Árvore Binária de Busca

Estrutura de Dados — QXD0010

Prof. Atílio Gomes Luiz
gomes.atilio@ufc.br

Universidade Federal do Ceará

1º semestre/2022

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Leituras para esta aula

- **Capítulo 14 (Árvores Binárias)** do livro Algoritmos em Linguagem C do prof. Paulo Feofiloff. Disponível no link: `https://b-ok.lat/book/2281834/e16397`

- `https://www.ime.usp.br/~pf/algoritmos/aulas/binst.html`

Introdução

Tempo para Inserção, Remoção e Busca

Usando Listas Duplamente Encadeadas:

- Podemos inserir e remover em $O(1)$
- Mas buscar demora $O(n)$

Tempo para Inserção, Remoção e Busca

Usando Listas Duplamente Encadeadas:

- Podemos inserir e remover em $O(1)$
- Mas buscar demora $O(n)$

Se usarmos vetores não-ordenados:

- Podemos inserir e remover em $O(1)$

Tempo para Inserção, Remoção e Busca

Usando Listas Duplamente Encadeadas:

- Podemos inserir e remover em $O(1)$
- Mas buscar demora $O(n)$

Se usarmos vetores não-ordenados:

- Podemos inserir e remover em $O(1)$
  - insira no final
  - para remover, troque com o último e remova o último

Tempo para Inserção, Remoção e Busca

Usando Listas Duplamente Encadeadas:

- Podemos inserir e remover em $O(1)$
- Mas buscar demora $O(n)$

Se usarmos vetores não-ordenados:

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

Tempo para Inserção, Remoção e Busca

Usando Listas Duplamente Encadeadas:

- Podemos inserir e remover em $O(1)$
- Mas buscar demora $O(n)$

Se usarmos vetores não-ordenados:

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

Se usarmos vetores ordenados:

- Podemos buscar em $O(\lg n)$ usando Busca binária
- Mas inserir e remover leva $O(n)$

Tempo para Inserção, Remoção e Busca

Usando Listas Duplamente Encadeadas:

- Podemos inserir e remover em $O(1)$
- Mas buscar demora $O(n)$

Se usarmos vetores não-ordenados:

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

Se usarmos vetores ordenados:

- Podemos buscar em $O(\lg n)$ usando Busca binária
- Mas inserir e remover leva $O(n)$

Veremos árvores binárias de busca

- Inserção, Remoção e Busca levam $O(\lg n)$ se a árvore for balanceada

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

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:

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 $e \in T_e$

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:

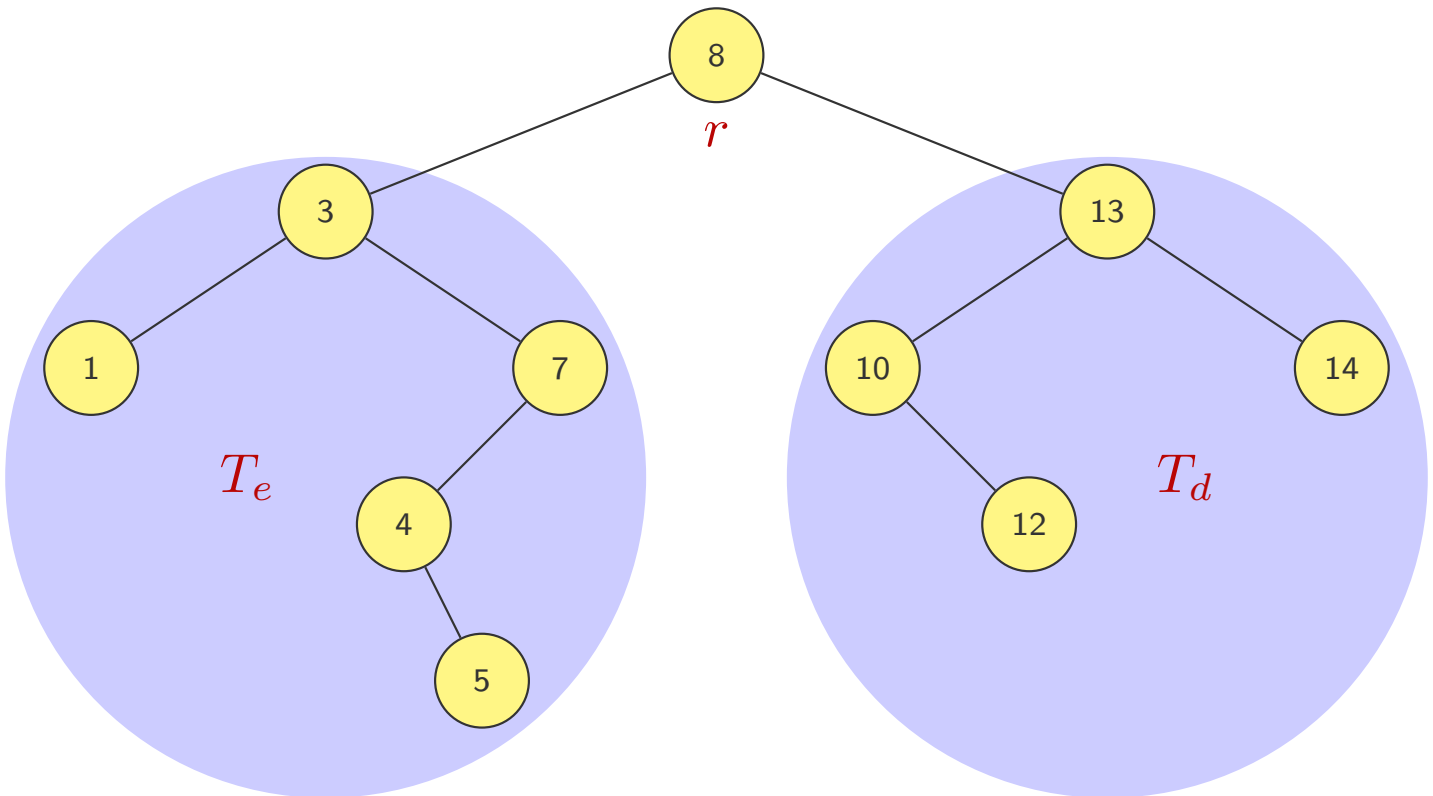
1. $e < r$ para todo elemento $e \in T_e$
2. $d > r$ para todo elemento $d \in T_d$

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 $d \in T_d$



Implementação em C++

Implementação

- O nó da árvore será um struct com três campos:
  - uma chave inteira e dois ponteiros, um para o filho esquerdo e outro para o filho direito.
- A implementação das operações de árvore usará recursividade.

Implementação

- O nó da árvore será um struct com três campos:
  - uma chave inteira e dois ponteiros, um para o filho esquerdo e outro para o filho direito.
- A implementação das operações de árvore usará recursividade.

```
1  /*********************************
2   * Definicao do struct Node
3   *********************************/
4  struct Node {
5      int key;
6      Node *left;
7      Node *right;
8
9      Node(int k, Node *l, Node *r)
10         : key(k), left(l), right(r)
11         { }
12
13     ~Node() {
14         cout << "removido: " << this->key << endl;
15     }
16 };
```

```
1  class BST { // classe BST (Binary Search Tree)
2  public:
3      BST();
4      ~BST();
5      void add(int key);          // Adicionar chave
6      void remove(int key);       // Remover chave
7      bool contains(int key);     // A arvore contem esta chave?
8      int minimum();              // Devolve chave minima
9      int maximum();              // Devolve chave maxima
10     int successor(int k);       // Devolve chave sucessora de k
11     int predecessor(int k);     // Devolve chave antecessora de k
12 private:
13     Node *root;
14     Node *add(Node *node, int key);
15     Node *search(Node *node, int key);
16     Node *clear(Node *node);
17     Node *minimum(Node *node);
18     Node *maximum(Node *node);
19     Node *ancestral_sucessor(Node *x, Node *raiz);
20     Node *ancestral_predecessor(Node *x, Node* raiz);
21     Node *sucessor(Node *x, Node *raiz);
22     Node *predecessor(Node *x, Node* raiz);
23     Node *remove(int k, Node *node);
24     Node *removeRoot(Node *node);
25 };
```

Implementação - Construtor e Destrutor

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1 BST::BST() {  // Construtor
2     root = nullptr;
3 }
```

Implementação - Construtor e Destrutor


```
1 BST::BST() {  // Construtor
2     root = nullptr;
3 }
4
5 BST::~BST() {  // Destrutor
6     root = clear(root);
7 }
```

Implementação - Construtor e Destrutor

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  BST::BST() {  // Construtor
2      root = nullptr;
3  }
4
5  BST::~BST() {  // Destrutor
6      root = clear(root);
7  }
8
9  // Esta funcao recebe um ponteiro para um node e libera
10 // os nos da arvore enraizada nesse node. A funcao devolve
11 // nullptr apos apagar a arvore enraizada em node
12 Node *BST::clear(Node *node) {
13     if(node != nullptr) {
14         node->left = clear(node->left);
15         node->right = clear(node->right);
16         delete node;
17     }
18     return nullptr;
19 }
```

Busca por um valor

A ideia é semelhante àquela da busca binária:

Busca por um valor

A ideia é semelhante àquela da busca binária:

- Ou o valor a ser buscado está na raiz da árvore

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

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

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

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

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

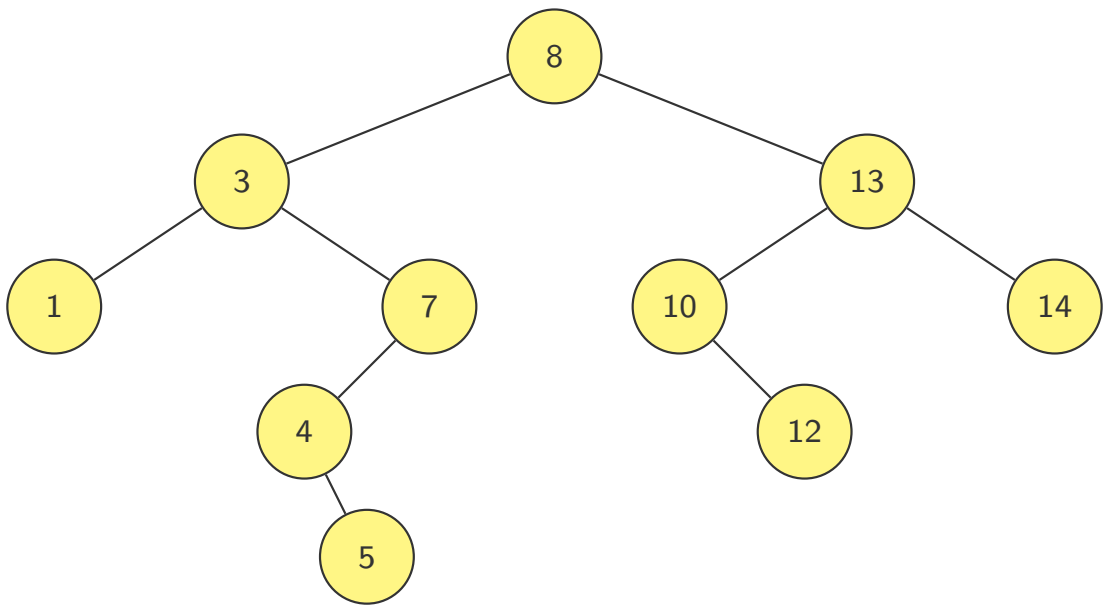
Ex: Buscando por 4

Busca por um valor

A ideia é semelhante 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

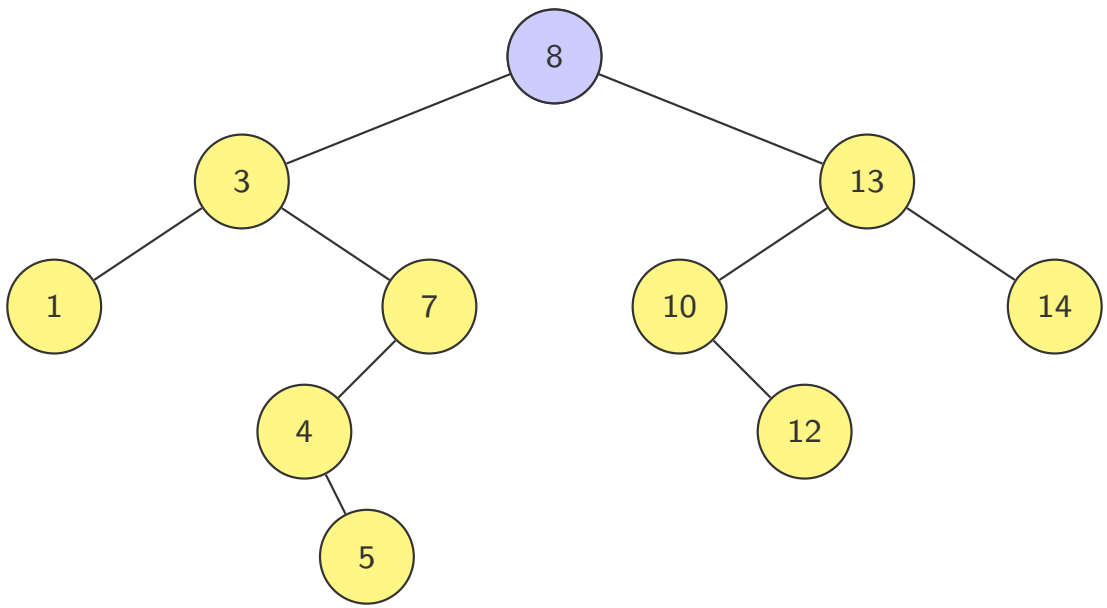


Busca por um valor

A ideia é semelhante 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

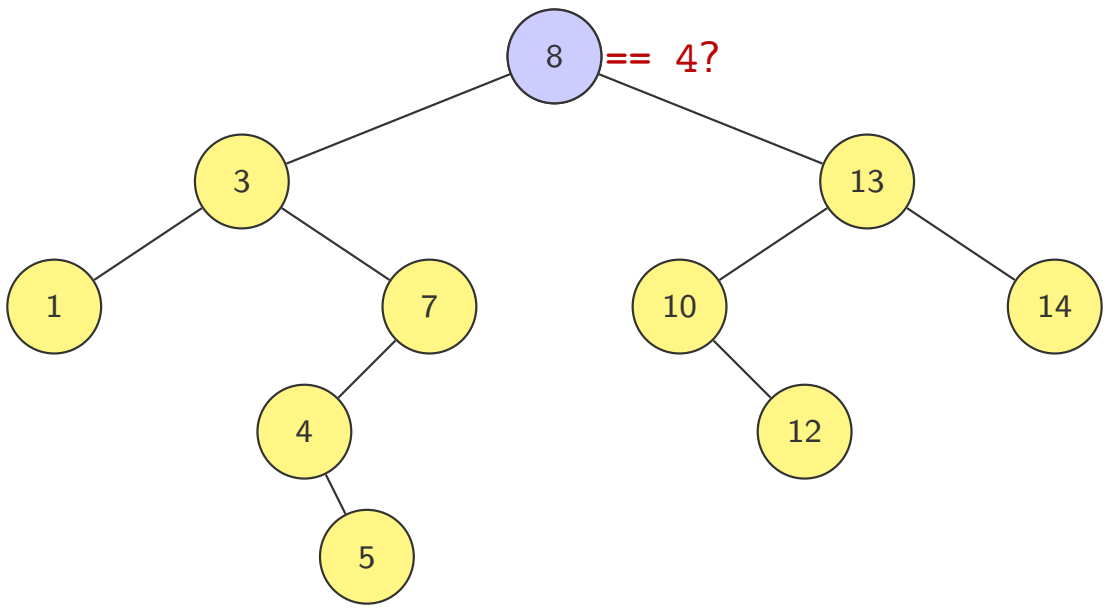


Busca por um valor

A ideia é semelhante 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

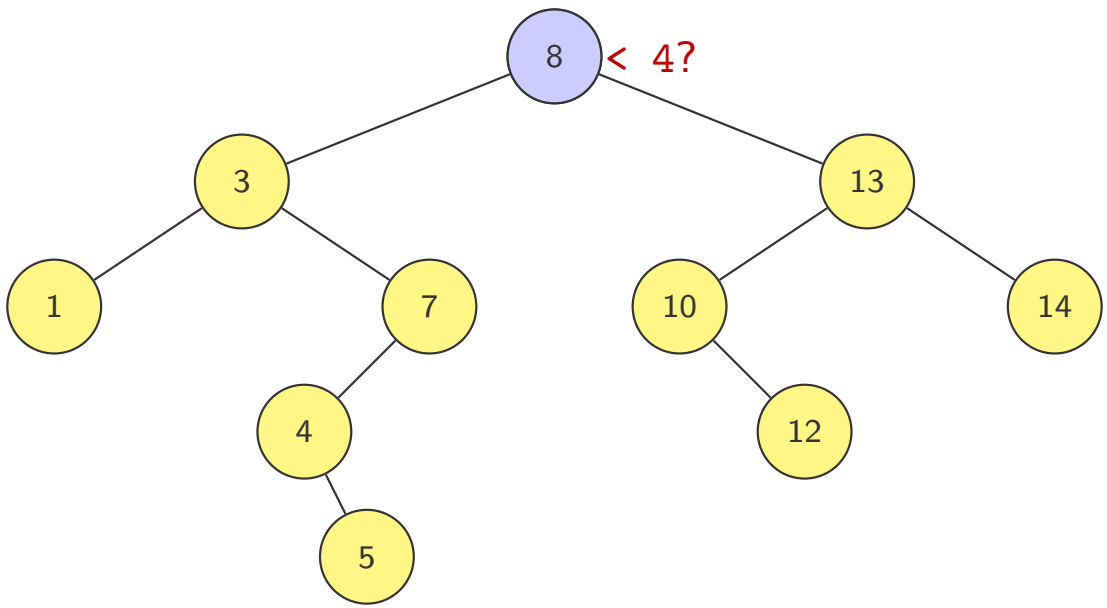


Busca por um valor

A ideia é semelhante 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

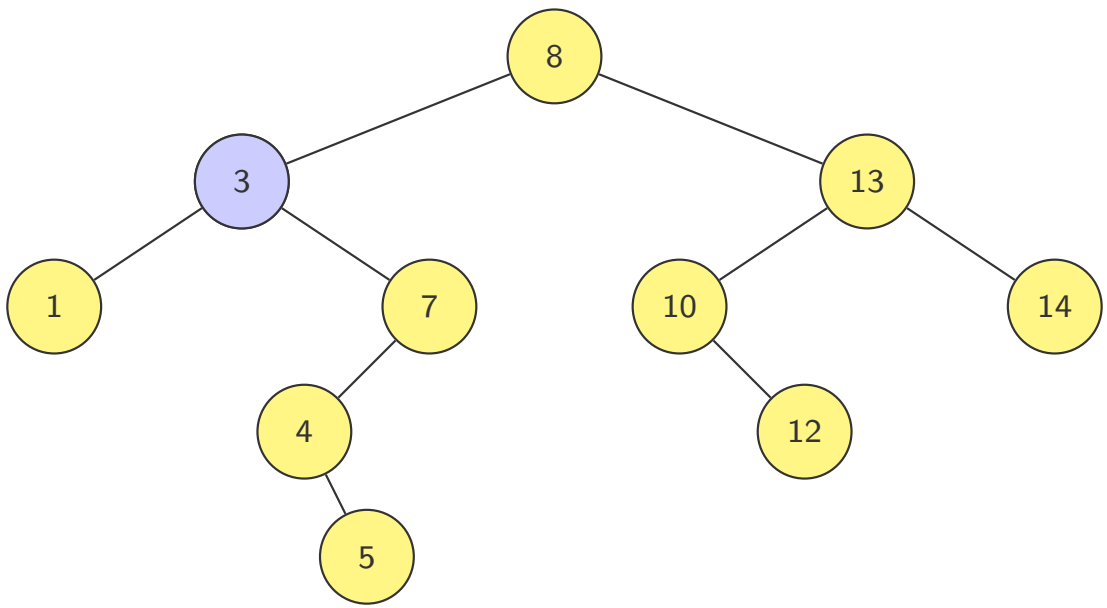


Busca por um valor

A ideia é semelhante 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

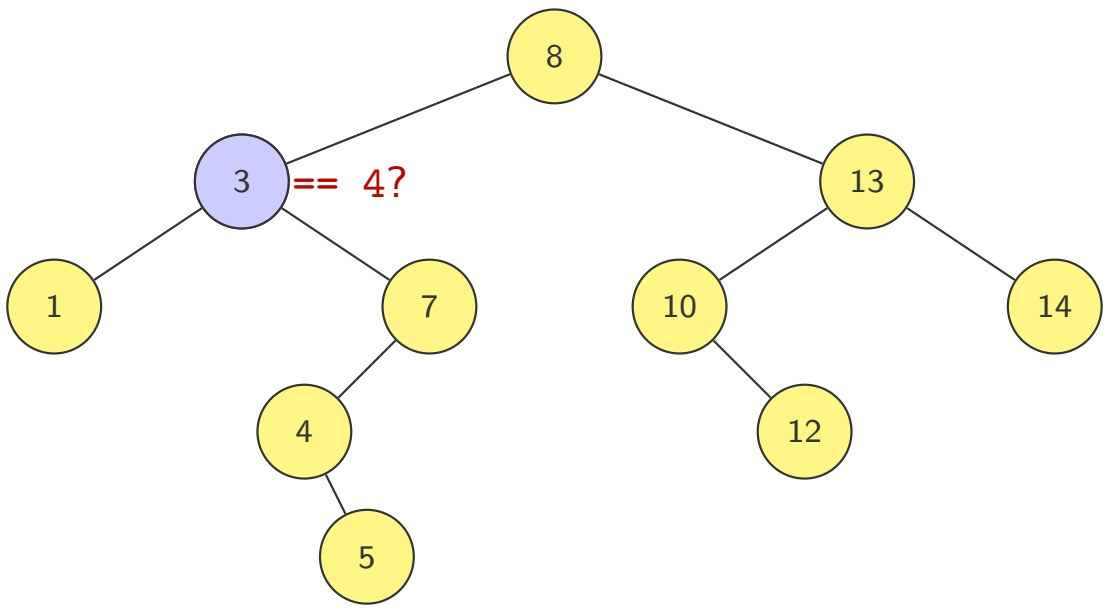


Busca por um valor

A ideia é semelhante 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

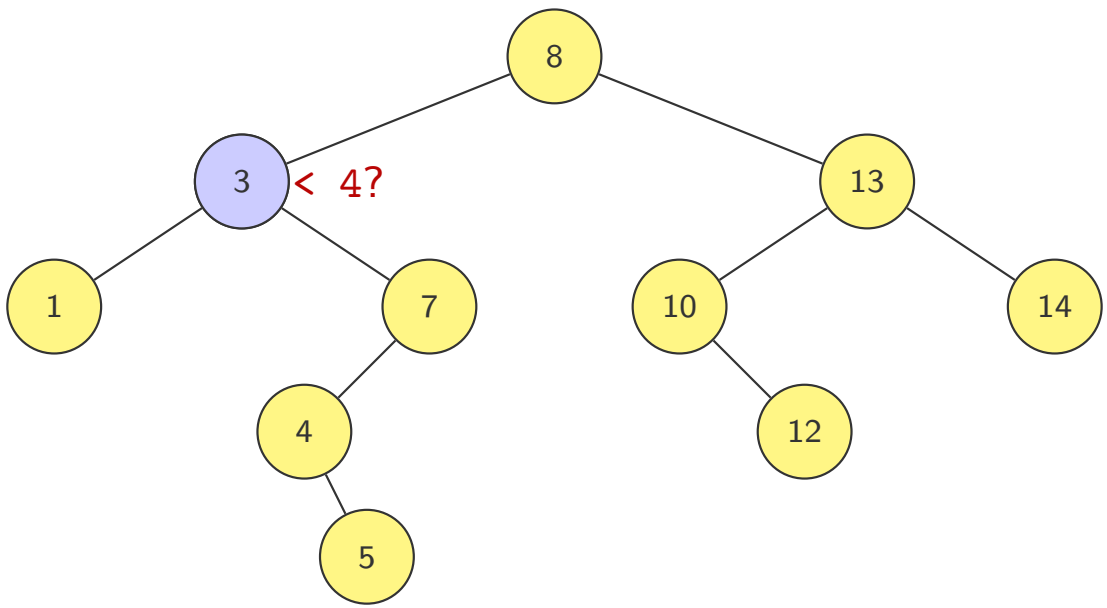


Busca por um valor

A ideia é semelhante 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

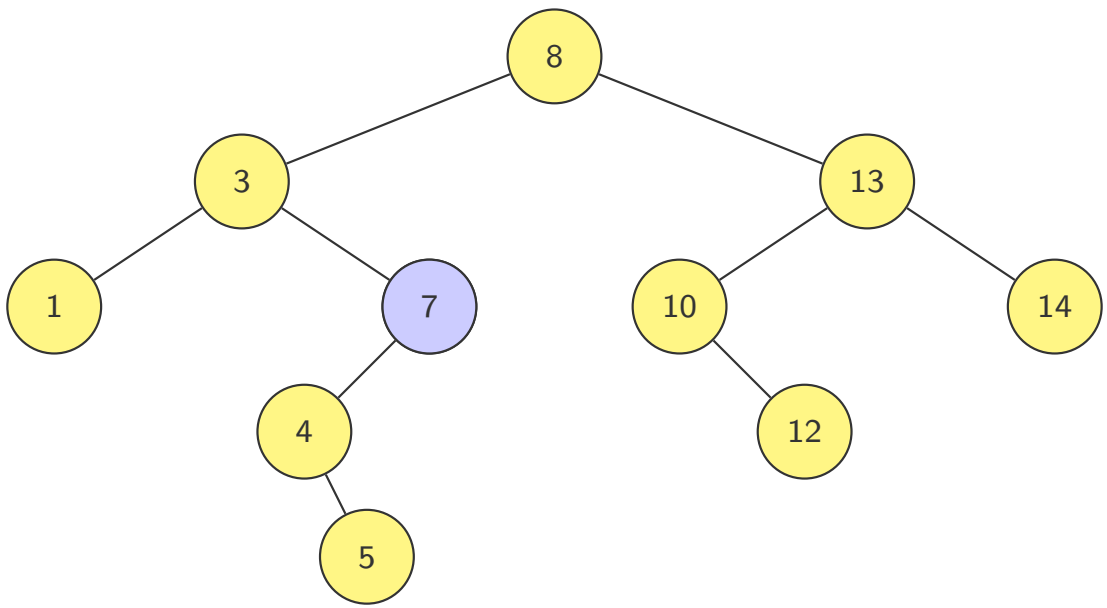


Busca por um valor

A ideia é semelhante 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

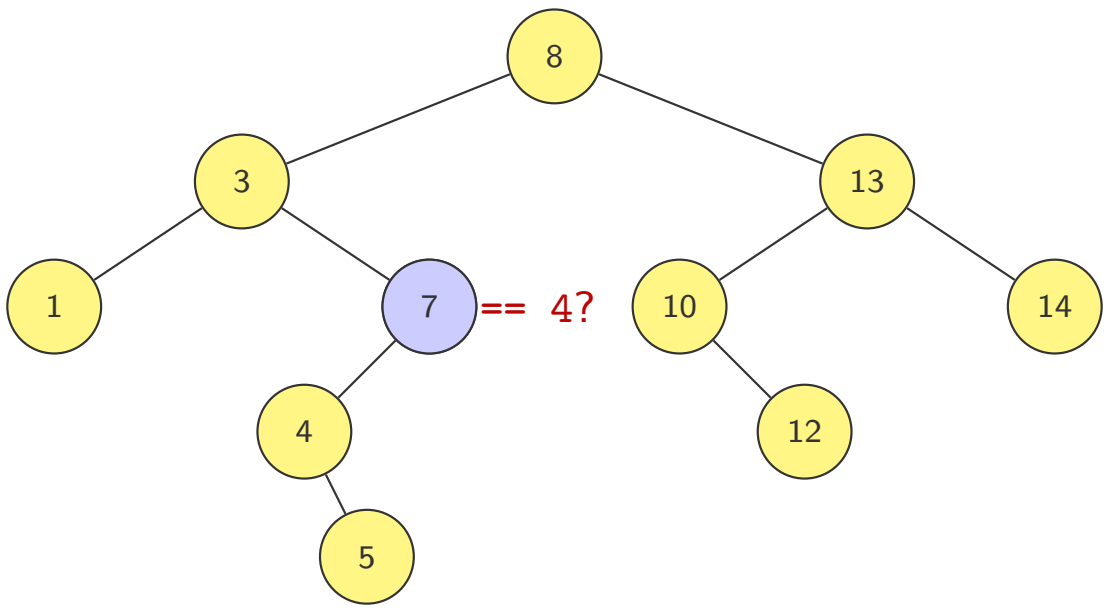


Busca por um valor

A ideia é semelhante 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

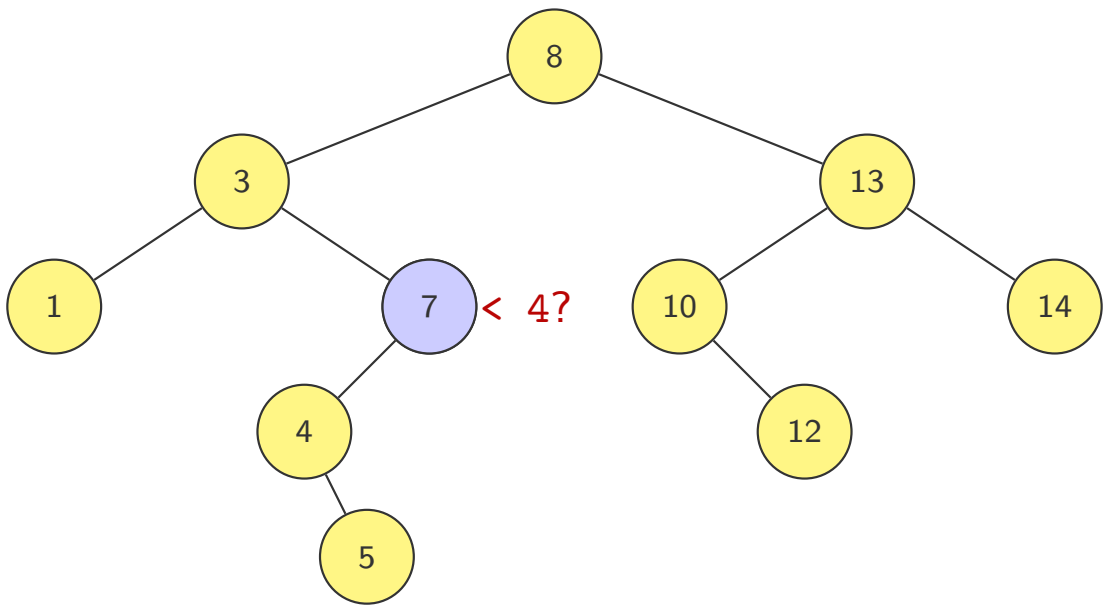


Busca por um valor

A ideia é semelhante 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

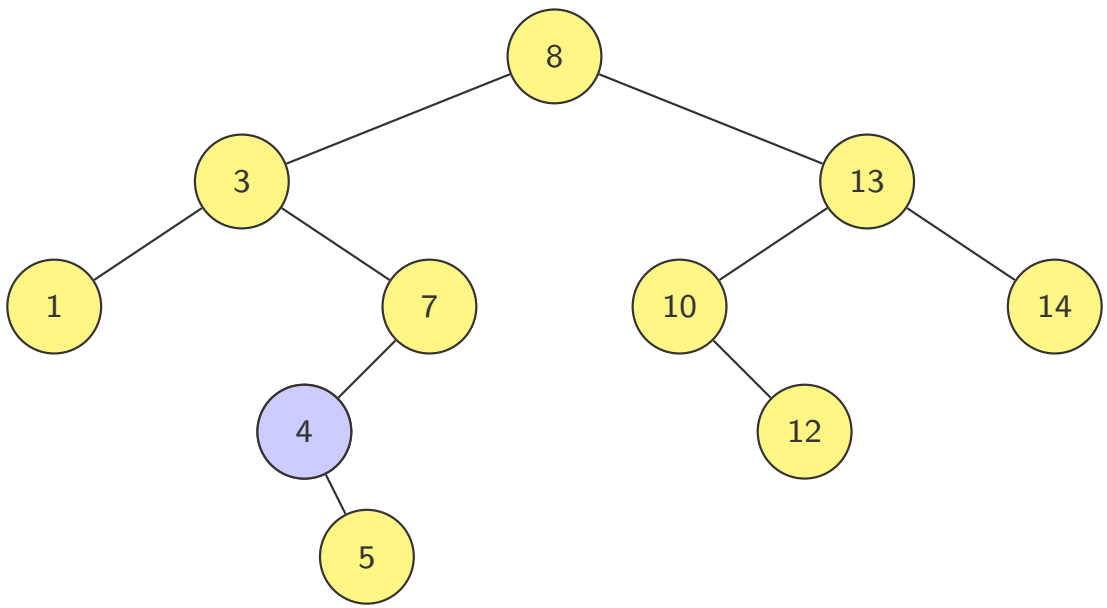


Busca por um valor

A ideia é semelhante 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

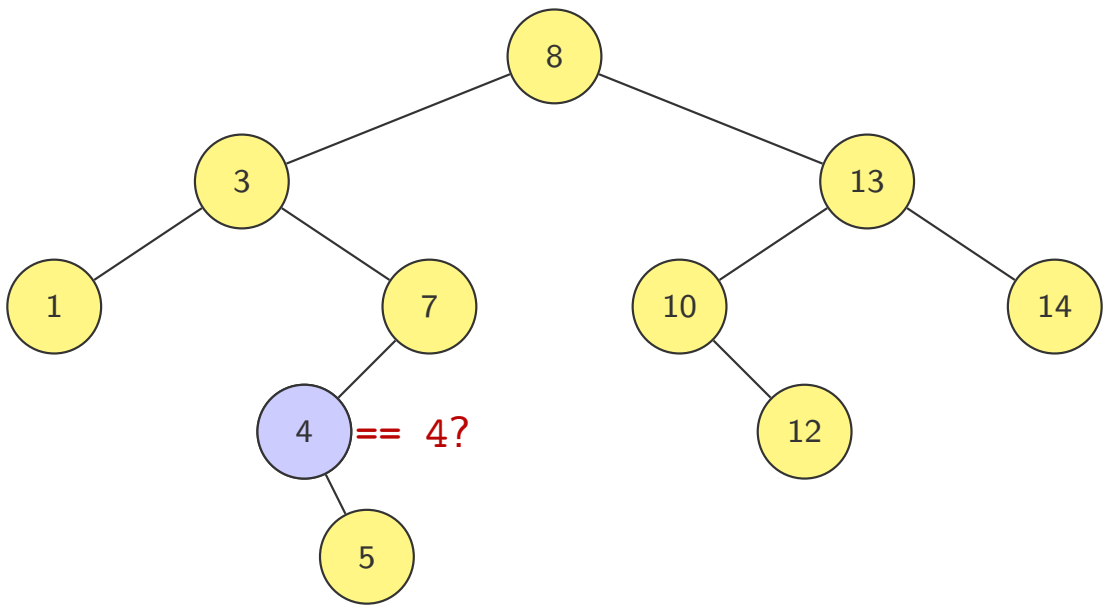


Busca por um valor

A ideia é semelhante 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

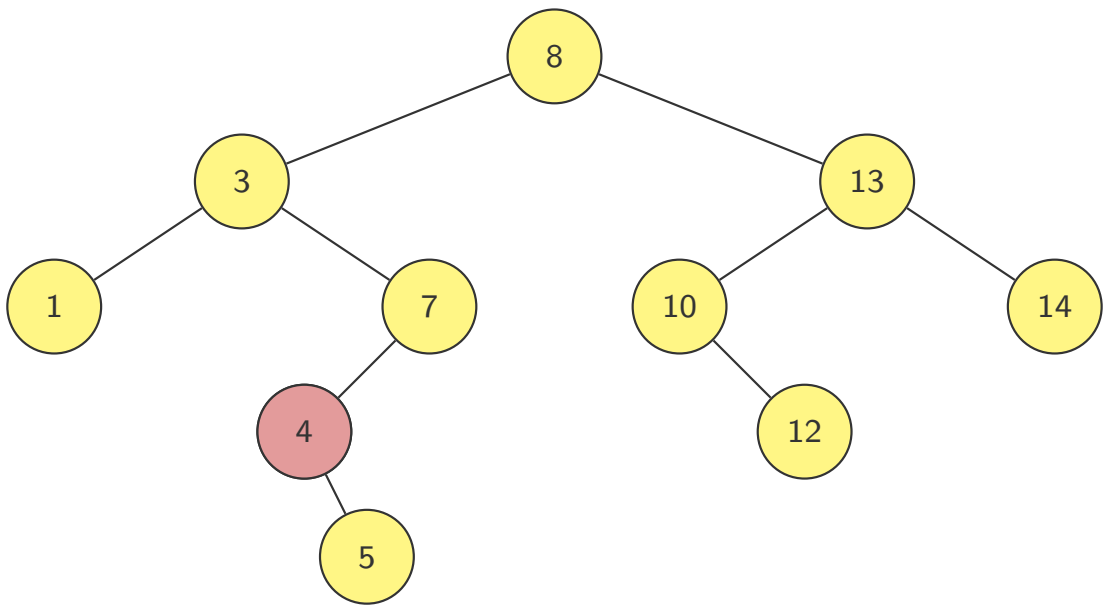


Busca por um valor

A ideia é semelhante 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

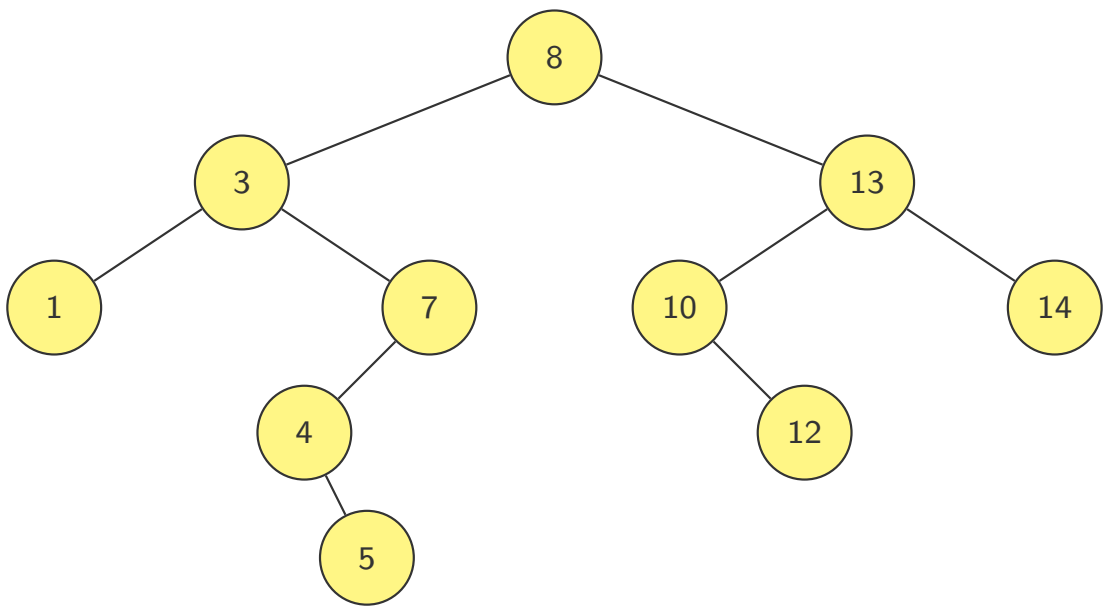


Busca por um valor

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

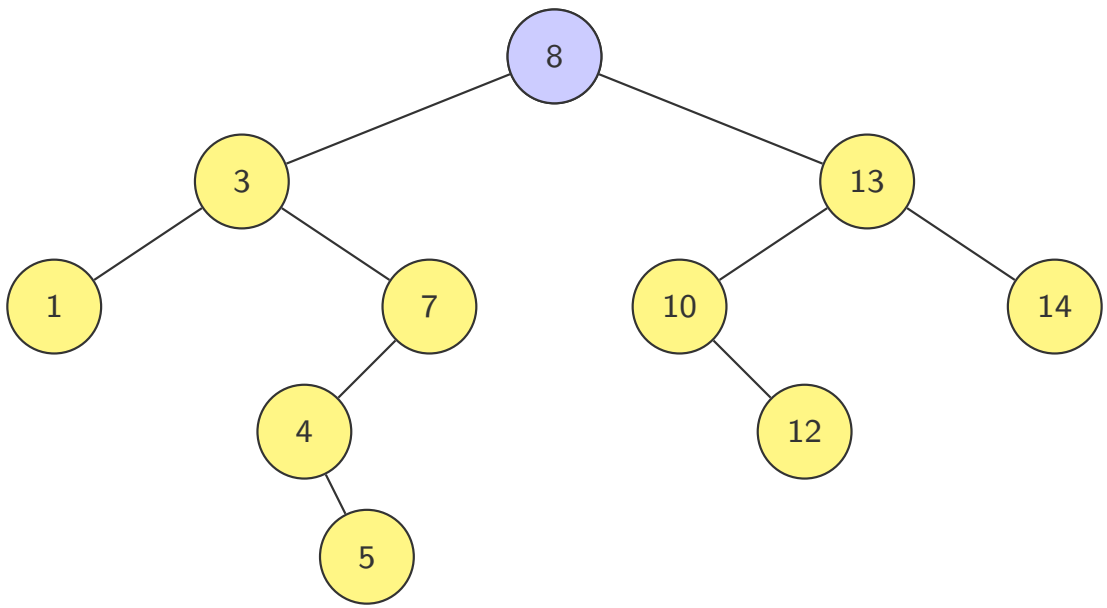


Busca por um valor

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

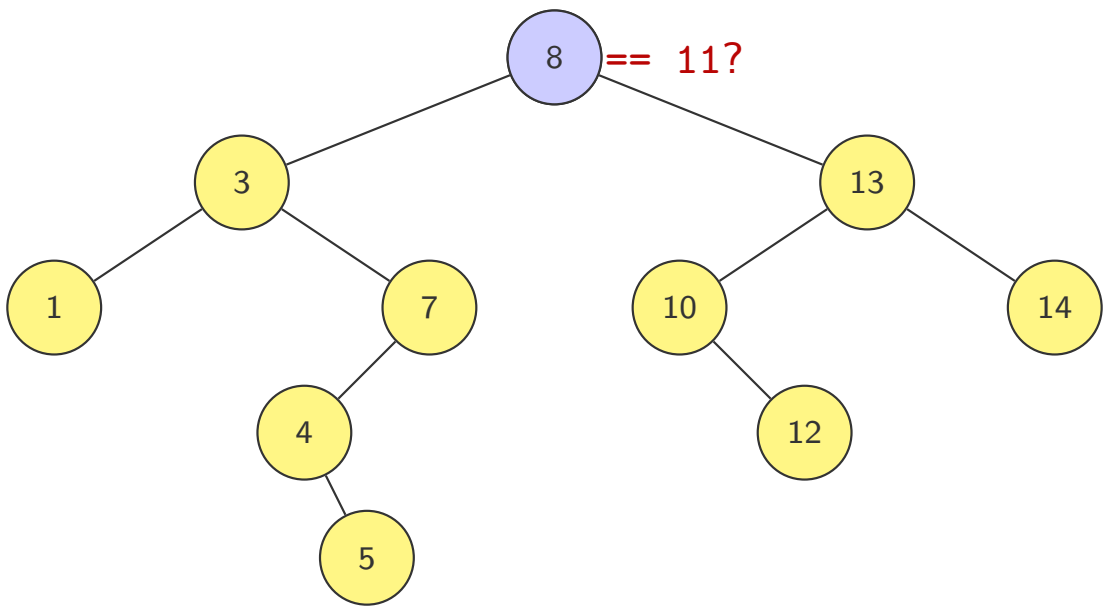


Busca por um valor

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

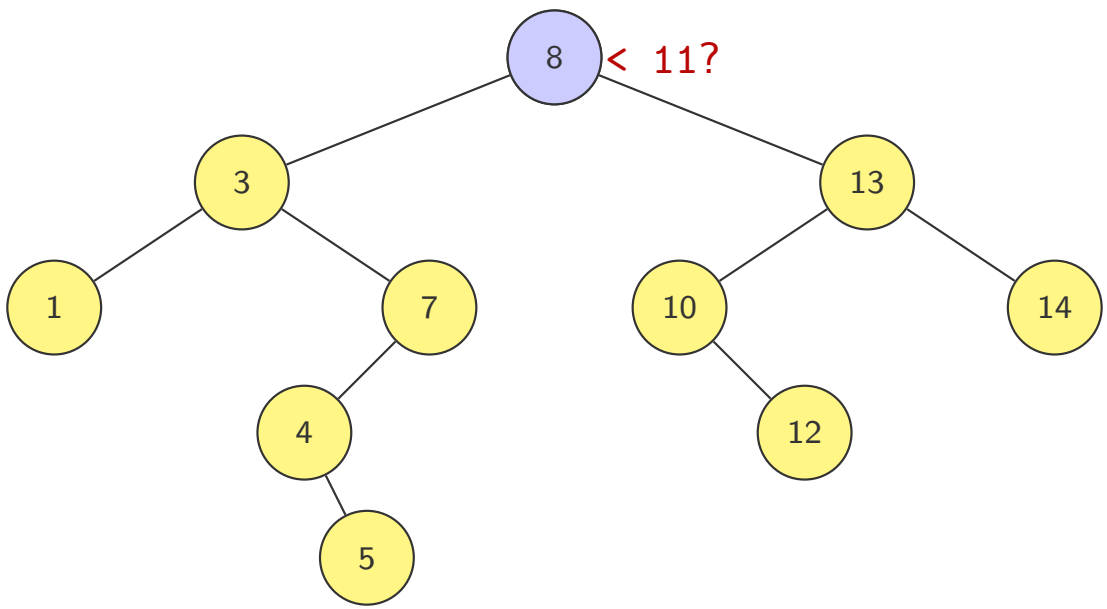


Busca por um valor

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

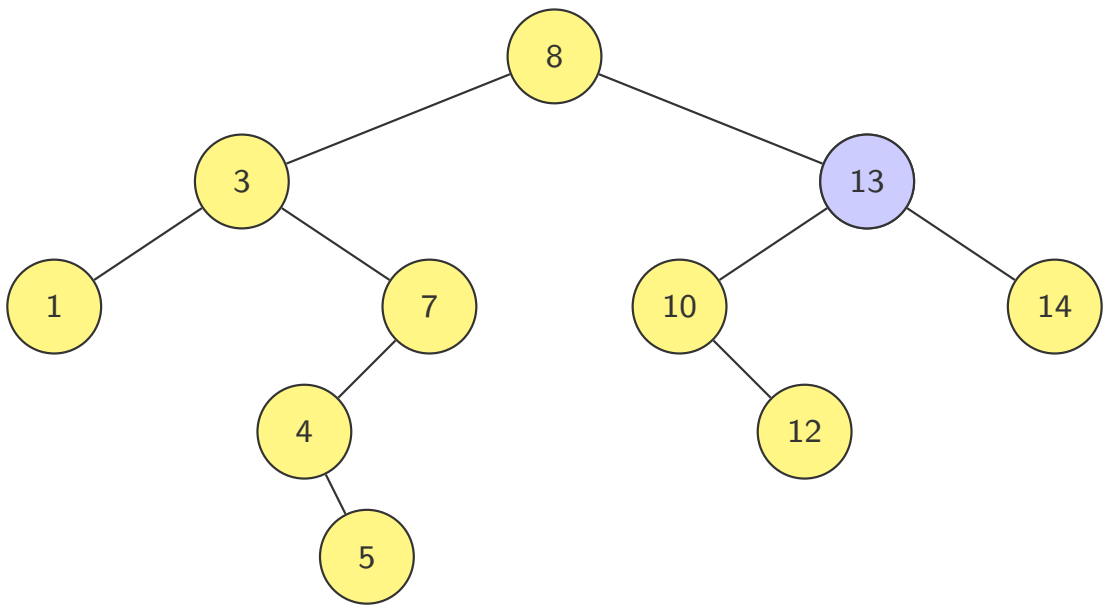


Busca por um valor

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

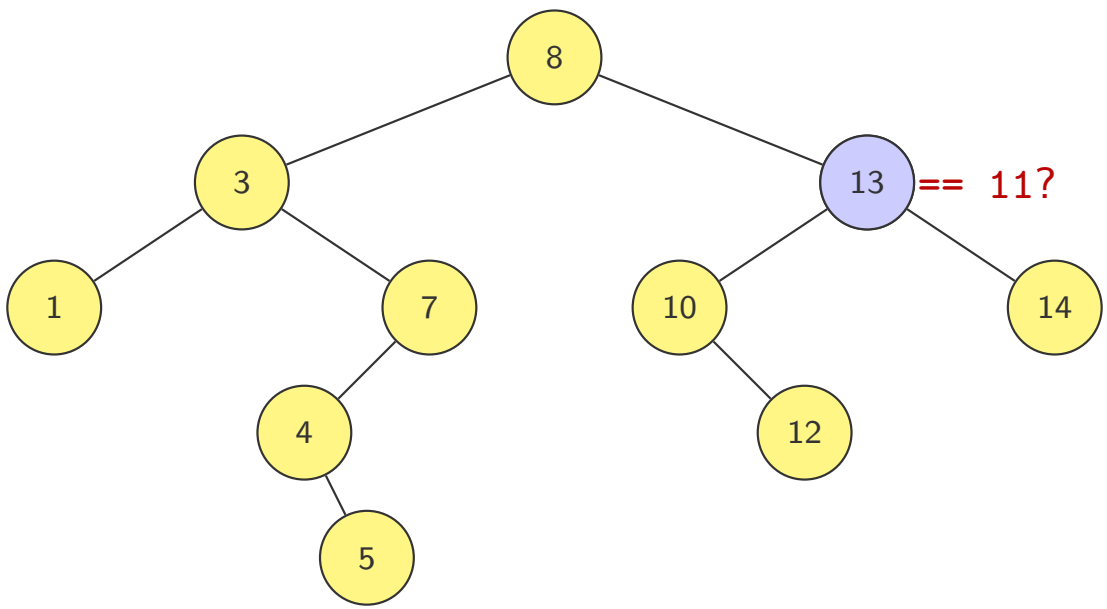


Busca por um valor

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

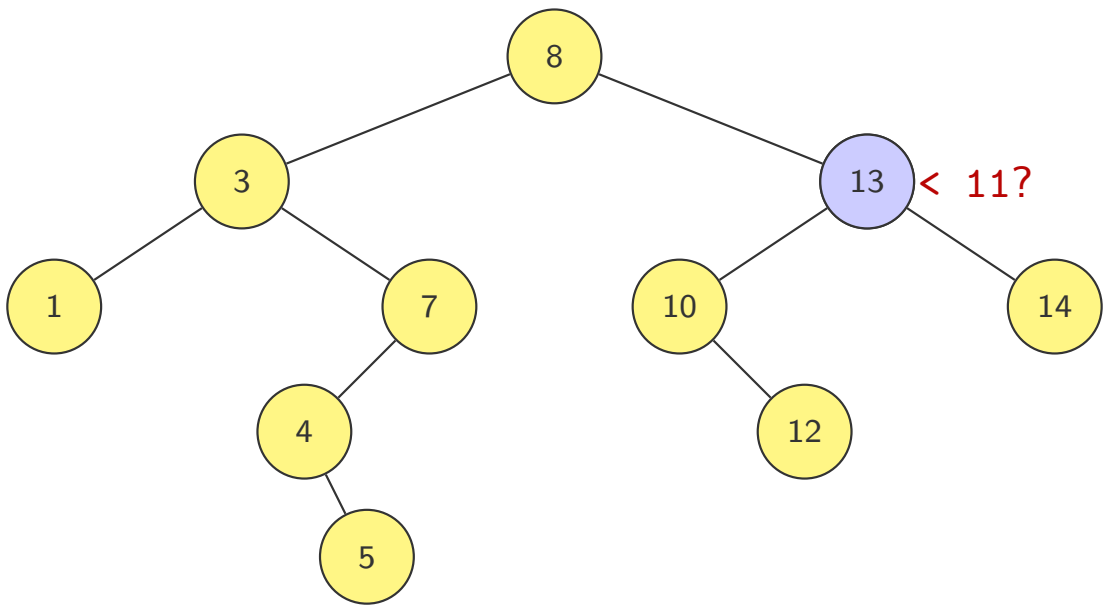


Busca por um valor

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

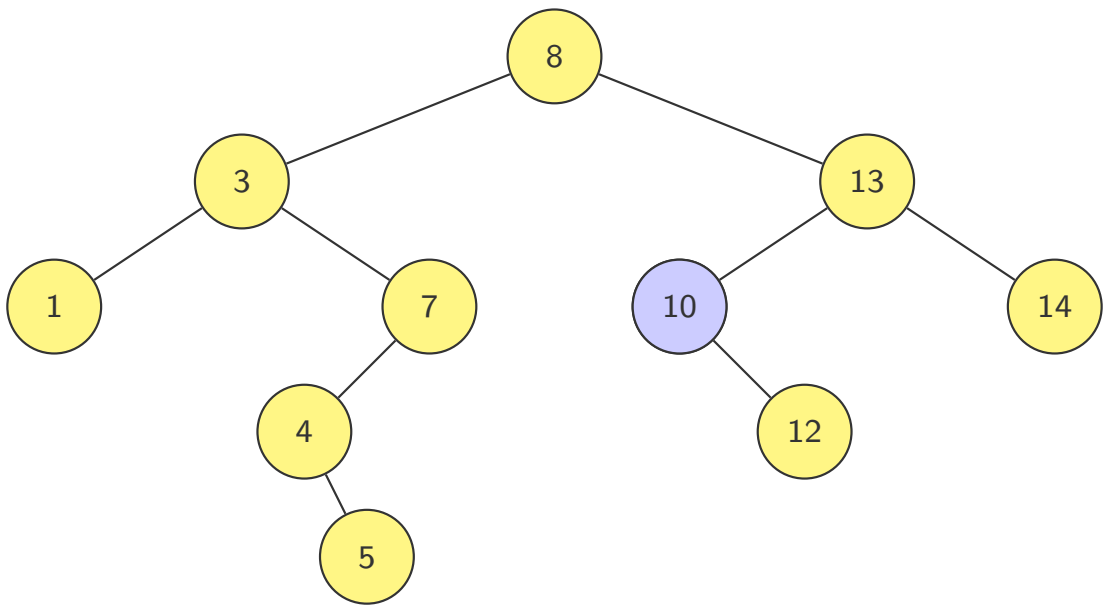


Busca por um valor

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

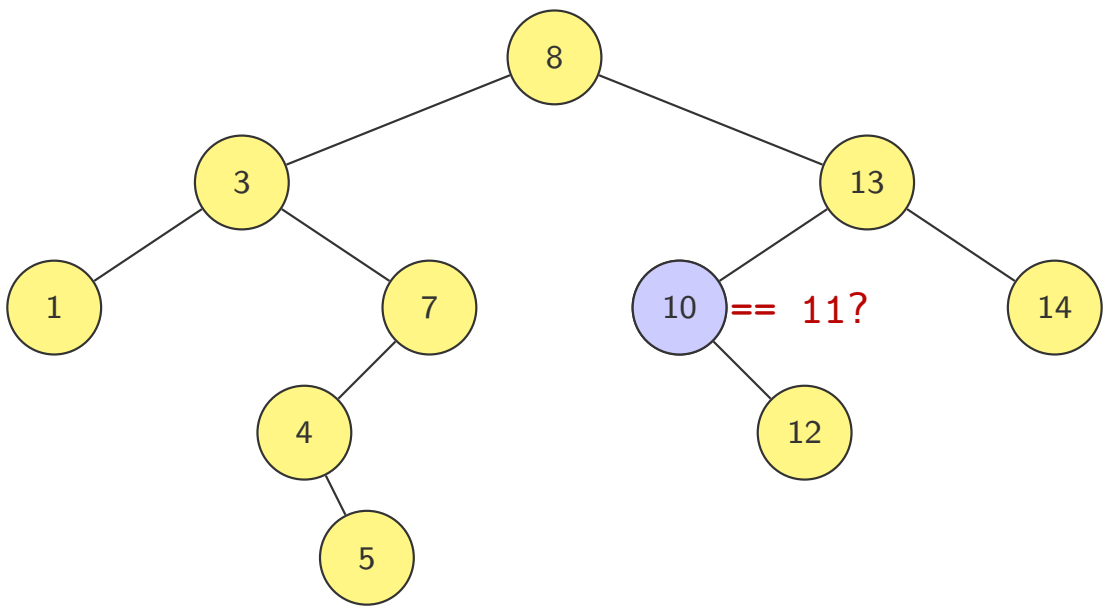


Busca por um valor

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

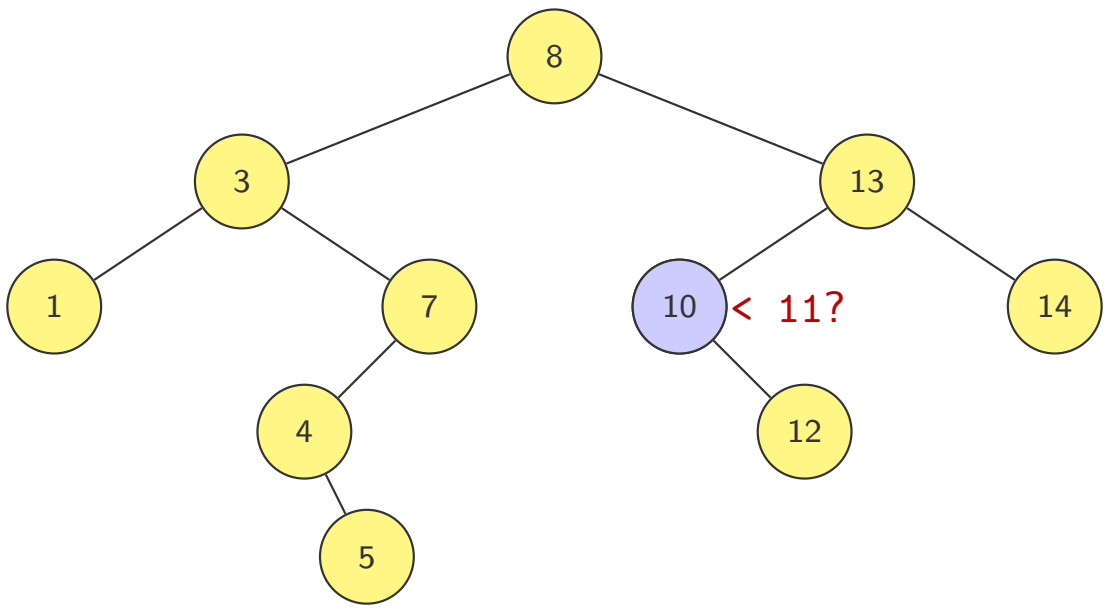


Busca por um valor

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

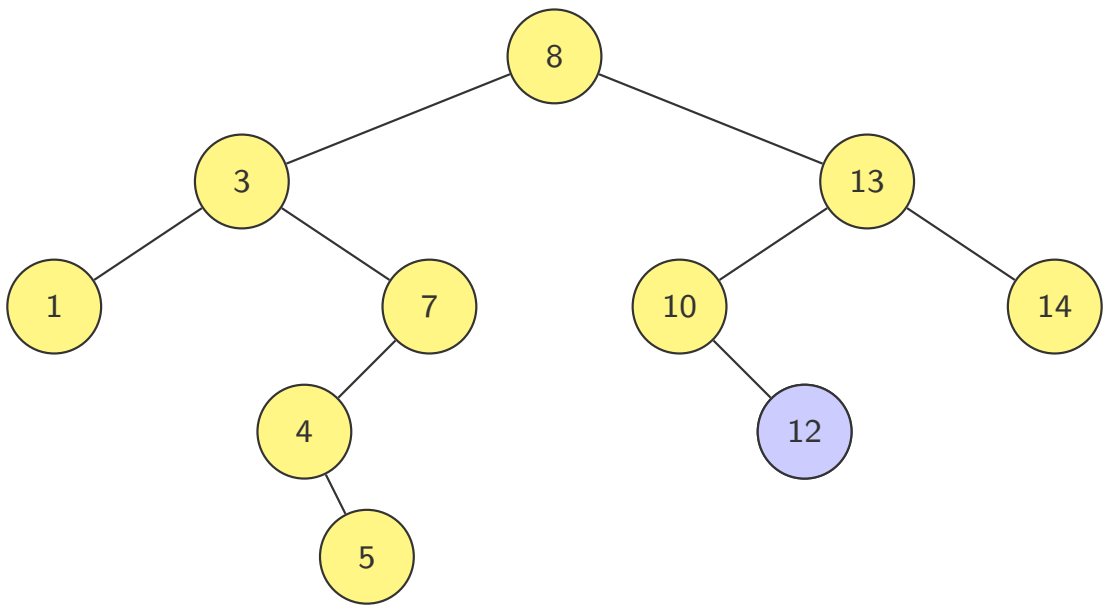


Busca por um valor

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

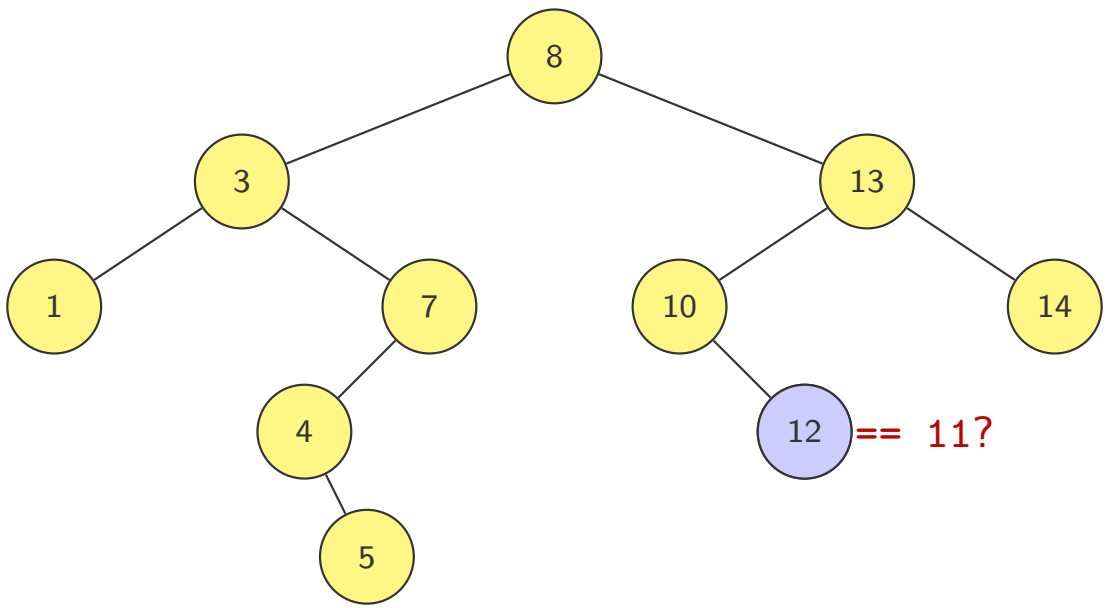


Busca por um valor

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

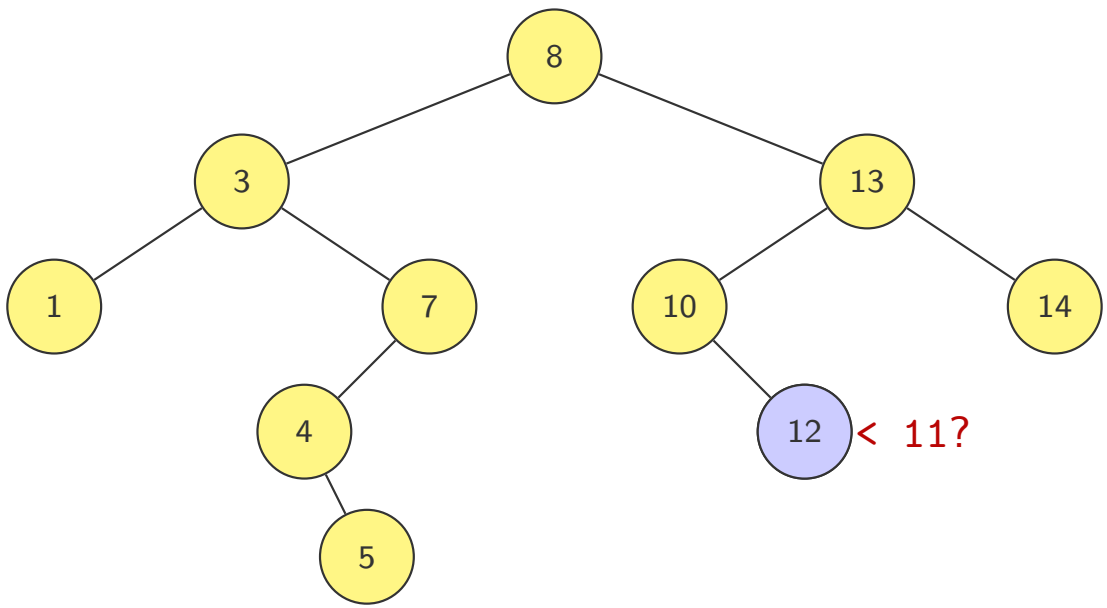


Busca por um valor

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

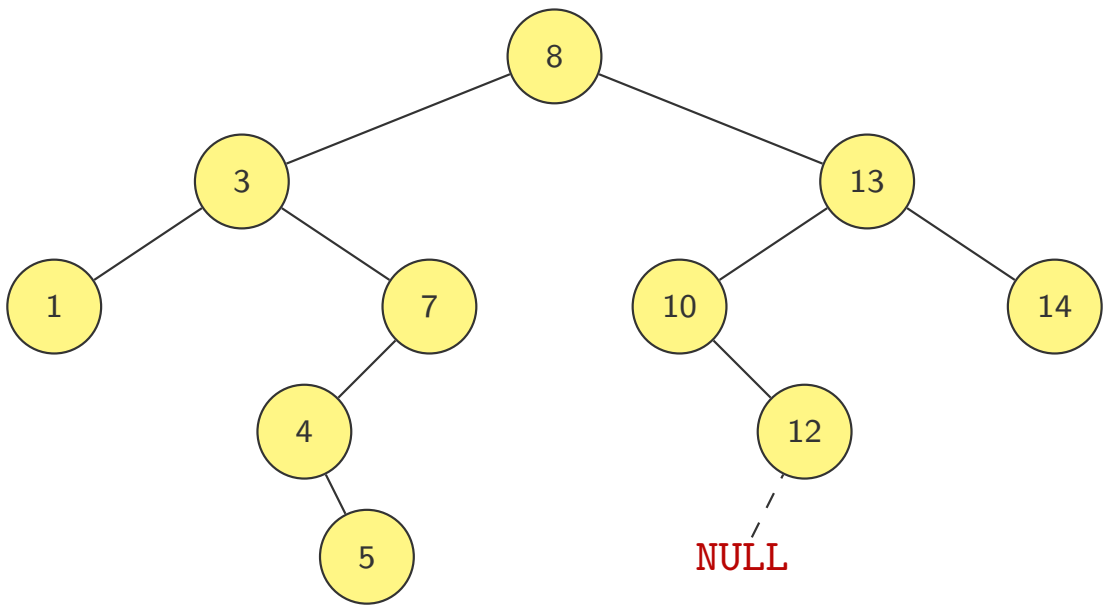


Busca por um valor

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



Implementação - Busca

Implementação - Busca

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  // Esta funcao recebe um ponteiro para node e uma key e:
2  // (1) devolve nullptr caso key nao esteja na arvore;
3  // (2) devolve um ponteiro para o no contendo key caso
       contrario.
4  Node *BST::search(Node *node, int key) {
5      if(node == nullptr || node->key == key)
6          return node;
7      if(key > node->key)
8          return search(node->right, key);
9      else
10         return search(node->left, key);
11 }
```

Implementação - Busca

Universidade Federal do Ceará
Campus Quixadá

```
1  // Esta funcao recebe um ponteiro para node e uma key e:
2  // (1) devolve nullptr caso key nao esteja na arvore;
3  // (2) devolve um ponteiro para o no contendo key caso
       contrario.
4  Node *BST::search(Node *node, int key) {
5      if(node == nullptr || node->key == key)
6          return node;
7      if(key > node->key)
8          return search(node->right, key);
9      else
10         return search(node->left, key);
11 }
12
13 bool BST::contains(int key) {
14     return search(root, key) != nullptr;
15 }
```

Eficiência da busca

Qual é o tempo da busca?

Eficiência da busca

Qual é o tempo da busca?

- depende da forma da árvore...

Eficiência da busca

Qual é o tempo da busca?

- depende da forma da árvore...

Ex: 31 nós

Eficiência da busca

Qual é o tempo da busca?

- depende da forma da árvore...

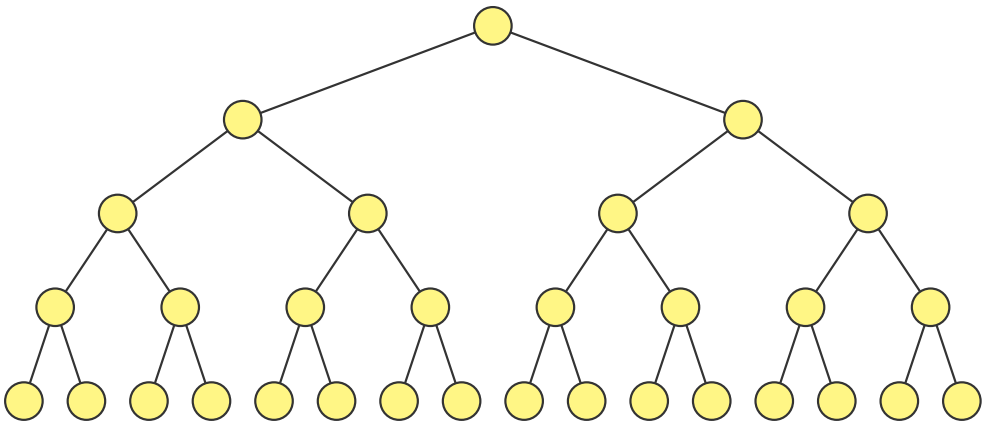
Ex: 31 nós

Eficiência da busca

Qual é o tempo da busca?

- depende da forma da árvore...

Ex: 31 nós



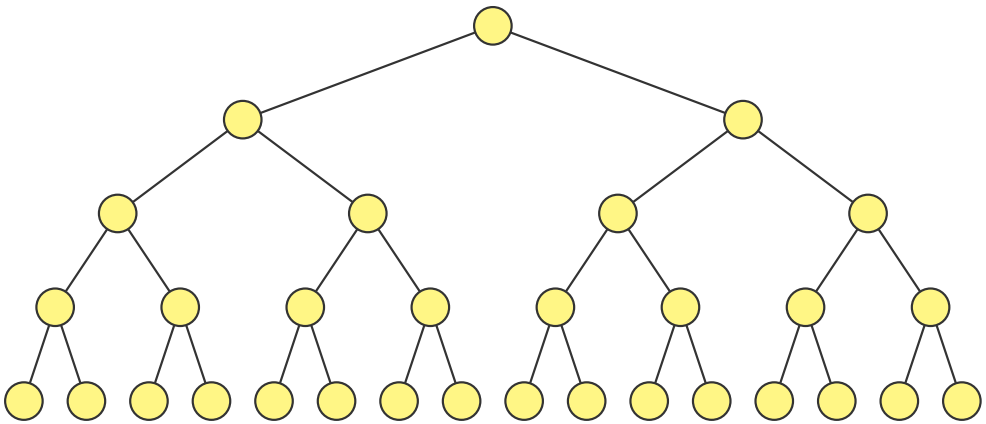
Melhor árvore: $O(\lg n)$

Eficiência da busca

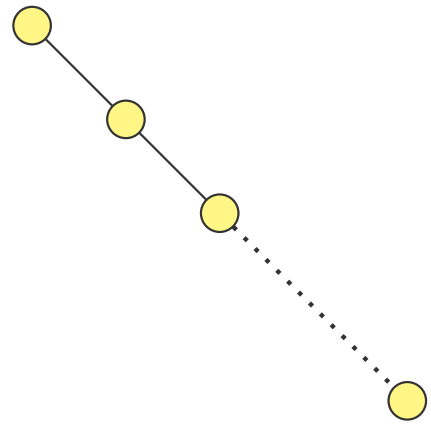
Qual é o tempo da busca?

- depende da forma da árvore...

Ex: 31 nós



Melhor árvore: $O(\lg n)$



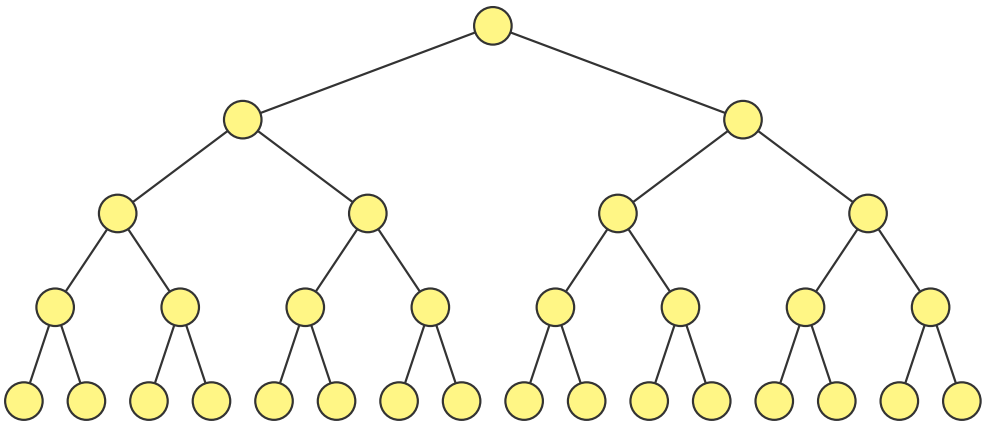
Pior árvore: $O(n)$

Eficiência da busca

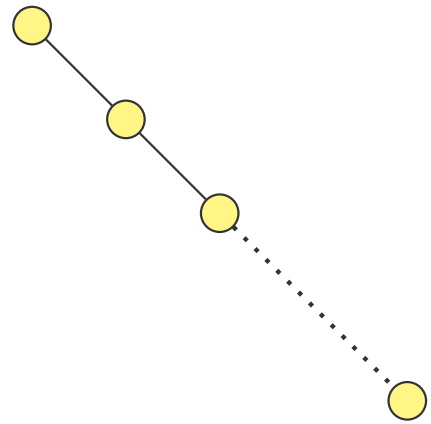
Qual é o tempo da busca?

- depende da forma da árvore...

Ex: 31 nós



Melhor árvore: $O(\lg n)$



Pior árvore: $O(n)$

Para ter a pior árvore basta inserir em ordem crescente...

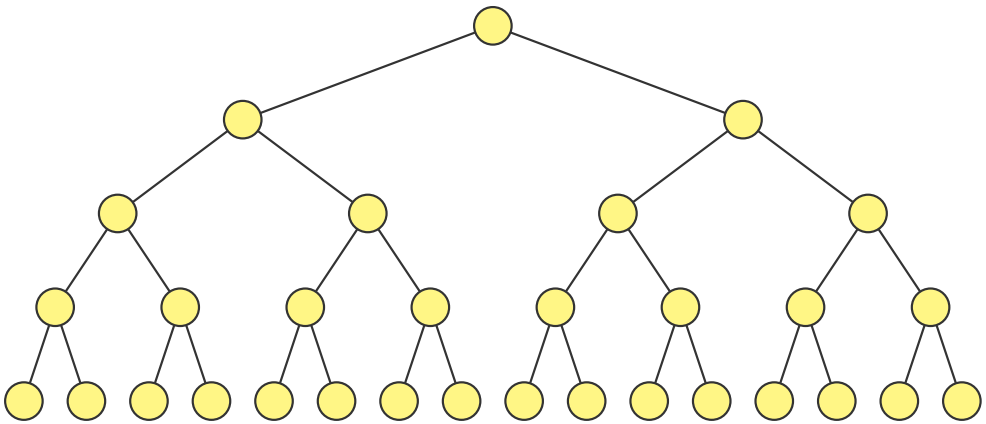
Eficiência da busca

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

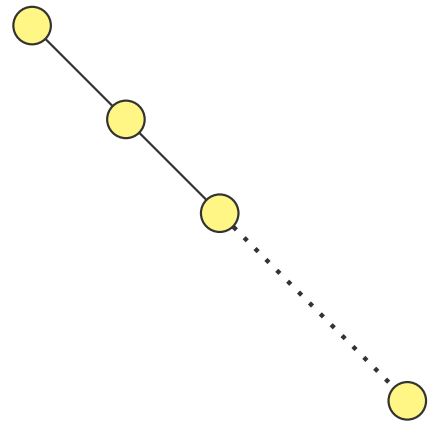
Qual é o tempo da busca?

- depende da forma da árvore...

Ex: 31 nós



Melhor árvore: $O(\lg n)$



Pior árvore: $O(n)$

Para ter a pior árvore basta inserir em ordem crescente...

Caso médio: em uma árvore com n elementos adicionados em ordem aleatória a busca demora (em média) $O(\lg n)$

Inserindo um valor

Precisamos determinar onde inserir o valor:

Inserindo um valor

Precisamos determinar onde inserir o valor:

- fazemos uma busca pelo valor

Inserindo um valor

Precisamos determinar onde inserir o valor:

- fazemos uma busca pelo valor
- e colocamos ele na posição onde deveria estar

Inserindo um valor

Precisamos determinar onde inserir o valor:

- fazemos uma busca pelo valor
- e colocamos ele na posição onde deveria estar

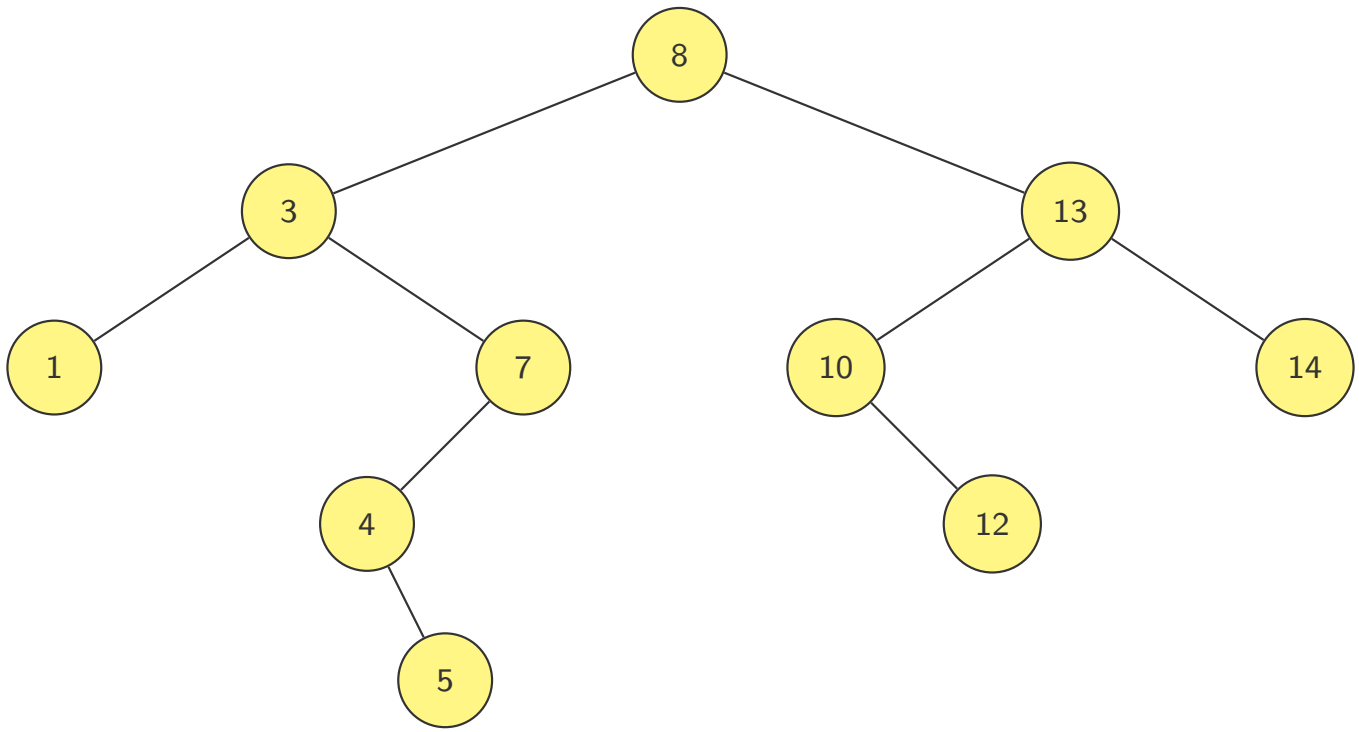
Ex: Inserindo 11

Inserindo um valor

Precisamos determinar onde inserir o valor:

- fazemos uma busca pelo valor
- e colocamos ele na posição onde deveria estar

Ex: Inserindo 11

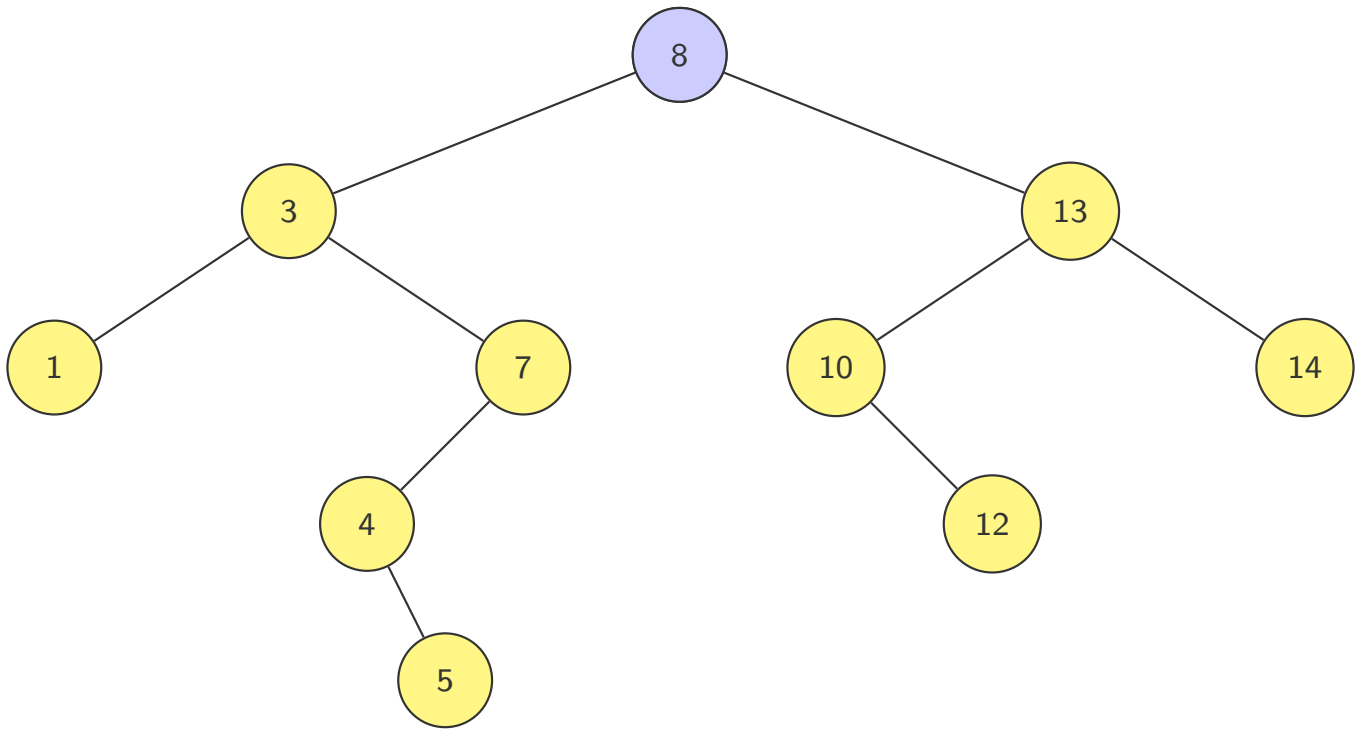


Inserindo um valor

Precisamos determinar onde inserir o valor:

- fazemos uma busca pelo valor
- e colocamos ele na posição onde deveria estar

Ex: Inserindo 11

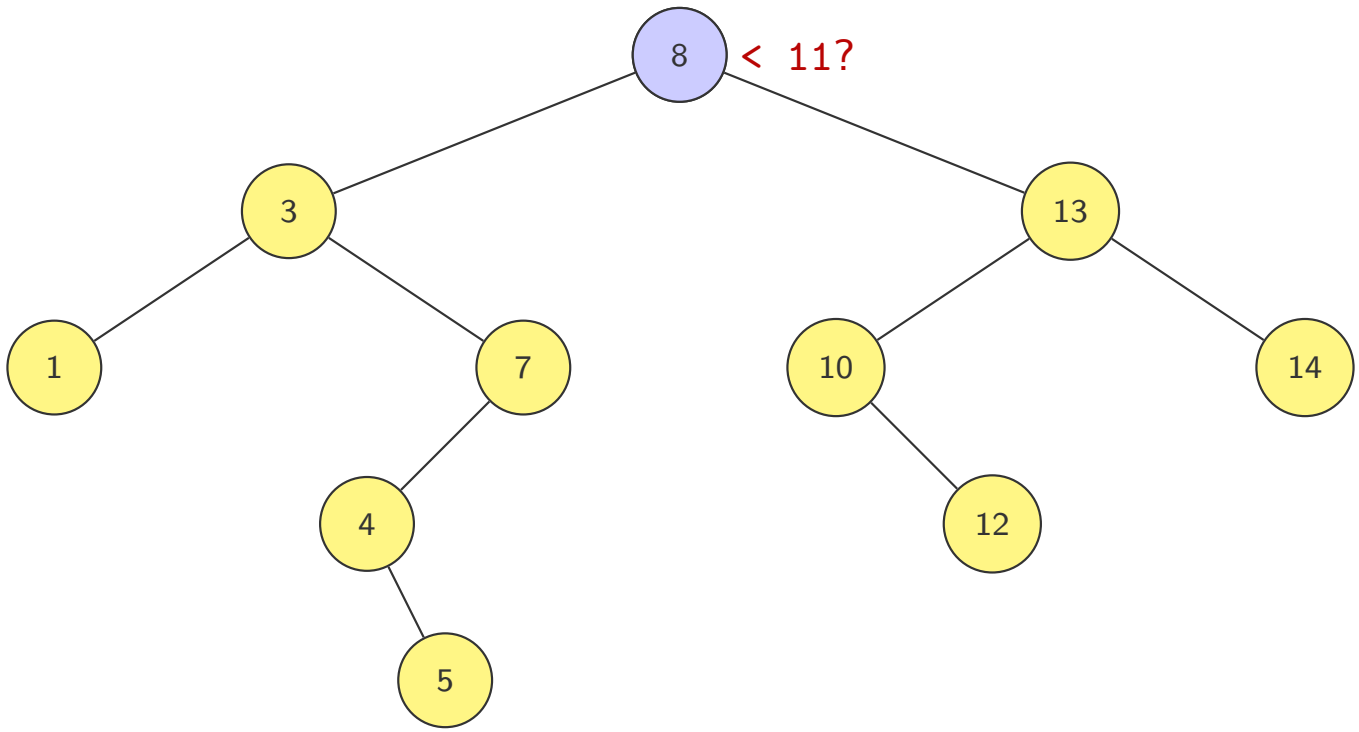


Inserindo um valor

Precisamos determinar onde inserir o valor:

- fazemos uma busca pelo valor
- e colocamos ele na posição onde deveria estar

Ex: Inserindo 11

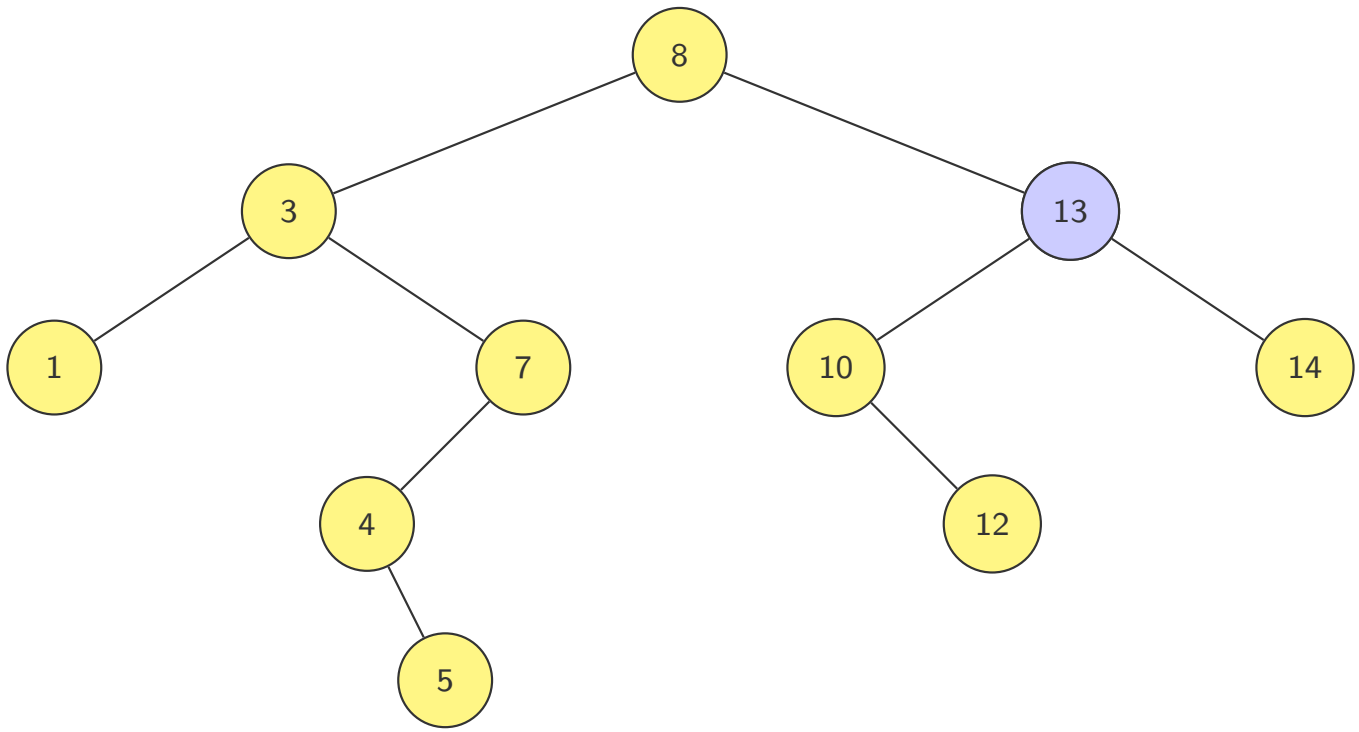


Inserindo um valor

Precisamos determinar onde inserir o valor:

- fazemos uma busca pelo valor
- e colocamos ele na posição onde deveria estar

Ex: Inserindo 11

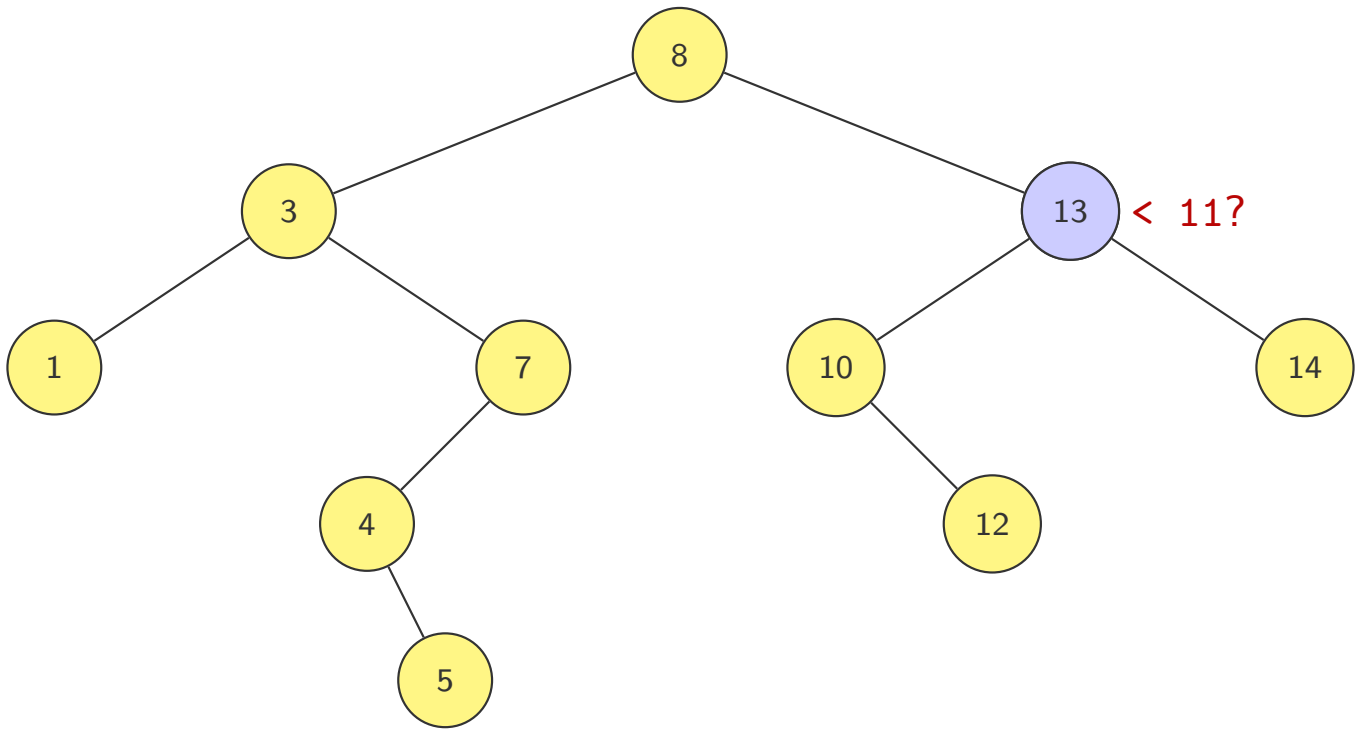


Inserindo um valor

Precisamos determinar onde inserir o valor:

- fazemos uma busca pelo valor
- e colocamos ele na posição onde deveria estar

Ex: Inserindo 11

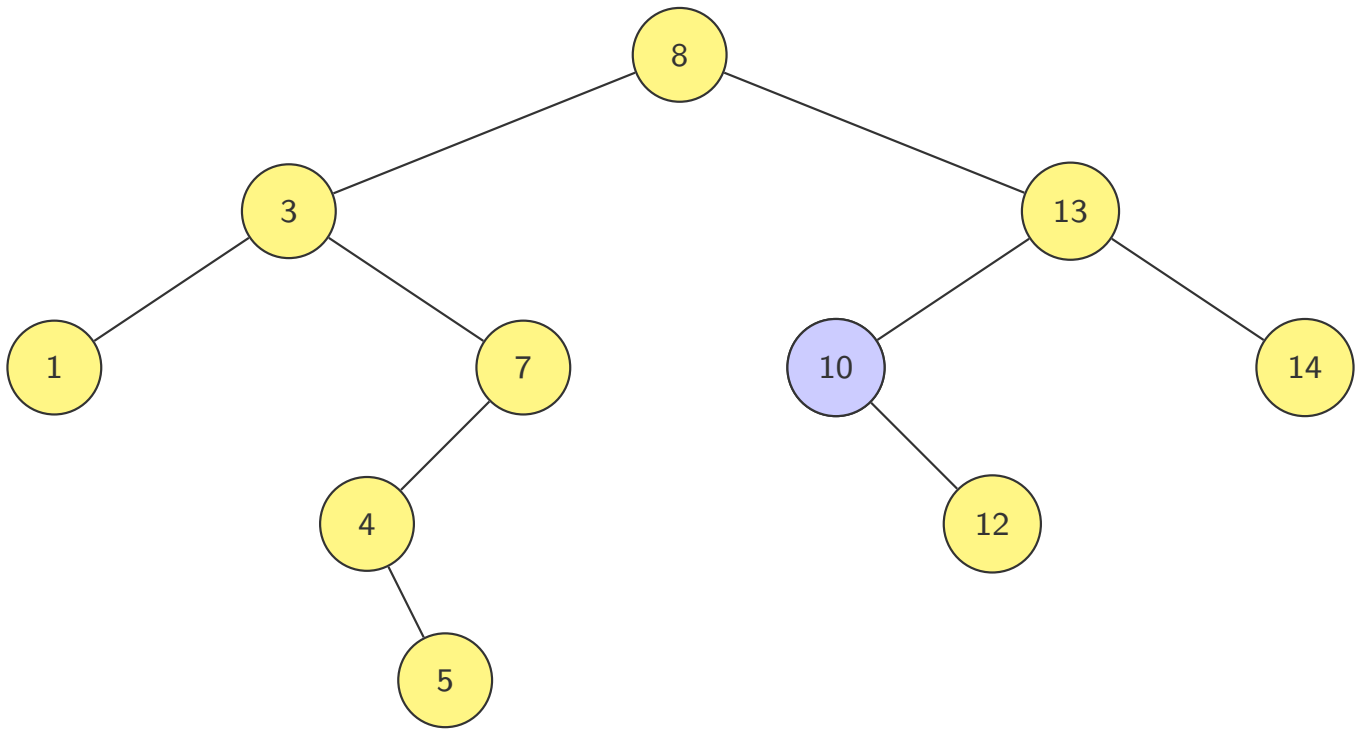


Inserindo um valor

Precisamos determinar onde inserir o valor:

- fazemos uma busca pelo valor
- e colocamos ele na posição onde deveria estar

Ex: Inserindo 11

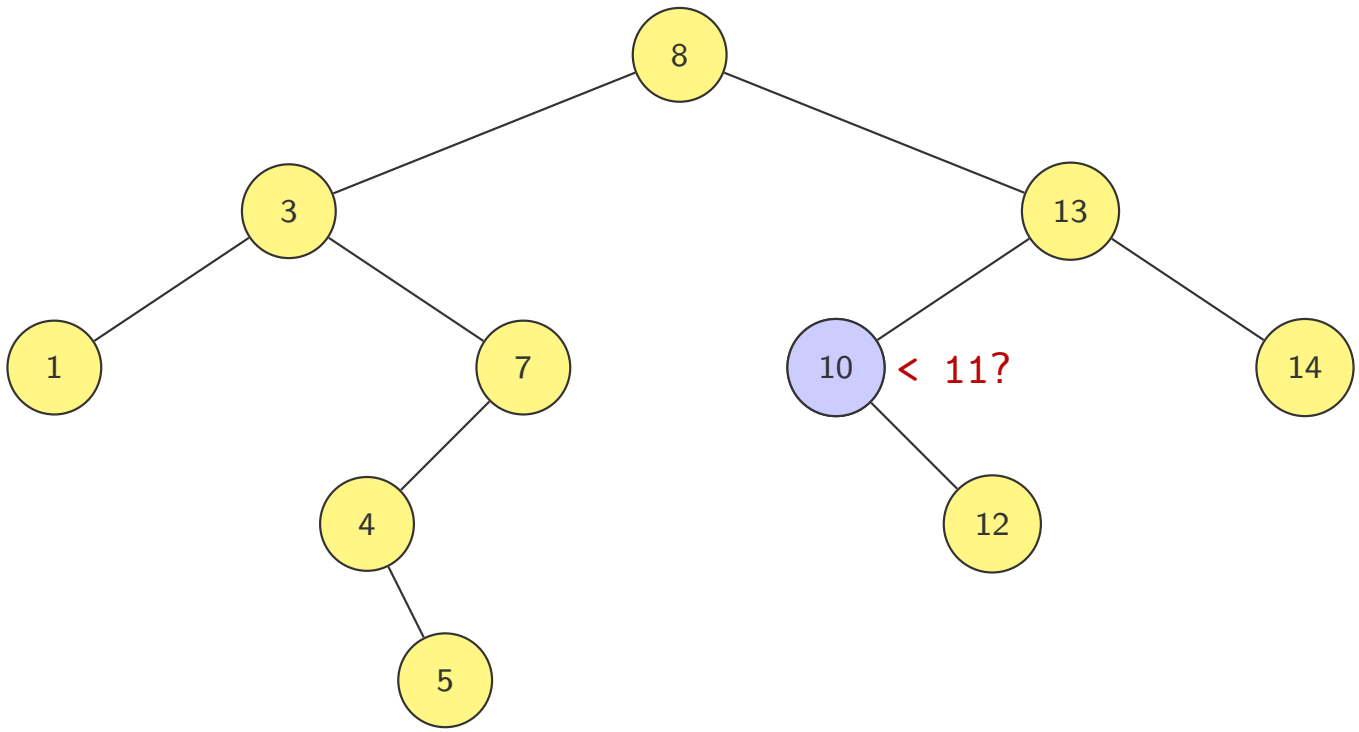


Inserindo um valor

Precisamos determinar onde inserir o valor:

- fazemos uma busca pelo valor
- e colocamos ele na posição onde deveria estar

Ex: Inserindo 11

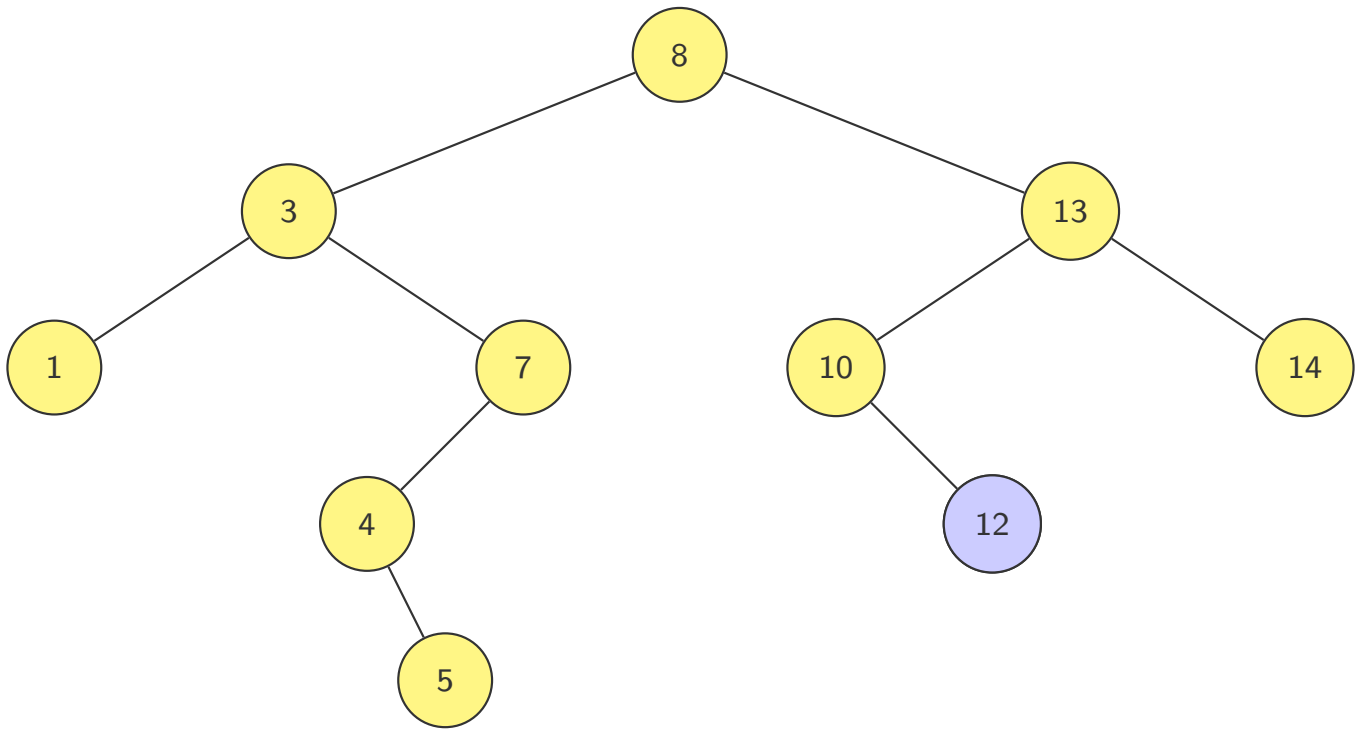


Inserindo um valor

Precisamos determinar onde inserir o valor:

- fazemos uma busca pelo valor
- e colocamos ele na posição onde deveria estar

Ex: Inserindo 11

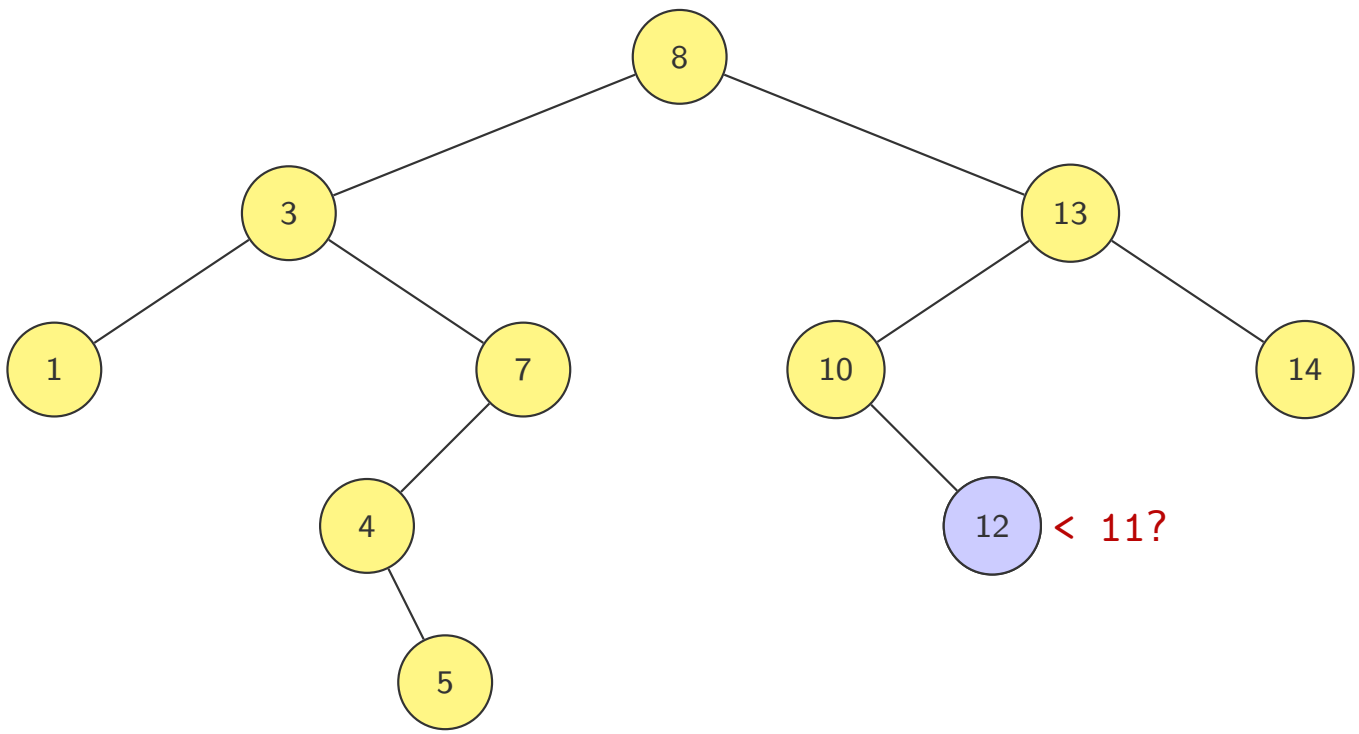


Inserindo um valor

Precisamos determinar onde inserir o valor:

- fazemos uma busca pelo valor
- e colocamos ele na posição onde deveria estar

Ex: Inserindo 11

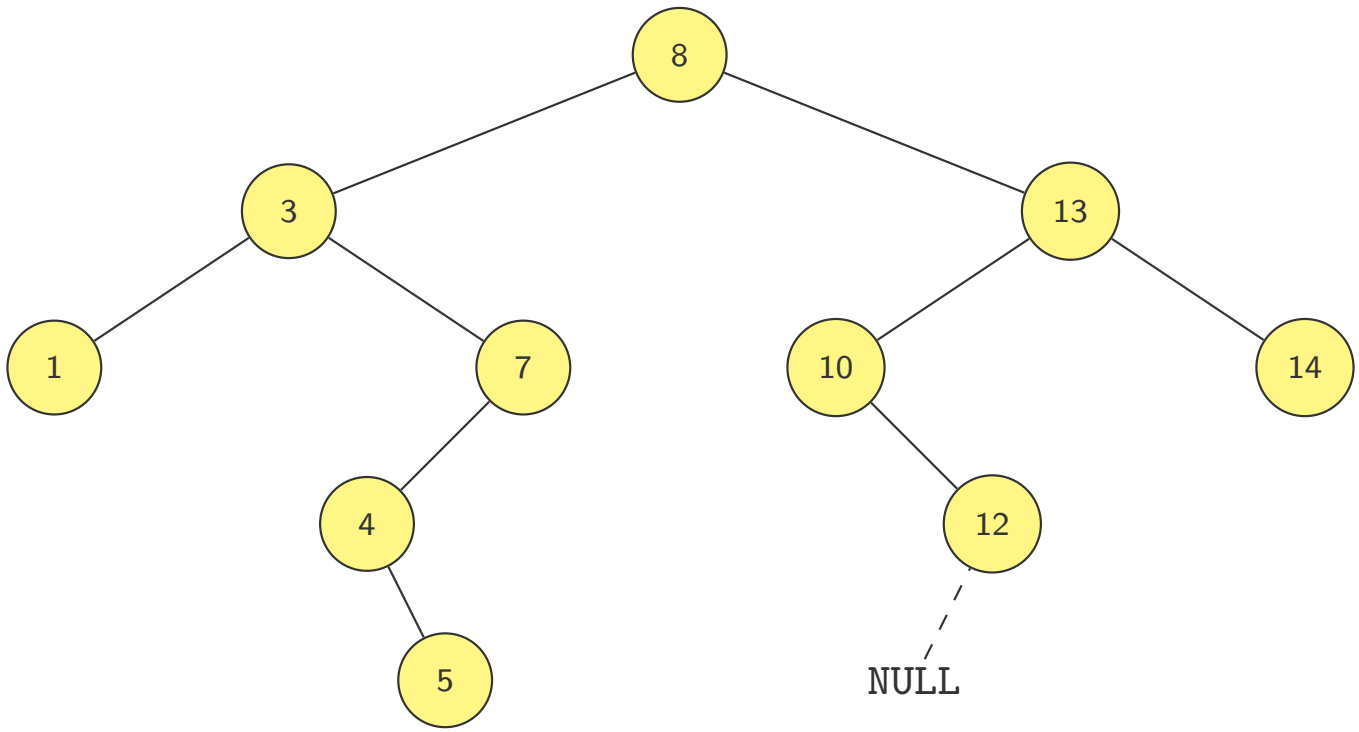


Inserindo um valor

Precisamos determinar onde inserir o valor:

- fazemos uma busca pelo valor
- e colocamos ele na posição onde deveria estar

Ex: Inserindo 11

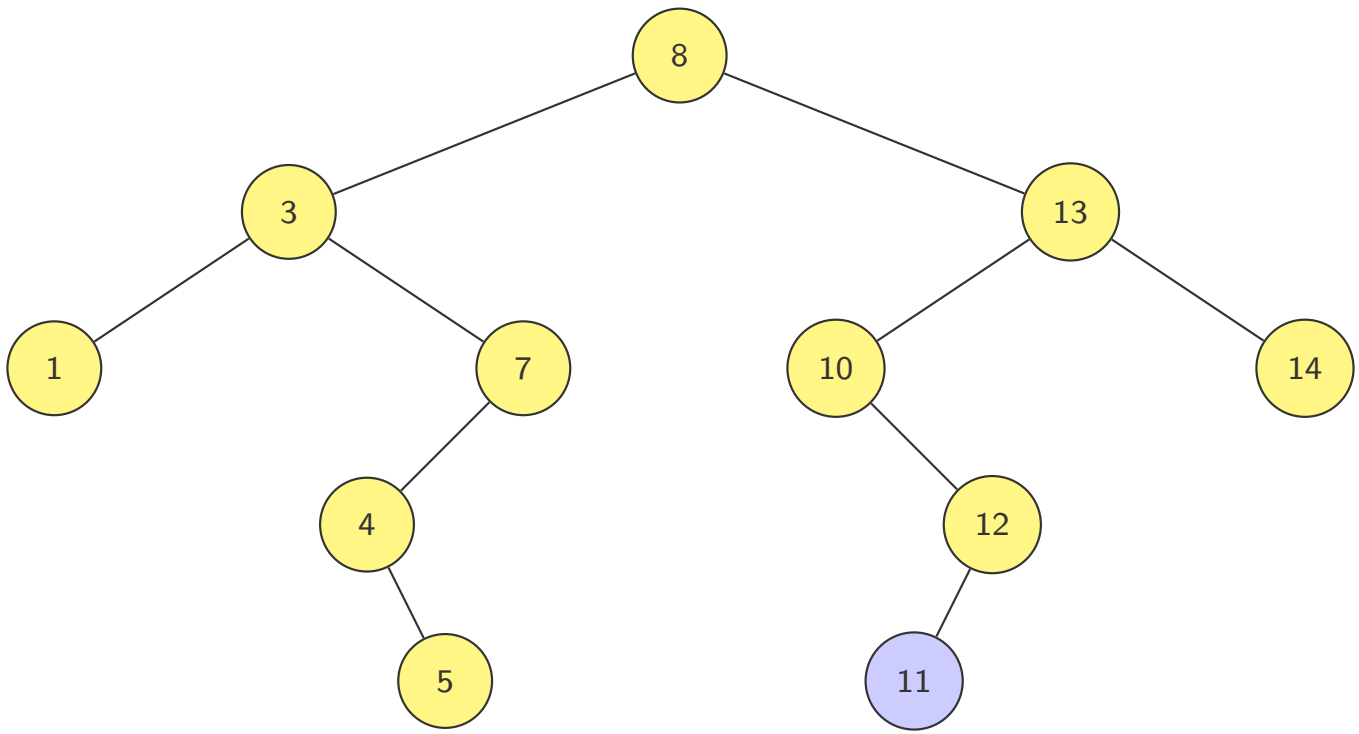


Inserindo um valor

Precisamos determinar onde inserir o valor:

- fazemos uma busca pelo valor
- e colocamos ele na posição onde deveria estar

Ex: Inserindo 11



Inserção — Implementação

Inserção — Implementação

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  // Esta funcao insere o no com chave key na arvore
2  // enraizada em node, somente se a chave key nao for repetida.
3  // Devolve um ponteiro para a raiz da nova arvore
4  // enraizada em node.
5  Node *BST::add(Node *node, int key) {
6      if(node == nullptr) { // Condicao de Parada
7          node = new Node(key, nullptr, nullptr);
8          return node;
9      }
10     // Casos Gerais
11     if(key > node->key)
12         node->right = add(node->right, key);
13     else if(key < node->key)
14         node->left = add(node->left, key);
15     return node;
16 }
```

Inserção — Implementação

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

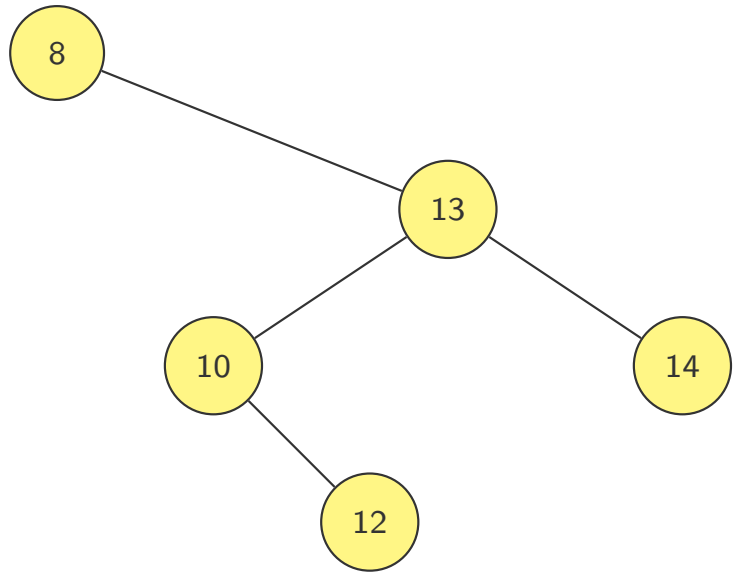
```
1  // Esta funcao insere o no com chave key na arvore
2  // enraizada em node, somente se a chave key nao for repetida.
3  // Devolve um ponteiro para a raiz da nova arvore
4  // enraizada em node.
5  Node *BST::add(Node *node, int key) {
6      if(node == nullptr) { // Condicao de Parada
7          node = new Node(key, nullptr, nullptr);
8          return node;
9      }
10     // Casos Gerais
11     if(key > node->key)
12         node->right = add(node->right, key);
13     else if(key < node->key)
14         node->left = add(node->left, key);
15     return node;
16 }
17
18 void BST::add(int key) {
19     root = add(root, key);
20 }
```

Mínimo da Árvore Binária de Busca

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

Mínimo da Árvore Binária de Busca

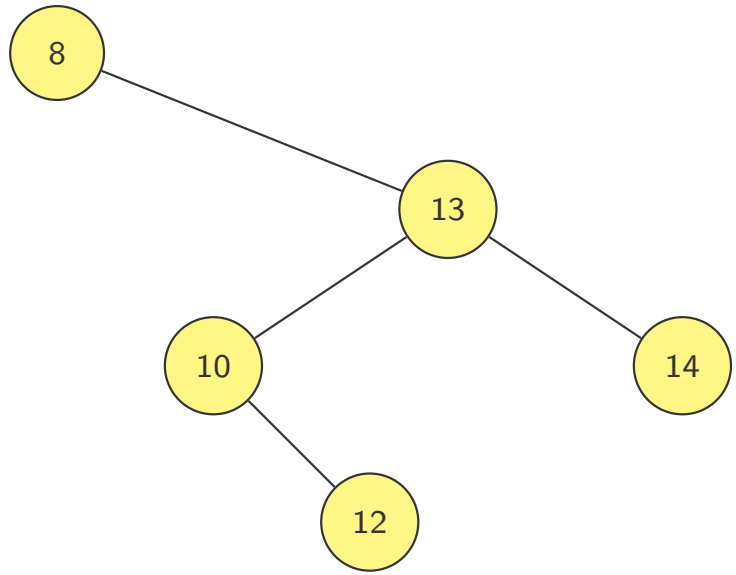
Onde está o nó com a menor chave de uma árvore?



Mínimo da Árvore Binária de Busca

Universidade Federal do Ceará
Campus Quixadá

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

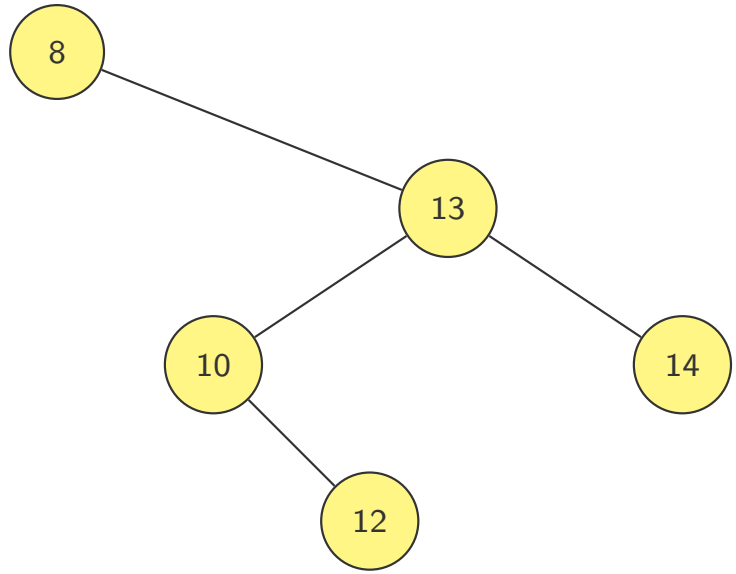


Quem é o mínimo para essa árvore?

Mínimo da Árvore Binária de Busca

Universidade Federal do Ceará
Campus Quixadá

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



Quem é o mínimo para essa árvore?

- É a própria raiz

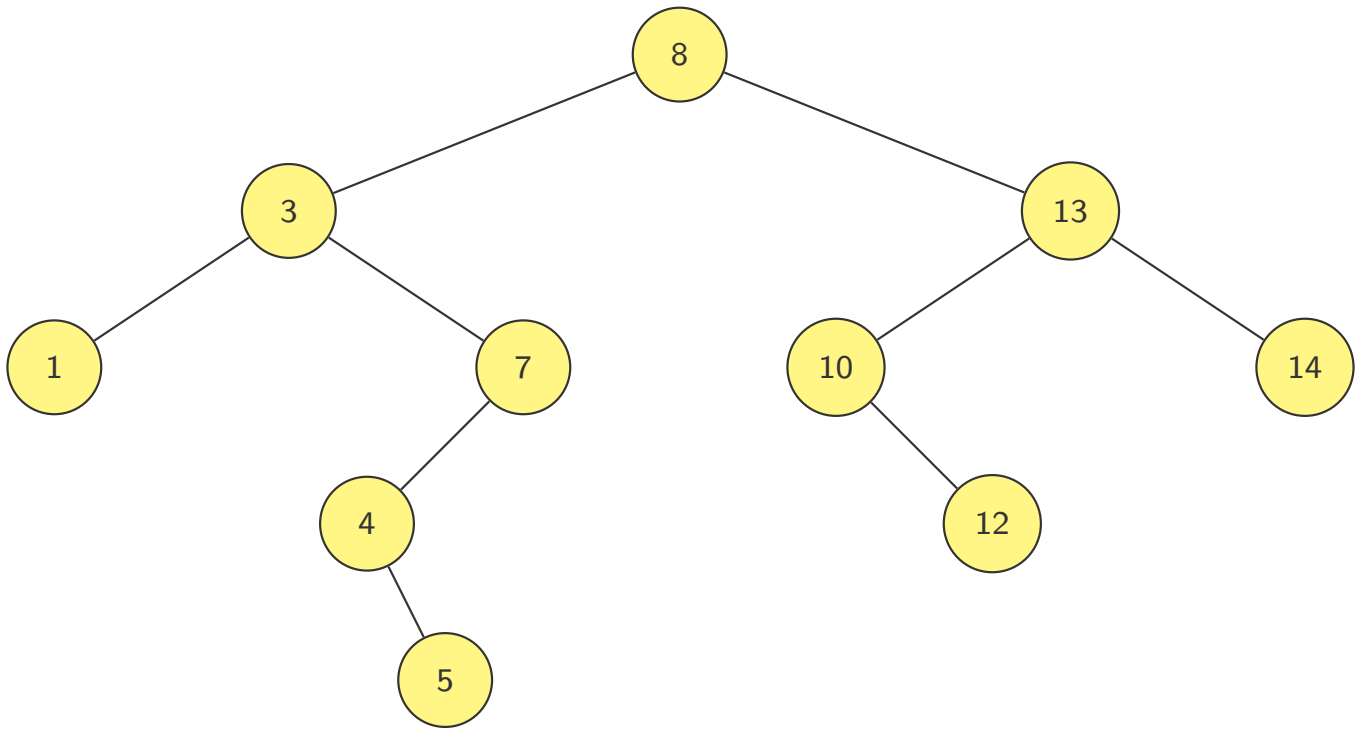
Mínimo da Árvore Binária de Busca

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

Mínimo da Árvore Binária de Busca

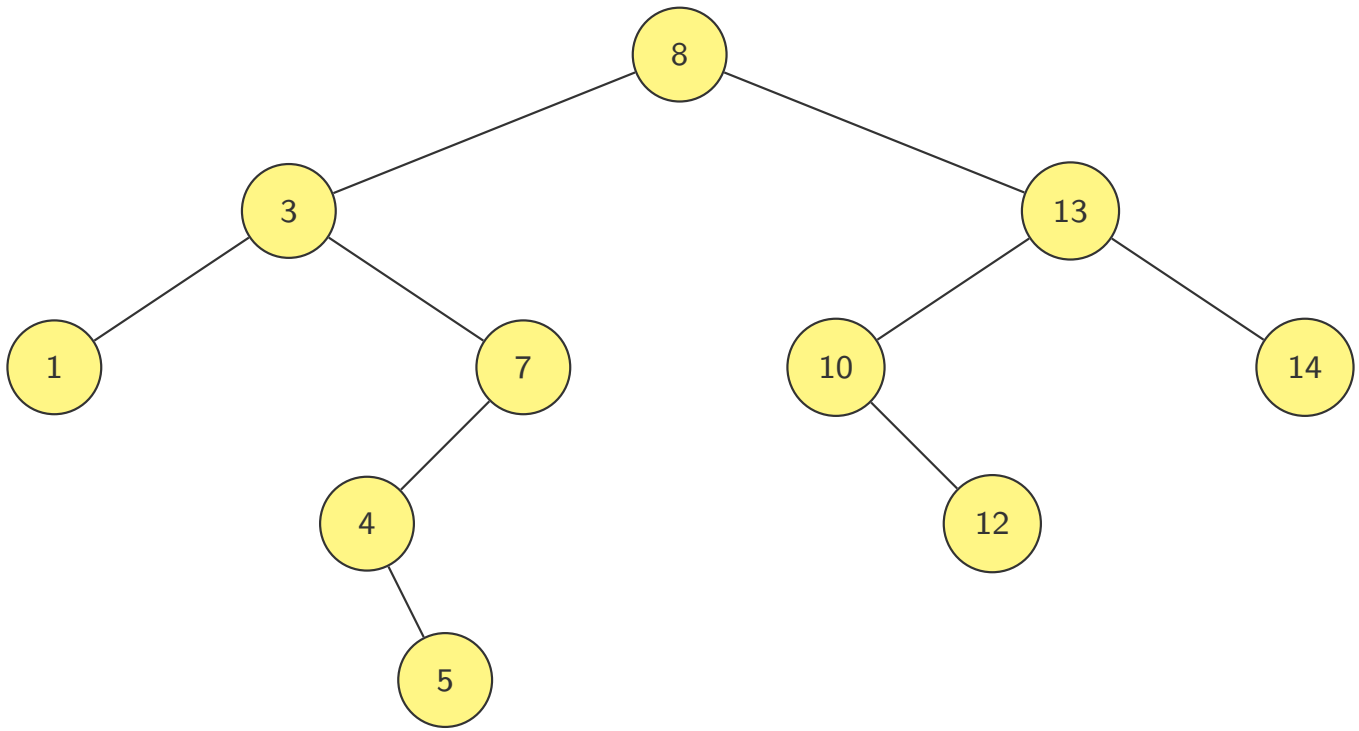
Universidade Federal do Ceará
Campus Quixadá

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



Mínimo da Árvore Binária de Busca

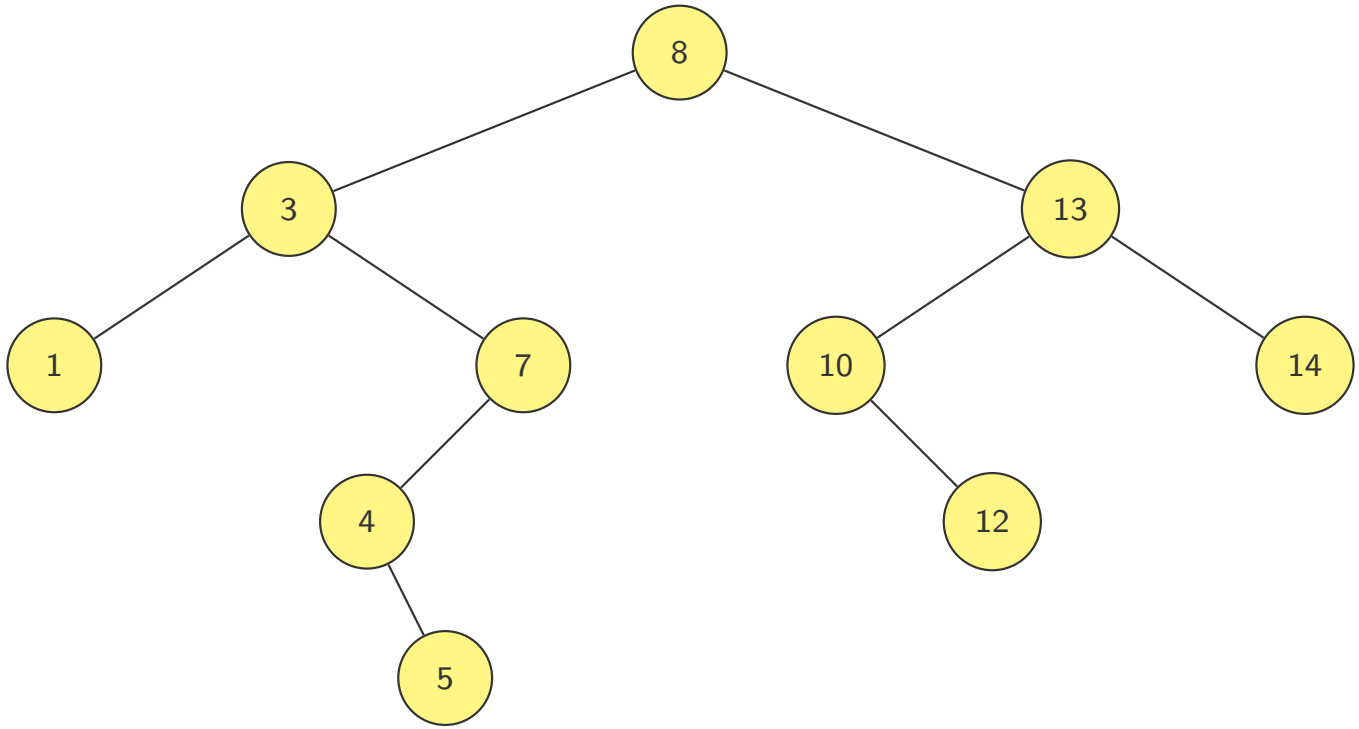
Onde está o nó com a menor chave de uma árvore?



Quem é o mínimo para essa árvore?

Mínimo da Árvore Binária de Busca

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 - Implementação

Mínimo - Implementação


```
1 // Recebe um ponteiro para a arvore enraizada em node e:
2 // (1) devolve o ponteiro para o no contendo o minimo; ou
3 // (2) devolve nullptr se a arvore for vazia.
4 Node *BST::minimum(Node *node) {
5     if(node != nullptr && node->left != nullptr)
6         return minimum(node->left);
7     else
8         return node;
9 }
```

Mínimo - Implementação

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1 // Recebe um ponteiro para a arvore enraizada em node e:
2 // (1) devolve o ponteiro para o no contendo o minimo; ou
3 // (2) devolve nullptr se a arvore for vazia.
4 Node *BST::minimum(Node *node) {
5     if(node != nullptr && node->left != nullptr)
6         return minimum(node->left);
7     else
8         return node;
9 }
```

```
1 // funcao publica. Devolve a chave minima da arvore
2 int BST::minimum() {
3     if(root == nullptr) // arvore vazia
4         throw std::runtime_error("erro: arvore vazia");
5     else
6         return minimum(root)->key;
7 }
```

Mínimo - Implementação


```
1 // Recebe um ponteiro para a arvore enraizada em node e:
2 // (1) devolve o ponteiro para o no contendo o minimo; ou
3 // (2) devolve nullptr se a arvore for vazia.
4 Node *BST::minimum(Node *node) {
5     if(node != nullptr && node->left != nullptr)
6         return minimum(node->left);
7     else
8         return node;
9 }
```

```
1 // funcao publica. Devolve a chave minima da arvore
2 int BST::minimum() {
3     if(root == nullptr) // arvore vazia
4         throw std::runtime_error("erro: arvore vazia");
5     else
6         return minimum(root)->key;
7 }
```

Para encontrar o máximo, basta fazer a operação simétrica (Tarefa)

Mínimo - Implementação

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1 // Recebe um ponteiro para a arvore enraizada em node e:
2 // (1) devolve o ponteiro para o no contendo o minimo; ou
3 // (2) devolve nullptr se a arvore for vazia.
4 Node *BST::minimum(Node *node) {
5     if(node != nullptr && node->left != nullptr)
6         return minimum(node->left);
7     else
8         return node;
9 }
```

```
1 // funcao publica. Devolve a chave minima da arvore
2 int BST::minimum() {
3     if(root == nullptr) // arvore vazia
4         throw std::runtime_error("erro: arvore vazia");
5     else
6         return minimum(root)->key;
7 }
```

Para encontrar o máximo, basta fazer a operação simétrica (Tarefa)

- Se a subárvore direita existir, é o seu máximo

Mínimo - Implementação


```
1 // Recebe um ponteiro para a arvore enraizada em node e:
2 // (1) devolve o ponteiro para o no contendo o minimo; ou
3 // (2) devolve nullptr se a arvore for vazia.
4 Node *BST::minimum(Node *node) {
5     if(node != nullptr && node->left != nullptr)
6         return minimum(node->left);
7     else
8         return node;
9 }
```

```
1 // funcao publica. Devolve a chave minima da arvore
2 int BST::minimum() {
3     if(root == nullptr) // arvore vazia
4         throw std::runtime_error("erro: arvore vazia");
5     else
6         return minimum(root)->key;
7 }
```

Para encontrar o máximo, basta fazer a operação simétrica (Tarefa)

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

Sucessor

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

Sucessor

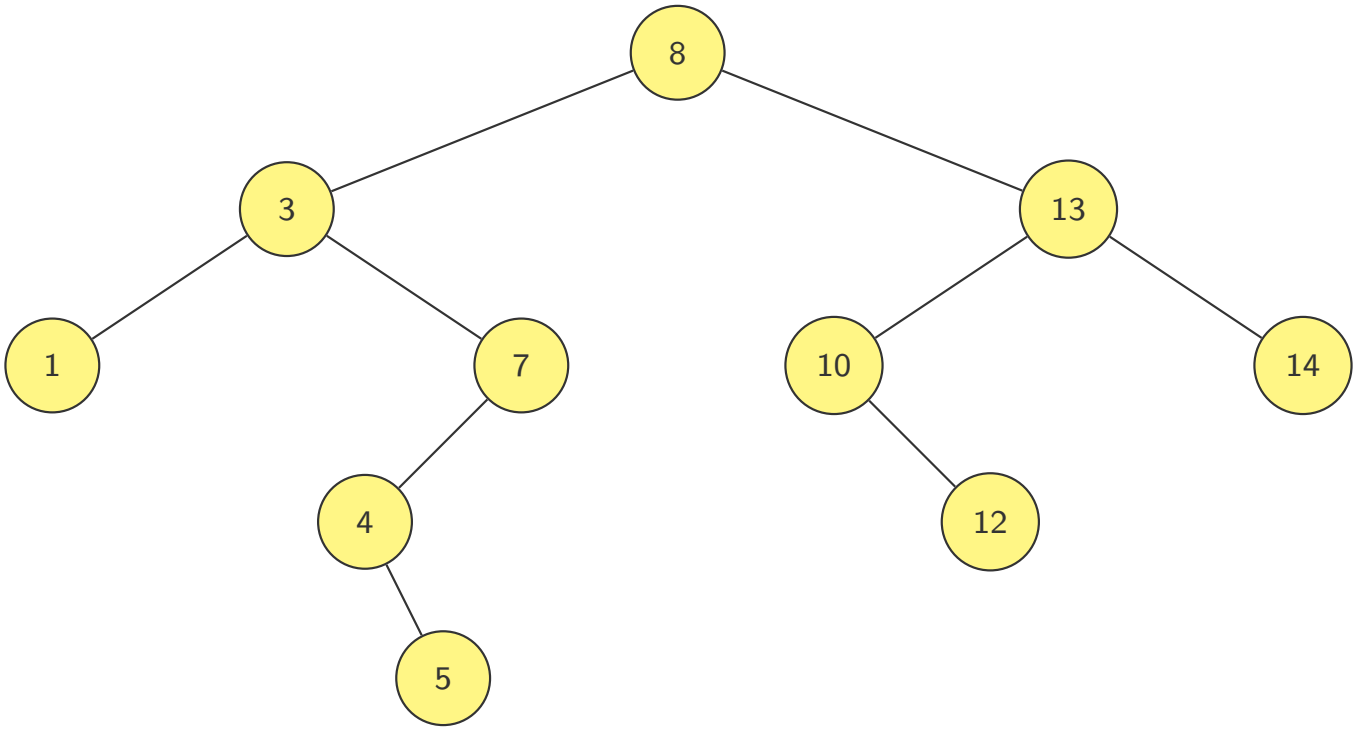
Dado um nó da árvore, onde está o seu sucessor?

- O sucessor é o próximo nó na ordenação

Sucessor

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

- O sucessor é o próximo nó na ordenação



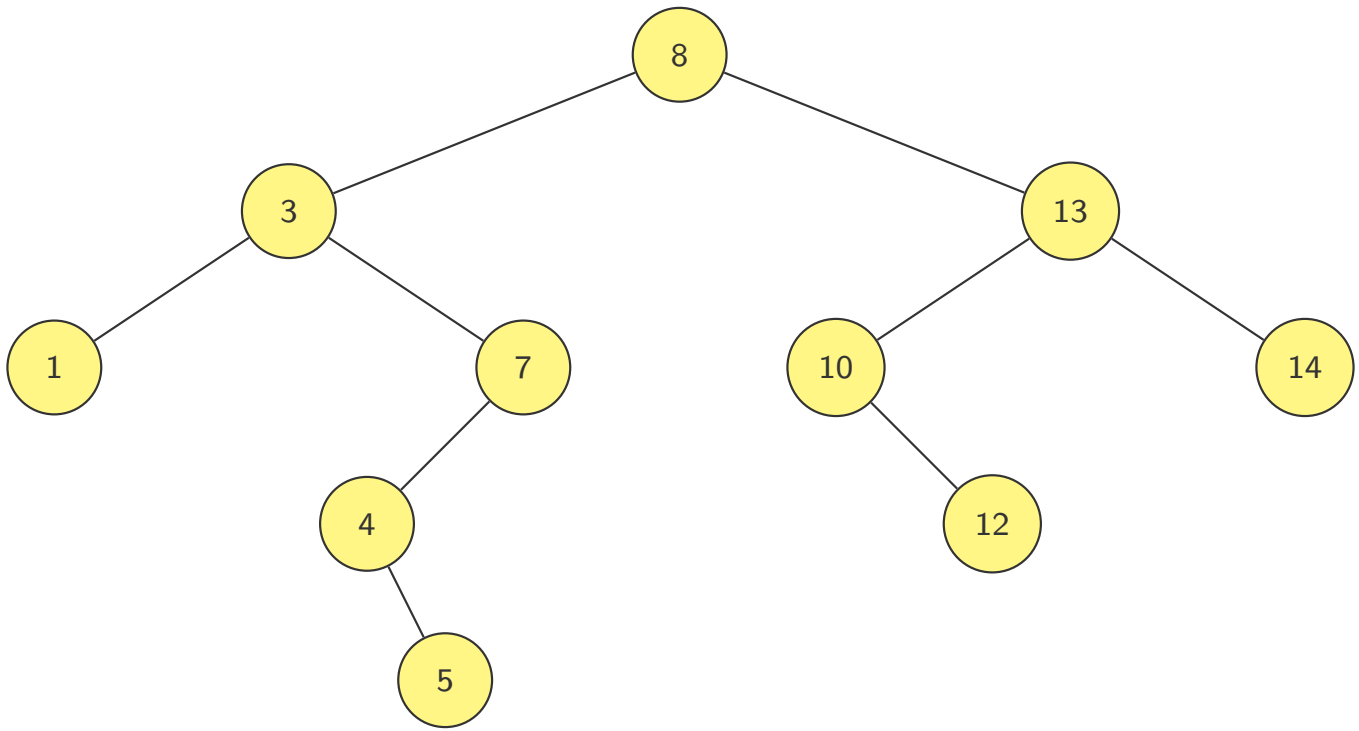
Quem é o sucessor de 3?

Sucessor

Universidade Federal do Ceará
Campus Quixadá

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

- O sucessor é o próximo nó na ordenação



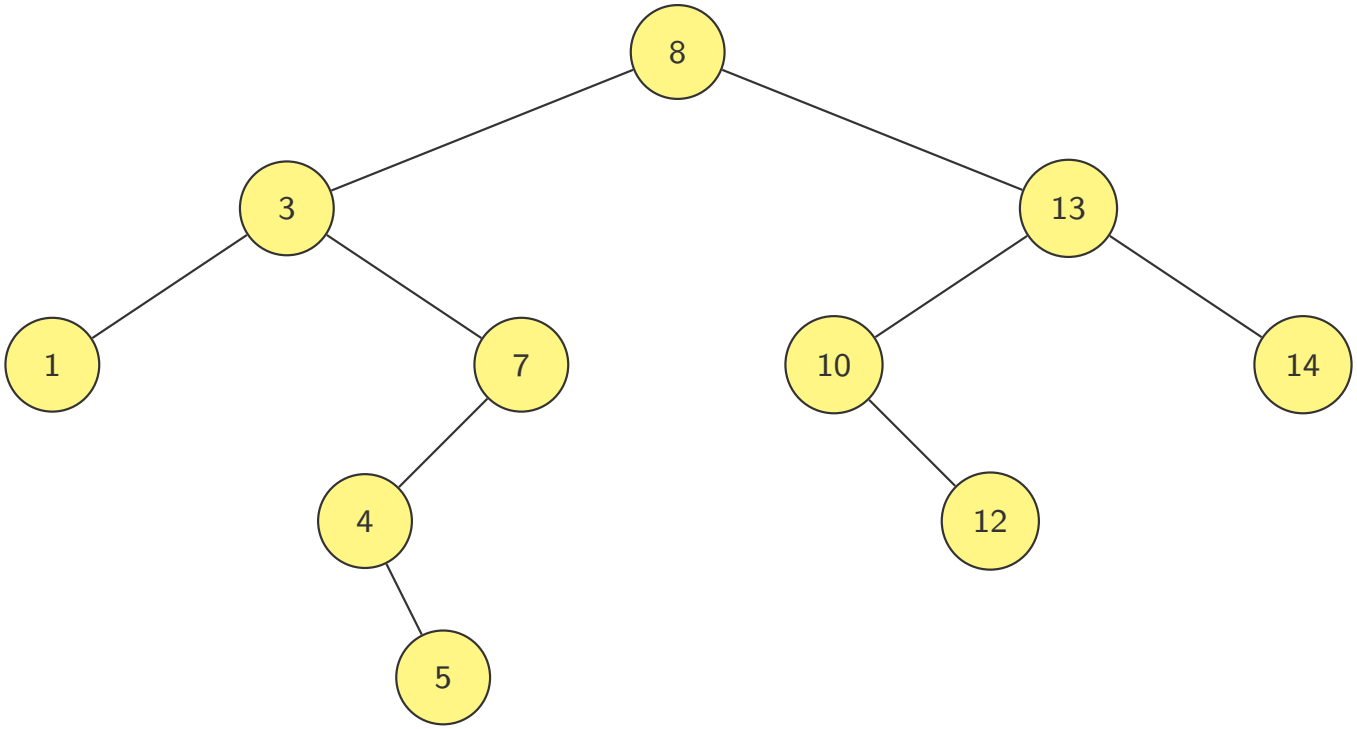
Quem é o sucessor de 3?

- É o mínimo da subárvore direita de 3

Sucessor

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

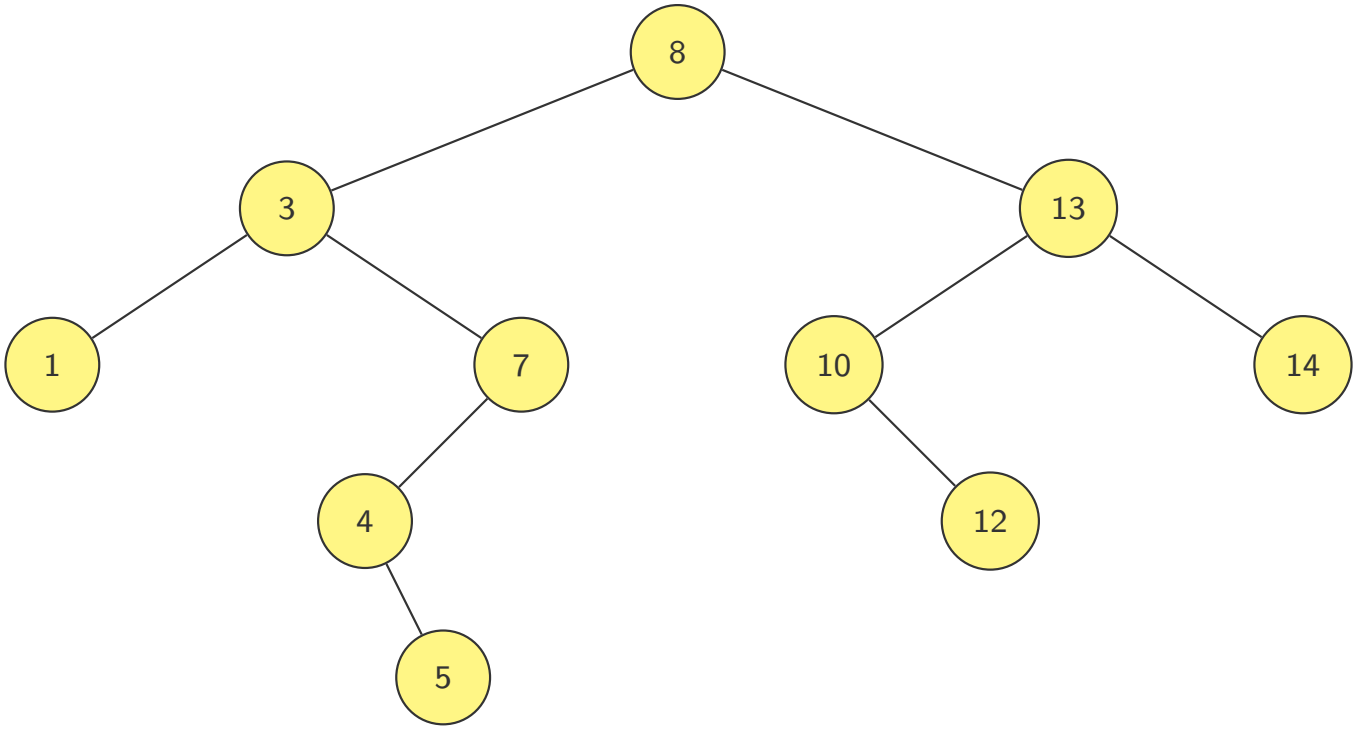
- O sucessor é o próximo nó na ordenação



Sucessor

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

- O sucessor é o próximo nó na ordenação

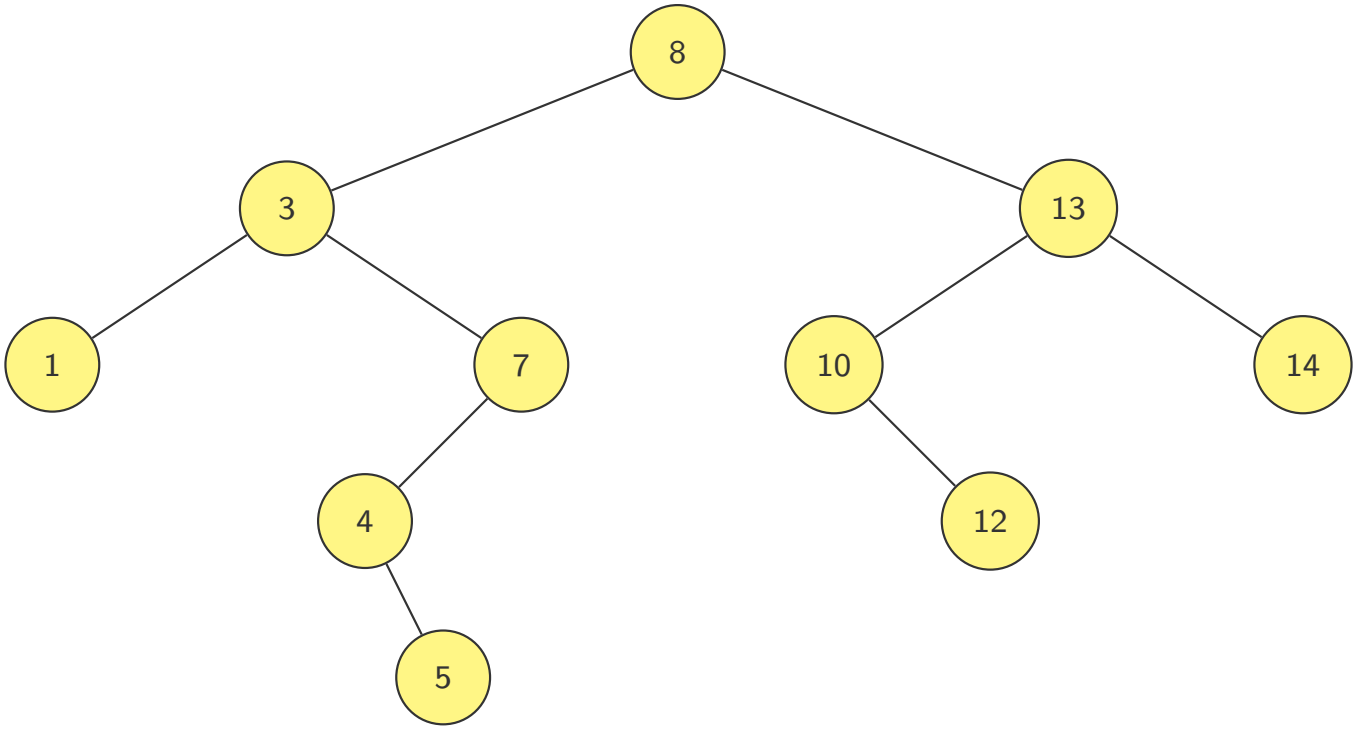


Quem é o sucessor de 7?

Sucessor

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

- O sucessor é o próximo nó na ordenação



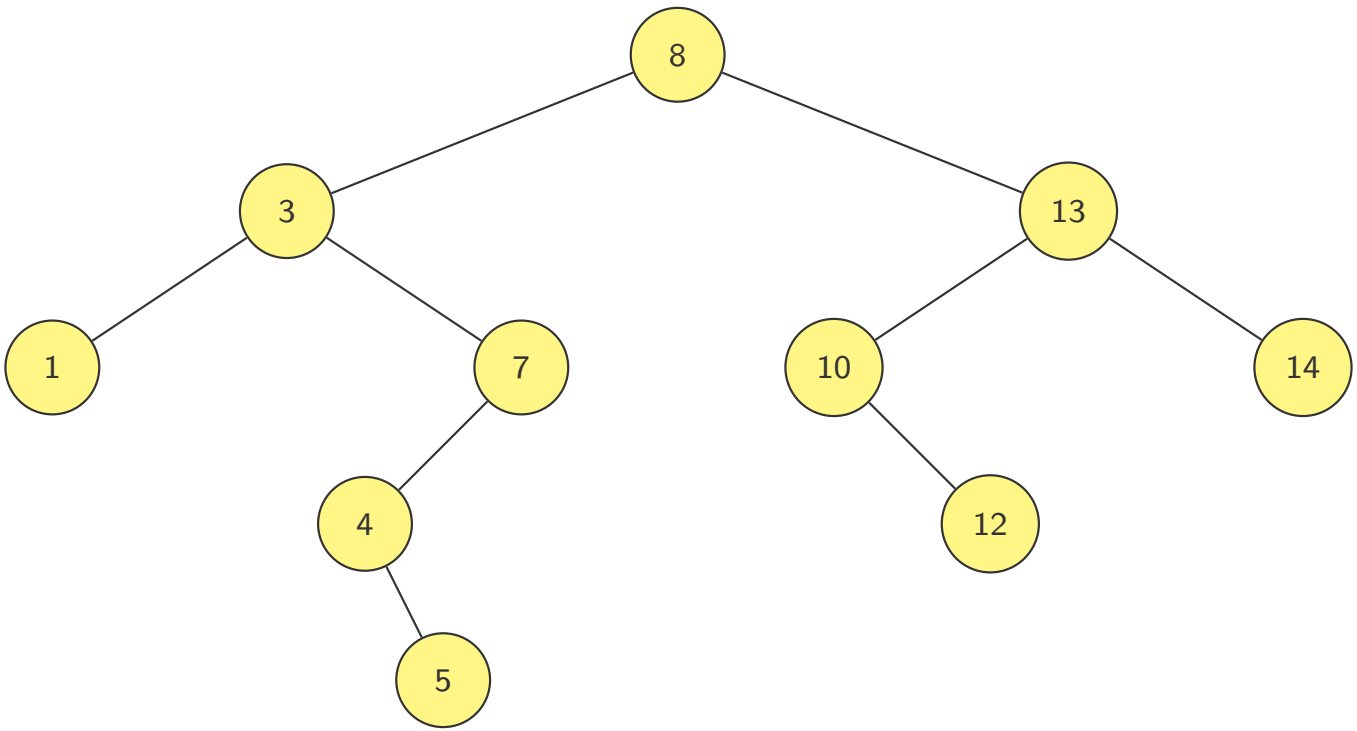
Quem é o sucessor de 7?

- É primeiro ancestral à direita

Sucessor

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

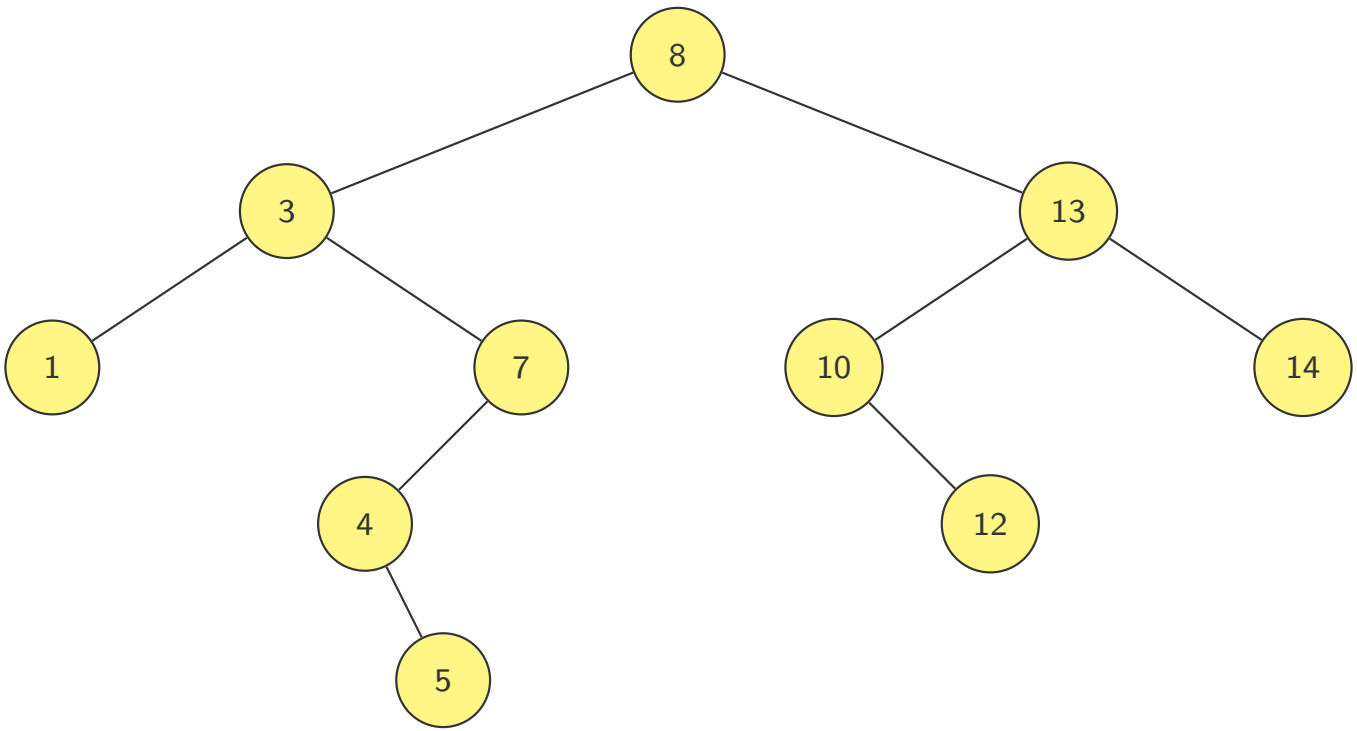
- O sucessor é o próximo nó na ordenação



Sucessor

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

- O sucessor é o próximo nó na ordenação



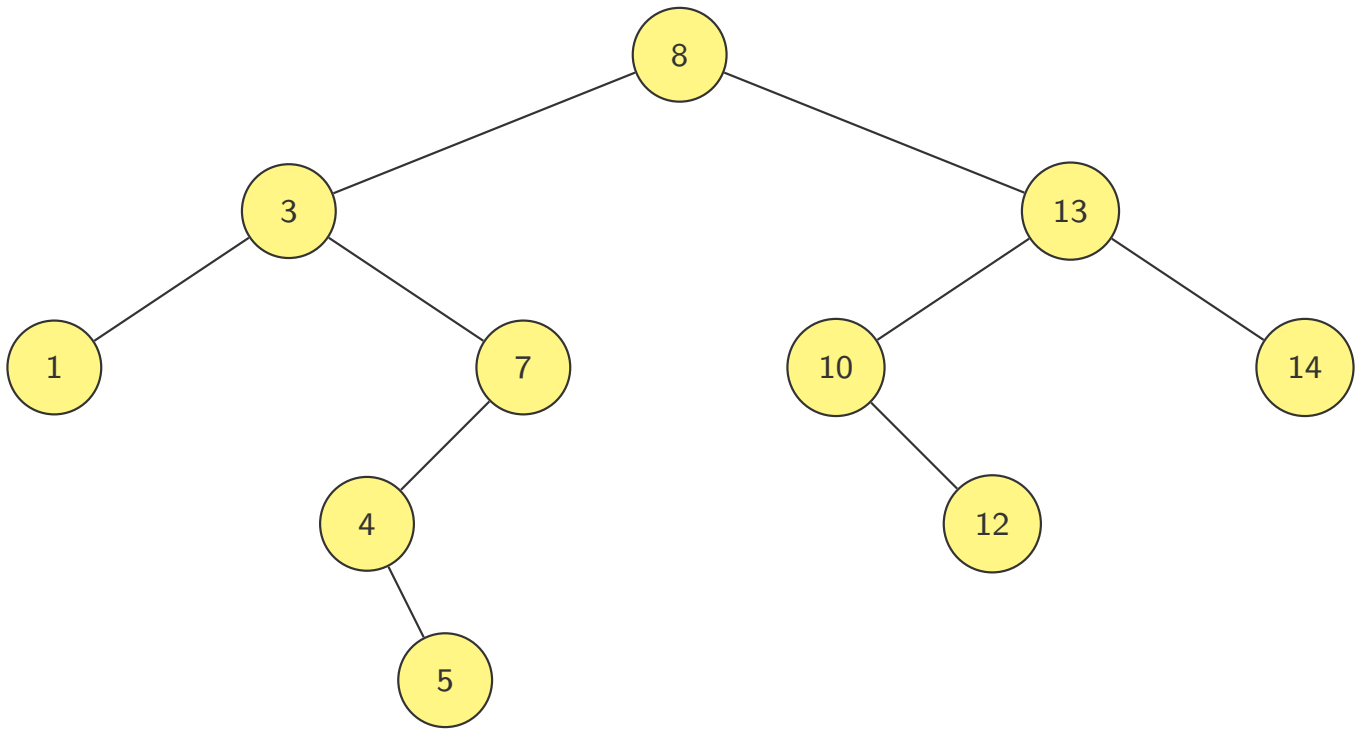
Quem é o sucessor de 14?

Sucessor

Universidade Federal do Ceará
Campus Quixadá

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

Implementação da função pública:

```
1  int BST::successor(int k) {
2      Node *temp = search(root, k);
3      if(temp != nullptr) {
4          Node *succ = sucessor(temp, root);
5          if(succ != nullptr)
6              return succ->key;
7          else
8              throw std::runtime_error("nao existe sucessor");
9      }
10     else throw std::runtime_error("chave inexistente");
11 }
```

Sucessor - Implementação

Implementação da função pública:

```
1  int BST::successor(int k) {
2      Node *temp = search(root, k);
3      if(temp != nullptr) {
4          Node *succ = sucessor(temp, root);
5          if(succ != nullptr)
6              return succ->key;
7          else
8              throw std::runtime_error("nao existe sucessor");
9      }
10     else throw std::runtime_error("chave inexistente");
11 }
```

Sucessor - Implementação


```
1  // Devolve o ponteiro para o no sucessor do no x
2  // passado como parametro. A funcao tambem recebe
3  // como parametro a raiz da arvore.
4  Node *BST::sucessor(Node *x, Node *raiz) {
5      if(x == nullptr || raiz == nullptr)
6          return nullptr;
7      else if(x->right != nullptr)
8          return minimum(x->right);
9      else
10         return ancestral_sucessor(x, raiz);
11 }
```

Sucessor - Implementação


```
1  // Devolve o ponteiro para o no sucessor do no x
2  // passado como parametro. A funcao tambem recebe
3  // como parametro a raiz da arvore.
4  Node *BST::sucessor(Node *x, Node *raiz) {
5      if(x == nullptr || raiz == nullptr)
6          return nullptr;
7      else if(x->right != nullptr)
8          return minimum(x->right);
9      else
10         return ancestral_sucessor(x, raiz);
11 }
```

- Exercício para casa: Implementar a função:

  `Node *ancestral_sucessor(Node *x, Node *raiz)`

  Ela recebe o nó `x`, a raiz da árvore e, então, retorna o ancestral de `x` que é também seu sucessor.

Antecessor - Implementação

Exercício para casa:

- A implementação das funções:
  int predecessor(int k)
  `Node* predecessor(Node *x, Node* raiz)`
  Node *ancestral_predecessor(Node *x, Node* raiz)
  são simétricas às do sucessor. Implemente-as também.

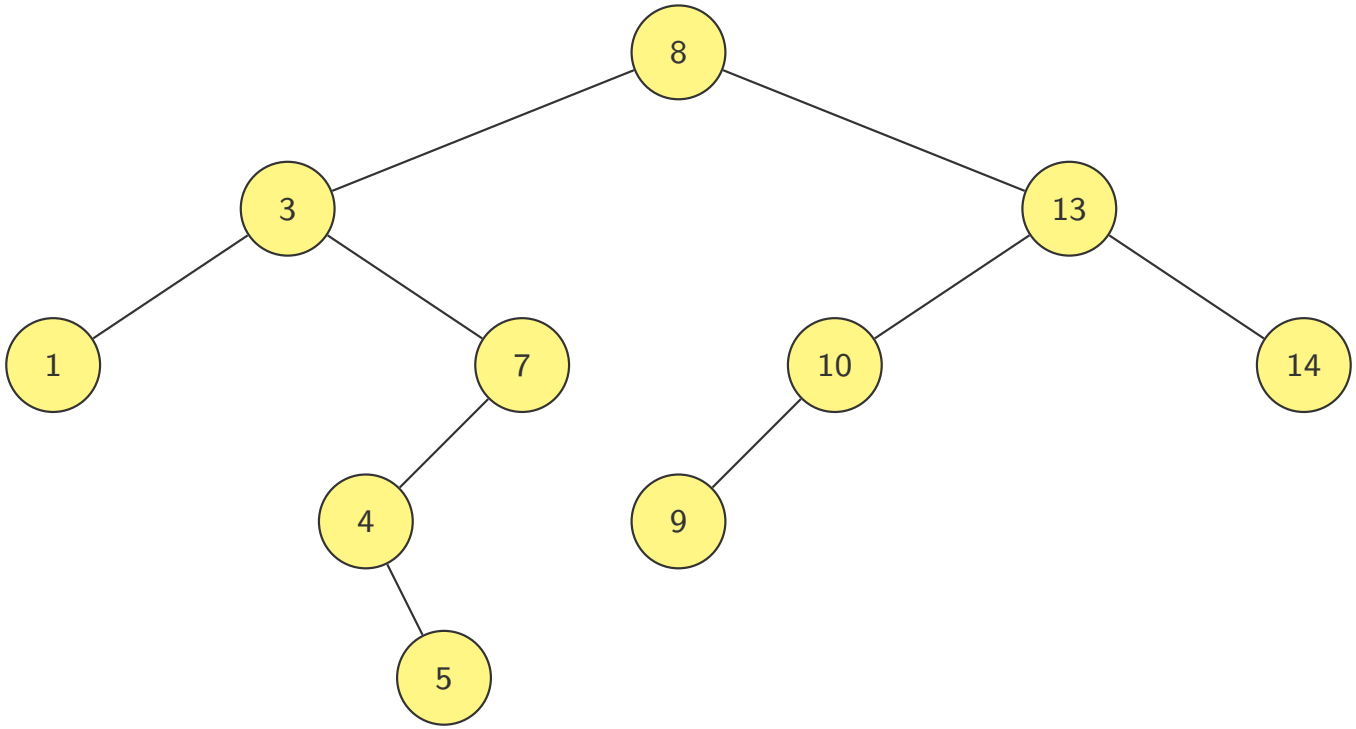
Remoção

Remoção

- Considere o problema de remover um nó de uma árvore binária de busca de tal forma que a árvore resultante continue de busca.
- Primeiro, precisamos fazer uma busca pelo nó a ser removido.
- Uma vez encontrado o nó, quais dificuldades podem surgir que dificultam a simples remoção do nó?

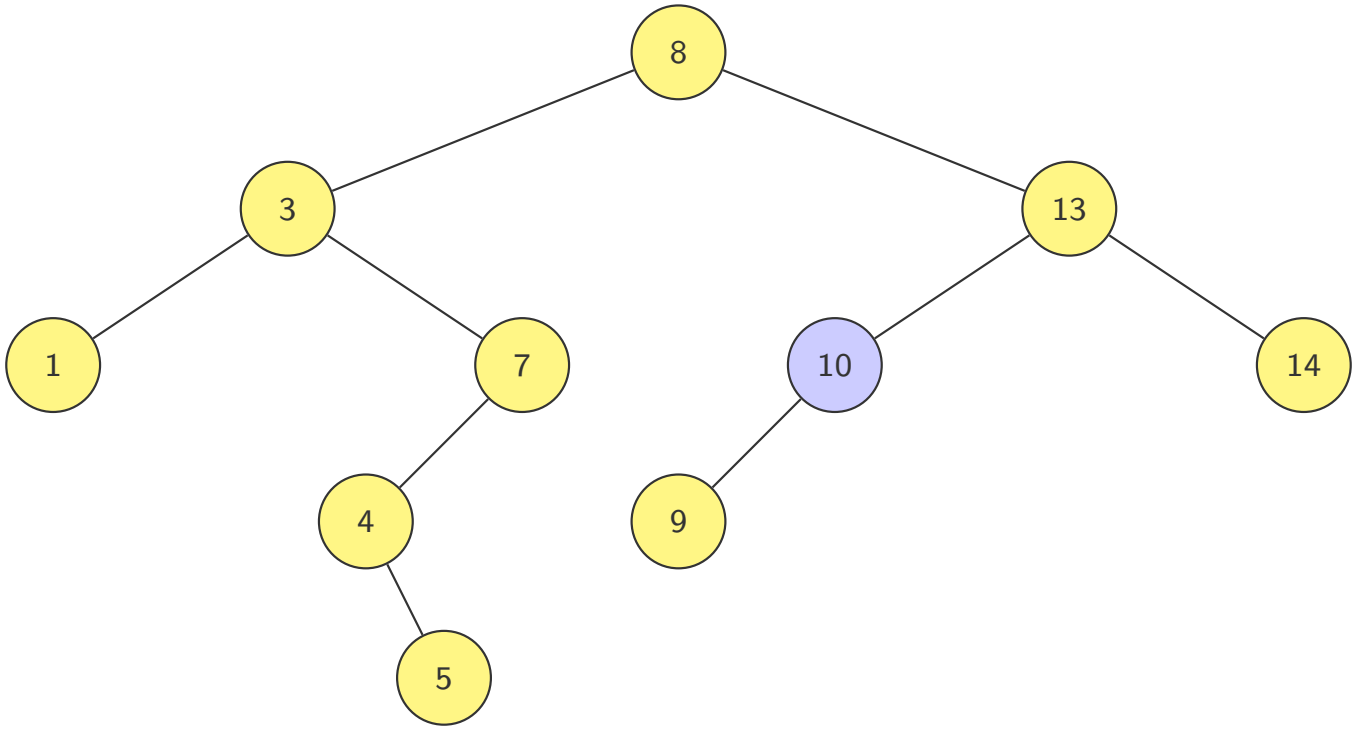
Remoção

Exemplo: queremos remover a chave 10



Remoção

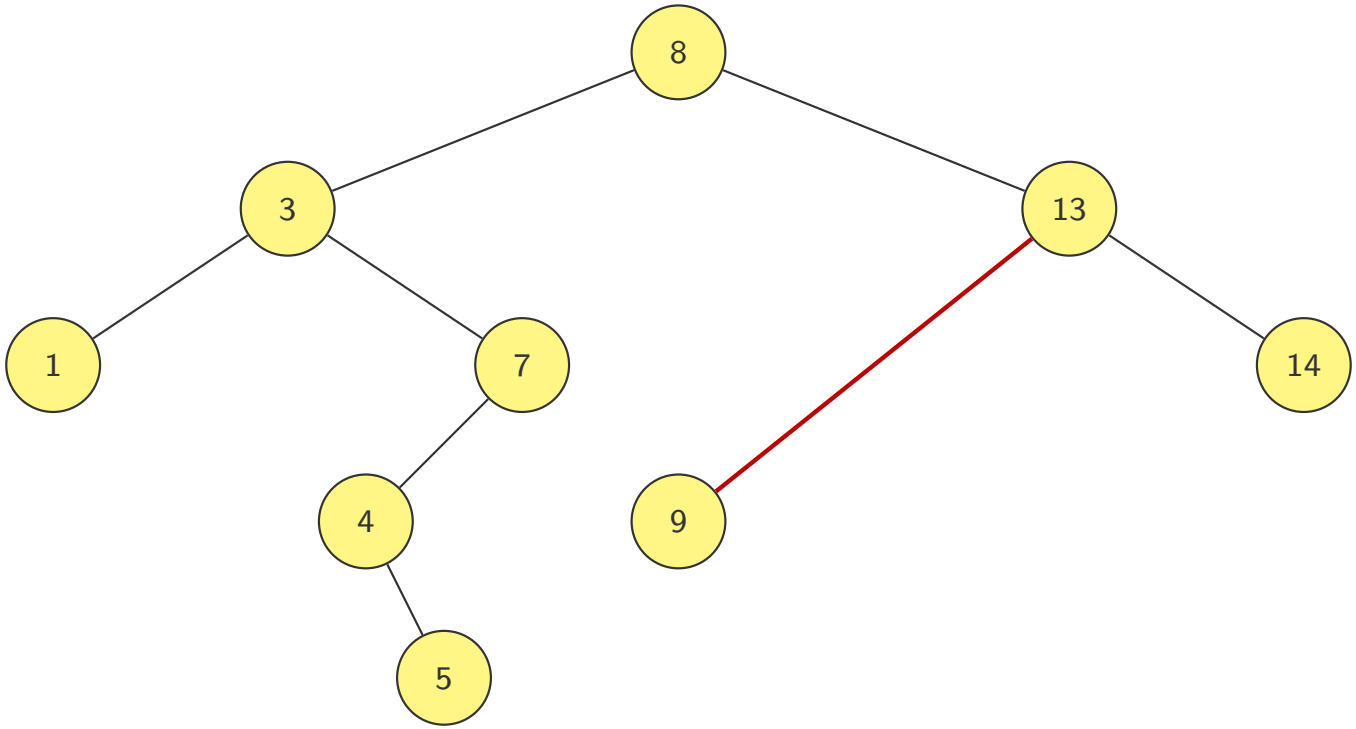
Exemplo: queremos remover a chave 10



- O nó x a ser removido pode ter exatamente um filho.

Remoção

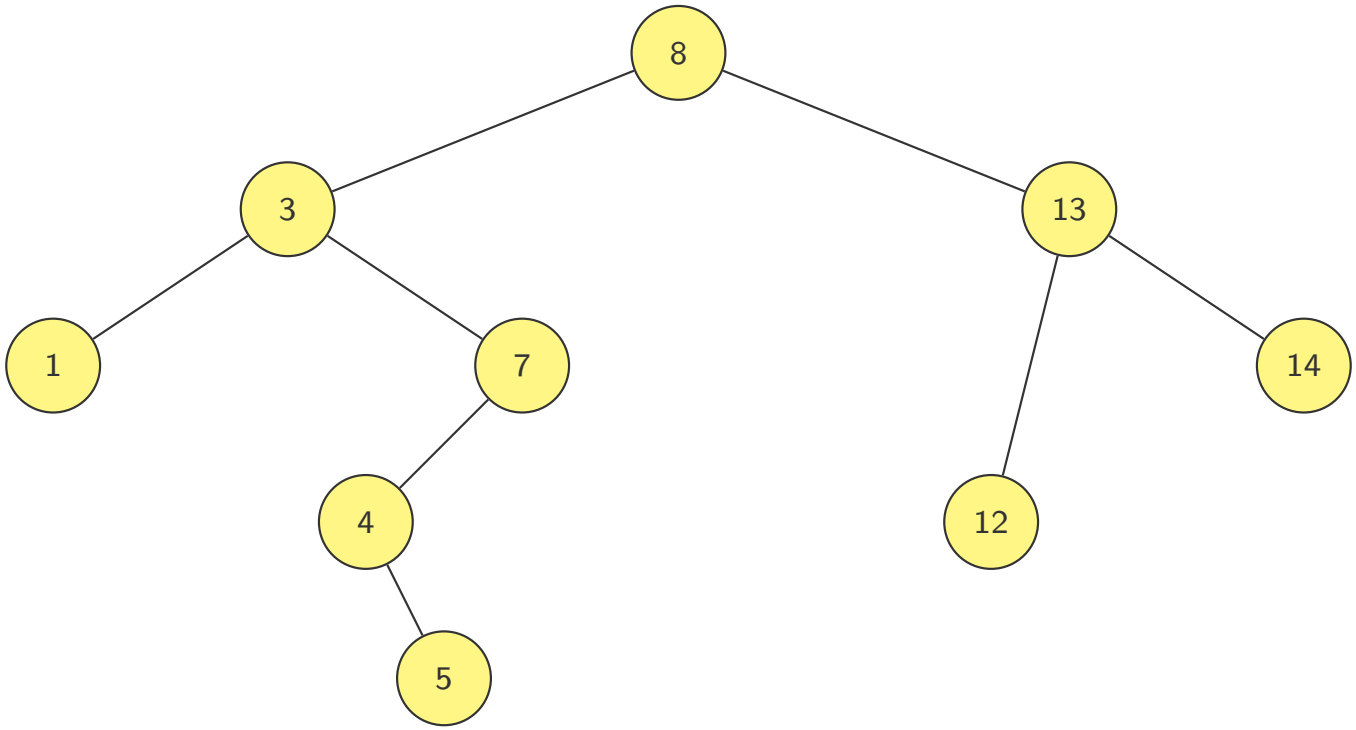
Exemplo: queremos remover a chave 10



- O nó x a ser removido pode ter exatamente um filho.
- Neste caso, fazemos o único filho de x ser filho do seu pai e depois removemos o nó x.

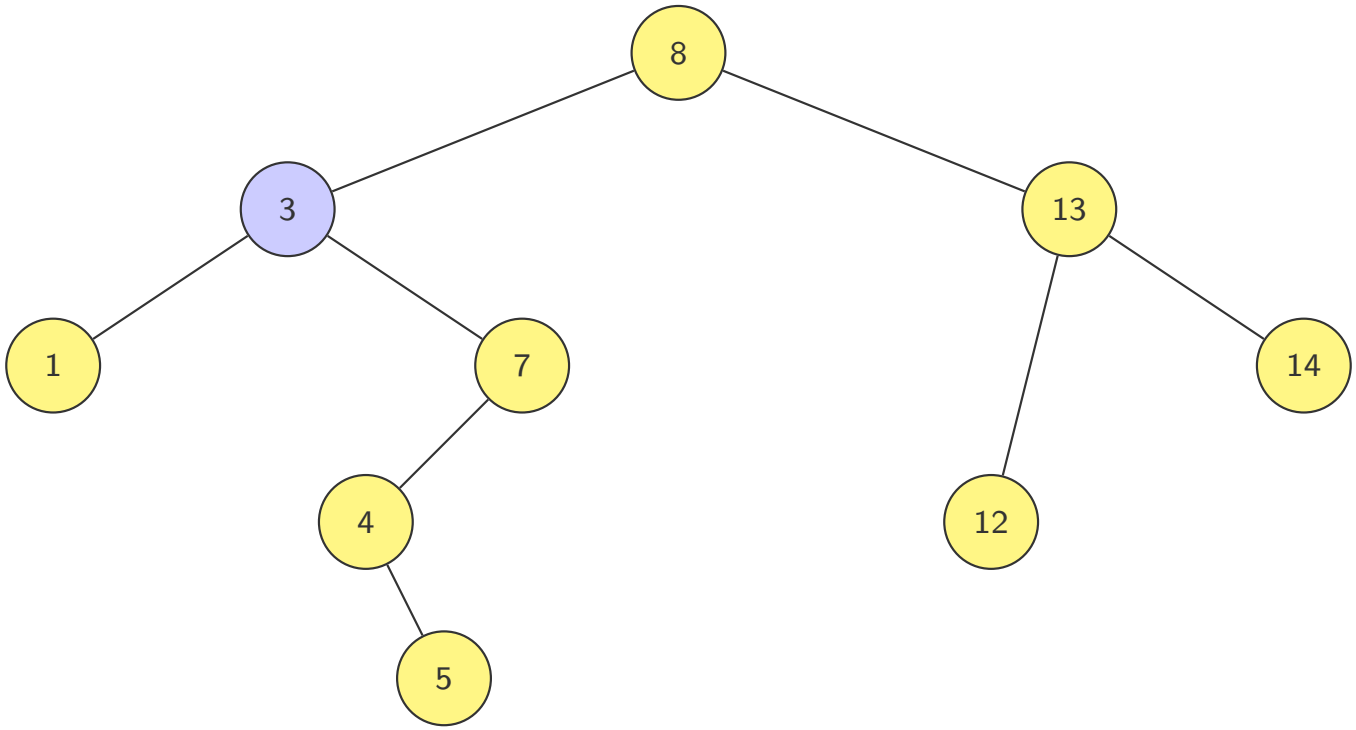
Remoção

Exemplo: removendo a chave 3



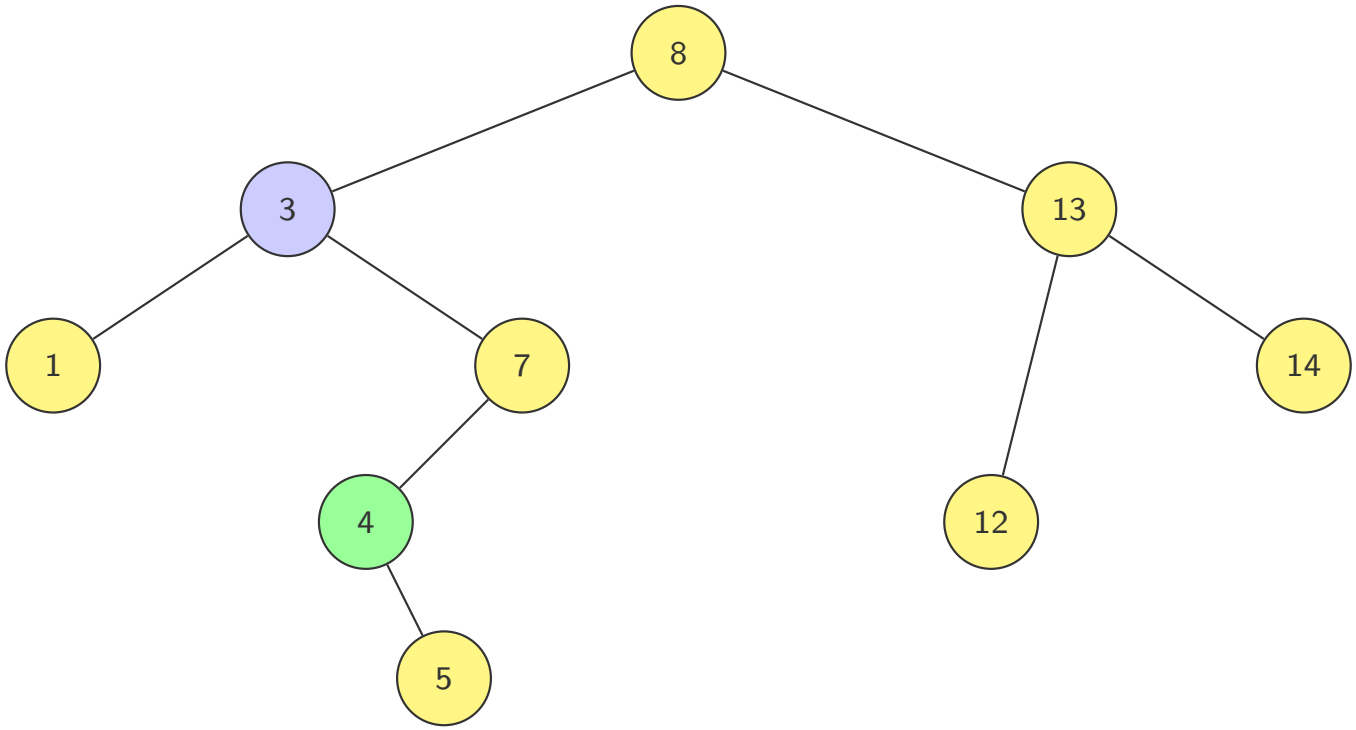
Remoção

Exemplo: removendo a chave 3



Remoção

Exemplo: removendo a chave 3

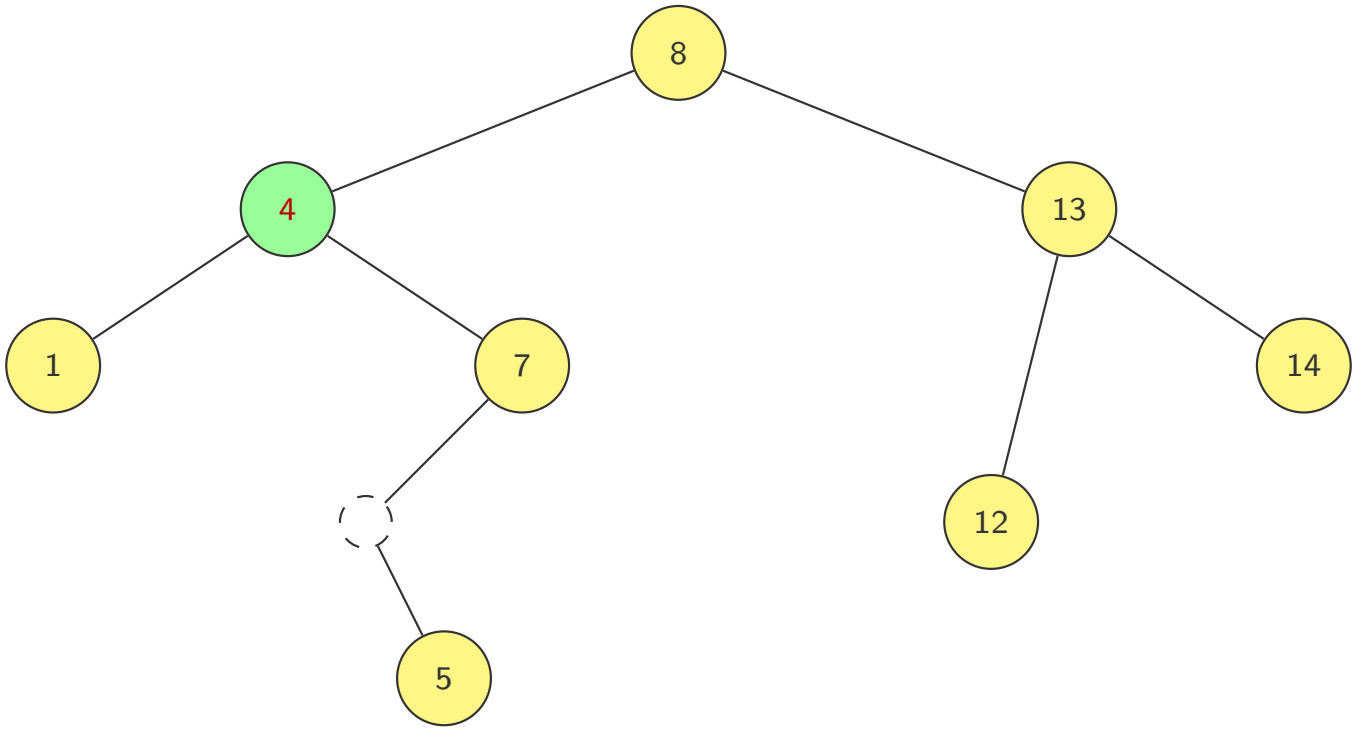


Podemos colocar o sucessor de 3 em seu lugar

- Isso mantém a propriedade da árvore binária de busca

Remoção

Exemplo: removendo a chave 3

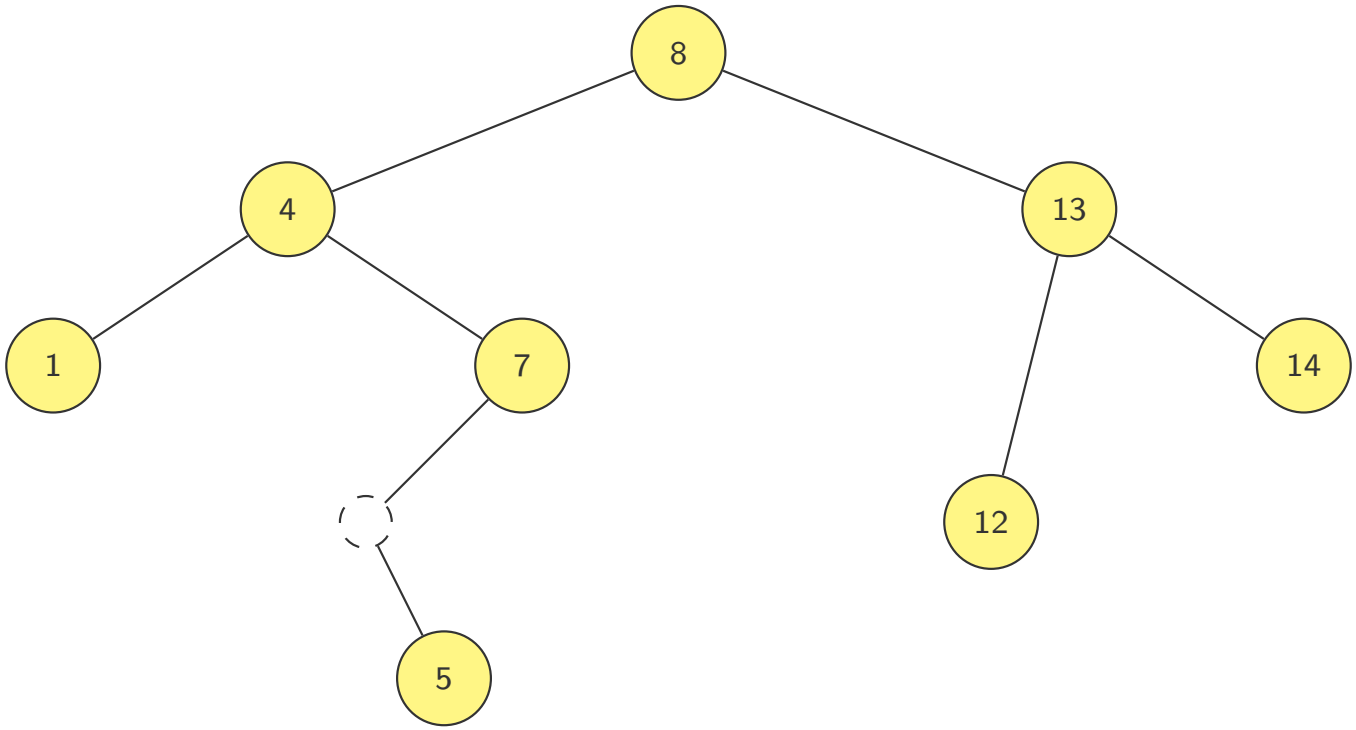


Podemos colocar o sucessor de 3 em seu lugar

- Isso mantém a propriedade da árvore binária de busca

Remoção

Exemplo: removendo a chave 3



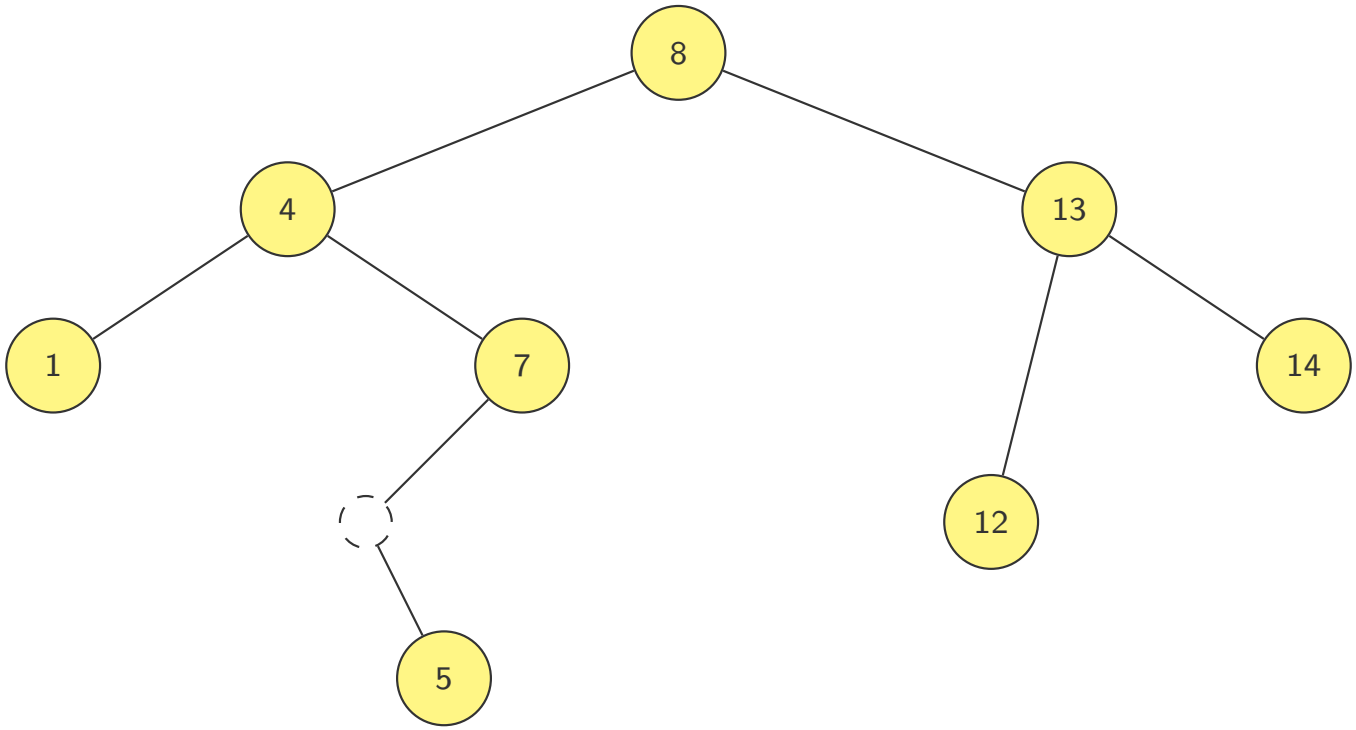
Podemos colocar o sucessor de 3 em seu lugar

- Isso mantém a propriedade da árvore binária de busca

E agora colocamos o filho direito do sucessor no seu lugar

Remoção

Exemplo: removendo a chave 3



Podemos colocar o sucessor de 3 em seu lugar

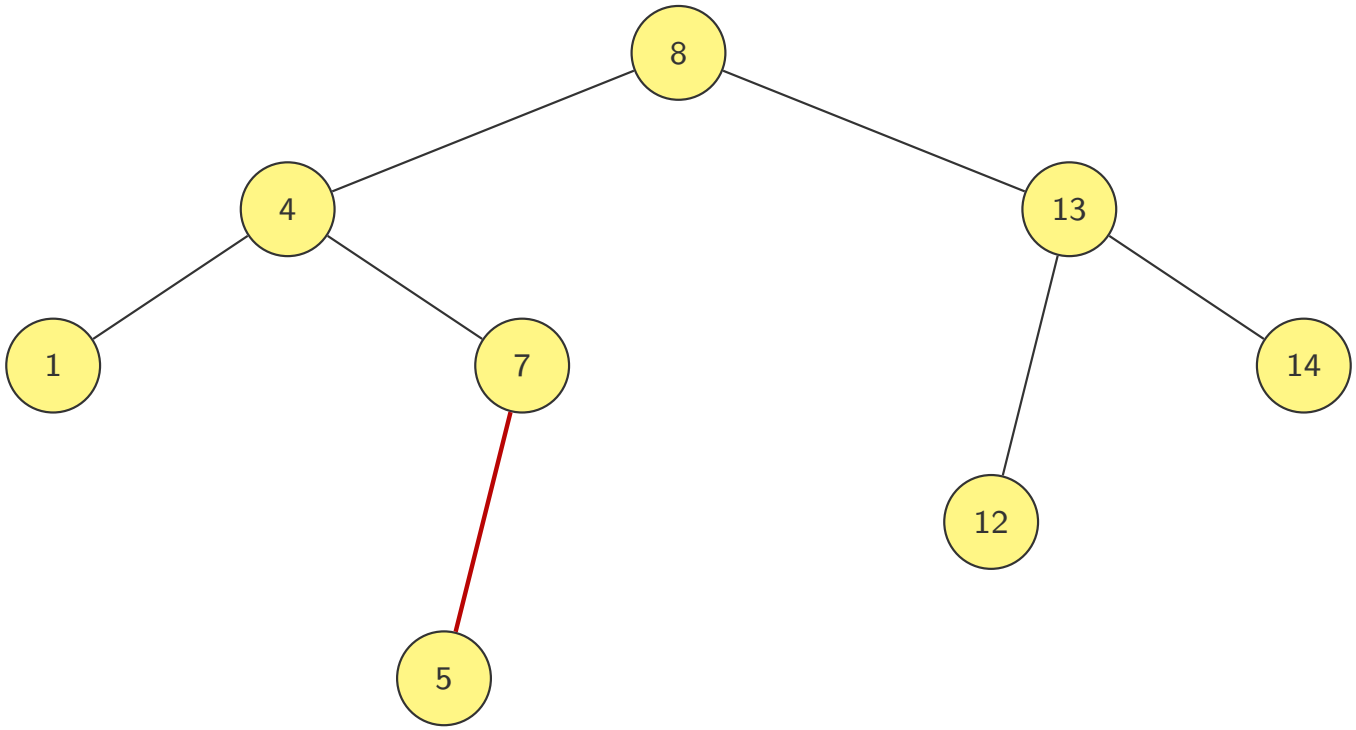
- Isso mantém a propriedade da árvore binária de busca

E agora colocamos o filho direito do sucessor no seu lugar

- O sucessor nunca tem filho esquerdo!

Remoção

Exemplo: removendo a chave 3



Podemos colocar o sucessor de 3 em seu lugar

- Isso mantém a propriedade da árvore binária de busca

E agora colocamos o filho direito do sucessor no seu lugar

- O sucessor nunca tem filho esquerdo!

Remoção

- Note que o nó a ser removido é a raiz de uma árvore. Essa raiz pode ou não ter filhos.
- Logo, convém tratar esse problema como a remoção da raiz de uma árvore.
- Seguimos os seguintes passos na remoção da raiz:
  1. Se a raiz não tiver filhos, ela é simplesmente removida e a árvore resultante fica vazia;
  2. Se a raiz tiver apenas o filho esquerdo, ele assume o papel de raiz;
  3. Senão, basta fazer com que o nó sucessor à raiz assuma o papel da raiz.

Remoção - Implementação


```
1 // Essa funcao recebe uma chave e remove o no contendo
2 // essa chave somente se ela existir na arvore
3 void BST::remove(int key) {
4     root = remove(key, root);
5 }
```

Remoção - Implementação


```
1  // Essa funcao recebe uma chave e remove o no contendo
2  // essa chave somente se ela existir na arvore
3  void BST::remove(int key) {
4      root = remove(key, root);
5  }
6
7
8  Node *BST::remove(int k, Node *node) {
9      if(node == nullptr) // Arvore vazia
10         return nullptr;
11     if(k == node->key) // Achou o no a ser removido
12         return removeRoot(node); // funcao auxiliar
13     // Ainda nao achamos o no, vamos busca-lo
14     if(k < node->key)
15         node->left = remove(k, node->left);
16     else
17         node->right = remove(k, node->right);
18     return node;
19 }
```

Remoção - Implementação (continuação)

Universidade Federal do Ceará
Campus Quixadá

```
1  // Recebe um ponteiro node para a raiz de uma arvore e
2  // remove a raiz, rearranjando a arvore de modo que ela
3  // continue sendo de busca. Devolve o endereco da nova raiz
4  Node *BST::removeRoot(Node *node) {
5      Node *pai, *q;
6      if(node->right == nullptr)
7          q = node->left;
8      else {
9          pai = node;
10         q = node->right;
11         while(q->left != nullptr) {
12             pai = q;
13             q = q->left;
14         }
15         if(pai != node) {
16             pai->left = q->right;
17             q->right = node->right;
18         }
19         q->left = node->left;
20     }
21     delete node;
22     return q;
23 }
```

main.cpp


```cpp
#include <iostream>
#include <string>
#include <sstream>
#include "bst.h"
using namespace std;

int main() {
    BST t;
    string skeys;
    int k;

    cout << "Digite as chaves separadas por espacos: ";
    getline(cin, skeys);
    stringstream ss { skeys };

    while(ss >> k) t.add(k);

    cout << "Menor chave: " << t.minimum() << endl;
    cout << "Maior chave: " << t.maximum() << endl;
    return 0;
}
```

Exercícios

Exercícios

- Conclua a implementação das funções que foram deixadas em aberto nos slides anteriores.

- Suponha que todo nó da BST tenha agora um ponteiro para nó pai. Reimplemente as operações vistas nessa aula considerando este novo ponteiro.

- Escreva uma função que receba como argumento uma BST vazia e um vetor $A[p..q]$ com $q - p + 1$ inteiros em ordem crescente e popule a BST com os inteiros do vetor A de modo que ela seja uma árvore binária de busca completa (altura igual a $\lceil \log_2 (n + 1) \rceil$). Sua função pode ter o seguinte protótipo:
  `void construirBST(BST *t, int A[], int p, int q);`

- Escreva uma função que transforme uma árvore binária de busca em um vetor crescente.

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

FIM