Estruturas de Dados

Árvores Binárias de Busca

Universidade Estadual Vale do Acaraú – UVA

Paulo Regis Menezes Sousa

paulo_regis@uvanet.br

Árvores Binárias de Busca

- Inserção
- Busca
- Remoção
- Desempenho

- BST = *binary search tree* = árvore binária de busca.
- As BST precisam ter valores (*chaves de busca*) comparáveis.
- Em uma BST todos os filhos à esquerda tem chaves menores que a raiz, todos os filhos à direita tem chaves maiores que a raiz e isso se aplica a todas as subárvores.

Código 1: Estrutura básica de uma BST

```
1  struct BST {
2      void *value;
3      BST *left;
4      BST *right;
5      int (*compar)(void*, void*);
6  };
```

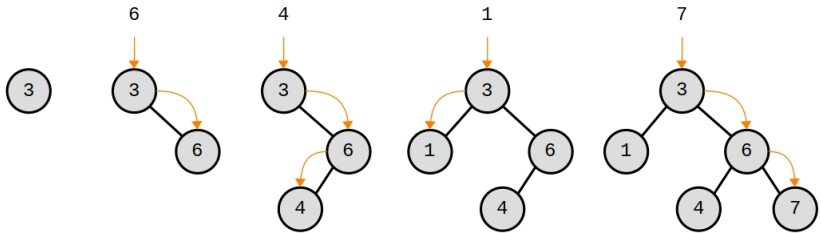
- Um exemplo de TAD para um tipo BST com algumas funções básicas.

```
1  typedef struct BST BST;
2
3  BST  *BST_alloc(int (*compar)(void*, void*));
4  void  BST_free(BST *bst);
5  void  BST_insert(BST *bst, void *value);
6  void *BST_search(BST *bst, void *value);
7  BST  *BST_remove(BST *bst, void *value);
8  void  BST_print(BST *bst, void (*print)(void*));
```

```
1  BST *BST_alloc(int (*compar)(void*, void*)) {
2      BST *bst = NULL;
3
4      if (compar) {
5          bst = (BST*) malloc(sizeof(BST));
6          bst->value = NULL;
7          bst->left = NULL;
8          bst->right = NULL;
9          bst->compar = compar;
10     }
11
12     return bst;
13 }
```

```
1  void BST_free(BST *bst) {
2      if (bst != NULL) {
3          BST_free(bst->left);
4          BST_free(bst->right);
5          free(bst);
6      }
7  }
```

- Após inserir um novo em uma BST a árvore resultante deve também ser BST.
- **Regra de inserção:** *comparar* a chave do novo nó com a chave da raiz e inserir à esquerda se ela é menor, ou à direita se ela é maior.
- **Exemplo:** inserção da sequência: 3, 6, 4, 1, 7.



```
1  void BST_insert(BST *bst, void *value) {
2
3      if (bst && value)
4          if (bst->value == NULL)
5              bst->value = value;
6          else
7              if (bst->compar(bst->value, value) < 0) {
8                  if (bst->right == NULL)
9                      bst->right = BST_alloc(bst->compar);
10
11                 BST_insert(bst->right, value);
12             }
13             else
14                 if (bst->compar(bst->value, value) > 0) {
15                     if (bst->left == NULL)
16                         bst->left = BST_alloc(bst->compar);
17
18                     BST_insert(bst->left, value);
19                 }
20 }
```

- Dada uma árvore de busca r e um número k, encontrar um nó de r cuja chave seja k.

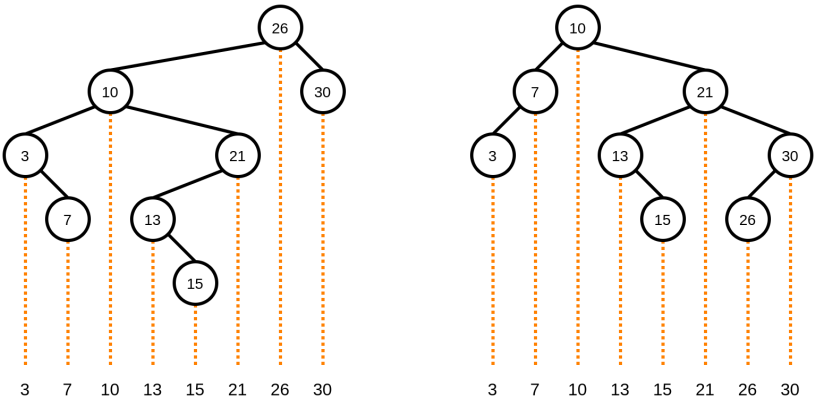
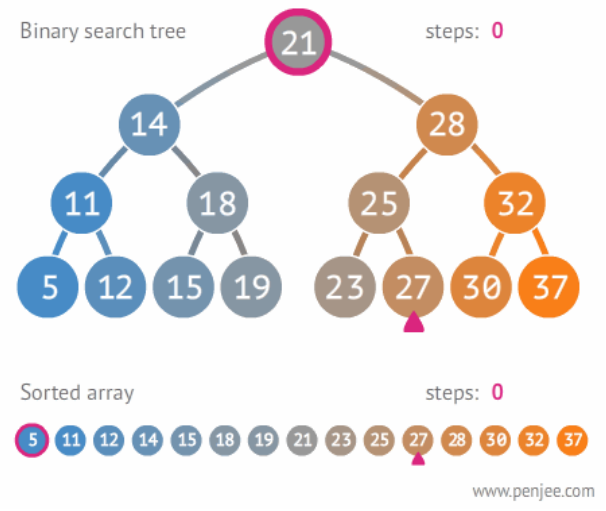
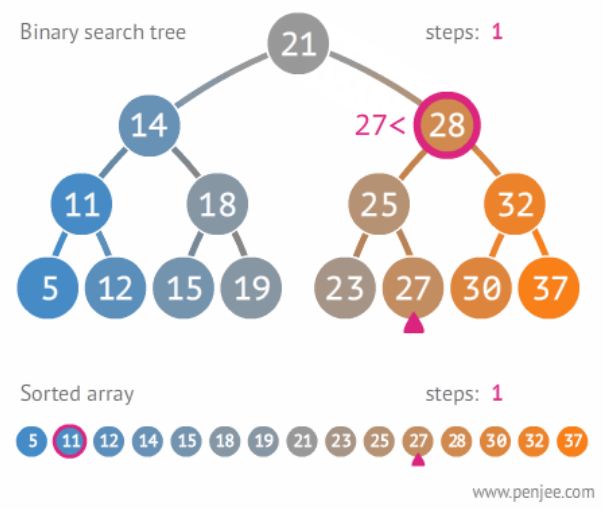
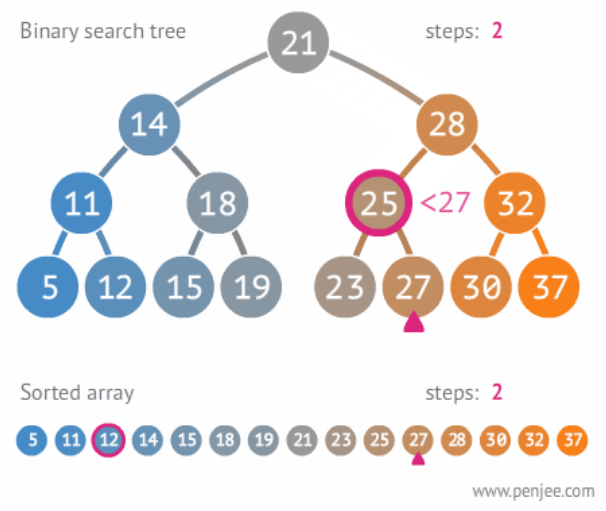


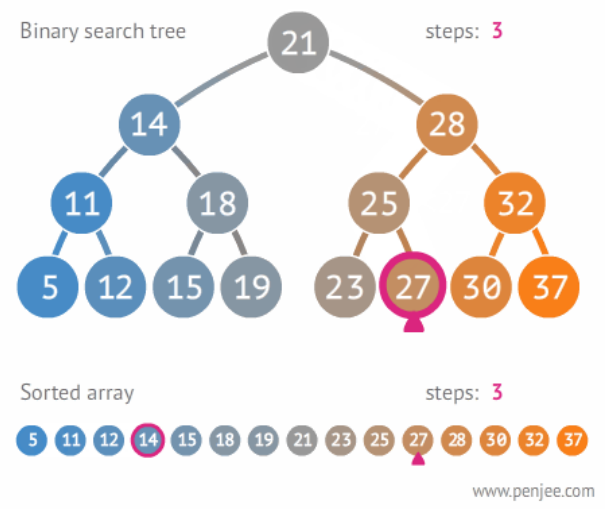
Figura 1: Duas BSTs com o mesmo conjunto de chaves, inseridas em ordens diferentes.

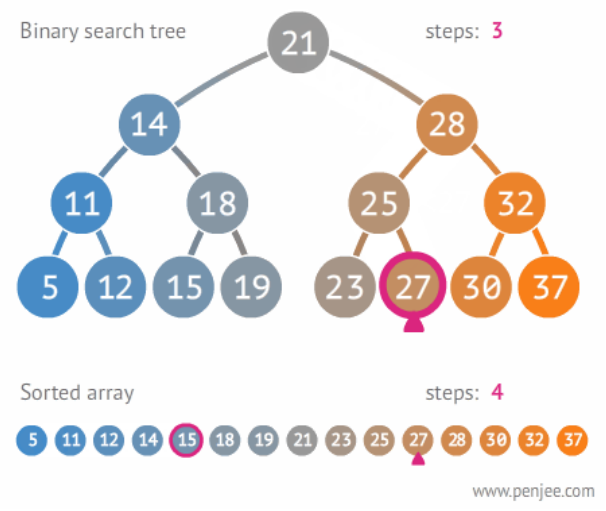
- No pior caso, a busca consome tempo proporcional à altura da árvore.
- Se a árvore for balanceada, o consumo será proporcional a $\log n$, sendo n o número de nós.
- Esse tempo é da mesma ordem que a busca binária num vetor ordenado.

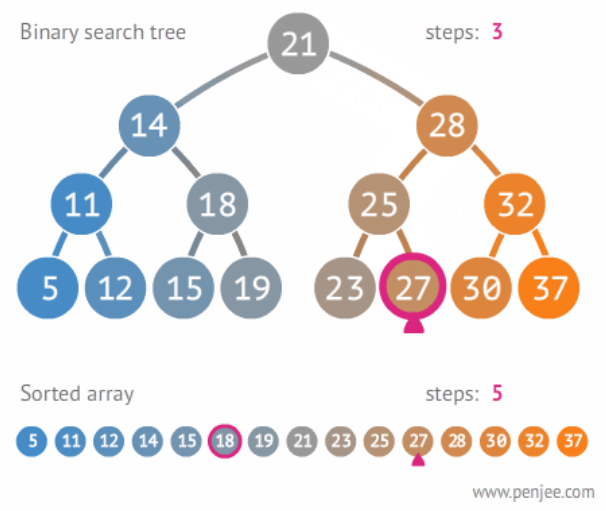

Binary search tree
steps: 0
21
14
28
11
18
25
32
5
12
15
19
23
27
30
37
Sorted array
steps: 0
5 11 12 14 15 18 19 21 23 25 27 28 30 32 37
www.penjee.com

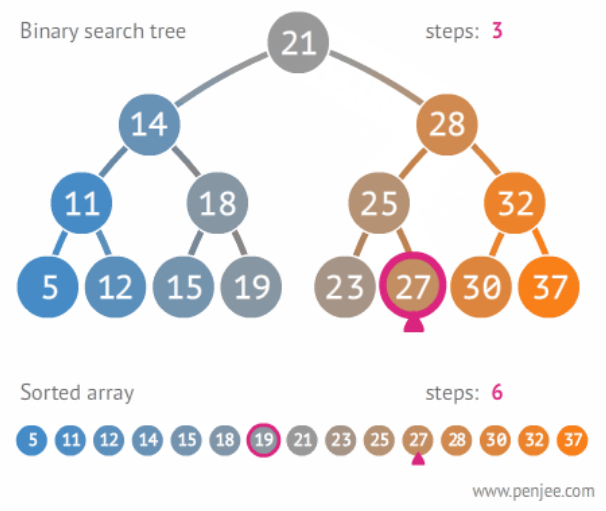

Binary search tree
steps: 1
21
14
27<
28
11
18
25
32
5
12
15
19
23
27
30
37
Sorted array
steps: 1
5 11 12 14 15 18 19 21 23 25 27 28 30 32 37
www.penjee.com

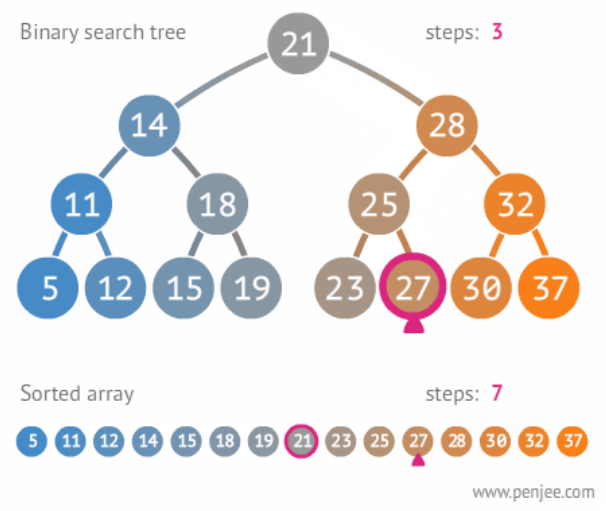

Binary search tree
steps: 2
21
14
28
11
18
25
<27
32
5
12
15
19
23
27
30
37
Sorted array
steps: 2
5 11 12 14 15 18 19 21 23 25 27 28 30 32 37
www.penjee.com

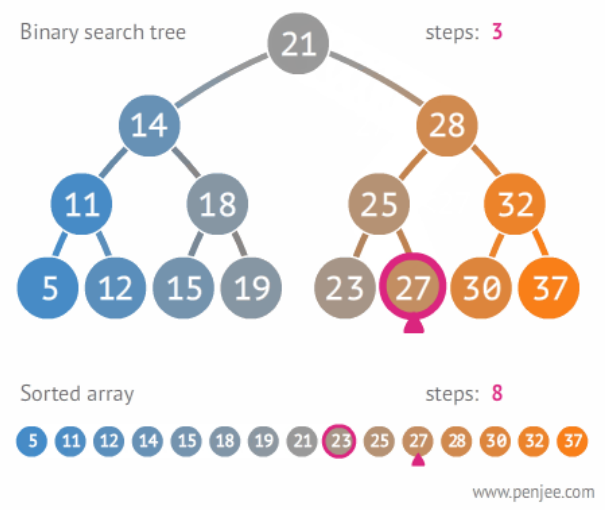

Binary search tree
steps: 3
21
14
28
11
18
25
32
5
12
15
19
23
27
30
37
Sorted array
steps: 3
5 11 12 14 15 18 19 21 23 25 27 28 30 32 37
www.penjee.com

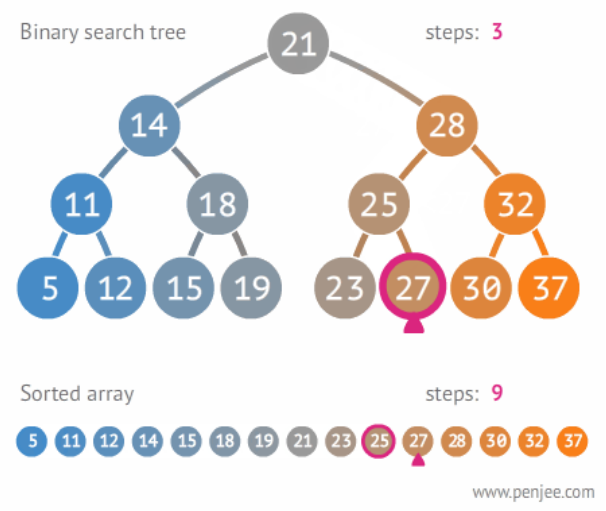

Binary search tree
steps: 3
21
14
28
11
18
25
32
5
12
15
19
23
27
30
37
Sorted array
steps: 4
5 11 12 14 15 18 19 21 23 25 27 28 30 32 37
www.penjee.com

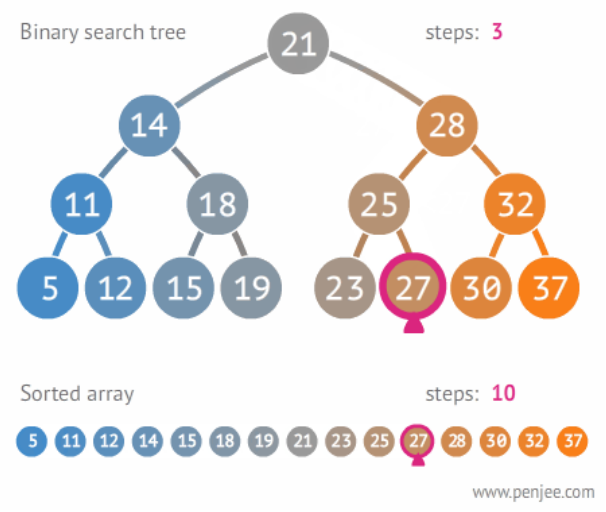

Binary search tree
steps: 3
21
14
28
11
18
25
32
5
12
15
19
23
27
30
37
Sorted array
steps: 5
5 11 12 14 15 18 19 21 23 25 27 28 30 32 37
www.penjee.com


Binary search tree
steps: 3
21
14
28
11
18
25
32
5
12
15
19
23
27
30
37
Sorted array
steps: 6
5 11 12 14 15 18 19 21 23 25 27 28 30 32 37
www.penjee.com


Binary search tree
steps: 3
21
14
28
11
18
25
32
5
12
15
19
23
27
30
37
Sorted array
steps: 7
5 11 12 14 15 18 19 21 23 25 27 28 30 32 37
www.penjee.com


Binary search tree
steps: 3
21
14
28
11
18
25
32
5
12
15
19
23
27
30
37
Sorted array
steps: 8
5 11 12 14 15 18 19 21 23 25 27 28 30 32 37
www.penjee.com

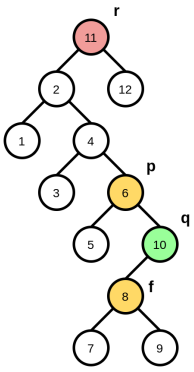

Binary search tree
steps: 3
21
14
28
11
18
25
32
5
12
15
19
23
27
30
37
Sorted array
steps: 9
5 11 12 14 15 18 19 21 23 25 27 28 30 32 37
www.penjee.com


Binary search tree
steps: 3
21
14
28
11
18
25
32
5
12
15
19
23
27
30
37
Sorted array
steps: 10
5 11 12 14 15 18 19 21 23 25 27 28 30 32 37
www.penjee.com

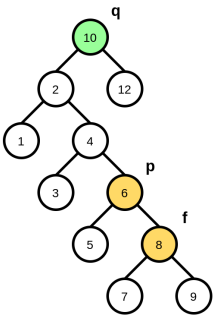
```c
void *BST_search(BST *bst, void *value) {
    if (bst && value)
        if (bst->value)
            if (bst->compar(bst->value, value) == 0)
                return bst->value;
            else
                if (bst->compar(bst->value, value) < 0)
                    return BST_search(bst->right, value);
                else
                    return BST_search(bst->left, value);
    return NULL;
}
```

- Problema: Remover um nó de uma árvore de busca de tal forma que a árvore continue sendo de busca.
- Caso em que o nó a ser removido é a raiz da árvore:
  - Se a raiz não tem um dos filhos, basta que o outro filho assuma o papel de raiz.
  - Senão, faça com que o nó anterior à raiz na ordem e-r-d assuma o papel de raiz.

Antes da remoção



Depois da remoção



- O maior nó à esquerda de r é q.
- O nó q é colocado no lugar de r, os filhos de r passam a ser filhos de q, e f passa a ser filho (direito) de p.

- Toda operação de busca ou inserção visita $1 + p$ nós, sendo p a profundidade do último nó visitado.
- Logo, o número de nós visitados não passa de $1 + h$, sendo h a altura da BST.
- No pior caso, todas as operações sobre uma BST consomem tempo proporcional à altura da árvore.
- Uma BST com N nós, tem altura no máximo $N - 1$ e no mínimo $\lfloor \log N \rfloor$. Se a altura estiver perto de $\log N$, a BT é balanceada.

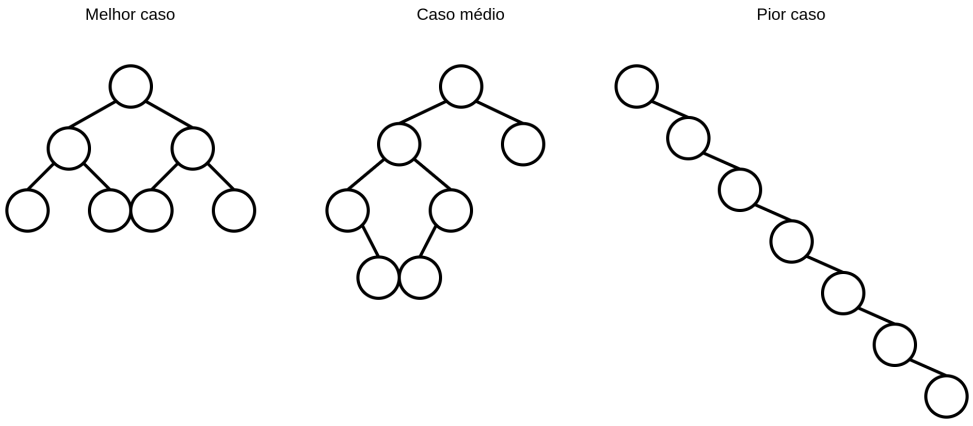
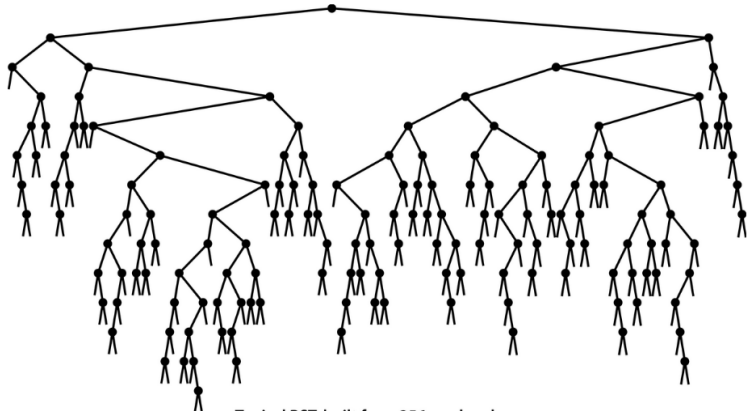


Figura 2: Possíveis estados de uma BST

- BST típica construída com 256 chaves inseridas em ordem aleatória



Typical BST, built from 256 random keys

Problema 1

Escreva uma função não-recursiva que recebe uma árvore binária de busca bst como parâmetro e retorna o ponteiro para o nó cuja chave possui o valor mínimo ou `NULL` caso a árvore esteja vazia.

```
BST *BST_min(BST bst);
```

Problema 2

Escreva uma função não-recursiva que verifica a existência de um dado valor na árvore.

```
int BST_isIn(BST bst, void *value);
```

Problema 3

Considere que você dado um vetor ordenado você precisa construir uma árvore de busca que contenha os mesmos elementos. Como você faria a construção da árvore para evitar que esta ficasse desbalanceada?

```
BST vecToTree(int vec[], int length);
```