Fundamentals of Programming 2

Binary Search Trees

Arkadiusz Chrobot

Department of Information Systems

May 30, 2025

Outline

Introduction

Trees and *binary trees* are non-linear data structures used to store hierarchically ordered data. In this case non-linearity means that each node has at most one ”predecessor”, called a *parent* and an unlimited number of ”successors”, called *children*. For binary trees the number of children that a single node can have is limited to two. It is worth to notice that trees and binary trees are *distinct types of data structures*. They are also subclasses of *graphs*, which will be discussed in the next lecture.
Today's lecture is about *binary search trees* (BSTs), that are a subtype of binary trees. Some definitions concerning binary trees and trees are given in the next slides.

Definitions

The Binary Tree

The binary tree is a finite set of nodes that is either empty or contains a node, called the *root*, and two disjoint binary trees, called the *left* and the *right* subtree. If these subtrees are not empty then their roots are called, respectively, the *left child* and the *right child* of the binary tree's root. Conversely, the root is called their *parent*. The *degree* of a node (the number of its children) in the binary tree is limited to two. Nodes with a non-zero degree are called *internal nodes*, and nodes with a degree equal to zero are called *leaves*. Each node also has a property called a *level*. The level of the root is zero and the level of every other node is greater by one than the level of the root in the smallest subtree that the node is part of. The *height* of the tree is greater by one than the maximal level of its nodes. The binary tree is also an *ordered tree* or a *flat tree*, because the order of its subtrees is important.

Definitions

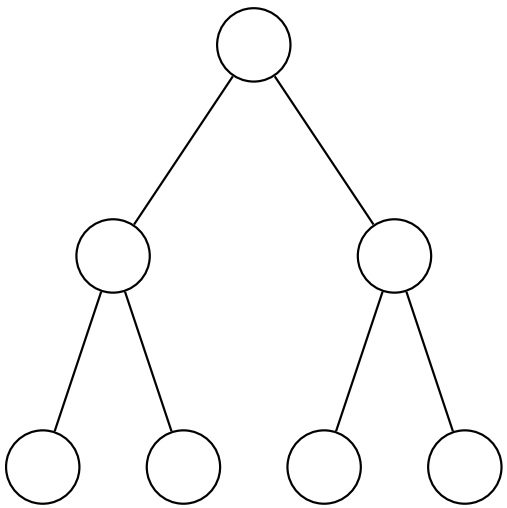
The Binary Tree And The Tree

A *tree* differs from a binary tree in the respect, that it *always* has at least one node, and the degree of each node in the trees is not limited.

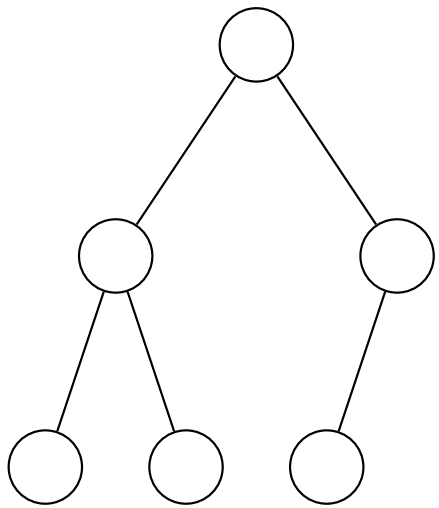
Definitions

The Full Binary Tree And The Complete Binary Tree

If each node in a binary tree has a degree of either two or zero, then it is a *full binary tree*. If a binary tree of a given height has all possible nodes, perhaps with the exception of the last level, then it is the *complete binary tree*.



A full binary tree



A complete binary tree

It's worth to notice that in Computer Science the trees grow upside-down

Definitions

Binary Search Tree (BST)

The binary search tree (BST) is a binary tree that each node stores a data item called a *key*. With the key *may be* associated another data item, called a *value*. Such a binary tree can be used to build a dictionary. It is another data structure where the key identifies the value. In the BST the order of the keys is determined by the following principle:

The order of keys in the BST

Let x be a node in the BST. If y is a node in the left subtree of x, then $\boldsymbol{key}(x) \geq \boldsymbol{key}(y)$. If y if a node in the right subtree of x then $\boldsymbol{key}(x) \leq \boldsymbol{key}(y)$.

The Implementation of BST

In the lecture, an implementation of the BST in a form of a dynamic data structure is presented, that adheres to the definition given in the previous slide. It stores only keys. Please notice however, that the definition is a little ambiguous. If the operation of adding a node to the BST should work according to it, then it would have a problem when the key in the new node was already in the tree — should the new node be added on the left side of the node with the same key or on the right side? To avoid this issue the keys in the demonstrated BST are unique. It is the most common approach to this issue.

The Implementation of BST

The Data Type of BST Node

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<time.h>
4
5  struct bst_node
6  {
7          int key;
8          struct bst_node *left_child, *right_child;
9  } *root;
```

The Implementation of BST

The Data Type of BST Node

The header files included to the program (lines 1–3) provide declarations of functions, that allow the program to communicate with the user, to allocate and deallocate the heap memory and to use pseudorandom numbers generator (PRNG).
Please notice, that the definition of the data type for a single BST node (lines 5–6) reminds a lot, a similar construct from the program that uses the doubly linked list. However, the purpose of the structure members is different. The *key* member stores the key, which is an integer of the `int` type. In the line no. 8 are declared two pointer fields, called `left_child` and `right_child`. They are used for storing the addresses of, respectively, the left and the right child of a given node. If the node is a leaf (has no children) then the value of these fields will be `NULL`. In an inner node, only one pointer field can be an empty pointer.

The Implementation of BST

The Data Type of BST Node

In some implementations of the BST each node has another pointer field, usually called `parent`, that points to the node's parent. Only in the root this field is an empty pointer.
In the line no. 9 of the slide no. 9 is declared a global variable named `root`. It is a pointer to the root of the BST. Its default value is `NULL` meaning that the BST also is initially empty.

The add_node() Function

```
1  void add_node(struct bst_node **node, int number)
2  {
3          while(*node && (*node)->key != number)
4                  if((*node)->key > number)
5                          node = &(*node)->left_child;
6                  else
7                          node = &(*node)->right_child;
8          if(!*node) {
9                  *node = (struct bst_node
                    ↪  *)malloc(sizeof(struct bst_node));
10                 if(*node) {
11                         (*node)->key = number;
12                         (*node)->left_child =
                            ↪  (*node)->right_child = NULL;
13                 }
14         }
15 }
```

The `add_node()` Function

The `add_node()` function is responsible for adding a new node with a given key to the BST. It returns no value, but it has two parameters. The first of them is a pointer to a pointer called `node`. This parameter is used for passing the address of the `root` pointer. The second one is a variable of the `int` type called `number`. By this parameter the function gets the key that should be stored in the new node.

The `add_node()` Function

The objective of the `while` loop (lines 3–7) is finding a place in the BST for the new node, using the key the node will store. The loop is performed as long as the pointer pointed by `node` is not empty and the node pointed by this pointer has a different key than the one that will be stored in the new node. If both these conditions are met, then the loop checks if the key in the node, whose address is stored in the pointer pointed by the `node` parameter, is greater than the key for the new node (line no. 4). If so, then the address of that node's `left_child` field is assigned to the `node` parameter (line no. 5). The smaller keys are stored in the left subtree and this is where the new node should be added. If the condition from the 4^{th} line is not fulfilled, then the address of the node's `right_child` field is assigned to the `node` parameter, meaning that the new key should be in the right subtree, where larger keys are stored.

The `add_node()` Function

After the `while` loop stops, the `add_node()` verifies, if the pointer pointed by the `node` parameter **is empty** (line no. 8). If not, then it means that the loop has found a node that has the same key as the one that is stored in the `number` parameter. The keys in the BST have to be unique, so the `add_node()` will exit in that case without doing anything else. However, if the condition in the 8^{th} line is met, then the function will try to allocate memory for the new node (line no. 9). It checks in the line no. 10 whether the operation was successful, and if so, it assigns the key stored in the `number` parameter to the new node (line no. 11) and initializes both of the node's pointer fields with the `NULL` value (line no. 12).

Please notice, that if the `while` loop stops at once, then it means that the BST is empty and `add_node()` will add the first node to the BST. If, however, the loop performs several iterations and stops when the `*node` expression in the 3^{rd} line is false, then it means that the `node` stores an address of one of the pointer fields of the node that should be the parent of the new node.

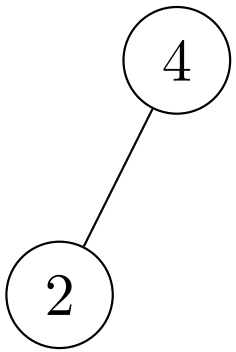
Insertion in BST

The next slide shows an animation that illustrates conceptually how several nodes are added to the BST. The order of the nodes' keys is as follows: 4, 2, 1, 3, 5.

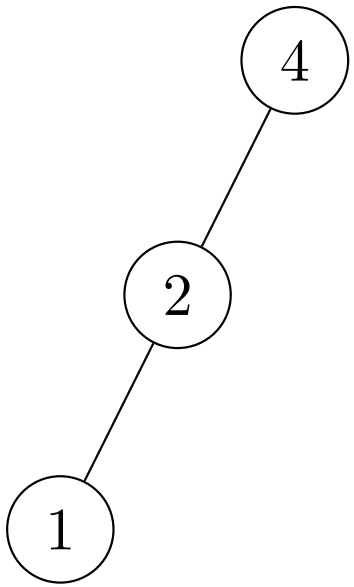
Insertion in BST

Insertion in BST

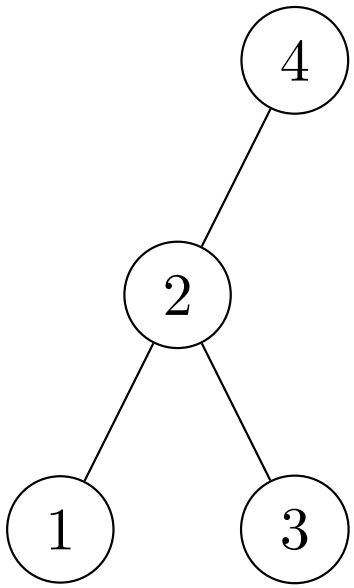
Insertion in BST



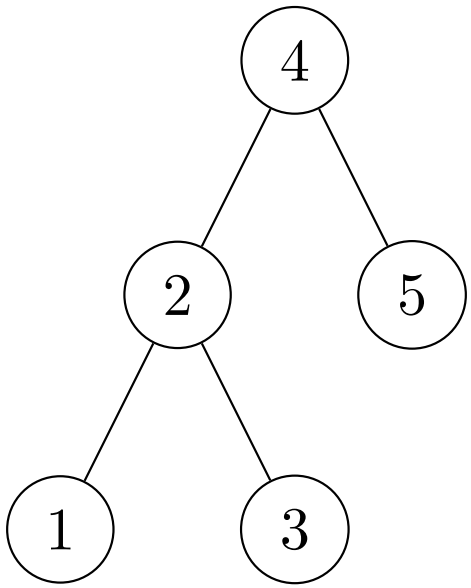
Insertion in BST



Insertion in BST



Insertion in BST



Binary Tree Traversal

There are three recursive algorithms for traversing a binary tree (in fact any tree):

1. *in-order* traversal,
2. *pre-order* traversal,
3. *post-order* traversal.

In all these algorithms the left subtree is traversed recursively before the right subtree. Only the root is visited in a different order in each of them. Most of other binary tree algorithms are based on these three. The binary tree traversal algorithms are usually implemented in a form of recursive functions.

Binary Tree Traversal

In-Order Traversal

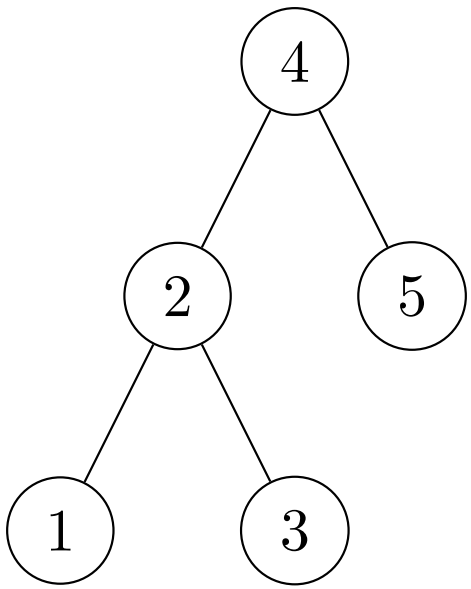
The in-order traversal algorithm is as follows:

1. traverse the left subtree recursively,
2. visit the root,
3. traverse the right subtree recursively.

The next slide shows an example BST and the outcome of a function that displays the keys stored in this BST using the in-order algorithm.

Binary Tree Traversal

In-Order Traversal



Outcome

1, 2, 3, 4, 5

The print_bst_inorder() Function

```
1  void print_bst_inorder(struct bst_node *node)
2  {
3          if(node) {
4                  print_bst_inorder(node->left_child);
5                  printf("%4d ",node->key);
6                  print_bst_inorder(node->right_child);
7          }
8  }
```

The `print_bst_inorder()` Function

The `print_bst_inorder()` function uses the *in-order* traversal algorithm to print the keys stored in the BST on the screen. It doesn't return any value, but it has a parameter named `node` which is a pointer to a node. The argument for this function is the address of the tree root.

The function first checks if the `node` is not an empty pointer (line no. 3). If it is so, then `print_bst_inorder()` invokes itself recursively for the left child of the node pointed by the `node` parameter and, in consequence, for the entire left subtree of that node (line no. 4). When the function returns from the recursive calls, then it prints the key stored in the node pointed by the `node` parameter (line no. 6) and once again it invokes itself recursively, but this time for the right child of the node, and in consequence, for its whole right subtree.

The `print_bst_inorder()` Function

It's worth noting that the sequence of recursive calls ends, when the function is invoked for a non-existing node. In that case, the condition in the 3^{rd} line is not satisfied, this instance of the function immediately exits and the control returns to the function's earlier instance.

Binary Tree Traversal

Pre-Order Traversal

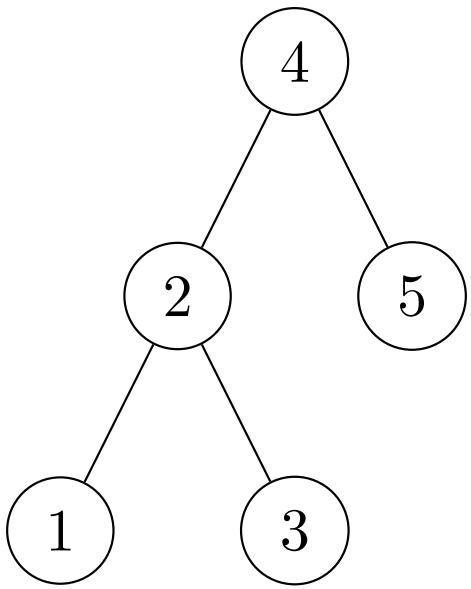
The *pre-order* traversal algorithm is as follows:

1. visit the root,
2. traverse recursively the left subtree,
3. traverse recursively the right subtree.

This algorithm, unlike the *in-order* traversal algorithm, visits the root, before traversing the subtrees. The next slide shows an example BST and the outcome of a function that uses the *post-order* traversal algorithm to print keys stored in this tree.

Binary Tree Traversal

Pre-Order Traversal



Outcome

4, 2, 1, 3, 5

The print_bst_preorder() Function

```
1  void print_bst_preorder(struct bst_node *node)
2  {
3          if(node) {
4                  printf("%4d ",node->key);
5                  print_bst_preorder(node->left_child);
6                  print_bst_preorder(node->right_child);
7          }
8  }
```

The `print_bst_preorder()` Function

The `print_bst_preorder()` is similar to the `print_bst_inorder()` function. The only two differences are the name and that the former is invoked recursively (lines 5–6) after the key from the node currently pointed by the `node` parameter is displayed (line no. 4).

Binary Tree Traversal

Post-Order Traversal

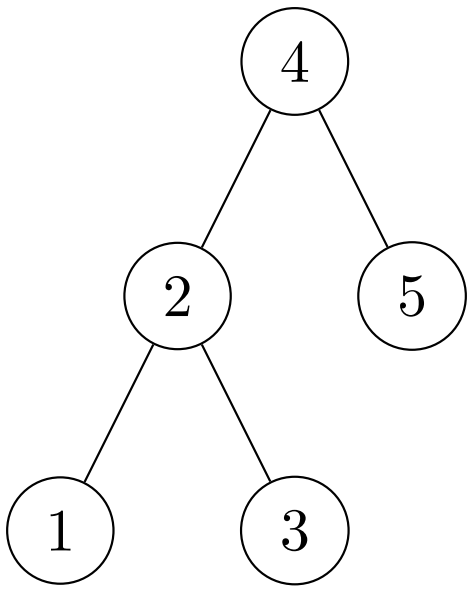
The post-order traversal algorithm is defined as follows:

1. traverse recursively the left subtree,
2. traverse recursively the right subtree,
3. visit the root.

It differs from the two previous in that it traverses both subtrees (first the left one then the right) before visiting the root. The next slide shows an example BST and the outcome of a function that applies the algorithm to display the keys stored in that BST.

Binary Tree Traversal

Post-Order Traversal



Outcome

1, 3, 2, 5, 4

The print_bst_postorder() Function

```
1  void print_bst_postorder(struct bst_node *node)
2  {
3          if(node) {
4                  print_bst_postorder(node->left_child);
5                  print_bst_postorder(node->right_child);
6                  printf("%4d ",node->key);
7          }
8  }
```

The `print_bst_preorder()` Function

The `print_bst_postorder()` function is also similar to the two previously presented. It has however a different name and it invokes itself recursively for the children of the node pointed by the `node` parameter (lines 4–5), before printing the key stored in that node on the screen.

Number of BST Nodes

Displaying the keys stored in the BST is not the only use of the binary tree traversal algorithms. Let's consider how to count the nodes of a BST. It turns out that the Divide-And-Conquer method could be useful in this case:

- if a tree is empty, then the number of its nodes is zero,
- if the tree is not empty, then the total number of its nodes is the sum of the number of nodes in its left subtree, plus its root (one node) and of the number of nodes in its right subtree.

Please notice, that the last point corresponds to the *in-order* traversal algorithm, although any of the binary tree traversal algorithms could be applied, due to the commutativity of addition.

The count_nodes() Function

```
1  unsigned int count_nodes(struct bst_node *node)
2  {
3          if(node)
4                  return count_nodes(node->left_child) +
                    ↪  1 + count_nodes(node->right_child);
5          else
6                  return 0;
7  }
```

The `count_nodes()` Function

The function, presented in the previous slide, counts the number of BST nodes using the algorithm described in the slide no. 32. It returns a number of the `unsigned int` type (the number of nodes is always natural) and takes the address of the BST root as an argument.

The find_minimum() And find_maximum() Functions

```
1  struct bst_node *find_minimum(struct bst_node *node)
2  {
3          while(node && node->left_child)
4                  node = node->left_child;
5          return node;
6  }
7
8  struct bst_node *find_maximum(struct bst_node *node)
9  {
10         while(node && node->right_child)
11                 node = node->right_child;
12         return node;
13 }
```

The `find_minimum()` And `find_maximum()` Functions

Finding a BST node with a minimum key is easy. It is the leftmost node. Similarly, the node with the maximum key is the rightmost node. The first function presented in the previous slide searches for the node with minimal key and returns its address. It takes as an argument the address of BST's root, which is passed by the `node` parameter. The `while` loop inside the function checks if BST node pointed by the `node` parameter exists and if its `left_child` field is not an empty pointer (line no. 3). If both expressions in the loop condition are true, then the address from the `left_child` field is assigned to the `node` parameter (line no. 4). It means that after the statement is performed, the `node` will point to the left child of the current node. The loop stops when it locates a node without a left child. The `find_minimum()` function returns then the address of this node, because it is the BST leftmost node (line no. 5). Please notice, that the function returns `NULL` only in one case — when it is invoked for an empty BST.

The `find_minimum()` And `find_maximum()` Functions

The `find_maximum()` function is similar to `find_minimum()`, but in the second expression of the `while` loop condition it checks if the `right_child` field of the node pointed by the `node` parameter is an empty pointer. If not, then it assigns the address stored in that field to the `node` parameter (line no. 11). When the loop stops the function returns the address of the node pointed by the `node` parameter, because it is the rightmost BST node with the maximal key.

The locate() Function

```
1  struct bst_node *locate(struct bst_node *node, int
    ↪  number)
2  {
3          while(node && node->key != number)
4                  if(node->key > number)
5                          node = node->left_child;
6                  else
7                          node = node->right_child;
8          return node;
9  }
```

The `locate()` Function

The objective of the `locate()` function is to find a BST node that stores a key passed by the `number` parameter. It also takes another argument, which is the address of BST root, passed by the `node` parameter. The function returns the address of the node with the specified key or `NULL` if such a node doesn't exist. Please notice, that the `while` loop in this function (lines 3–7) is quite similar to the `while` loop in the `add_node()` function. This time however, the loop uses a first level ("regular") pointer. If the pointer is not empty and points to the node without the given key (line no. 3) then the function checks if the key in that node is greater than the specified key (line no. 4). If so, then it assigns the address of the node's left child to the `node` parameter (or `NULL`, if the child doesn't exist), otherwise it assigns the address of the node's right child (or `NULL`, if the child doesn't exist). After the loop stops the value stored in the `node` parameter is returned by the function.

Searching For a Node With a Given Key

Performance

The main advantage of the BST is the time complexity of locating a node with a given key. It is proportional to the height of the tree. If the *shape* of the tree is close to the shape of a complete tree, then its height is estimated as $log_2(n)$, where n is the total number of nodes in the BST.

The remove_bst_nodes() Function

```
1  void remove_bst_nodes(struct bst_node **node)
2  {
3          if(*node) {
4                  remove_bst_nodes(&(*node)->left_child);
5                  remove_bst_nodes(&(*node)->right_child);
6                  free(*node);
7                  *node=NULL;
8          }
9  }
```

The `remove_bst_nodes()` Function

The function that removes all nodes from a binary tree uses the *post-order* traversal algorithm, as it causes the removal to start from the leaves. This assures that there is no danger of passing addresses of non-existent node's fields to the recursive calls of the function. After the function exits, the value of the `root` pointer should be `NULL`. That's why `remove_bst_nodes()` assigns this value to the dereferenced `node` pointer (line no. 8). This means, that the `NULL` value is assigned to each pointer field of a node, before this node is deleted, and eventually it will be assigned to the `root` pointer.

Deletions in BST

The goal of deleting a BST node is actually to remove a key from this data structure. It is a quite complex operation. The function that implements it should properly handle the following cases:

1. there is no node of a given key — this case doesn't require any deletion to be preformed,
2. the node to be deleted doesn't have any children — the node may be removed, but the `NULL` value should be assigned to its parent's `left_child` field (if the node to delete is its left child) or to the `right_child` (if the node to delete is its right child) field,
3. the node to be deleted has only one child — before the node is deleted, the address of its child should be assigned to its parent's pointer field that now points to this node,
4. the node to be deleted has two children — this is the hardest case — the node cannot be just deleted, another BST node has to be found that will be removed instead.

Deletions in BST

The ”another node” mentioned in the previous slide is either the successor or the predecessor of the node that should be deleted. The successor is a node with a key that is directly greater than the key in the node to be deleted. The predecessor stores a key that is directly smaller, than the key stored in the node to be deleted. Also, the successor of a node is its right subtree leftmost node and the predecessor of a node is its left subtree rightmost node. Before the predecessor/successor can be deleted, the key from this node has to be assigned to the node that was originally intended to be deleted. In the implementation of this operation, that is discussed in this lecture, the predecessor of the node with two children is always deleted. A function that unlinks the predecessor from the BST is described as first, and then the function that handles all the cases of deleting a single node from the BST.

The isolate_predecessor() Function

```
1  struct bst_node *isolate_predecessor(struct bst_node
    ↪  **node)
2  {
3          while((*node)->right_child)
4                  node = &(*node)->right_child;
5          struct bst_node *predecessor = *node;
6          *node = (*node)->left_child;
7          return predecessor;
8  }
```

The `isolate_predecessor()` Function

The `isolate_predecessor()` function returns the address of the predecessor of the node originally intended to be deleted. It is invoked only from within the function that deletes a BST node and only if the node has two children. As an argument it takes the address of a pointer that points to the left child (the root of the left subtree) of the node originally intended to be deleted. The `while` loop (lines 3–4) looks for the rightmost node in the left subtree of the node originally intended to be deleted. Please notice, that in each iteration of the loop the address of the pointer field that points to the current node right child is assigned to the `node` parameter. The loop stops when it finds a node that has no right child. It is the predecessor of the node originally intended to be deleted. The function assigns its address to the local pointer named `predecessor` (line no. 5). After that it assigns the address stored in the predecessor's `left_child` field to the pointer pointed by the `node` parameter (line no. 6).

The `isolate_predecessor()` Function

The predecessor doesn't have the right child for sure, however it may have the left one. If this child exists, then its address should be assigned to the pointer field of the predecessor's parent, that points to the predecessor. This assures that the child won't be lost, possibly together with its subtrees. If the child doesn't exist then the `NULL` value should be assigned to the pointer field of the predecessor's parent, that pointed to the predecessor — after the predecessor is unlinked its parent won't have this child.
When the predecessor is unlinked from the BST, then the function returns the predecessor's address and exits (line no. 7).

The delete_node() Function

```
1  void delete_node(struct bst_node **node, int number)
2  {
3          while(*node && (*node)->key != number)
4                  if ((*node)->key > number)
5                          node = &(*node)->left_child;
6                  else
7                          node = &(*node)->right_child;
8          if (*node) {
9                  struct bst_node *node_to_delete = *node;
10                 if(!node_to_delete->left_child)
11                         *node = (*node)->right_child;
12                 else if(!node_to_delete->right_child)
13                         *node = (*node)->left_child;
14                 else {
15                         node_to_delete =
                           ↪  isolate_predecessor(&(*node)->left_child);
16                         (*node)->key = node_to_delete->key;
17                 }
18                 free(node_to_delete);
19         }
20 }
```

The `delete_node()` Function

The `delete_node()` function is responsible for deleting a BST node that stores a given key. It doesn't return any value, but takes two arguments: the address of a pointer that points to the BST root and the key that should be stored in the node to be deleted. Please notice, that the `while` loop (lines 3–7) is the same as in the `add_node()` function. However, when the loop stops the `delete_node()` function, unlike `add_node()`, checks if the node, that the loop was supposed to find, exists (line no. 8). If not, then the function exits — there is no node in the BST that stores the given key, so there is nothing to delete. If the node exists, then the function stores its address in a local pointer named `node_to_delete` (line no. 9). Then it checks if the left child of that node **doesn't exist** (line no. 10). If it is so, then the node still can have a right child. That's why the `delete_node()` function assigns the address stored in the `right_child` field of the node to be deleted to the pointer pointed by the `node` parameter (line no. 11).

The `delete_node()` Function

That way, if the node has a parent, the proper field pointer of the node's parent will start pointing to the node's right child. If, however, the node's right child doesn't exist, then the pointer field will get the `NULL` value, which in this case is also correct — after the node that stores the given key is deleted its parent won't have this child. If the condition in the 10^{th} line is not met then the function checks the condition in the 12^{th} line, meaning if the right child of the node to be deleted **doesn't exist**. If the condition is met, then it is known for sure, that the left child exists (because the condition in the 10^{th} line is not met). In that case the `delete_node()` function assigns the address of this child to the pointer pointed by the `node` parameter (line no. 13).
If both conditions in the lines no. 10 and no. 12 are not met, then the node to be deleted has two children. In that case it has to be replaced by its predecessor.

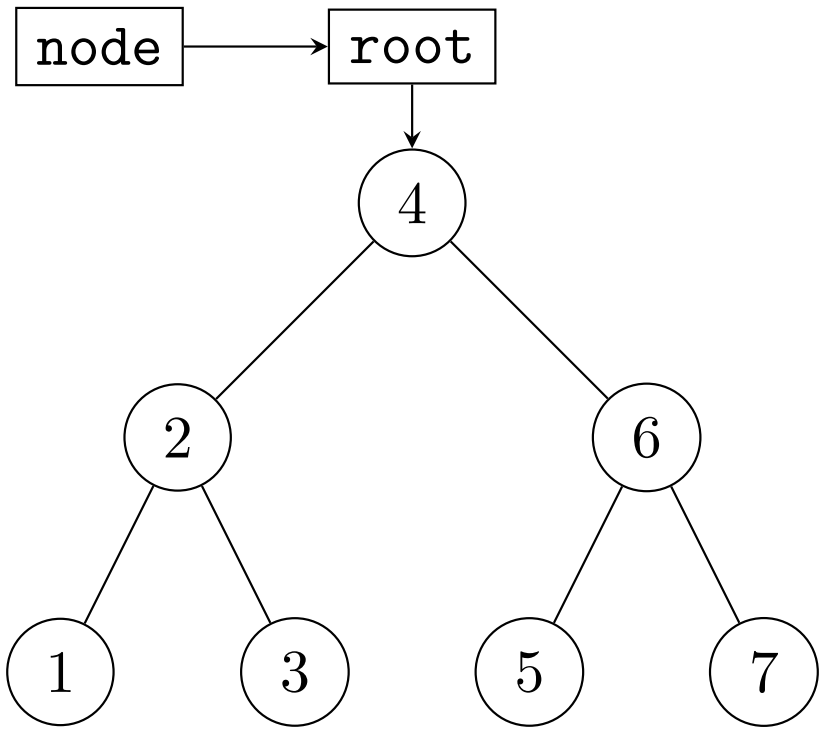
The `delete_node()` Function

That's why, in the 15^{th} line the `delete_node()` function invokes `isolate_predecessor()` passing, as an argument to that latter function, the address of the `left_child` filed of the node, which originally should be deleted. The result of `isolate_predecessor()` is assigned to the `node_to_delete` pointer (line no. 15). The outcome of that function is the address of the predecessor of the node originally intended to be deleted. The key from the predecessor is assigned to the node that originally was to be deleted (line no. 15).

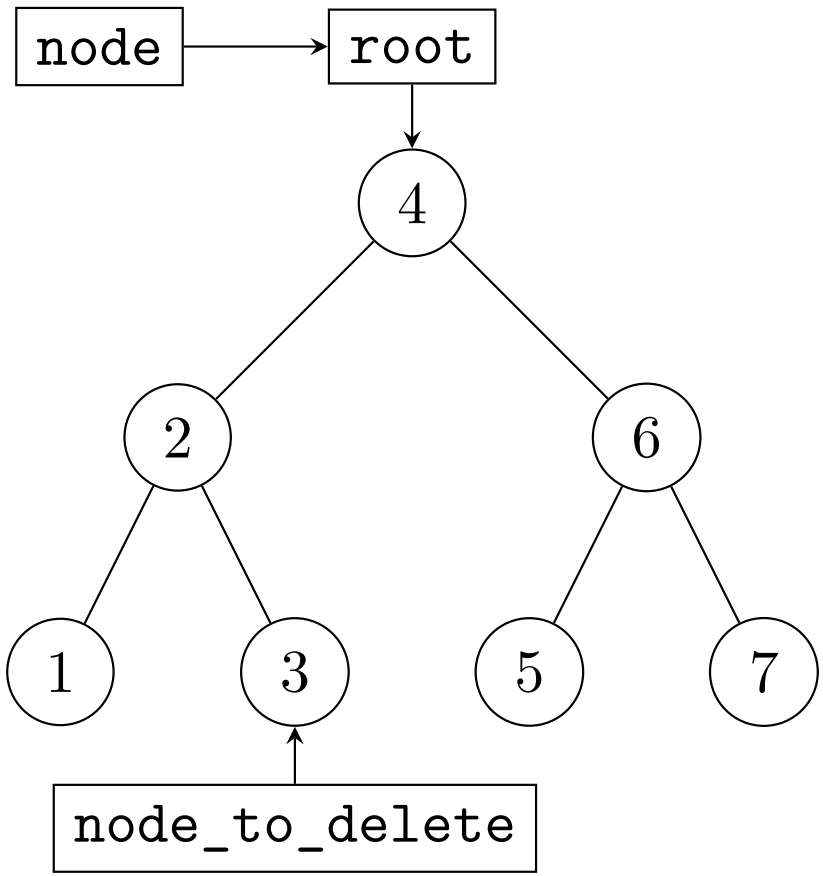
In all the cases that `delete_node()` handles, it eventually invokes the `free()` function to release the memory allocated for the BST node pointed by the `node_to_delete` (line no. 18) and quits.

The next slide illustrates a simple case of deleting a key stored in a node that has two children.

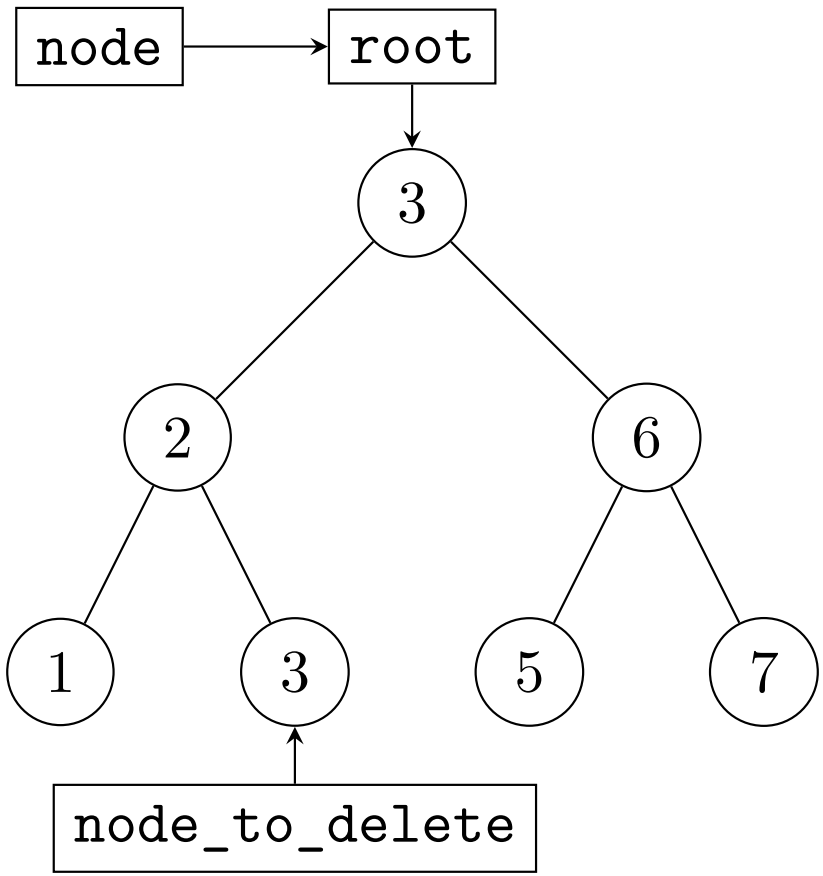
Deleting a Node With Two Children



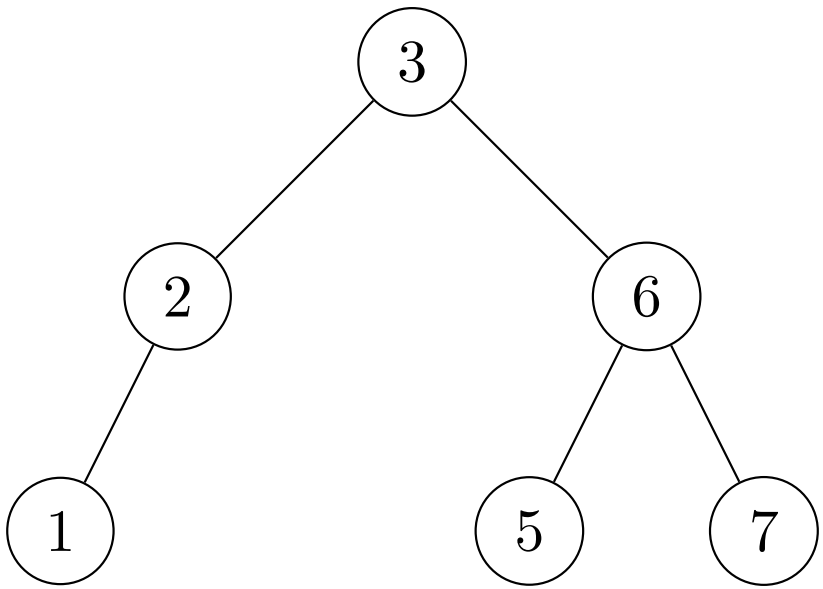
Deleting a Node With Two Children



Deleting a Node With Two Children



Deleting a Node With Two Children



The main() Function

First Part

```
1  int main(void)
2  {
3          srand(time(0));
4          for(int i=0;i<10;i++)
5                  add_node(&root, -10 + rand() % 21);
6          printf("Number of nodes in the binary search tree:
           ↪  %u\n", count_nodes(root));
7          print_bst_inorder(root);
8          puts("");
9          print_bst_preorder(root);
10         puts("");
11         print_bst_postorder(root);
12         puts("");
```

The main() Function

First Part

First, the `main()` function initializes the pseudorandom number generator (line no. 3) and in the `for` loop it tries to insert into the BST 10 nodes, which keys are integers randomly chosen from the $[-10, 10]$ interval (lines 4–5). Then, in the 6^{th} line, it displays the number of nodes in the BST, which is returned by the `count_node()` function. In the next lines the `main()` calls functions that display keys stored in the BST using, respectively the *in-order*, *pre-order* and *post-order* traversal algorithms. Please notice, that after each of the function exits, the `put()` is called to move the cursor to the next line on the screen.

The main() Function

Second Part

```
1           if(root) {
2                   printf("Minimum key: %d\n",
                     ↪  find_minimum(root)->key);
3                   printf("Maximum key: %d\n",
                     ↪  find_maximum(root)->key);
4           }
5           int number_to_delete = -10+rand()%21;
6           printf("Key to be deleted: %d\n", number_to_delete);
7           struct bst_node *result = locate(root,
             ↪  number_to_delete);
8           if(result)
9                   printf("The key is in the binary search tree:
                     ↪  %d\n", result->key);
10          else
11                  puts("The key is not in the binary search
                     ↪  tree.");
```

The main() Function

Second Part

The `main()` function also calls functions that find the nodes storing the minimal and maximal key (lines 1–4). Because these functions return `NULL` if and only if they are invoked on an empty BST, then the `main()` checks in the line no. 1, if it is not the case. When the condition is met, then the pointer returned by these functions can be immediately dereferenced and the keys stored in nodes pointed by them can be displayed (lines 2–3). Next the `main()` function randomly chooses a key to be removed and stores it in the `number_to_delete` local variable (line no. 6). The key is then displayed and the `locate()` function is invoked, which tries to find a node that stores such a key (line no. 7). The outcome of that latter function is assigned to a local pointer named `result`. The `main()` function checks if it is not an empty pointer (line no. 8) and prints the key stored in the node that this pointer points to. Otherwise, it prints an appropriate message.

The main() Function

Third Part

```
1           delete_node(&root, number_to_delete);
2           printf("The BST after trying to delete the key
             ↪  %d.\n",number_to_delete);
3           printf("Number of nodes in the binary search tree:
             ↪  %u\n", count_nodes(root));
4           puts("Keys in the BST.");
5           print_bst_inorder(root);
6           puts("");
7           remove_bst_nodes(&root);
8           return 0;
9   }
```

The main() Function

Third Part

Next, the `main()` function calls the function that deletes the BST node with a given key (line no. 1). After that the program displays messages, informing the user, which key it tried to delete (line no. 2) and how many nodes there are in the BST now (line no. 3). Then it displays all keys stored in the BST using the *in-order* traversal algorithm (line no. 5). This order allows the user to quickly determine, which key has been deleted. Finally, the `main()` calls the `remove_bst_nodes()` function to delete all the nodes in the BST (line no. 7) and exits (line no. 8).

Summary

The main advantage of the BST is the time complexity of operation performed on that tree, which is proportional to its height. If the shape of the BST is close to the shape of a complete binary tree, then its height is $log_2(n)$, where n is the number of nodes in the BST. Unfortunately, if keys are added to the tree in an increasing or decreasing order, then the resulting BST will become a list and its height will be n.

To avoid such edge cases the *balanced binary trees*, such as the AVL trees and red-black trees, can be used instead of regular BSTs. Most of the operations on BST can be easily implemented with the use of both the recursive and non-recursive (using loops) functions. The tree traversal (and related) operations are an exception. They are implemented the best as recursive functions.

Summary

Please notice, that the binary trees may be used to represent arithmetic expressions. These sorts of binary trees are called *binary expression trees*. If we apply the *in-order* traversal algorithm to print the content of the tree nodes on the screen then we will get the expression in conventional (infix) notation. If we use the *pre-order* traversal algorithm for the same purpose, then we will get the expression in the Polish (prefix) notation (or PN). Finally, the *post-order* traversal algorithm will give us the expression in the reverse Polish (postfix) notation (or RPN).

Questions

?

The End

Thank You For Your Attention!