# Fundamentals of Programming 2

## Binary Search Trees (BST) — Part One

Arkadiusz Chrobot

Department of Computer Science

April 27, 2020

# Outline

# Introduction

Trees are nonlinear abstract data structures used for representing hierarchically ordered data. They are a special case of other data structures called graphs that will be discussed in future lectures. The Binary Search Tree can be described as a tree with a specific order of elements (nodes). Trees and graphs in Computer Science are implementations of some mathematical concepts. The next slides present several definitions originating from this discipline that relate to the trees.

# Definitions

## Tree

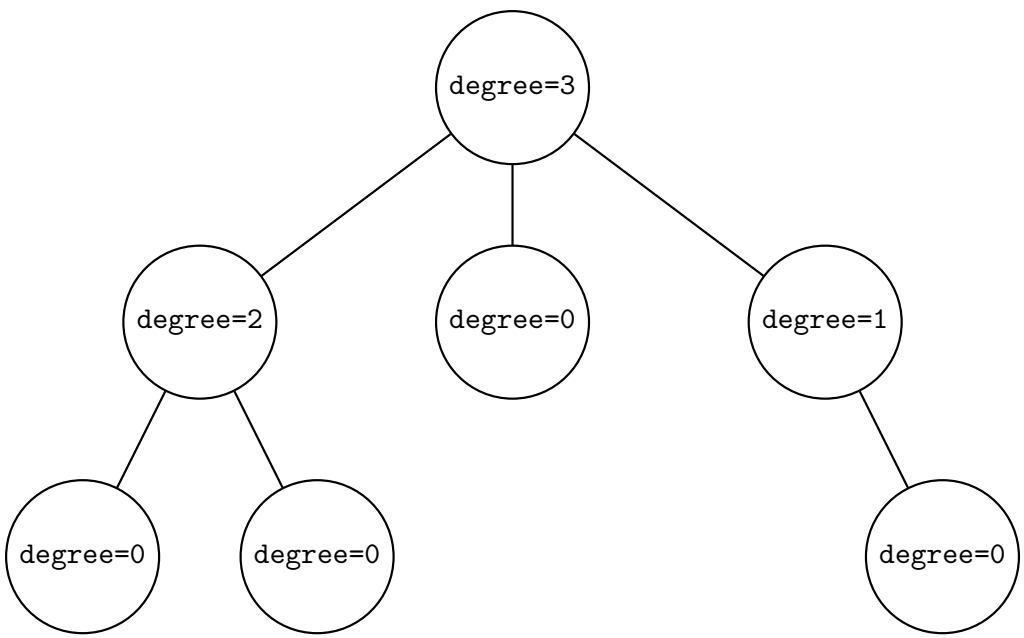
A tree is a set $T$ of one or more elements called nodes or *vertexes* satisfying the following conditions:

- there is one special node called *root*,
- the rest of the nodes is partitioned into $m \geq 0$ disjoint sets $T_1, \ldots, T_m$, which are also trees. The trees $T_1, \ldots, T_m$ are called *subtrees* of the root.

# Definitions

## Degree of Node

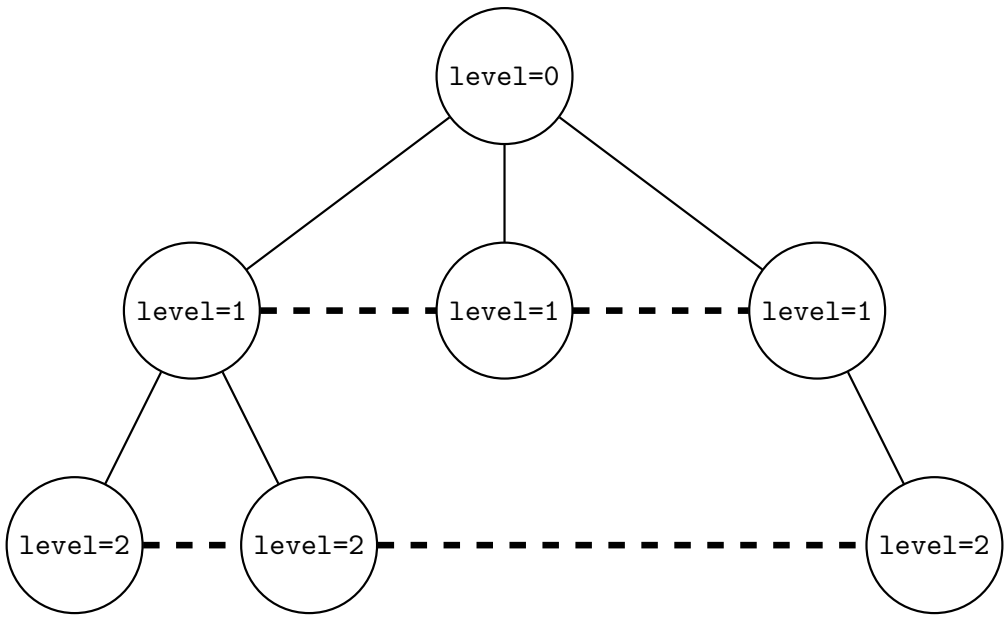
The number of subtrees of a single node is called its *degree*. Nodes with the degree higher than zero are called *internal nodes* or *inner nodes*. Nodes with the degree equal zero are called *external nodes* or *leafs*.



# Definitions

## Level of Node

The *level of a node* is defined recursively: the root has a level equal 0 and the level of each other node is higher by one than the level of the root in a smallest subtree that includes the node.



# Definitions

## Ordered Trees

If the order of the subtrees in a tree does matter then the tree is an *ordered tree*.

# Definitions

## The Hight of the Tree

The *hight of a tree* is the maximal level of its nodes plus one.

# Definitions

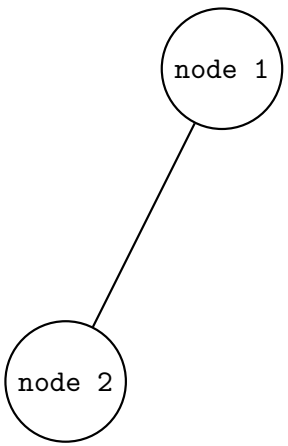
## Binary Tree

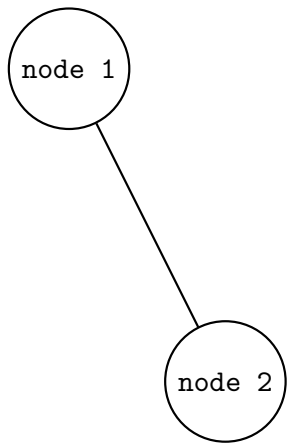
A *binary tree* is a finite set of nodes, which either is empty of it consists of the root node and two disjoint binary trees called the *left* and the *right* subtree. Therefore, the degree of each node in a binary tree doesn't exceeds two. If the left and/or right subtree of a given node is not empty, then the root of that subtree is called a *descendant* (or a *child*) of the given node, while the node is called an *ancestor* (or a *parent* or an *ascendant*) of that root. Interesting fact: according to the introduced definitions the binary tree is not a tree, because it can have no elements and the tree has to have at least one element.

# Definitions

## Differences Between Binary Trees and Trees

Let's assume that the graphs presented in this slide are binary trees. In that case those are two different tress, otherwise the graphs are the same tree.

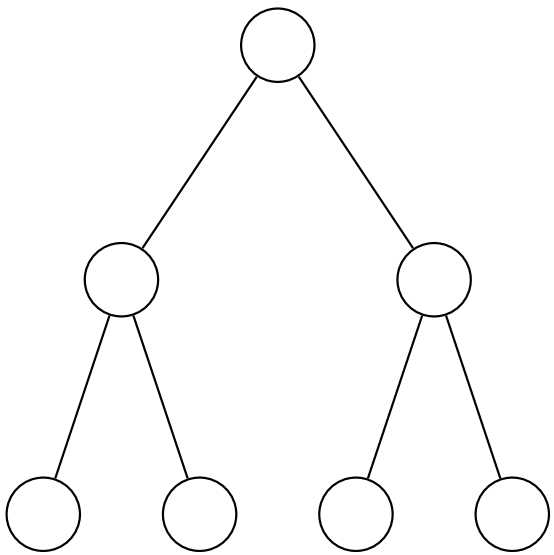


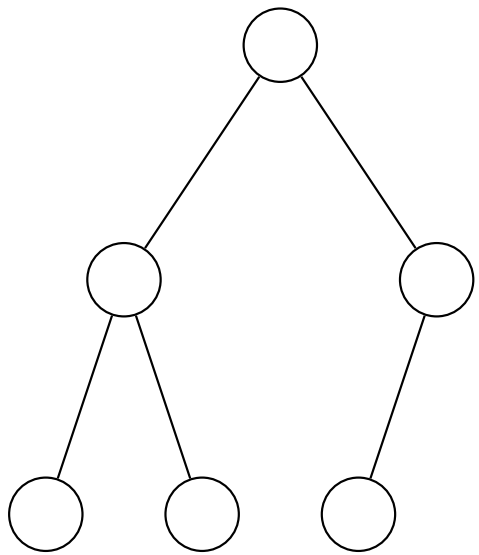
# Definitions

## Full and Complete Binary Tree

If a binary tree of a given height has all possible nodes, then it is called a *full binary tree*. If a binary tree of a given height has all possible nodes on every level, except possibly a few on the last one, it is called a *complete binary tree*.



The Full Binary Tree



The Complete Binary Tree

In Computer Science trees are drawn upside down (with the root up).

# Definitions

## The BST

The Binary Search Tree (BST) is used for implementing *dictionaries* that store pairs of keys and values. Each value is identified uniquely by a key. In the BST the order of keys follow this rule:

### Key Order in the BST

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

Peter Braß[1] calls the BSTs *the first model of search trees.*

[1] Peter Brass "Advanced Data Structures", Cambridge University Press, Cambridge, 2008

# Implementation

In this lecture an implementation of the BST is discussed that stores key-value pairs, where the key is an ASCII of a character and the value is the character. The rule of the key order in the BST allows the key of a given node to be repeated in the tree. Unfortunately, this complicates the implementation of adding a new node to the tree. Thus, some compromise has to be made. Some computer scientists think that keys in the BST should not repeat, others state that the keys may repeat, but those that do should be always placed in the left or always in the right subtree of the node that stores the same key. The first approach seems to be plausible, but may cause issues for example when the BST is applied for storing personal data, where the surname of a person is the key. Peter Braß suggests to allow the keys to repeat, but the values should be stored only in the leaves. In this lecture yet another approach is taken — the keys are allowed to repeat and each node stores a value.

# Implementation

Just like other abstract data structures the binary trees can be implemented with the use of arrays or as dynamically allocated data structures. In this lecture the latter approach is discussed. As in the case of linked lists and other similar structures the base type of BST (the data type of a single node) and operations for the tree have to be defined. As for the operations, in this lecture only adding a new node to the BST, printing the content of the tree and removing all nodes from the BST are defined. The printing operation uses several binary tree traversing algorithms and can also displays the shape of the tree on the screen. According to the definition that is introduced in the lecture an empty binary tree is an existing tree, so formally there is no operation of removing or creating a binary tree (also a BST). The operation of removing a single node from a BST is difficult and it will be discussed in the next part of the lecture, together with some other additional operations.

# Implementation

## Header Files

```
1  #include<stdlib.h>
2  #include<time.h>
3  #include<curses.h>
4  #include<locale.h>
```

# Implementation

## Header Files

Aside from header files that are also used in programs discussed in previous lectures, there are included the `curses.h` and `locale.h` files. Some of the functions declared in the `curses.h` header file are applied for printing the content of the tree on the screen in a way that shows the shape of the BST. The `time()` function declared in the `time.h` header file is applied for initializing the PRNG, that is used for generating keys and values added to the BST.

# Base Type and Root Pointer

```
1  struct tree_node
2  {
3      int key;
4      char value;
5      struct tree_node *left_child, *right_child;
6  } *root;
```

# Base Type

The definition of the BST base type resembles the definition used for the doubly linked list, but the pointer fields serve a different purpose. The `left_child` field points the left child of the node and in general the whole left subtree of the node, the `right_child` field points the right child of the node and in result the whole right subtree of the node. If the node is a leaf then both pointer fields have the value of `NULL`. If the node has only one child then only one of those fields has such a value. There are implementations of the BST where a third pointer field is used. This field points to the parent of the node. The only node that stores `NULL` value in such a field is the root. The `key` and `value` fields are used for storing, respectively, the key and the associated value. Also in the previous slide a global variable named `root` is declared. The variable is a pointer to the root node of the tree.

# BST Operations

Some of the operations for the BST can be easily implemented either as recursive or iterative functions. Others are usually implemented only as recursive functions. Defining them as iterative functions would be challenging and they would be at most as efficient as their recursive counterparts. In this lecture both iterative and recursive implementation of an operation are presented, whenever both of them are easy to create. First, the implementation of adding a node operation is discussed.

# Adding Node — Recursive Approach

```
1  void add_node(struct tree_node **root, int key, char value)
2  {
3      if(*root==NULL)
4      {
5          *root = (struct tree_node *)
6                          malloc(sizeof(struct tree_node));
7          if(*root) {
8              (*root)->key = key;
9              (*root)->value = value;
10             (*root)->left_child = (*root)->right_child = NULL;
11         }
12     } else
13     if((*root)->key >= key)
14         add_node(&(*root)->left_child,key,value);
15     else
16         add_node(&(*root)->right_child,key,value);
17 }
```

# Adding Node — Recursive Approach

The `add_node()` function doesn't return any value, but has three parameters. The first one is a pointer to a pointer. It is used for passing an address of the root pointer or, when the function is called recursively, an address of one of the currently visited node pointer fields, that point to its children. The last two parameters are used for passing the key and the value. The pair (the key and the value) is stored in the new node of the BST. After the function is called it checks if the value of the pointer pointed by the `root` pointer is `NULL`. If it is the first (non-recursive) call of the function and the condition is fulfilled then the tree is empty and the function adds its first node. The function allocates memory for the node (lines no. 5 and 6) and initializes fields of the node (lines no. 8, 9 and 10). Please observe, that both pointer fields get the `NULL` value. If the condition in the 3rd line is not met then the tree already has some nodes. In that case the function evaluates the expression in the 13th line, i.e. check the relation between the key stored in a visited node and the one that should be assigned to the new node.

# Adding Node — Recursive Approach

If the key in the visited node is greater or equal to the new key, then the `add_node()` function calls itself for the left child of the node, which is also the root of its left subtree. Thus, all values with the key equal to the key of the visited node are stored in the left subtree of the node. If the condition in the 13th line is not satisfied, then the function is invoked recursively for the right child of the visited node. Please note, that in both cases the first argument of the function is the address of the appropriate pointer field, hence the value of the field can be modified by subsequent instances of the function. The `add_node()` function stops calling itself when one of its instances is invoked for a pointer field that has a value of `NULL`. In that case the instance creates a new node of the BST by performing the same statements (lines no. 4–11) as for the first node and exits. After the new node is created the other instances of the function also exit.

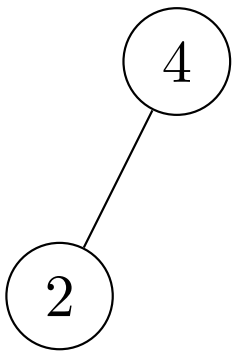
# Adding Node — Recursive Approach

The next slide shows how the nodes storing the following keys: `4`, `2`, `1`, `3` and `5` are added to the BST.

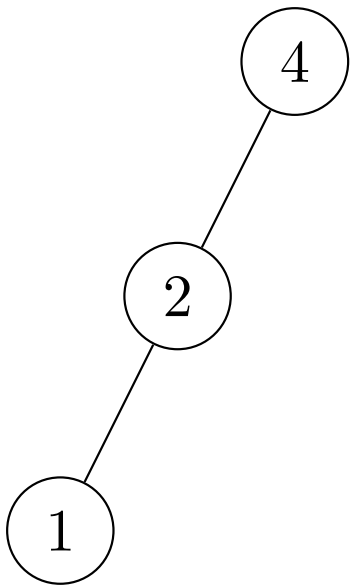
# Adding Node

# Adding Node

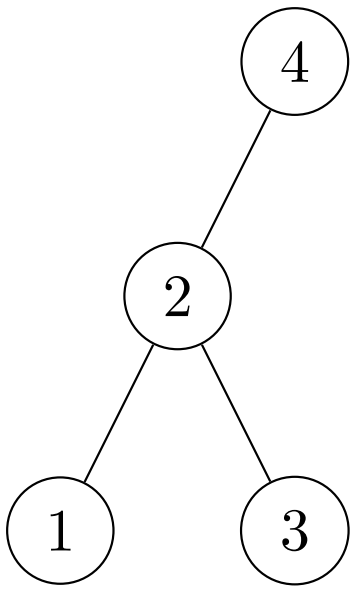
# Adding Node



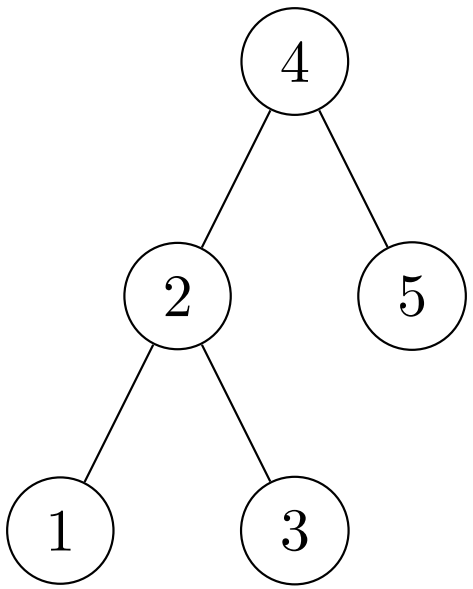
# Adding Node



# Adding Node



# Adding Node



# Adding Node — Iterative Approach

```
1  void add_node(struct tree_node **root, int key, int value)
2  {
3      while(*root!=NULL) {
4          if((*root)->key>=key)
5              root = &(*root)->left_child;
6          else
7              root = &(*root)->right_child;
8      }
9      *root = (struct tree_node *)
10                         malloc(sizeof(struct tree_node));
11     if(*root) {
12         (*root)->key = key;
13         (*root)->value = value;
14         (*root)->left_child = (*root)->right_child = NULL;
15     }
16 }
```

# Adding Node — Iterative Approach

It proves that the iterative version of `add_node()` function is as simple as its recursive counterpart. The prototype of the function is the same. The first element of its body is the `while` loop (lines no. 3–8). If the tree is empty the loop will perform no iteration. If it already has nodes then the relation between the key in a visited node and the new key is verified inside the loop. If the key in the node is greater or equal to the new key then the address of the pointer field that points to the left child of the node is assigned to the `root` parameter (5th line), otherwise the address of the pointer field that points to the right child of the node is assigned to this parameter (7th line). If one of the fields has the `NULL` value then the loop stops. Next, the new node is created (line no. 9 and 10) and its address is assigned to field pointer pointed by the `root` parameter. If the `while` loop hasn't performed a single iteration, the address of the new node will be assigned to the `root` pointer — the node will become the root of the BST. The fields of the new node are initialized in lines no. 12–14.

# Traversing the Binary Tree

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

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

In all of those algorithms the left subtree is traversed before the right subtree. The only difference is when the root is visited.

# Traversing the Binary Tree

## *In-Order* Traversal

The in-order traversal algorithm for a binary tree is defined as follows:

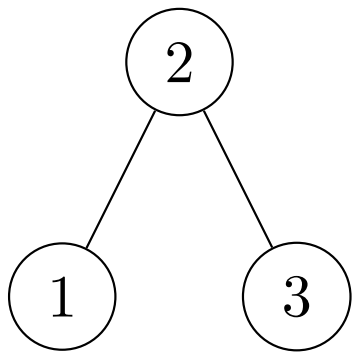
1. traverse the left subtree,
2. visit the root,
3. traverse the right subtree.

The algorithm is usually implemented in a form of a recursive function. In the next slide is defined a function that applies this algorithm for printing the keys stored in a BST. There is also a figure that shows an example of such a tree and a result of traversing the tree with the use of this algorithm.

# Traversing the Binary Tree

## *In-Order* Traversal

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




### The Result

```
1, 2, 3
```

# Traversing the Binary Tree

## *Pre-Order* Traversal

The pre-order traversal algorithm for a binary tree is defined as follows:

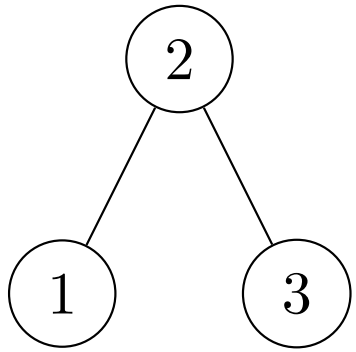
1. visit the root,
2. traverse the left subtree,
3. traverse the right subtree.

This algorithm is also usually implemented in a from of recursive function. The difference between the in-order and pre-order traversal algorithms is that the latter visits the root first and then both of the subtrees. In the next slide is defined a function that applies the algorithm for printing the keys stored in a BST. There is also a part of the slide that illustrates how the algorithm traverses an example BST and what is its result for that tree — just like in the case of the in-order traversal algorithm.

# Traversing the Binary Tree

## *Pre-Order* Traversal

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




### The Result

2, 1, 3

# Traversing the Binary Tree

## *Post-Order* Traversal

The post-order traversal algorithm for a binary tree is defined as follows:

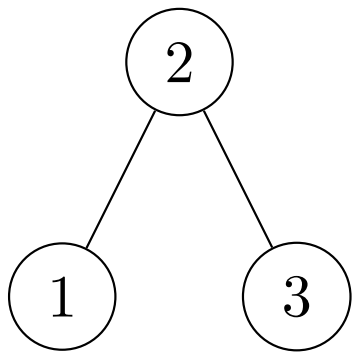
1. traverse the left subtree,
2. traverse the right subtree,
3. visit the root.

It is usually implemented in a form of a recursive function, just like the two previously discussed algorithms. In this algorithm the subtrees are visited first, then the root. The next slide presents an example implementation of the algorithm together with a figure that illustrates how the algorithm can be applied to an example BST — just like in the cases of previously discussed algorithms.

# Traversing the Binary Tree

## *Post-Order* Traversal

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




### The Result

```
1, 3, 2
```

# Printing the Content of Binary Tree

The functions presented in the previous slides display on the screen only the keys stored in the BST, using a specified binary tree traversal algorithm. The next slides show definitions of functions that print the whole content of the tree nodes, i.e. the key - value pairs. The functions use to this end respectively: the in-order, pre-order and post-order traversal algorithm. Each key - value pair is printed in a separate line of the screen.

# Printing the Content of BST — In-Order Traversal

```
1  void print_inorder(struct tree_node *root)
2  {
3      if(root) {
4          print_inorder(root->left_child);
5          printw("key: %4d, value: %4c\n",
6                                          root->key,root->value);
7          print_inorder(root->right_child);
8      }
9  }
```

# Printing the Content of BST — Pre-Order Traversal

```
1  void print_preorder(struct tree_node *root)
2  {
3      if(root) {
4          printw("key: %4d, value: %4c\n",
5                                          root->key,root->value);
6          print_preorder(root->left_child);
7          print_preorder(root->right_child);
8      }
9  }
```

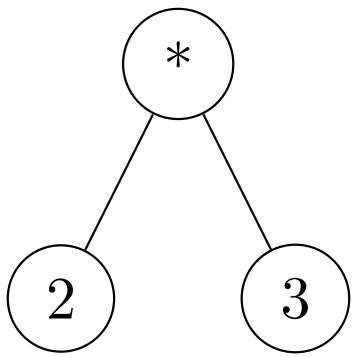
# Printing the Content of BST — Post-Order Traversal

```
1  void print_postorder(struct tree_node *root)
2  {
3      if(root) {
4          print_postorder(root->left_child);
5          print_postorder(root->right_child);
6          printw("key: %4d, value: %4c\n",
7                                          root->key,root->value);
8      }
9  }
```

# Arithmetic Expression Tree

Tree traversal algorithms have many others applications than just printing the content of the BST or other binary tree. For example, a binary tree may represent an arithmetic expression, like $2 \cdot 3$. The multiplication operator can be stored in the root and the leaves may represent the arguments. In effect, the shape of the tree determines what operation is performed on which arguments. Such a tree is called an *arithmetic expression tree*. If the tree is traversed using the in-order traversal algorithm then the result will be the arithmetic expression in the Infix Notation. If however, the pre-order traversal algorithm is applied then the resulting expression will be in Polish Notation. Finally, if the post-order traversal algorithm is applied to this tree, then the expression will be in Reversed Polish Notation. The next slides contain figures that show those operations. The arithmetic expressions represented by the arithmetic expression tree can be much more complex than the one used in this lecture.

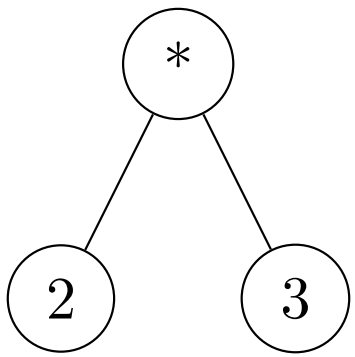
# Arithmetic Expression Tree




## In-Order Traversal Algorithm Result

2 * 3

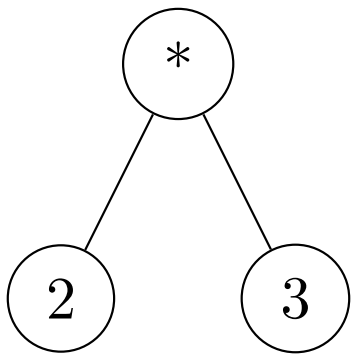
# Arithmetic Expression Tree




## Pre-Order Traversal Algorithm Result

* 2 3

# Arithmetic Expression Tree




## Post-Order Traversal Algorithm Result

```
2 3 *
```

# Displaying Shape of BST

It would be interesting to print the content of the BST in a way that would show the shape of the tree. To this end the `curses` library is applied. The subsequent levels of the BST[2] are separated from each other by two lines of the screen, to make the shape of the tree more visible. The pre-order traversal algorithm is used for displaying the shape of the tree, because it starts with the root which has to be printed first and then prints the subtrees. Since printing pairs of a key and a value could deteriorate the visibility of the tree shape, two functions are presented that print separately the keys and the values.

[2] A single level of the tree include all existing nodes that have the same level.

# Displaying Shape of BST — Printing Values

```
1  void print_values(struct tree_node *root, int x, int y,
2                                              unsigned int distance)
3  {
4      if(root) {
5        mvprintw(y,x,"%4c",root->value);
6        print_values(root->left_child,x-distance,y+2,distance/2);
7        print_values(root->right_child,x+distance,y+2,distance/2);
8      }
9  }
```

# Displaying Shape of BST — Printing Values

The `print_values()` function has four parameters. The first one is a pointer used for passing the address of the root of the BST or addresses of its subtrees roots. Next two parameters are used for passing the coordinates of the place on the screen where the value of the currently visited node should be displayed. By the last parameter is passed the half of the distance between the root and one of its child nodes along the x-axis. It is taken into account by the function even if only one of the nodes exists. The function first checks if it is invoked for an existing node of the tree (4th line). If so, then it prints the value stored in the node in a place on the screen determined by the passed coordinates. Next, it it calls itself recursively for the left and then the right child of the node. Please notice, how the value of the forth parameter is used in the second argument of the recursive calls. For the left child it is subtracted from the abscissa of the parent of the node, and for the right child it is added to the same value. The ordinate is increased by 2 and the distance between the root and one of its child nodes is halved.

# Displaying Shape of BST — Printing Keys

```
1  void print_keys(struct tree_node *root, int x, int y,
2                                          unsigned int distance)
3  {
4      if(root) {
5          mvprintw(y,x,"%4d",root->key);
6          print_keys(root->left_child,x-distance,y+2,distance/2);
7          print_keys(root->right_child,x+distance,y+2,distance/2);
8      }
9  }
```

# Displaying Shape of BST — Printing Keys

The function that prints keys of the BST is defined almost in the same way as the function that prints values. The only difference is the statement in the 5th line, where instead of the `value` field the `key` field is printed.

# Removing All Nodes From Binary Tree

```
1  void remove_tree_nodes(struct tree_node **root)
2  {
3      if(*root)
4      {
5          remove_tree_nodes(&(*root)->left_child);
6          remove_tree_nodes(&(*root)->right_child);
7          free(*root);
8          *root = NULL;
9      }
10 }
```

# Removing All Nodes From Binary Tree

The function that removes all the nodes from a binary tree uses the post-order traversal algorithm for traversing the tree, so there is no danger that the recursive calls of the function will receive addresses of nonexistent fields. Since all nodes are removed from the tree, the root pointer has to store the `NULL` value after the function exits. That's why in the 8th line the function assigns the value to the variable pointed by the `root` parameter. The statement also assigns the `NULL` value to each of pointer fields of tree nodes, before they are deleted. Owing to the applied algorithm, the function removes nodes from the tree starting with leaves and finishing with the root.

# The main() Function

## First Part

```
1  int main(void)
2  {
3   if(setlocale(LC_ALL,"")==NULL) {
4     fprintf(stderr,"The language settings exception!\n");
5     return -1;
6   }
7   if(!initscr()) {
8     fprintf(stderr,"The curses library initialization error!\n");
9     return -1;
10  }
11  int i;
12  srand(time(0));
13  for(i=0;i<10;i++) {
14    int key = 0; char value = 0;
15    value='a'+rand()%('z'-'a'+1);
16    key = (int)value;
17    add_node(&root,key,value);
18  }
```

# The `main()` Function

## First Part

In the first part of the `main()` function the `curses` library and the PRNG are initialized and then ten nodes that store lowercase letters of Latin alphabet are added to the BST (lines no. 13–18).

# The main() Function

## Second Part

```
1       printw("Data in the tree (in-order traversal):\n");
2       print_inorder(root);
3       refresh();
4       getch();
5       erase();
6       printw("Data in the tree (post-order traversal):\n");
7       print_postorder(root);
8       refresh();
9       getch();
10      erase();
11      printw("Data in the tree (pre-order traversal):\n");
12      print_preorder(root);
13      refresh();
14      getch();
15      erase();
```

# The `main()` Function

## Second Part

In the second part of the `main()` function the content of the BST is displayed on the screen with the use of the tree traversal algorithms. After each of the functions implementing the algorithms exits the program stops and waits for the user to presses any key on the keyboard. After that the content of the screen is cleared and another function is performed.

# The main() Function

## Third Part

```
1       printw("The shape of the BST (values):");
2       print_values(root,COLS/2,1,20);
3       refresh();
4       getch();
5       erase();
6       printw("The shape of the BST (keys):");
7       print_keys(root,COLS/2,1,20);
8       refresh();
9       getch();
10      erase();
11      remove_tree_nodes(&root);
12      if(endwin()==ERR) {
13          fprintf(stderr,"The endwin() function exception!\n");
14          return -1;
15      }
16      return 0;
17  }
```

# The `main()` Function

## Third Part

In the third part of the `main()` function definition the values and keys stored in the BST nodes are displayed on screen in such a way that it shows the shape of the tree. Please notice the coordinates of the first displayed node, i.e. the root of the tree. The ordinate is `1`, which means that the data from the node are displayed in the second line of the screen. The value of the abscissa is set to `COLS/2`, where `COLS` is a constant specifying the number of columns in the screen. It means that the value or the key is displayed in the half length of the second row of the screen. That way the shape of the BST should be more visible, but the overall effect depends on the actual content and number of the nodes. Just like in the second part, the program stops after each function exits and waits for the user to press any key on the keyboard. Then the content of the screen is then cleared and the next function is performed. After the keys are printed all the nodes are removed from the BST, the `curses` library is finalized and the program exits.

# Summary

The properties of the BST will be closely discussed in the next lecture. Trees in general are flexible data structures which are applied in compilers (the arithmetic expression trees or syntax trees in general), operating systems (for example the CFS and CFQ algorithms used by Linux kernel), computer graphics (for example the BSP trees) and in many other programs ranging from the bookkeeping applications to the artificial intelligence software. In many cases trees are the most effective data structures.

# Questions

?

# The End

Thank You For Your Attention!