);
22
23 p_no sucessor(p_no x);
24
25 p_no antecessor(p_no x);
```

Busca por um valor

A ideia é semelhante àquela da busca binária:

- Ou o valor a ser buscado está na raiz da árvore
- Ou é menor do que o valor da raiz
  - Se estiver na árvore, está na subárvore esquerda
- Ou é maior do que o valor da raiz
  - Se estiver na árvore, está na subárvore direita

Ex: Buscando por 4

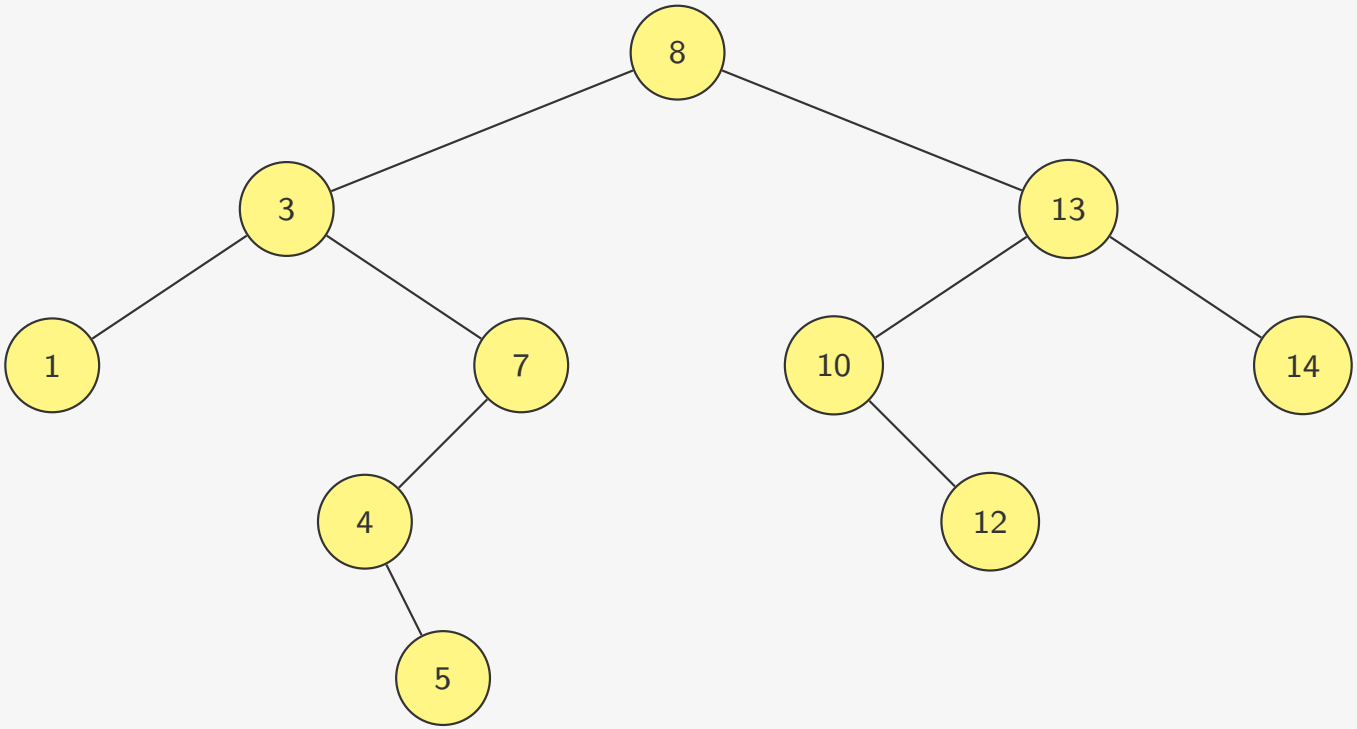


Busca por um valor

A ideia é semelhante a da busca binária:

- Ou o valor a ser buscado está na raiz da árvore
- Ou é menor do que o valor da raiz
  - Se estiver na árvore, está na subárvore esquerda
- Ou é maior do que o valor da raiz
  - Se estiver na árvore, está na subárvore direita

Ex: Buscando por 11



Busca

Versão recursiva:

```
1 p_no buscar(p_no raiz, int chave) {
2   if (raiz == NULL || chave == raiz->chave)
3     return raiz;
4   if (chave < raiz->chave)
5     return buscar(raiz->esq, chave);
6   else
7     return buscar(raiz->dir, chave);
8 }
```

Versão iterativa:

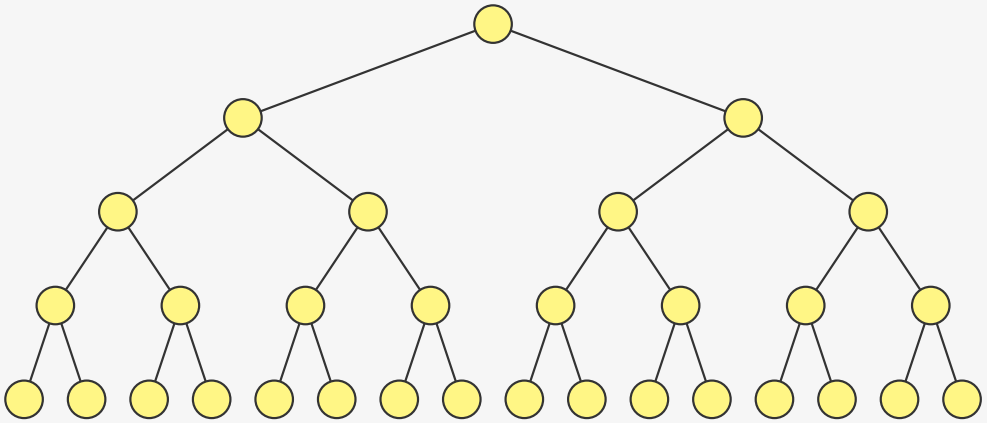
```
1 p_no buscar_iterativo(p_no raiz, int chave) {
2   while (raiz != NULL && chave != raiz->chave)
3     if (chave < raiz->chave)
4       raiz = raiz->esq;
5     else
6       raiz = raiz->dir;
7   return raiz;
8 }
```

Eficiência da busca

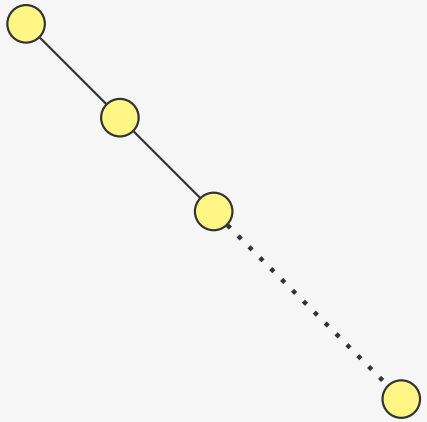
Qual é o tempo da busca?

- depende da forma da árvore...

Ex: 31 nós



Melhor árvore: $\mathrm{O}(\lg n)$



Pior árvore: $\mathrm{O}(n)$

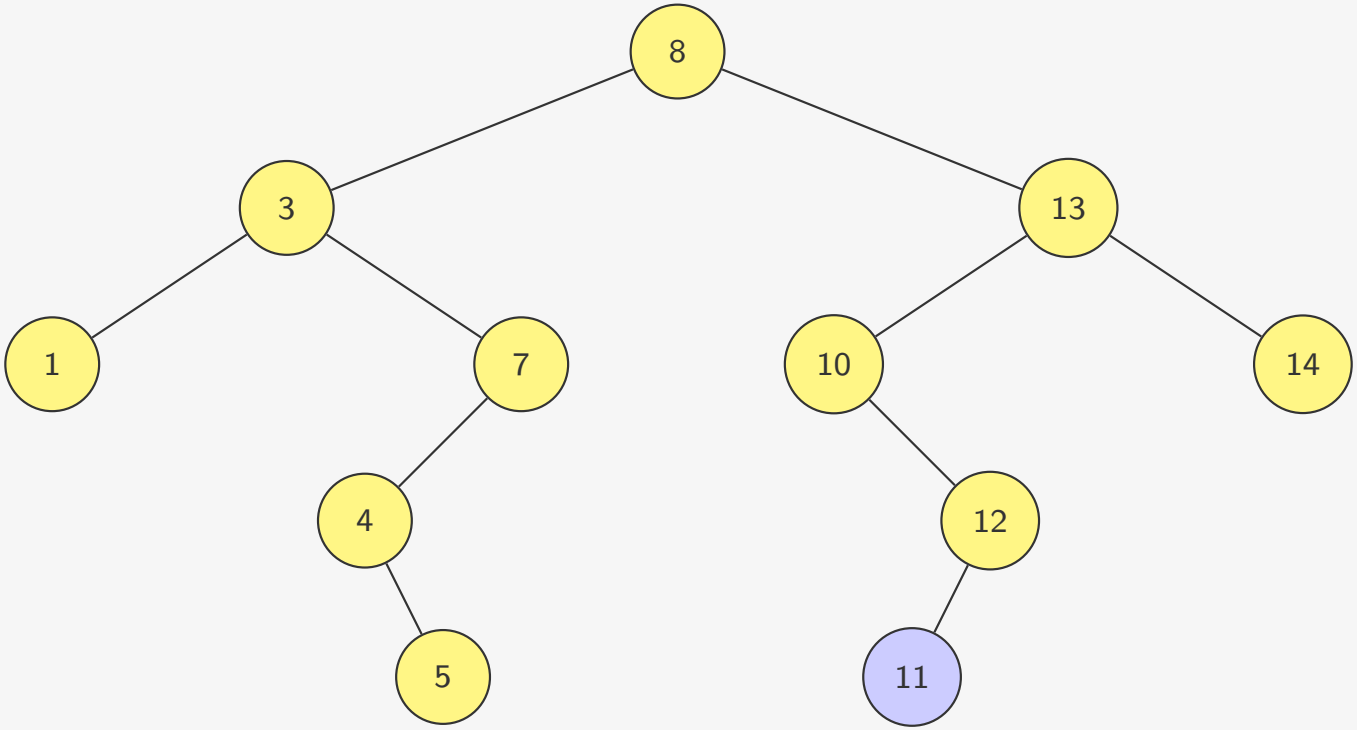
Caso médio: em uma árvore com n elementos adicionados aleatoriamente, a busca demora (em média) $\mathrm{O}(\lg n)$

Inserindo um valor

Precisamos determinar onde inserir o valor:

- fazemos uma busca pelo valor
- e inserimos onde ele deveria estar

Ex: Inserindo 11



Inserção — implementação

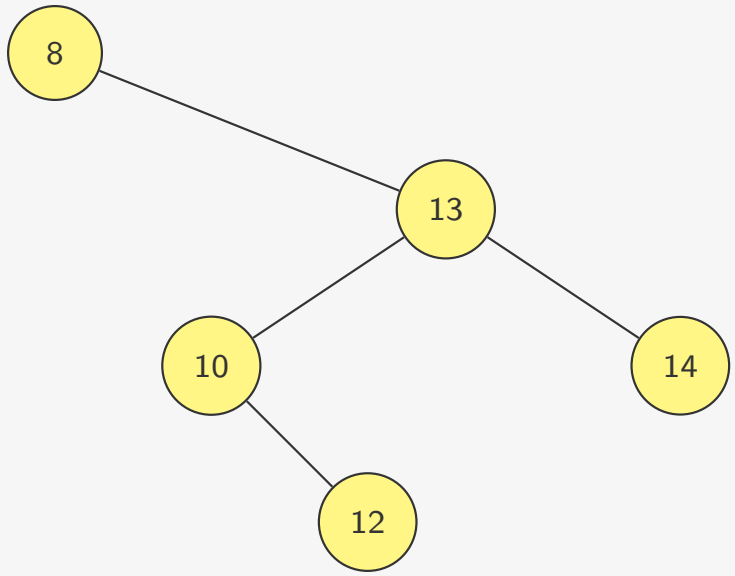
O algoritmo insere na árvore recursivamente

- devolve um ponteiro para a raiz da "nova" árvore
- assim como fizemos com listas ligadas

```
1  p_no inserir(p_no raiz, int chave) {
2    p_no novo;
3    if (raiz == NULL) {
4      novo = malloc(sizeof(struct no));
5      novo->esq = novo->dir = NULL;
6      novo->chave = chave;
7      return novo;
8    }
9    if (chave < raiz->chave)
10     raiz->esq = inserir(raiz->esq, chave);
11   else
12     raiz->dir = inserir(raiz->dir, chave);
13   return raiz;
14 }
```

Mínimo da Árvore

Onde está o nó com a menor chave de uma árvore?

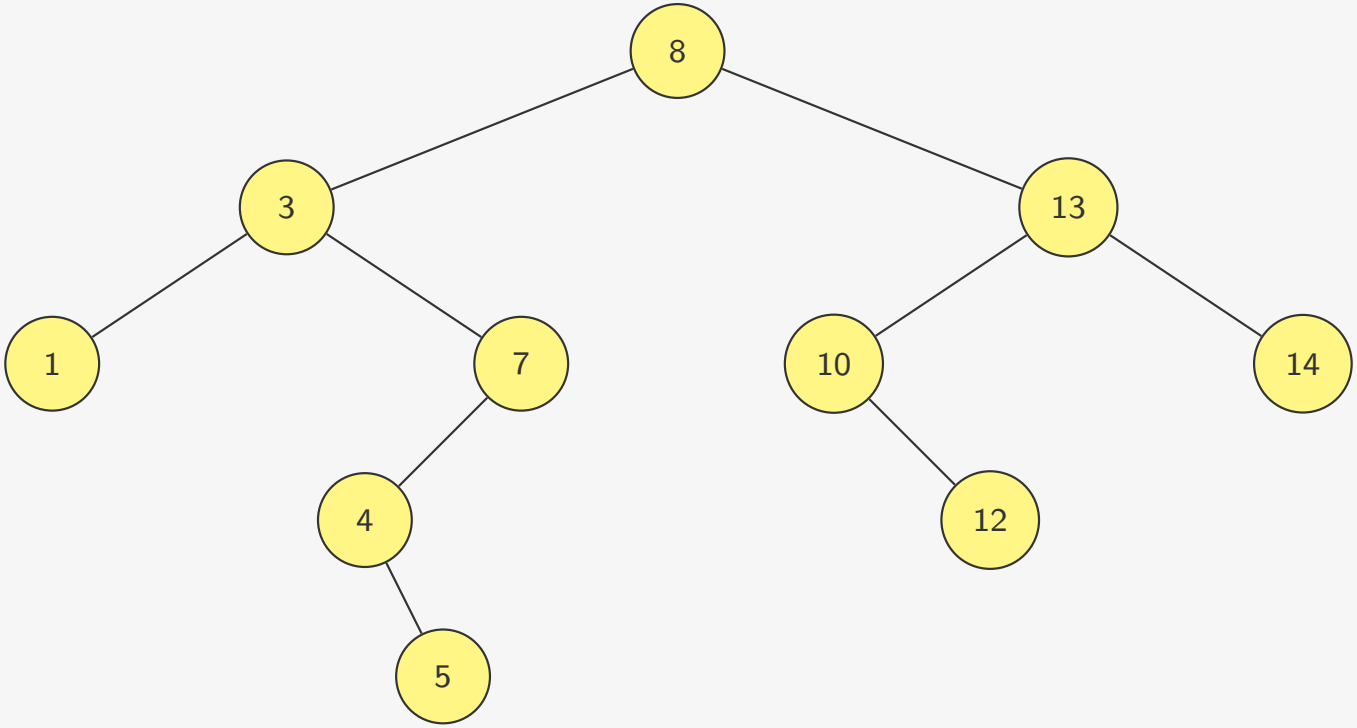


Quem é o mínimo para essa árvore?

- É a própria raiz

Mínimo da Árvore

Onde está o nó com a menor chave de uma árvore?



Quem é o mínimo para essa árvore?

- É o mínimo da subárvore esquerda

Mínimo — Implementações

Versão recursiva:

```
1 p_no minimo(p_no raiz) {
2   if (raiz == NULL || raiz->esq == NULL)
3     return raiz;
4   return minimo(raiz->esq);
5 }
```

Versão iterativa:

```
1 p_no minimo_iterativo(p_no raiz) {
2   while (raiz != NULL && raiz->esq != NULL)
3     raiz = raiz->esq;
4   return raiz;
5 }
```

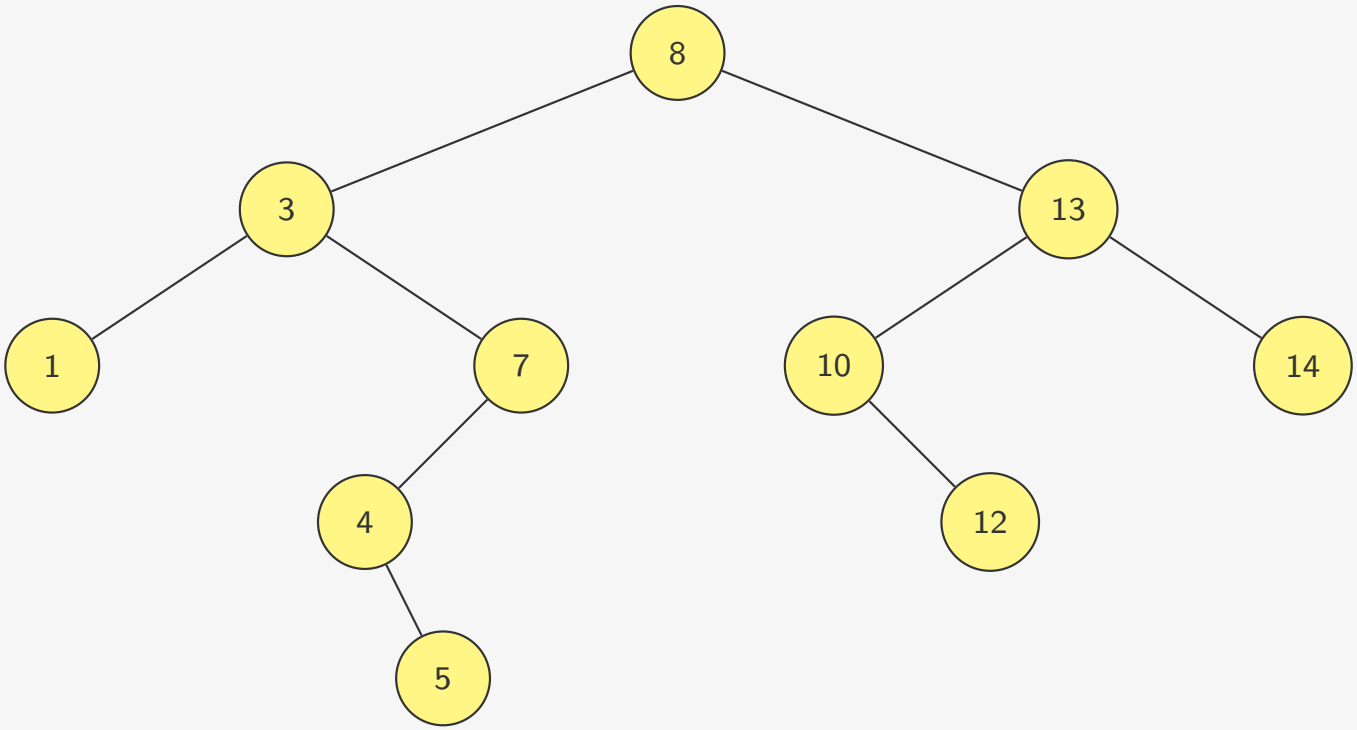
Para encontrar o máximo, basta fazer a operação simétrica

- Se a subárvore direita existir, ela contém o máximo
- Senão, é a própria raiz

Sucessor

Dado um nó da árvore, onde está o seu sucessor?

- O sucessor é o próximo nó na ordenação



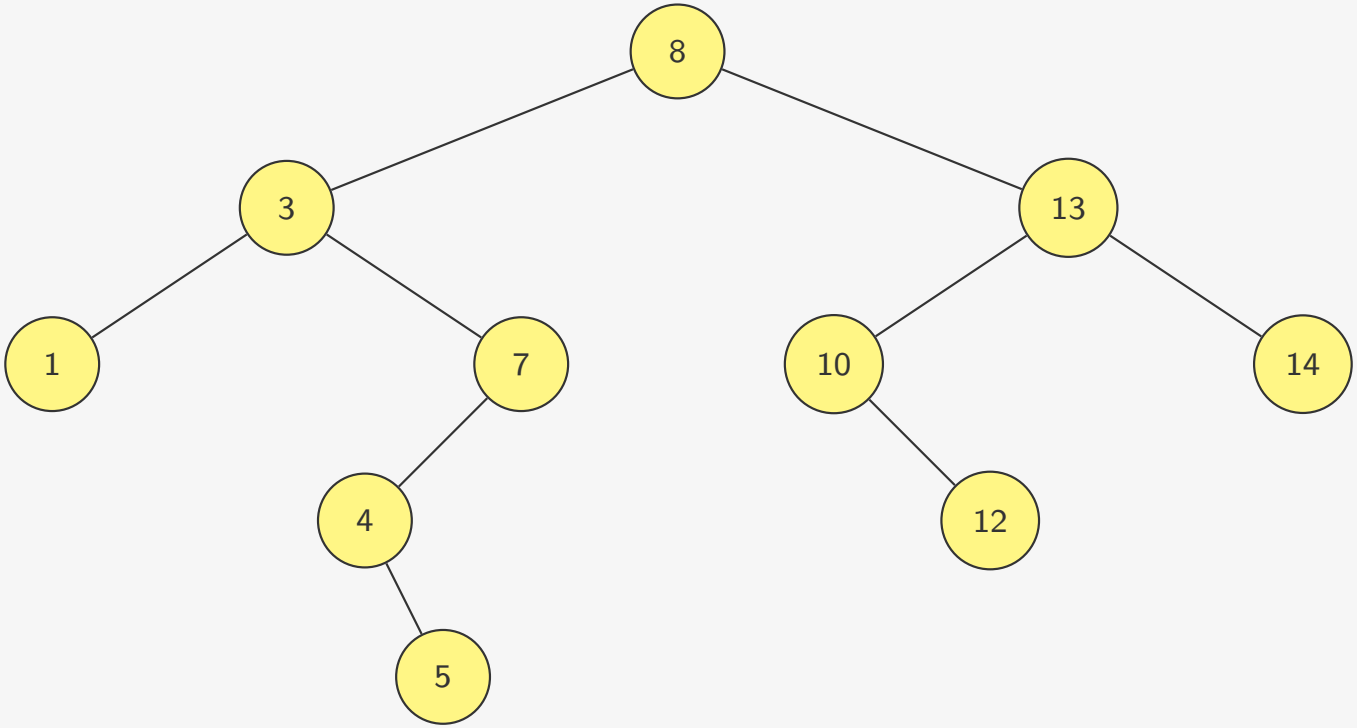
Quem é o sucessor de 3?

- É o mínimo da subárvore direita de 3

Sucessor

Dado um nó da árvore, onde está o seu sucessor?

- O sucessor é o próximo nó na ordenação



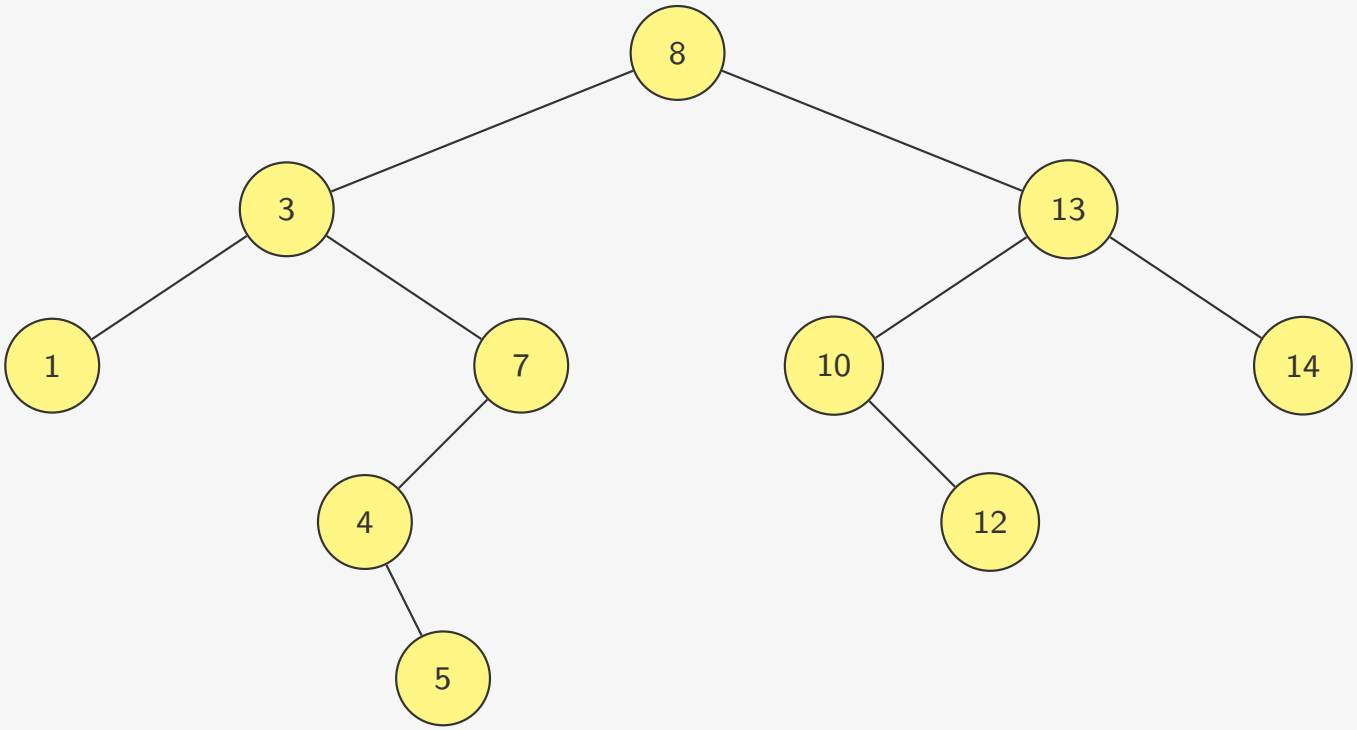
Quem é o sucessor de 7?

- É primeiro ancestral a direita

Sucessor

Dado um nó da árvore, onde está o seu sucessor?

- O sucessor é o próximo nó na ordenação



Quem é o sucessor de 14?

- não tem sucessor...

Sucessor — Implementação

```
1 p_no sucessor(p_no x) {
2   if (x->dir != NULL)
3     return minimo(x->dir);
4   else
5     return ancestral_a_direita(x);
6 }
```

```
1 p_no ancestral_a_direita(p_no x) {
2   if (x == NULL)
3     return NULL;
4   if (x->pai == NULL || x->pai->esq == x)
5     return x->pai;
6   else
7     return ancestral_a_direita(x->pai);
8 }
```

A implementação da função `antecessor` é simétrica

Remoção

Ex: removendo 10

Remoção

Ex: removendo 3

Podemos colocar o sucessor de 3 em seu lugar

- Isso mantém a propriedade da árvore binária de busca

E agora removemos o sucessor

- O sucessor nunca tem filho esquerdo!

Remoção — Implementação

Versão sem ponteiro para `pai` e que não libera o nó

```
1  p_no remover_rec(p_no raiz, int chave) {
2    if (raiz == NULL)
3      return NULL;
4    if (chave < raiz->chave)
5      raiz->esq = remover_rec(raiz->esq, chave);
6    else if (chave > raiz->chave)
7      raiz->dir = remover_rec(raiz->dir, chave);
8    else if (raiz->esq == NULL)
9      return raiz->dir;
10   else if (raiz->dir == NULL)
11     return raiz->esq;
12   else
13     remover_sucessor(raiz);
14   return raiz;
15 }
```

Remoção — Implementação

```
1  void remover_sucessor(p_no raiz) {
2    p_no min = raiz->dir; /*será o mínimo da subárvore direita*/
3    p_no pai = raiz;    /*será o pai de min*/
4    while (min->esq != NULL) {
5      pai = min;
6      min = min->esq;
7    }
8    if (pai->esq == min)
9      pai->esq = min->dir;
10   else
11     pai->dir = min->dir;
12   raiz->chave = min->chave;
13 }
```

Exercício

Faça uma função que imprime as chaves de uma ABB em ordem crescente

Solução

```
1 void imprime(p_no raiz) {
2     if (raiz != NULL) {
3         imprime(raiz->esq);
4         printf("%d ", raiz->chave);
5         imprime(raiz->dir);
6     }
7 }
```

Exercício

Faça uma implementação da função `ancestral_a_direita` que não usa o ponteiro `pai`

- Dica: você precisará da raiz da árvore pois não pode subir

Solução

```
1  p_no ancestral_a_direita(p_no x, p_no raiz) {
2      p_no atual = raiz;
3      p_no ancestral = NULL;
4      while (atual->chave != x->chave ) {
5          if (atual->chave > x->chave) {
6              ancestral = atual;
7              atual = atual->esq;
8          } else
9              atual = atual->dir;
10     }
11     return ancestral;
12 }
```

Dúvidas?