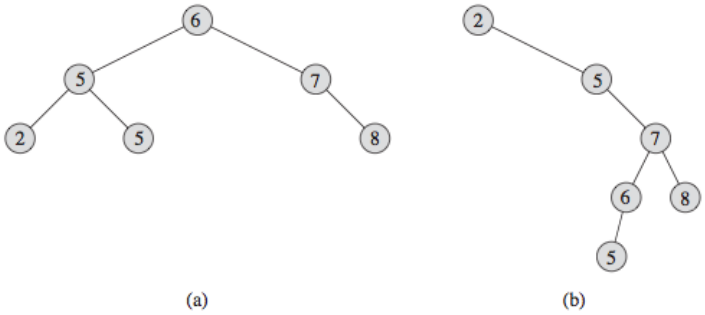
Estruturas de Dados

Árvores binárias de pesquisa

1. Árvores binárias de pesquisa (Cormen capítulo 12. Seções 12.1, 12.2 e 12.3)

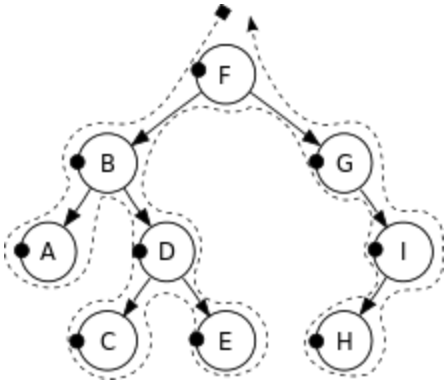
Uma árvore de pesquisa binária é organizada, conforme seu nome sugere, em uma árvore binária. Ela pode ser representada por uma estrutura de dados encadeada em que cada nó é um objeto. Além do campo *chave,* cada nó possui campos *esquerdo, direito* e *p*, que apontam para o filho da esquerda, o filho da direita e o pai, respectivamente. O nó raiz da árvore é o único que possui o campo *p* nulo.

As chaves em uma árvore de pesquisa binária sempre estão armazenadas de modo que satisfaçam à seguinte **propriedade**: Seja *x* um nó em uma árvore de pesquisa binária. Se *y* é um nó na subárvore esquerda de *x*, então *chave[y]* <= *chave[x]*. Se *y* é um nó na subárvore direita de *x*, então *chave[x]* <= *chave[y]*.



Podemos percorrer todas as chaves em uma árvore de pesquisa binária por um simples algoritmo recursivo de três formas diferentes:

- Percurso em ordem: percorra a subárvore da esquerda, visite a raíz, percorra a subárvore da direita.
- Percurso de pré-ordem: visite a raíz, percorra a subárvore da esquerda, percorra a subárvore da direita.
- Percurso de pós-ordem: percorra a subárvore da esquerda, percorra a subárvore da direita, visite a raíz.



Em ordem: A, B, C, D, E, F, G, H, I. Pré-ordem: F, B, A, D, C, E, G, I, H. Pós-ordem: A, C, E, D, B, H, I, G, F

O **percurso em ordem** é usado para recuperar os elementos em ordem.
O **percurso em pré** ordem pode ser usado para duplicar por completo uma árvore de pesquisa binária. Também é usado para fazer expressões em notação polonesa (notação prefixada).
O **percurso em pós ordem** pode ser usado para se deletar por completo uma árvore binária. Também é usado para fazer expressões em notação polonesa reversa (notação pós fixada).

infix: 5 + ((1 + 2) × 4) − 3 => posfix: 5 1 2 + 4 × + 3 − => prefix: - + 5 x + 1 2 4 3

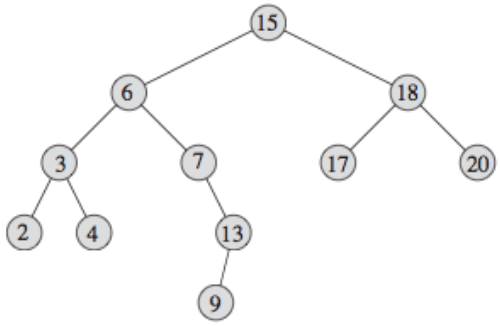
Pesquisa: a operação mais comum em uma árvore de pesquisa binária é a procura por uma certa chave armazenada na árvore. A busca por uma chave *k* começa na raiz, e um caminho descendente na árvore é traçado. Para cada nó *x* encontrado na árvore, o valor da chave *k* é comparado com *chave[x]*. Se *k* é menor que *chave[x]*, a pesquisa continua na subárvore da esquerda. Se *k* é maior que *chave[x]* a busca continua na subárvore da direita. Se *k* é igual a *chave[x],* a busca termina. O tempo de execução da busca é, no pior caso, O(h), onde *h* é a altura da árvore.

Mínimo e máximo: o elemento mínimo de uma árvore de pesquisa binária sempre pode ser encontrado seguindo-se os ponteiros filho da esquerda da árvore a partir da raiz. Da mesma forma, o elemento máximo sempre pode ser encontrado seguindo-se os ponteiros filho da direita a partir da raiz. Ambas as operações são O(h).

Sucessor e predecessor: o sucessor de certo nó *x* pode ser encontrado da seguinte forma: se a subárvore da direita de *x* for não vazia, então o sucessor de *x* será o nó mais à esquerda da subárvore da direita, e pode ser encontrado usando a função *tree-minimum(direita[x])*. Se a subárvore da direita de *x* for vazia e *x* possuir um sucessor, esse sucessor é o ancestral mais baixo de *x* cujo filho da esquerda também é um ancestral de *x*. Para encontrá-lo, basta subirmos a árvore desde *x* até encontrarmos um nó que seja o filho da esquerda de seu pai (incluindo *x*).

O predecessor de *x* é o nó mais a direita da subárvore da esquerda (se existir uma subárvore à esquerda). Encontrado usando *tree-maximum(esquerda[x])*. Caso contrário, subimos a árvore desde *x* até encontrarmos um nó que seja o filho da direita de seu pai.

As operações sucessor e predecessor são O(h).



Sucessor de 13 -> 15. Sucessor de 2 -> 3.
Predecessor de 17 -> 15. Predecessor de 9 -> 7.

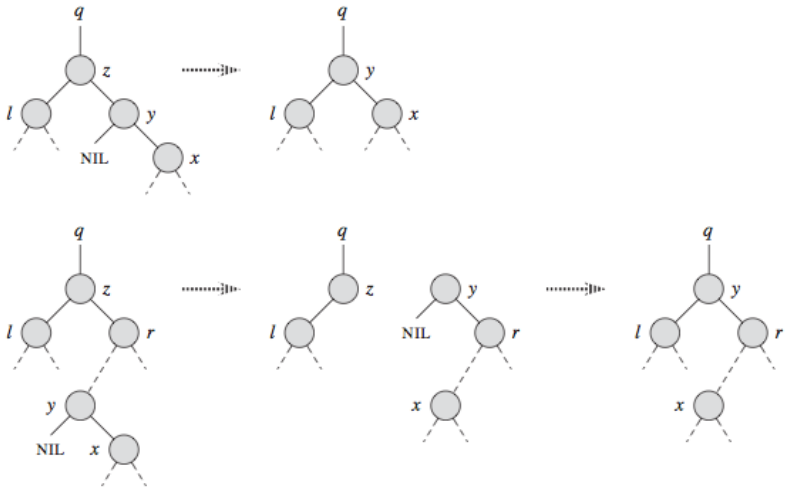
Inserção e remoção: As operações de remoção e inserção causam mudanças na estrutura da árvore binária de pesquisa para que sua propriedade continue válida.

Inserção: A inserção é simples. Basta percorrer a árvore a partir da raiz, buscando o local de inserção. A inserção pode ser feita em O(h). Inserção de números ordenados gera árvores ruins.

Remoção: A remoção de um nó *z* é um pouco mais complexa. Existem três casos. Os dois casos mais simples são:

z não possui filhos. A remoção é simples, bastante remover o nó.
z possui somente um filho. Basta remover *z* e elevar a posição do filho de *z* na árvore.

O caso mais complexo ocorre quando *z* possui dois filhos. Nesse caso, devemos substituir *z* pelo seu **sucessor**. Nesse caso existem duas situações possíveis. Se o sucessor de *z* for seu filho da direita, basta substituir *z* por ele. Caso contrário substituímos o sucessor *y* de *z* por seu filho da direita e, em seguida, substituimos *z* por *y*.



As operações de remoção possuem, no pior caso, complexidade O(h).

Finamente, vimos que a complexidade de quase todas as operações em uma árvore de pesquisa binária são da ordem O(h). No entanto, podemos provar que uma **árvore de pesquisa binária construída aleatoriamente** (uma árvore construída através da inserção de chaves distintas em ordem aleatória) possui altura esperada $h = lg\ n$. Logo, as operações possuem complexidade esperada O(lg n).