



CENTRO DE CIÊNCIA E TECNOLOGIA
LABORATÓRIO DE CIÊNCIAS MATEMÁTICAS
UNIVERSIDADE ESTADUAL DO NORTE FLUMINENSE

Árvores de Busca Binária

Implementação TAD

Disciplina: Estrutura de Dados I

Prof. Fermín Alfredo Tang Montané

Curso: Ciência da Computação

Árvores de Busca Binária

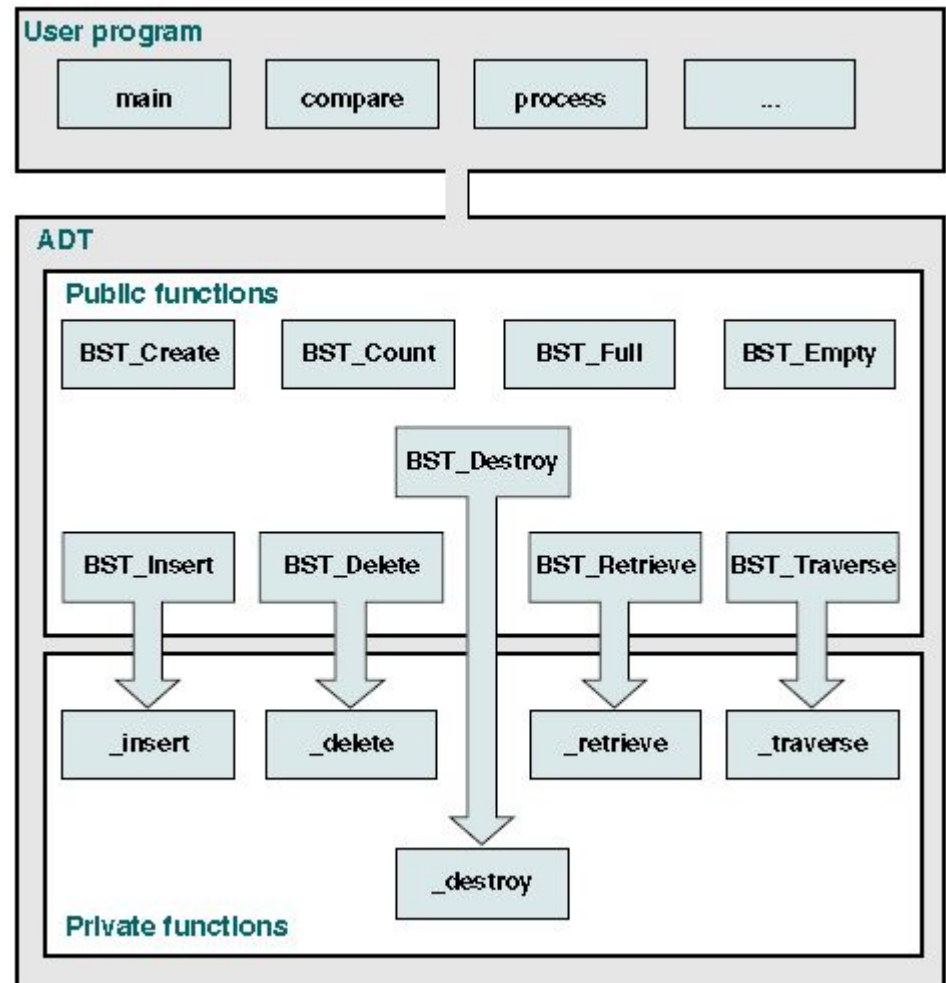
Algoritmos

- Algoritmos básicos para uma árvore de busca binária (*Binary Search Tree*, *BST*) são apresentados a seguir.
- Outros algoritmos podem ser necessários dependendo da aplicação. Por exemplo, no caso de percursos, costuma ser necessário definir uma função a nível de aplicação para processar o dado do nó.
- O tipo abstrato de dados deve ser capaz dar suporte a diferentes estruturas. Por isso, o cabeçalho da árvore é armazenado na memória dinâmica.

Árvores de Busca Binária

Algoritmos

- Definem-se 9 funções públicas e 5 funções privadas.



Árvores de Busca Binária

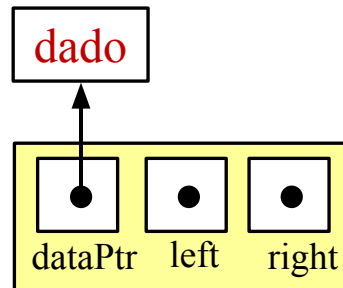
Estruturas e Protótipos

- O tipo abstrato de dado para uma árvore binária utiliza duas estruturas:
 - Uma estrutura para o **cabeça-lho da árvore**;
 - Outra estrutura para o **nó da árvore**.
- O **cabeça-lho da árvore**, é parecido ao utilizado em uma lista. Ele possui três campos:
 - Um contador;
 - Um ponteiro ao nó raiz da árvore;
 - O endereço de uma função de comparação, necessária para realizar buscas na árvore.
- Qualquer programa de aplicação somente terá acesso ao ponteiro para nó cabeça-lho da árvore.
- Por outro lado, a função de comparação é definida de maneira particular para cada programa de aplicação, uma vez que depende dos dados da árvore.
- Um **nó da árvore** possui também possui três campos:
 - Um ponteiro para o dado armazenado;
 - Um ponteiro para a sub-árvore esquerda (nó raiz dessa sub-árvore);
 - Um ponteiro para a sub-árvore direita (nó raiz dessa sub-árvore)

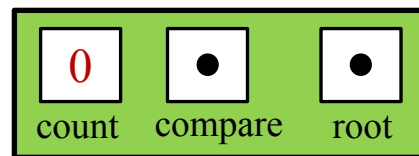
Árvores de Busca Binária

Estruturas e Protótipos

- O código de definição das duas estruturas é o seguinte:
- As estruturas são ilustradas graficamente:



NODE



BST_TREE

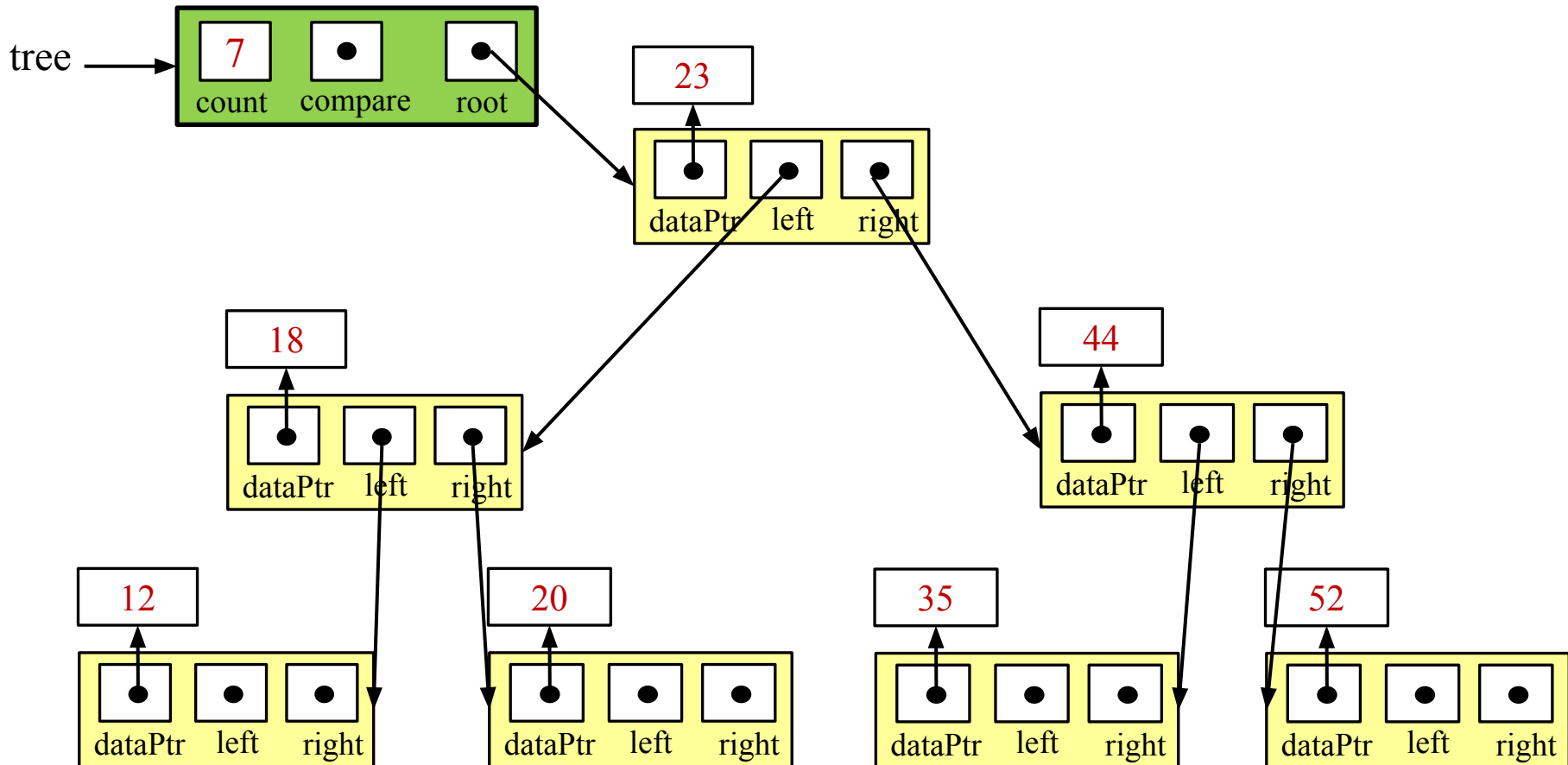
P7-01.h - Estruturas

```
6  #include <stdbool.h>
7
8  // Structure Declarations
9  typedef struct node
10 {
11     void*      dataPtr;
12     struct node* left;
13     struct node* right;
14 } NODE;
15
16 typedef struct
17 {
18     int    count;
19     int    (*compare) (void* argu1, void* argu2);
20     NODE*  root;
21 } BST_TREE;
22
```

Árvores de Busca Binária

Exemplo da Estrutura

- O exemplo ilustra a estrutura de árvore binária genérica com cabeçalho.



Árvores de Busca Binária

Estruturas e Protótipos

P7-01.h - Protótipos

- As declarações dos protótipos das funções são os seguintes:

```
23 // Prototype Declarations
24 BST_TREE* BST_Create
25     (int (*compare) (void* argu1, void* argu2));
26 BST_TREE* BST_Destroy (BST_TREE* tree);
27
28 bool BST_Insert (BST_TREE* tree, void* dataPtr);
29 bool BST_Delete (BST_TREE* tree, void* dltKey);
30 void* BST_Retrieve (BST_TREE* tree, void* keyPtr);
31 void BST_Traverse (BST_TREE* tree,
32     void (*process)(void* dataPtr));
33
34 bool BST_Empty (BST_TREE* tree);
35 bool BST_Full (BST_TREE* tree);
36 int BST_Count (BST_TREE* tree);
37
38 static NODE* _insert
39     (BST_TREE* tree, NODE* root,
40     NODE* newPtr);
41 static NODE* _delete
42     (BST_TREE* tree, NODE* root,
43     void* dataPtr, bool* success);
44 static void* _retrieve
45     (BST_TREE* tree,
46     void* dataPtr, NODE* root);
47 static void _traverse
48     (NODE* root,
49     void (*process) (void* dataPtr));
50 static void _destroy (NODE* root);
```

Árvores de Busca Binária

BST_Create (ABB_Criar)

- A operação `BST_Create()` cria o cabeçalho da árvore na memória dinâmica. Inicializa todos os campos do cabeçalho.

- 1 Tenta alocar memória para o nó cabeçalho da árvore;
- 2 Caso a alocação seja bem sucedida, inicializa os campos do cabeçalho:
 - **ponteiro raiz em nulo;**
 - **contador em zero;**
 - a** **ponteiro da função de comparação em compare.**
- 3 Retorna o ponteiro ao nó cabeçalho da árvore.

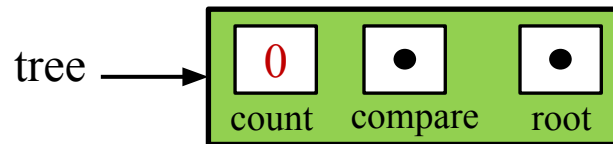
P7-02.h – BST_Create()

```
1  /* ===== BST_Create =====
2     Allocates dynamic memory for an BST tree head
3     node and returns its address to caller
4     Pre      compare is address of compare function
5             used when two nodes need to be compared
6     Post     head allocated or error returned
7     Return  head node pointer; null if overflow
8  */
9  BST_TREE* BST_Create
10      (int (*compare) (void* arg1, void* arg2))
11  {
12      // Local Definitions
13      BST_TREE* tree;
14
15      // Statements
16      1 tree = (BST_TREE*) malloc (sizeof (BST_TREE));
17      2 if (tree)
18          {
19              tree->root      = NULL;
20              tree->count      = 0;
21              a tree->compare = compare;
22          } // if
23
24      3 return tree;
25  } // BST_Create
```


Árvores de Busca Binária

BST_Create (ABB_Criar)

- O exemplo de BST_Create().



Árvores de Busca Binária

BST_Insert (ABB_Inserir)

- A operação **BST_Insert()** insere um novo nó em uma árvore de busca binária.

- 1 A função possui 2 parâmetros:
 - **ponteiro ao cabeçalho;**
 - **ponteiro ao elemento;**
- 2 Tenta alocar memória para o novo nó. Se não conseguir retorna False.
- 3 Inicializa o novo nó.
- 4 Se a árvore estiver vazia, o novo nó se torna raiz. Caso contrário, chama a **função _insert()** no nó **raiz da árvore.**
- 5 Atualiza o contador. Retorna True.

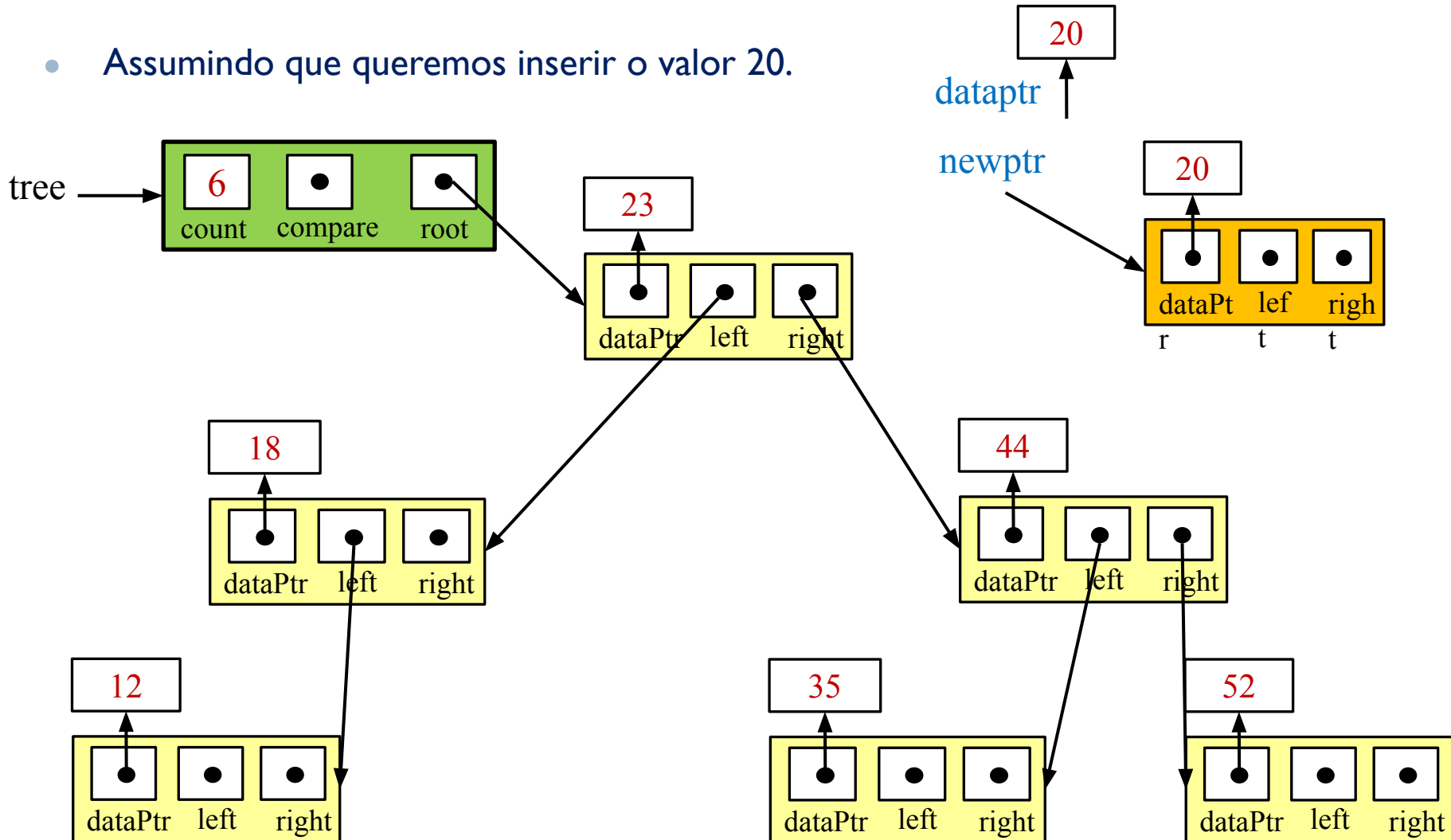
P7-03.h – BST_Insert()

```
1  /* ===== BST_Insert =====
2      This function inserts new data into the tree.
3          Pre    tree is pointer to BST tree structure
4          Post   data inserted or memory overflow
5          Return Success (true) or Overflow (false)
6  */
7  bool BST_Insert (BST_TREE* tree, void* dataPtr) 1
8  {
9      // Local Definitions
10     NODE* newPtr;
11
12     // Statements
13     2 newPtr = (NODE*)malloc(sizeof(NODE));
14     if (!newPtr)
15         return false;
16
17     3 newPtr->right = NULL;
18     newPtr->left = NULL;
19     newPtr->dataPtr = dataPtr;
20
21     4 if (tree->count == 0)
22         tree->root = newPtr;
23     else
24         _insert(tree, tree->root, newPtr);
25
26     5 (tree->count)++;
27     return true;
28 } // BST_Insert
```

Árvores de Busca Binária

BST_Insert (ABB_Inserir)

- Assumindo que queremos inserir o valor 20.



Árvores de Busca Binária

_Insert (_Inserir)

P7-04.h – _Insert()

- A função recursiva `_insert()` procura o local certo de inserção para o novo nó:
 - no cabeçalho ou
 - um nó folha.
- Retorna o ponteiro ao novo nó o que permite o encadeamento.
- 1 Possui 3 parâmetros:
 - ponteiro ao cabeçalho;
 - ponteiro ao nó raiz atual;
 - ponteiro ao novo nó.

```
1  /* ===== _insert =====
2      This function uses recursion to insert the new data
3      into a leaf node in the BST tree.
4      Pre    Application has called BST_Insert, which
5             passes root and data pointer
6      Post   Data have been inserted
7      Return pointer to [potentially] new root
8  */
9  1 NODE* _insert (BST_TREE* tree, NODE* root, NODE* newPtr)
10 {
11     // Statements
12     if (!root)
13         // if NULL tree
14         return newPtr;
```

Árvores de Busca Binária

_Insert (_Inserir)

- 1 Se a raiz é nula (caso base), encontrou a posição de inserção.
 - a Retorna o ponteiro ao novo nó.
- Caso contrário:
- 2 Verifica se o novo nó é menor que a raiz atual.
 - b Se menor, chama `_insert()` na subÁrvore Esquerda.
 - c Se maior ou igual, chama `_insert()` na SubÁrvore Direita.

P7-04.h – _Insert()

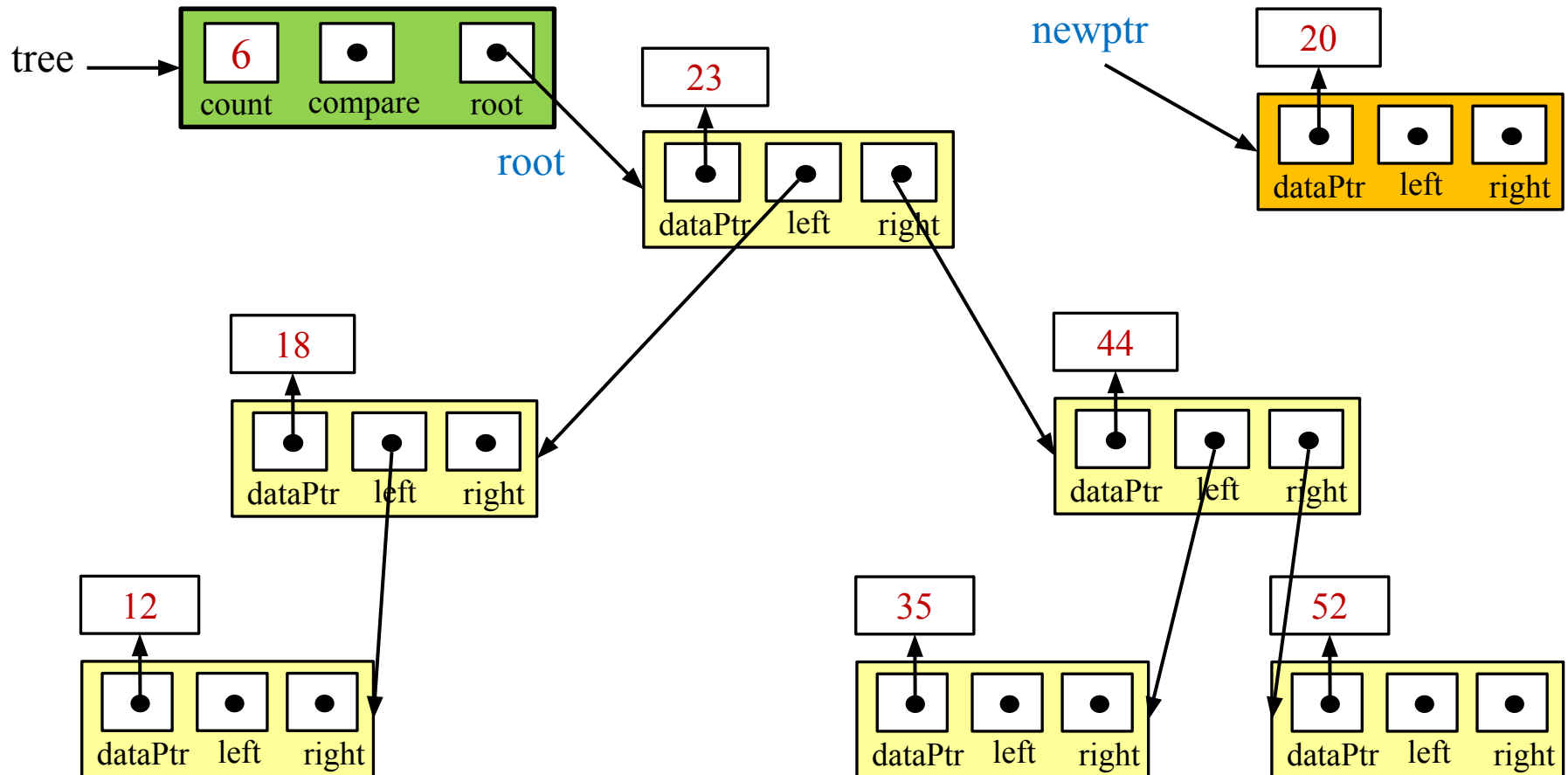
```
8  /*
9  NODE* _insert (BST_TREE* tree, NODE* root, NODE* newPtr)
10 {
11 // Statements
12 1 if (!root)
13 // if NULL tree
14 a return newPtr;
15
16 // Locate null subtree for insertion
17 2 if (tree->compare(newPtr->dataPtr,
18 root->dataPtr) < 0)
19 {
20 b root->left = _insert(tree, root->left, newPtr);
21 return root;
22 } // new < node
23 else
24 // new data >= root data
25 {
26 c root->right = _insert(tree, root->right, newPtr);
27 return root;
28 } // else new data >= root data
29 return root;
30 } // _insert
```

Função comparação (A,B)

Árvores de Busca Binária

BST_Insert (ABB_Inserir)

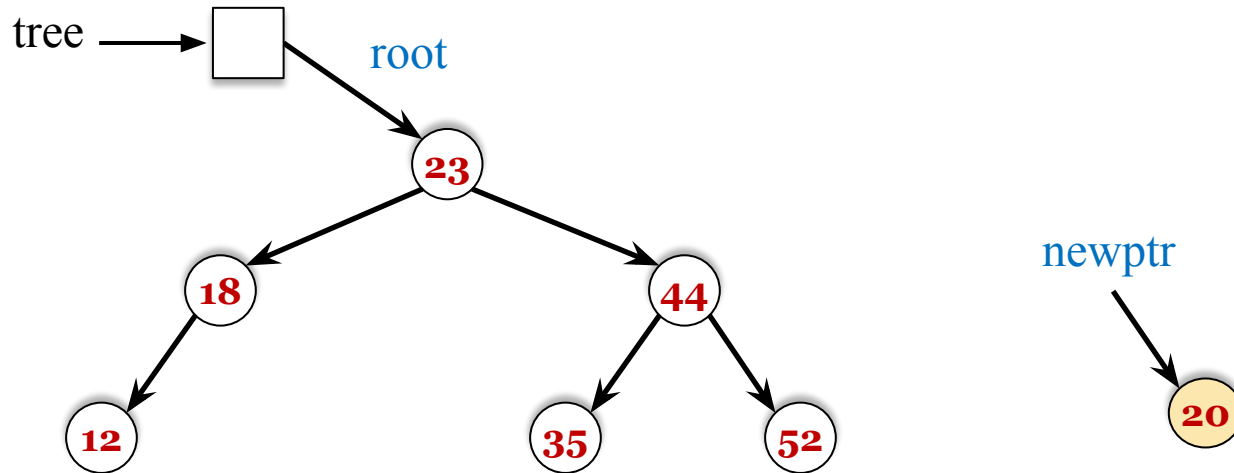
- Assumindo que queremos inserir o valor 20.



Árvores de Busca Binária

BST_Insert (ABB_Inserir)

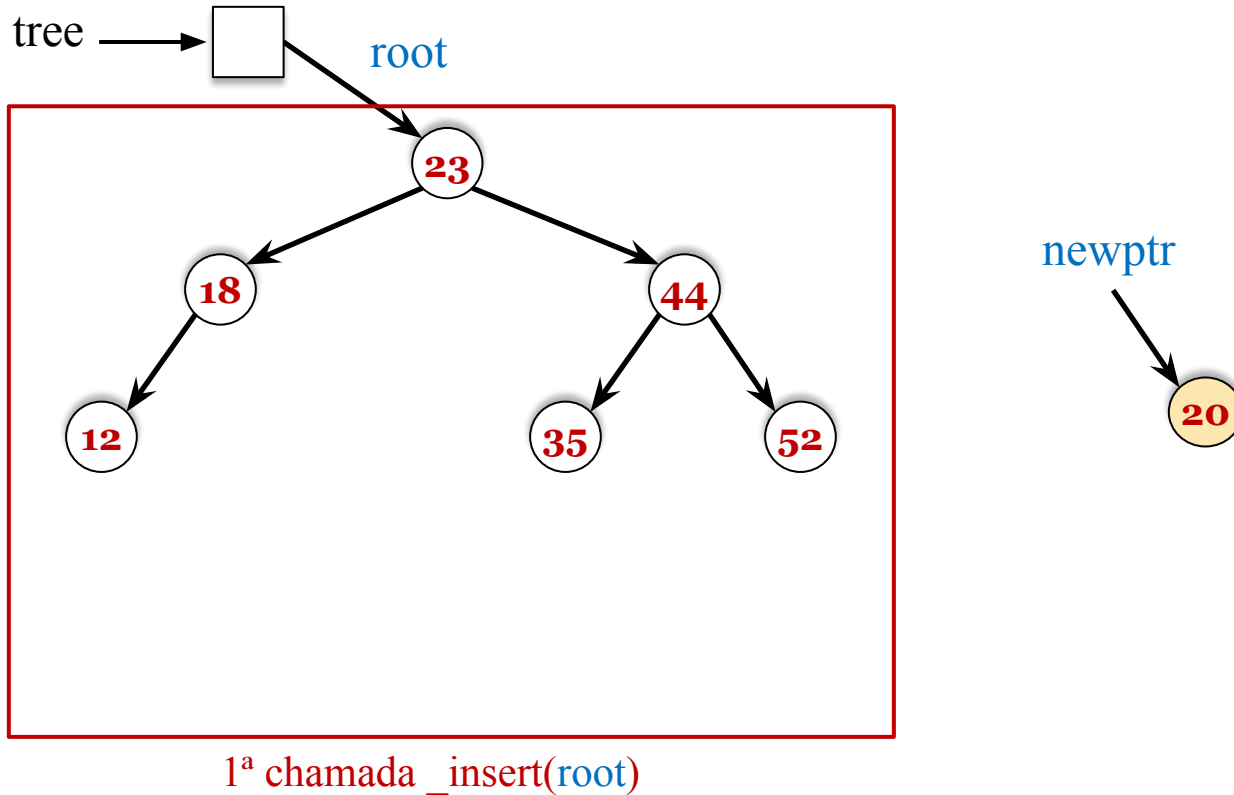
- Assumindo que queremos inserir o valor 20.



Árvores de Busca Binária

BST_Insert (ABB_Inserir)

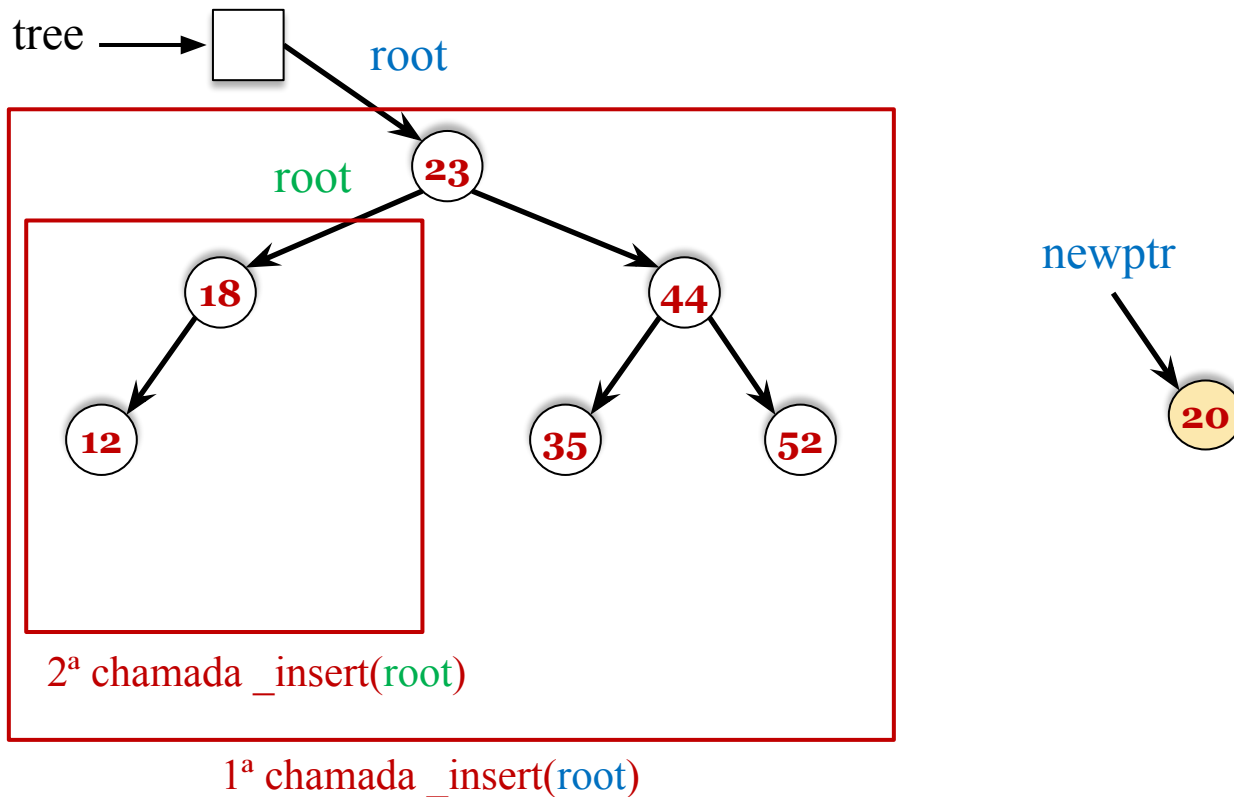
- Assumindo que queremos inserir o valor 20.



Árvores de Busca Binária

BST_Insert (ABB_Inserir)

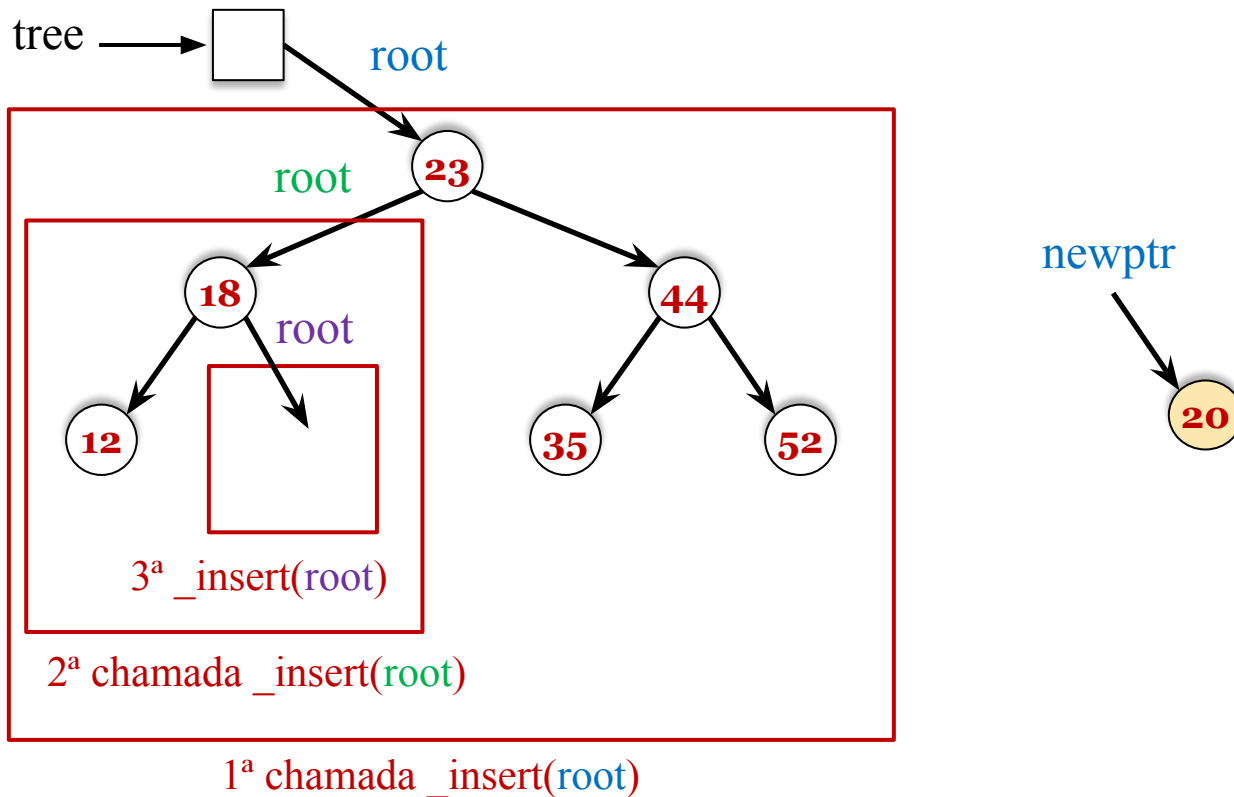
- Assumindo que queremos inserir o valor 20.



Árvores de Busca Binária

BST_Insert (ABB_Inserir)

- Assumindo que queremos inserir o valor 20.



Árvores de Busca Binária

ArvBinEliminar (BST_Delete)

P7-05.h – BST_Delete ()

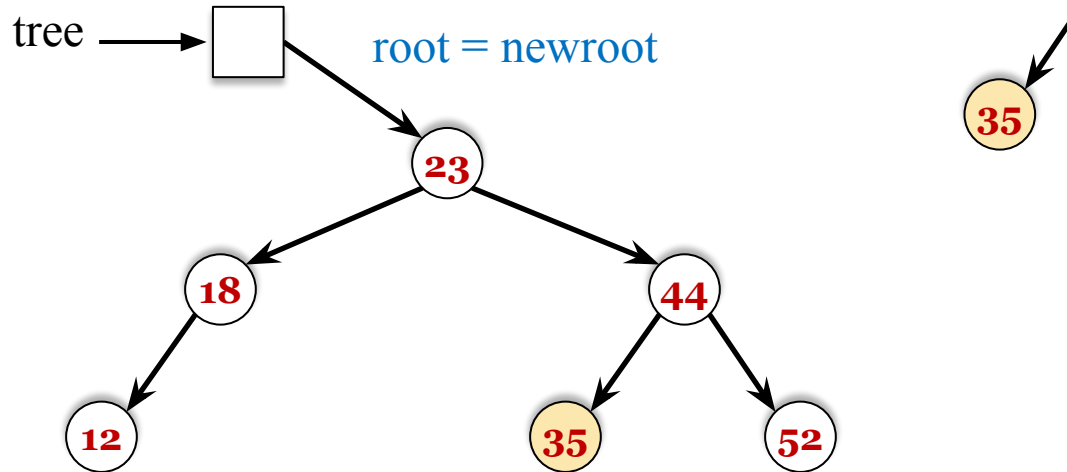
- A operação **BST_Delete()** tenta eliminar um nó com valor específico em uma árvore de busca binária.
- 1 A função possui 2 parâmetros:
 - **ponteiro ao cabeçalho;**
 - **ponteiro ao elemento;**
 - 2 Chama a função recursiva **_delete()**.
 - 3 Se a eliminação der certo:
 - **Atualiza a raiz no cabeçalho;**
 - **Atualiza o contador;**
 - 4 Se removeu o nó retorna **True**; Caso contrário retorna **False**.

```
1  /* ===== BST_Delete =====
2  This function deletes a node from the tree and
3  rebalances it if necessary.
4      Pre    tree initialized--null tree is OK
5             dltKey is pointer to data structure
6             containing key to be deleted
7      Post   node deleted and its space recycled
8             -or- An error code is returned
9      Return Success (true) or Not found (false)
10 */
11 bool BST_Delete (BST_TREE* tree, void* dltKey) 1
12 {
13     // Local Definitions
14     bool success;
15     NODE* newRoot;
16
17     // Statements
18     2 newRoot = _delete (tree, tree->root, dltKey,
19                        &success);
20     3 if (success)
21     {
22         tree->root = newRoot;
23         (tree->count)--;
24         if (tree->count == 0)
25             // Tree now empty
26             tree->root = NULL;
27     } // if
28     4 return success;
29 } // BST_Delete
```

Árvores de Busca Binária

BST_Insert (ABB_Inserir)

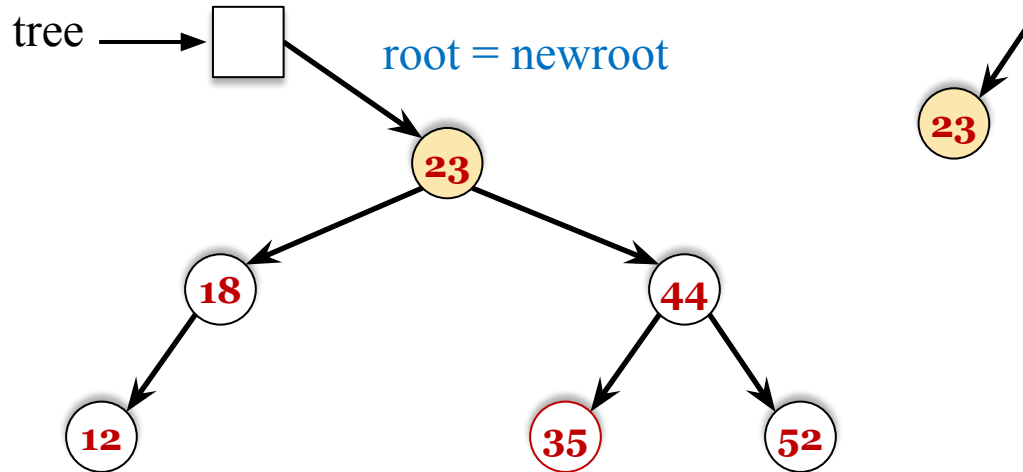
- Assumindo que queremos remover o valor 35.



Árvores de Busca Binária

BST_Insert (ABB_Inserir)

- Assumindo que queremos inserir o valor 23.



- Considerando o sucessor imediato.

Árvores de Busca Binária

_Delete (_Eliminar)

P7-06.h – _Delete ()

- A função recursiva `_delete()` procura na árvore o elemento a ser removido. Caso exista, trata 3 casos:
 - nó sem filhos;
 - nó com 1 filho;
 - nó com 2 filhos.
- A função possui 4 parâmetros:
 - ponteiro ao cabeçalho;
 - ponteiro ao nó raiz atual;
 - ponteiro o elemento;
 - booleano sucesso.
- 1 Se a raiz for nula:
 - Faz sucesso false;
 - Retorna nulo.

```
1  /* ===== _delete =====
2      Deletes node from the tree and rebalances
3      tree if necessary.
4      Pre    tree initialized--null tree is OK
5             dataPtr contains key of node to be deleted
6      Post   node is deleted and its space recycled
7             -or- if key not found, tree is unchanged
8             success is true if deleted; false if not
9      Return pointer to root
10 */
11 NODE* _delete (BST_TREE* tree,    NODE* root,
12               void*    dataPtr, bool* success)
13 {
14     // Local Definitions
15     NODE* dltPtr;
16     NODE* exchPtr;
17     NODE* newRoot;
18     void* holdPtr;
19
20     // Statements
21     1 if (!root)
22     {
23         *success = false;
24         return NULL;
25     } // if
26
```

Árvores de Busca Binária

_Delete (_Eliminar)

P7-06.h – Função recursiva _Delete ()

- 1 Compara o dado a ser eliminado ao dado na raiz:
 - a Se for menor;
 - b Se for maior;
- 2 Se for igual, o nó foi encontrado. Tratam-se os casos em que o nó a ser removido (nó root) não possui filhos ou possui um filho.
- 3 Caso sem filho esquerdo (sem antecessor) e talvez 1 filho direito.
 - c Elimina o nó;
 - d Retorna a sub-árvore direita ou nulo;
- 4 Caso com apenas 1 filho esquerdo.

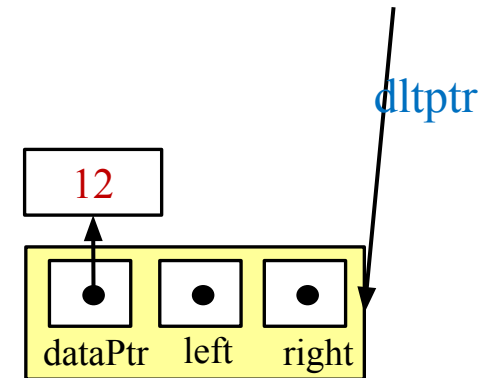
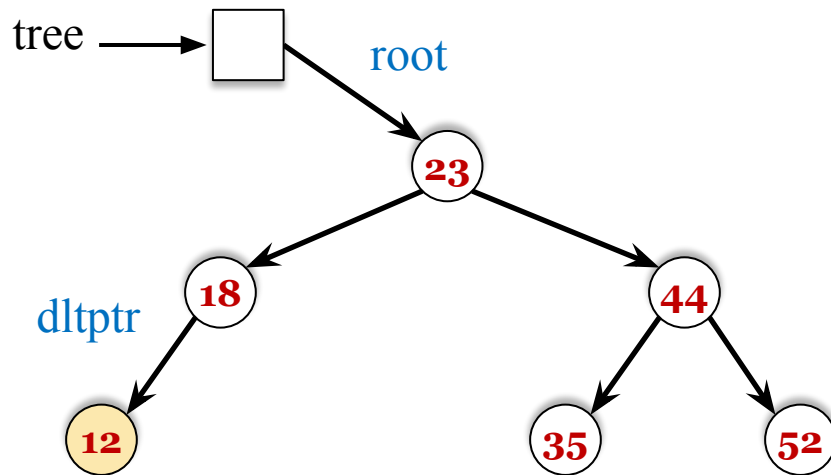
```
27 1 if (tree->compare(dataPtr, root->dataPtr) < 0)
28     a root->left = _delete (tree,    root->left,
29                           dataPtr, success);
30     else if (tree->compare(dataPtr, root->dataPtr) > 0)
31         b root->right = _delete (tree,    root->right,
32                                dataPtr, success);
33 2 else
34     // Delete node found--test for leaf node
35     {
36     3 dltPtr = root;
37       if (!root->left)
38         // No left subtree
39         {
40           free (root->dataPtr);      // data memory
41           newRoot = root->right;
42           c free (dltPtr);           // BST Node
43           *success = true;
44           d return newRoot;         // base case
45         } // if true
46     4 else
47       if (!root->right)
48         // only left subtree
49         {
50           newRoot = root->left;
51           free (dltPtr);
52           *success = true;
53           return newRoot;          // base case
54         } // if
```

free (root->dataPtr);

Árvores de Busca Binária

_Delete (_Eliminar)

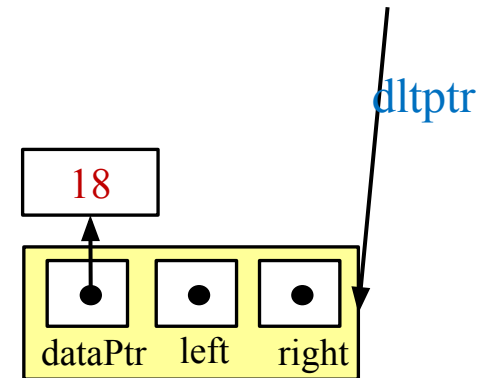
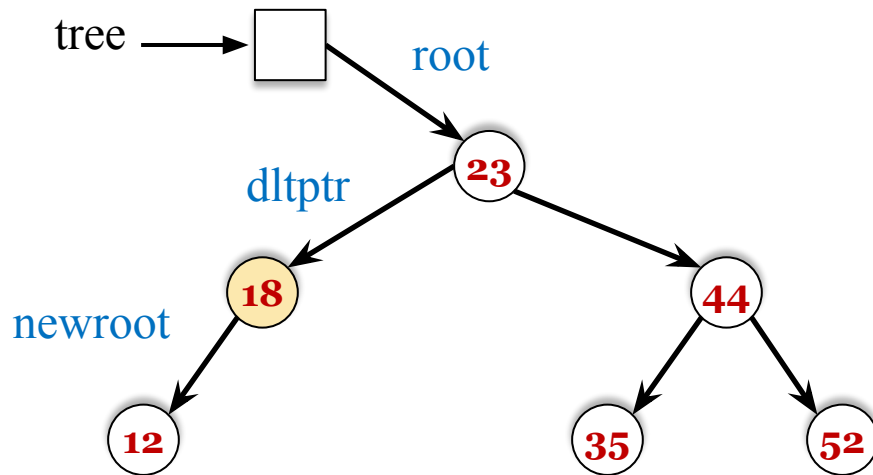
- Assumindo que queremos remover o valor 12.



Árvores de Busca Binária

_Delete (_Eliminar)

- Assumindo que queremos remover o valor 18.



Árvores de Busca Binária

_Delete (_Eliminar)

- Trata-se o caso em que o nó a ser removido (nó root) possui dois filhos.

- 1 Procura na sub-árvore esquerda de root, o nó antecessor (nó exchPtr).
- 2 Troca os dados da raiz e do nó antecessor.
 - a Salva o dado de root;
 - b Substitui o dado de root pelo dado do antecessor;
 - c Substitui o dado do antecessor pelo dado de root;
- 4 Chama `_delete()` para eliminar o nó antecessor na sub-árvore esquerda.

P7-06.h – Função recursiva `_Delete ()`

```
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
```

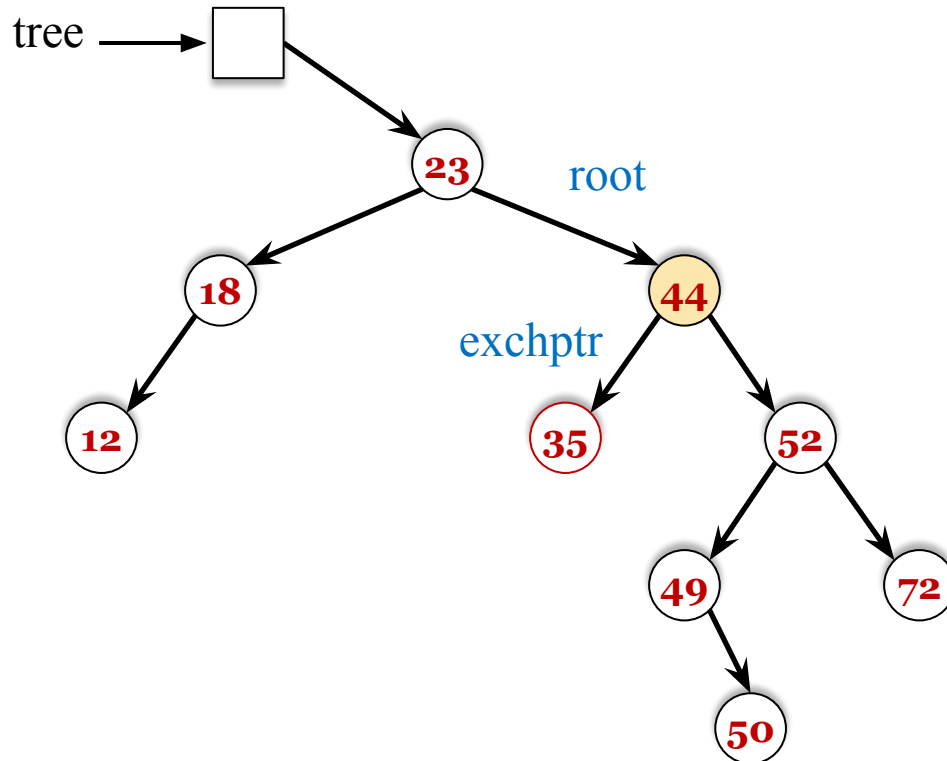
```
else
    // Delete Node has two subtrees
    {
        1 exchPtr = root->left;
        // Find largest node on left subtree
        2 while (exchPtr->right)
            exchPtr = exchPtr->right;

        3 // Exchange Data
        holdPtr      = root->dataPtr; a
        root->dataPtr = exchPtr->dataPtr; b
        exchPtr->dataPtr = holdPtr; c
        root->left =
        4 _delete (tree, root->left,
                    exchPtr->dataPtr, success);
    } // else
    } // node found
    return root;
} // _delete
```

Árvores de Busca Binária

_Delete (_Eliminar)

- Assumindo que queremos remover o valor 44.

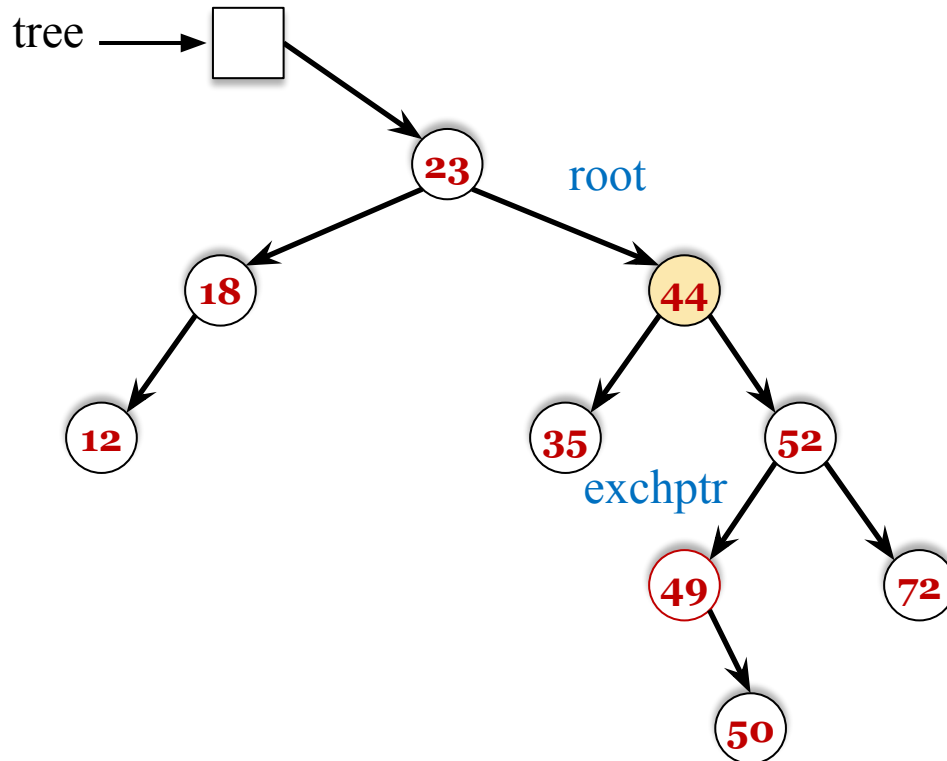


- Considerando o antecessor imediato.

Árvores de Busca Binária

_Delete (_Eliminar)

- Assumindo que queremos remover o valor 44.



- Considerando o sucessor imediato.

Árvores de Busca Binária

BST_Retrieve (ABB_Recupera)

- A operação **BST_Retrieve()** realiza uma busca de um elemento específico na árvore, começando pela raiz e descendo pela esquerda ou direita conforme o resultado da comparação do elemento procurado com o nó atual.

P7-07.h – BST_Retrieve ()

- 1 A função **BST_Retrieve()** possui dois parâmetros:
 - Ponteiro à árvore;
 - Ponteiro ao elemento;
- 2 Se a raiz não é nula:
 - Chama a função recursiva **_Retrieve()** de 3 parâmetros:
 - Ponteiro à árvore;
 - Ponteiro ao elemento;
 - Ponteiro ao nó atual;
- 3 Caso contrário, retorna nulo.

```
1  /* ===== BST_Retrieve =====
2  Retrieve node searches tree for the node containing
3  the requested key and returns pointer to its data.
4      Pre      Tree has been created (may be null)
5              data is pointer to data structure
6              containing key to be located
7      Post     Tree searched and data pointer returned
8      Return   Address of matching node returned
9              If not found, NULL returned
10 */
11 void* BST_Retrieve (BST_TREE* tree, void* keyPtr) 1
12 {
13     // Statements
14     2 if (tree->root)
15         return _retrieve (tree, keyPtr, tree->root);
16     3 else
17         return NULL;
18 } // BST_Retrieve
```

Árvores de Busca Binária

_Retrieve (_Recupera)

P7-08.h – _Retrieve()

1 A função recursiva `_retrieve()` realiza a busca de um elemento específico. Possui três parâmetros:

- Ponteiro à árvore;
- Ponteiro ao elemento;
- Ponteiro ao nó atual;

2 Se a raiz não é nula:

- Usa a função `compare` armazenada na árvore.

Compara o elemento com raiz:

- a Se menor. Vai SubArv. Esq.
- b Se maior. Vai SubArv. Dir.
- c Se Igual. Retorna o ponteiro ao elemento no nó atual.

3 Caso contrário, retorna nulo.

```
1  /* ===== _retrieve =====
2  Searches tree for node containing requested key
3  and returns its data to the calling function.
4      Pre    _retrieve passes tree, dataPtr, root
5             dataPtr is pointer to data structure
6             containing key to be located
7      Post   tree searched; data pointer returned
8      Return Address of data in matching node
9             If not found, NULL returned
10 */
11 void* _retrieve (BST_TREE* tree,
12                 void* dataPtr, NODE* root) 1
13 {
14     // Statements
15     2 if (root)
16     {
17         a if (tree->compare(dataPtr, root->dataPtr) < 0)
18             return _retrieve(tree, dataPtr, root->left);
19         b else if (tree->compare(dataPtr,
20                                root->dataPtr) > 0)
21             return _retrieve(tree, dataPtr, root->right);
22         c else
23             // Found equal key
24             return root->dataPtr;
25     } // if root
26     else
27         // Data not in tree
28         3 return NULL;
29 }
```

Árvores de Busca Binária

BST_Traverse (ABB_Percurso)

- A operação `BST_Traverse()` realiza um percurso em-ordem da árvore e utiliza uma função definida a nível de aplicação como função de processamento de cada nó. O percurso em-ordem determina a ordem em que os nós da árvore são visitados, enquanto a função de processamento determina o que será feito, cada vez que um nó é visitado.

P7-09.h – BST_Traverse()

- 1 A função `BST_Traverse()`
 - Recebe o ponteiro ao cabeçalho da árvore e o ponteiro a função de processamento dos nós.
- 2 Chama a função `_traverse()` que percorre os nós da árvore em-ordem e realiza um processamento em cada nó.

```
1  /* ===== BST_Traverse =====
2      Process tree using inorder traversal.
3      Pre   Tree has been created (may be null)
4            process "visits" nodes during traversal
5      Post  Nodes processed in LNR (inorder) sequence
6  */
7  void BST_Traverse (BST_TREE* tree,
8                    void (*process) (void* dataPtr)) 1
9  {
10     // Statements
11     2 _traverse (tree->root, process);
12     return;
13 } // end BST_Traverse
```

- Obs. A função de processamento tem como parâmetro: o ponteiro ao dado de um nó.

Árvores de Busca Binária

_Traverse (_Percorrer)

- A função interna `_traverse()` é uma função recursiva que realiza o percurso da árvore na modalidade em-ordem realizando o processamento dos nós com base em uma função do usuário.

P7-10.h – `_traverse()`

- 1 A função `_traverse()`
 - Recebe um ponteiro ao um nó qualquer e ao endereço da função de processamento.
- 2 Se o nó raiz não é nulo:
 - a Percorre-se a sub-árvore esquerda.
 - b Processa-se o nó raiz.
 - c Percorre-se a sub-árvore direita.
- 3 Caso contrário, retorna.

```
1  /* ===== _traverse =====
2      Inorder tree traversal. To process a node, we use
3      the function passed when traversal was called.
4      Pre   Tree has been created (may be null)
5      Post  All nodes processed
6  */
7  void _traverse (NODE* root,
8                  void (*process) (void* dataPtr)) 1
9  {
10     // statements
11     2 if (root)
12     {
13         a _traverse (root->left, process);
14         b process (root->dataPtr);
15         c _traverse (root->right, process);
16     } // if
17     3 return;
18 } // _traverse
```


Árvores de Busca Binária

BST_Empty (ABB_Vazia)

- A operação `BST_Empty()` verifica se a árvore está vazia. A verificação é realizada de maneira análoga às operações equivalentes em outras estruturas.

P7-11.h – `BST_Empty ()`

- A função simplesmente verifica o campo contador da árvore.
- 1 Se o contador for zero, a árvore está vazia e retorna `True`. Caso contrário, a árvore não está vazia e retorna `False`.

```
1  /* ===== BST_Empty =====
2  Returns true if tree is empty; false if any data.
3  Pre      Tree has been created. (May be null)
4  Returns  True if tree empty, false if any data
5  */
6  bool BST_Empty (BST_TREE* tree)
7  {
8  // Statements
9  1 return (tree->count == 0);
10 } // BST_Empty
```

- Outra forma de verificar: conferir se o ponteiro ao nó raiz é nulo.

`tree->root == NULL`

Árvores de Busca Binária

BST_Full (ABB_Cheia)

- A operação `BST_Full()` verifica se a árvore está cheia. A verificação é realizada de maneira análoga às operações equivalentes em outras estruturas.

P7-12.h – `BST_Full()`

- 1 Primeiro, tenta alocar memória para um novo nó da árvore.
- 2 Se a alocação for bem sucedida, a árvore não está cheia. Libera a memória alocada e retorna False.
- 3 Caso contrário, a árvore está cheia e retorna True.

```
1  /* ===== BST_Full =====
2     If there is no room for another node, returns true.
3     Pre      tree has been created
4     Returns  true if no room for another insert
5             false if room
6  */
7  bool BST_Full (BST_TREE* tree)
8  {
9      // Local Definitions
10     NODE* newPtr;
11
12     // Statements
13     1 newPtr = (NODE*)malloc(sizeof (*(tree->root)));
14     2 if (newPtr)
15     {
16         free (newPtr);
17         return false;
18     } // if
19     3 else
20         return true;
21 } // BST_Full
```

Poderia ser simplesmente:
(NODE*) malloc (sizeof (NODE));

Árvores de Busca Binária

BST_Count (ABB_Contador)

- A operação `BST_Count()` retorna o número de nós na árvore.

P7-13.h – `BST_Count()`

- 1 Retorna o campo contador no cabeçalho da árvore.

```
1  /* ===== BST_Count =====
2      Returns number of nodes in tree.
3      Pre      tree has been created
4      Returns tree count
5  */
6  int BST_Count (BST_TREE* tree)
7  {
8      // Statements
9      1 return (tree->count);
10 }
```

Árvores de Busca Binária

BST_Destroy (ABB_Destruir)

- A operação `BST_Destroy()` é utilizada para eliminar todos os nós da árvore, assim como a estrutura de cabeçalho, quando a árvore não é mais necessária. A operação é realizada de maneira análoga às operações equivalentes em outras estruturas.

P7-14.h – BST_Destroy()

- 1 Verifica se a árvore é válida (o ponteiro não é nulo).
 - Caso afirmativo:
- 2
 - chama a função recursiva `_destroy()` que percorre a árvore e elimina todo os nós.
- 3 Elimina o cabeçalho da árvore.
- 4 Retorna um ponteiro nulo.

```
1  /* ===== BST_Destroy =====
2  Deletes all data in tree and recycles memory.
3  The nodes are deleted by calling a recursive
4  function to traverse the tree in inorder sequence.
5  Pre      tree is a pointer to a valid tree
6  Post     All data and head structure deleted
7  Return   null head pointer
8  */
9  BST_TREE* BST_Destroy (BST_TREE* tree)
10 {
11     // Statements
12     1 if (tree)
13     2     _destroy (tree->root);
14
15     // All nodes deleted. Free structure
16     3 free (tree);
17     4 return NULL;
18 } // BST_Destroy
```

Árvores de Busca Binária

_Destroy (_Destruir)

- A função interna `_destroy()` é uma função recursiva que percorre a árvore segundo a modalidade em-ordem e elimina cada nó da árvore.

P7-15.h – `_Destroy()`

1. Verifica se o nó raiz existe (se o ponteiro não é nulo).
 - Caso afirmativo:
 - a. Destrói a subárvore esquerda;
 - b. Libera o dado alocado ao nó raiz;
 - c. Destrói a subárvore Direita;
 - d. Libera o nó raiz.
 - 2. Caso contrário, retorna.

```
1  /* ===== _destroy =====
2  Deletes all data in tree and recycles memory.
3  It also recycles memory for the key and data nodes.
4  The nodes are deleted by calling a recursive
5  function to traverse the tree in inorder sequence.
6  Pre      root is pointer to valid tree/subtree
7  Post     All data and head structure deleted
8  Return   null head pointer
9  */
10 void _destroy (NODE* root)
11 {
12     // Statements
13     1 if (root)
14     {
15         a _destroy (root->left);
16         b free (root->dataPtr);
17         c _destroy (root->right);
18         d free (root);
19     } // if
20     2 return;
21 } // _destroy
```

Árvores de Busca Binária

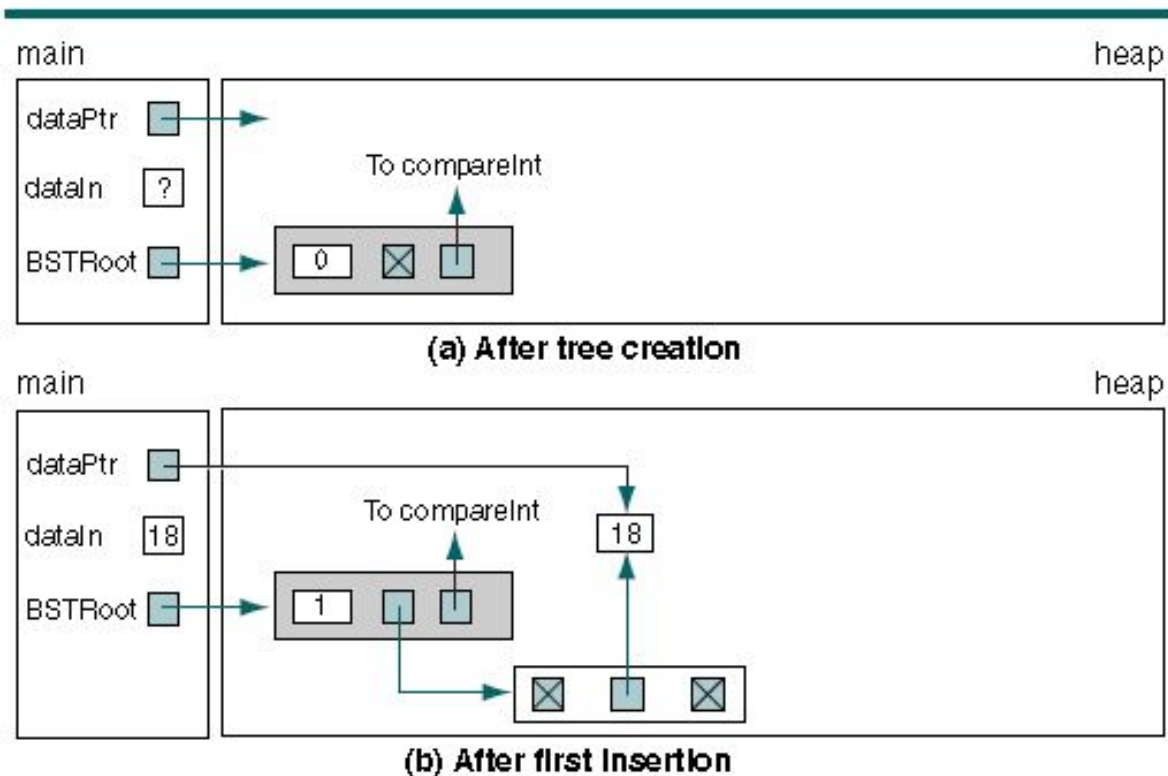
Aplicações

- São apresentadas dois exemplos de uso de árvores:
 - A primeira é uma aplicação simples que armazena números inteiros.
 - A segunda é uma aplicação que armazena a informação dos estudantes em uma turma e os identifica pelo número de matrícula (número id.)
- Vale observar em ambos casos a definição das funções *compare()* e *process()*, que somente podem ser definidas a nível de aplicação mas que são utilizadas pelas funções do tipo abstrato de dados.

Árvores de Busca Binária

Aplicação: ABB de Inteiros

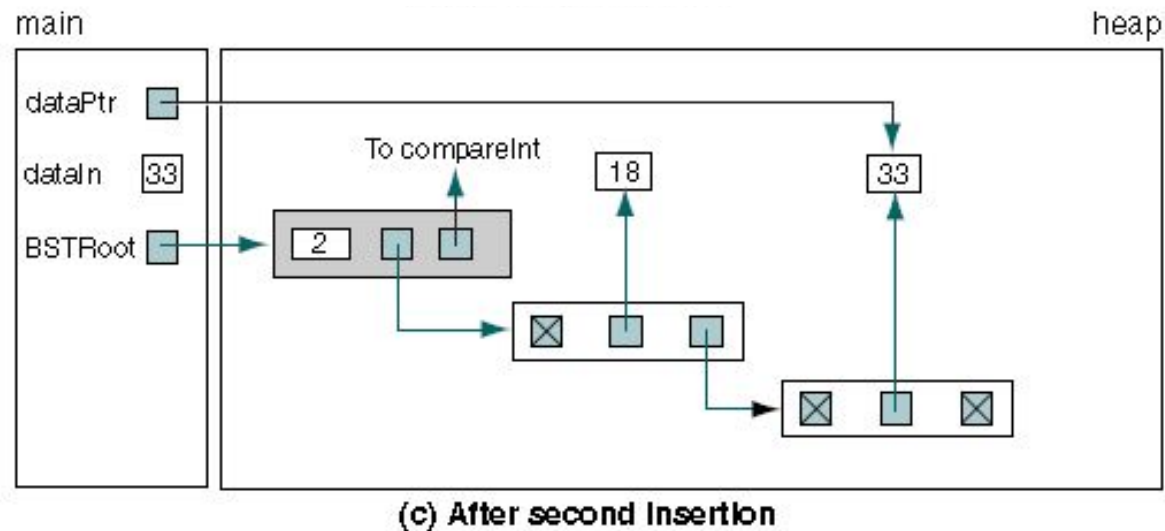
- Este programa lê uma sequência de números inteiros desde o teclado e insere-os (função `BST_Insert`), em uma árvore de busca binária. Logo, os números são impressos mediante a função `BST_Traverse`.



Árvores de Busca Binária

Aplicação: ABB de Inteiros

- (Continuação...)



Insertions into a BST

Árvores de Busca Binária

Aplicação: ABB de Inteiros

P7-16.c – Programa principal: main()

- O código ilustra a criação de uma árvore de busca binária de inteiros.
- Inclui o TAD de Árvore de Busca Binária: P7-BST-ADT.h
- Define as funções:
 - comparação compareInt()
 - processamento printBST()
- Cria a árvore de busca binária inteira usando a operação:
 - BST_Create()

```
1  /* This program builds and prints a BST.
2      Written by:
3      Date:
4  */
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include "P7-BST-ADT.h"
8
9  // Prototype Declarations
10 int compareInt (void* num1, void* num2);
11 void printBST  (void* num1);
12
13 int main (void)
14 {
15     // Local Definitions
16     BST_TREE* BSTRoot;
17     int*      dataPtr;
18     int       dataIn = +1;
19
20     // Statements
21     printf("Begin BST Demonstation\n");
22
23     BSTRoot = BST_Create (compareInt);
24
25     // Build Tree
26     printf("Enter a list of positive integers;\n");
27     printf("Enter a negative number to stop.\n");
```

← Cria a ABB

Árvores de Busca Binária

Aplicação: ABB de Inteiros

- O código possui um loop `do .. while` que realiza a leitura de dados inteiros enquanto forem não negativos.
- Cria um novo apontado por `dataPtr`.
- Insere o novo nó na árvore usando `BST_Insert()`.
- Realiza o percurso da árvore de busca binária em-ordem usando `BST_Traverse()` e a função de processamento `printBST()`.

P7-16.c – Programa principal: `main()`, continuação...

```
28
29
30     do
31     {
32         printf("Enter a number: ");
33         scanf ("%d", &dataIn);
34         if (dataIn > -1)
35         {
36             dataPtr = (int*) malloc (sizeof (int));
37             if (!dataPtr)
38             {
39                 printf("Memory overflow in add\n"),
40                 exit(100);
41             } // if overflow
42             *dataPtr = dataIn;
43             BST_Insert (BSTRoot, dataPtr);
44         } // valid data
45     } while (dataIn > -1);
46
47     printf("\nBST contains:\n");
48     BST_Traverse (BSTRoot, printBST);
49
50     printf("\nEnd BST Demonstration\n");
51     return 0;
52 } // main
```

Insere o novo
nó na árvore

Percorre a
ABB usando
em-ordem

Árvores Busca Binária

Aplicação: ABB de Inteiros

P7-16.c – Função de Comparação Inteira: `compareInt()`

- O código mostra a função de comparação `compareInt()` para dados inteiros definida na aplicação.

```
53  /* ===== compareInt =====
54      Compare two integers and return low, equal, high.
55      Pre num1 and num2 are valid pointers to integers
56      Post return low (-1), equal (0), or high (+1)
57  */
58  int compareInt (void* num1, void* num2)
59  {
60      // Local Definitions
61      int key1;
62      int key2;
63
64      // Statements
65      key1 = *(int*)num1;
66      key2 = *(int*)num2;
67      if (key1 < key2)
68          return -1;
69      if (key1 == key2)
70          return 0;
71      return +1;
72  } // compareInt
```

Valor inteiro apontado por num1

Valor inteiro apontado por num2

Árvores Busca Binária


Aplicação: ABB de Inteiros

P7-16.c – Função de Processamento: printBST()

- O código mostra a função de processamento `printBST()` definida na aplicação para imprimir dados inteiros.

```
73
74  /* ===== printBST =====
    Print one integer from BST.
    Pre num1 is a pointer to an integer
    Post integer printed and line advanced
75
76  */
77
78 void printBST (void* num1)
79 {
80  // Statements
81  printf("%4d\n", *(int*)num1);
82  return;
83 } // printBST
84
```

Valor inteiro
apontado por num1



Árvores de Busca Binária

Aplicações – Árvore de Inteiros: Resultado

- O código mostra como resultado o ingresso de dados inteiros em uma árvore de busca binária e a impressão desses dados pelo percurso em-ordem.

```
Results:
Begin BST Demonstation
Enter a list of positive integers;
Enter a negative number to stop.
Enter a number: 18
Enter a number: 33
Enter a number: 7
Enter a number: 24
Enter a number: 19
Enter a number: -1

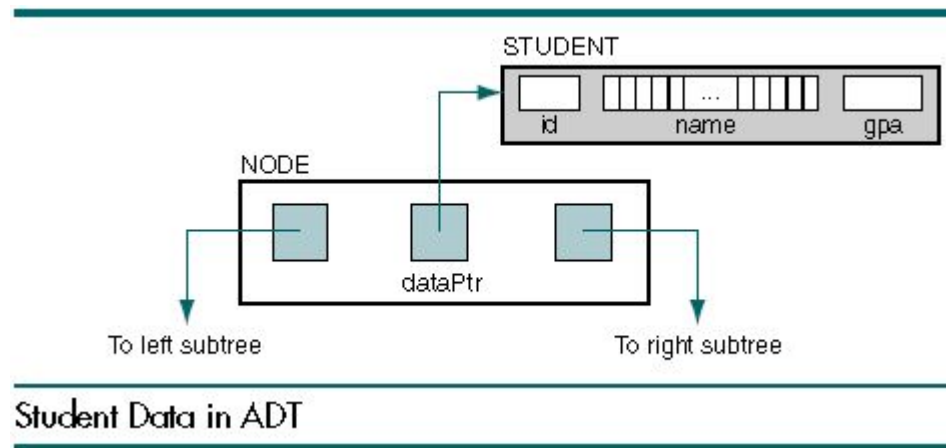
BST contains:
    7
   18
   19
   24
   33

End BST Demonstration
```

Árvores de Busca Binária

Aplicações – Árvore de Estudantes: Prog. 7-17

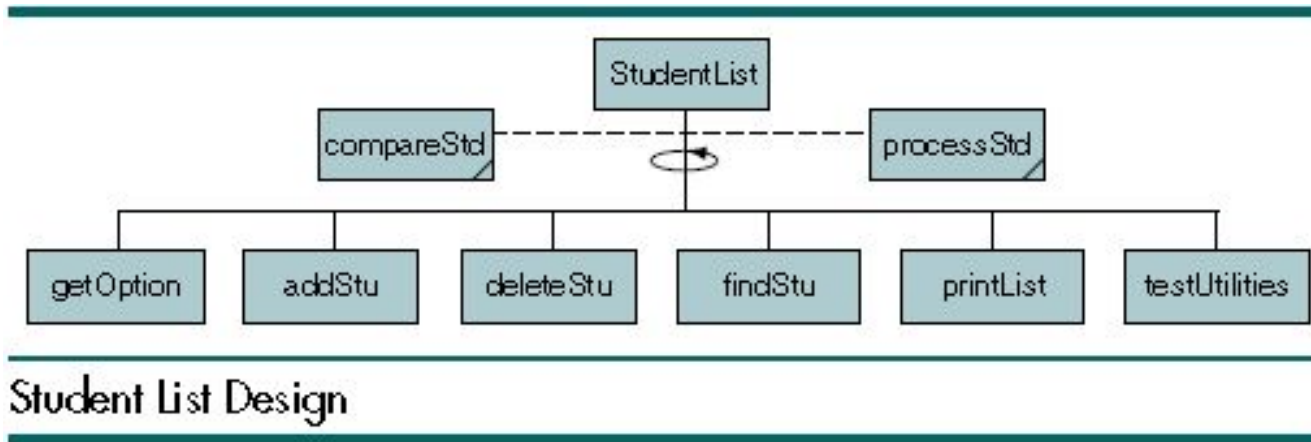
- O programa armazena as informações de uma lista de estudantes, entre elas:
 - ID. do estudante;
 - Nome do estudante;
 - Coeficiente de Rendimento.
- Os estudantes podem ser adicionados, eliminados a partir do teclado, recuperados individualmente ou listados.
- As estruturas usadas para os estudantes e a árvore são ilustradas.



Árvores de Busca Binária

Aplicações – Árvore de Estudantes: Prog. 7-17

- A estrutura do programa de aplicação compreende as seguintes funções:



- Ver código compartilhado no Classroom.

Referências

- Gilberg, R.F. e Forouzan, B.A. Data Structures_A Pseudocode Approach with C. Capítulo 7. Binary Search Trees. Segunda Edição. Editora Cengage, Thomson Learning, 2005.