Fundamentals of Programming 2

Binary Search Trees (BST) — Part One

Arkadiusz Chrobot

Department of Computer Science

May 21, 2019

Outline

Introduction

Trees are nonlinear abstract data structures used for representing hierarchically ordered data. They are a special case of other data structures that will be discussed in future lectures. Those structures are called graphs. The Binary Search Tree can be considered a tree with a specific order of elements (nodes). Trees and graphs in computer science are implementations of mathematical concepts. Therefore, some definition from mathematics related to the trees are introduced in the next slides.

Definitions

Tree

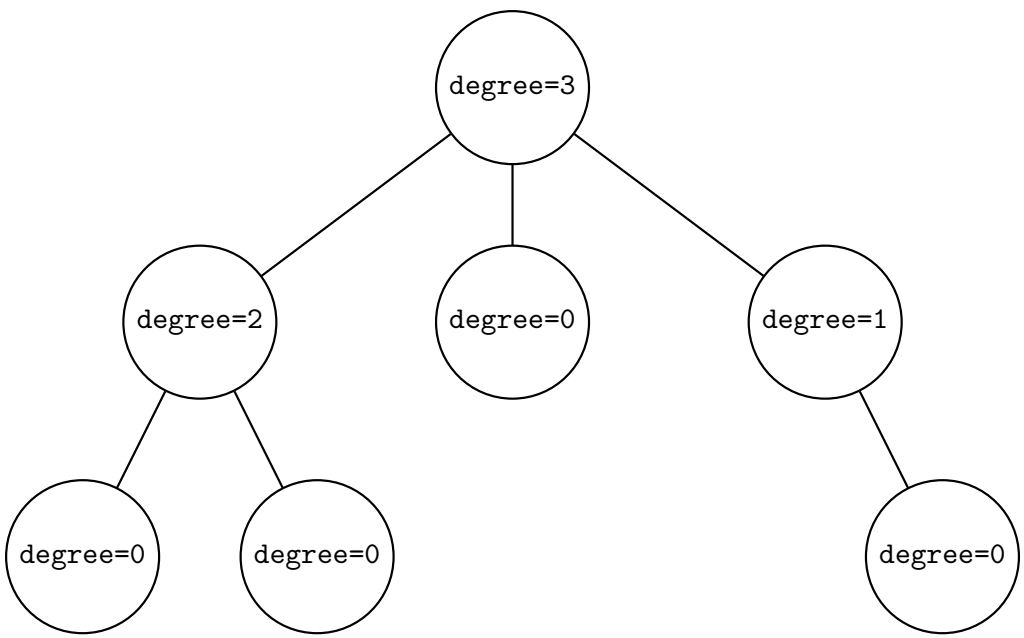
A tree is a set T of one or more elements called nodes or *vertexes* satisfying the following conditions:

- there is one special node called *root*,
- the rest of the nodes are 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

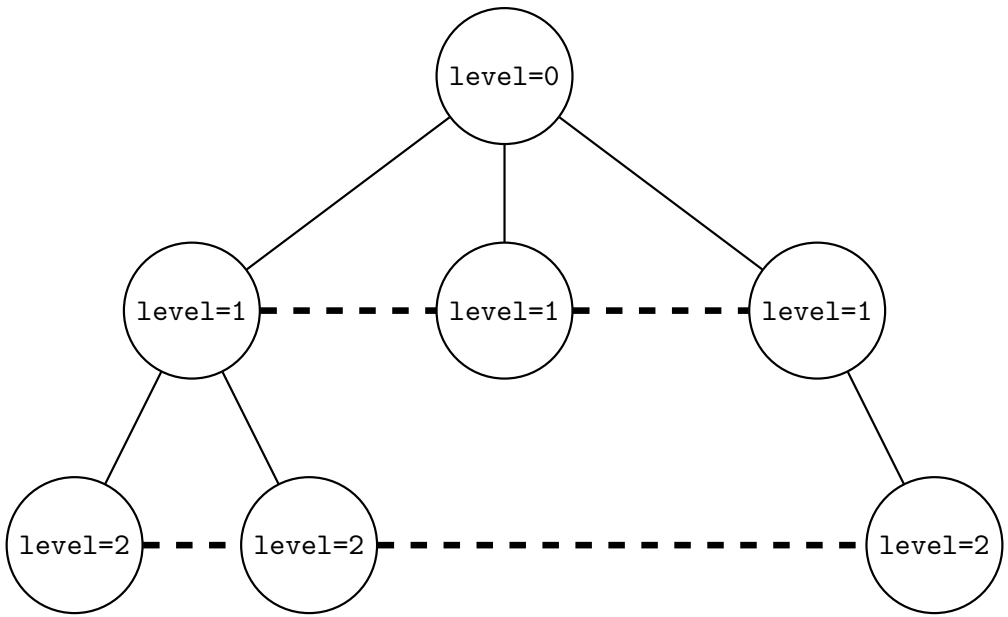
The number of subtrees of a single tree node is called a *degree*. The nodes with degree higher than zero are called *internal nodes* or *inner nodes*. The nodes with degree equal zero are called *external nodes* or *leafs*.



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

Definitions

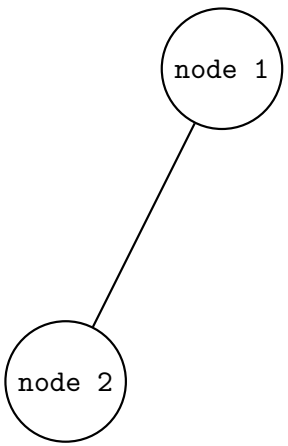
Binary Tree

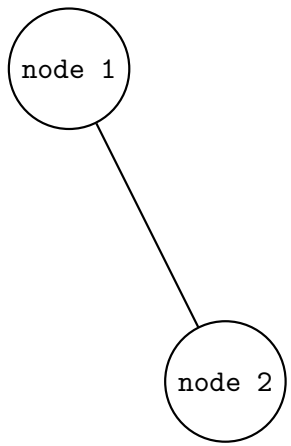
A *binary tree* is a finite set of nodes, which either is empty of it consists of the root node and two disjoint binary trees called the *left* and *right* subtrees. Therefore, the degree of each node in a binary tree doesn't exceeds two. If the left and/or right subtree of a given node is not empty, 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*) 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 Trees and Trees

If it is assumed that the figures in the slide show binary trees, then they are two different binary trees, otherwise those figures show the same tree.

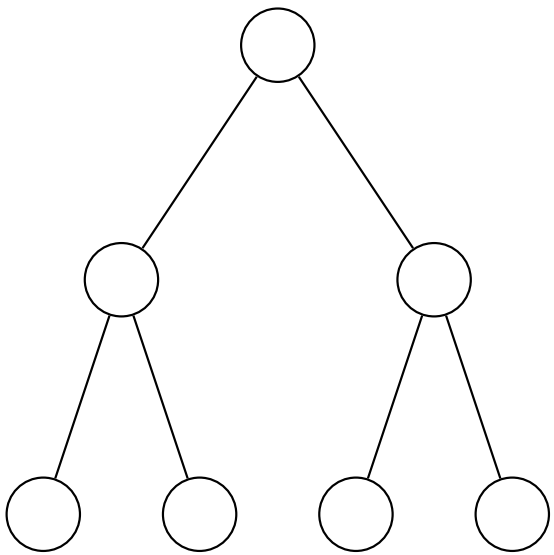




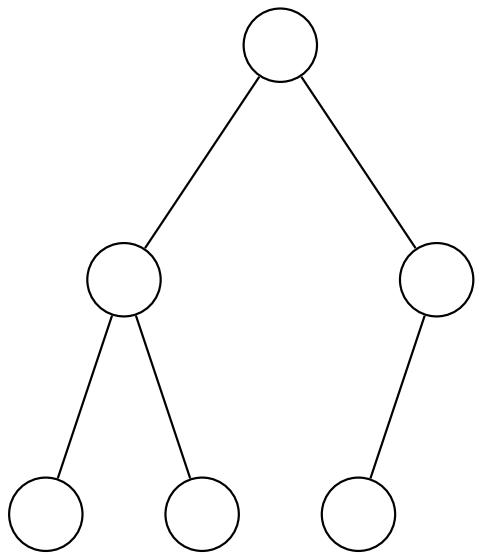
Definitions

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

It' worth to mention that in computer science the trees are drawn with the root up.

Definitions

The BST

Binary Search Tree (BST) is applied for implementing data structures called *dictionaries* in which every value is identified by a unique key. In the BST the keys are ordered according to the following 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 node in the right subtree of 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

The implementation is discussed with the use of a BST that stores key–value pairs, where the key is an ASCII code of character and the value is the character. The definition of the BST allows the key of a given node to be repeated in the left or right subtree of that node. 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 claim that the keys may repeat, but those that do should be placed always on the left or always on the right side of the node that stores the same key. The first approach seems to be plausible, but may cause problems when the BST is for example applied for storing personal data, where the surname of the person is the key. The aforementioned Peter Braß proposes to allow the keys to repeat, but store the values only in the tree leaves. In the lecture yet another approach is taken — the keys are allowed to repeat and each node stores a value.

Implementation

Just like other data structures the binary trees can be implemented with the use of arrays or as dynamically allocated data structures. In the lecture the latter approach is discussed. Similarly as for lists and other the like 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 the lecture are only defined adding a new node to the BST, printing the content of the tree and removing all nodes from the BST. The printing operation utilizes several binary tree traversing algorithms and can also visualise the structure 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 tree is complicated and it will be discussed in the next 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 for header files that are also included in programs discussed in the previous lectures, there are also included the `curses.h` and `locale.h` files. Some of the functions declared in the `curses.h` header file are used 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 BST base type resembles the definition of doubly linked list, but the pointer fields serve a different purpose. The `left_child` field points the left child of the node and thus the whole left subtree of the node in general, 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 `NULL` value. 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. The 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 related with the key value. In the previous slide also 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 implemented in a simple way both as recursive and iterative functions, others are usually implemented only as recursive functions. The definitions of the iterative counterparts of the latter functions would be complicated and at most as efficient as the recursive functions. In the lecture both iterative and recursive implementations of a given operation are presented, whenever both of them are easy to create. The description starts with the operation of adding a node to the BST.

Adding a 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 a 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 children of the currently visited node. The key is passed by the second parameter and by the third is passed the value. The pair (the key and the value) is stored in the new node of the BST. The function checks if the value of the pointer pointed by the `root` pointer is `NULL`, after it is called. 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 the first node to it. Therefore, the function allocates memory for the node (lines no. 5 and 6) and initializes fields of the node (lines 8, 9 and 10). Please observe, that both pointer fields get the `NULL` value. If the condition in the 3rd line is not fulfilled then the tree already has some nodes. In that case the function performs the statement in the 13th line, i.e. check the relation between the key stored in a visited node and the one that should be assigned in the new node.

Adding a 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 field can be modified by subsequent instances of the function, if needed. The function stops invoking 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 statements (lines no. 4–11), in the same way as it is described in the previous slide and terminates.

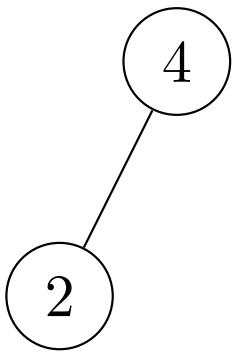
Adding a Node — Recursive Approach

After the new node is created the other instances of the `add_node()` also terminate. There is an animation in the next slide that shows how the nodes containing the following keys: `4`, `2`, `1`, `3` and `5` are added to the BST.

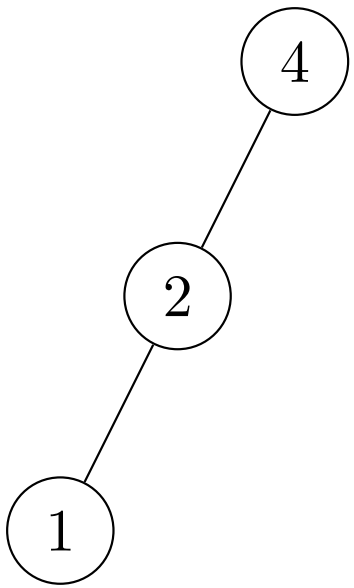
Adding a Node

Adding a Node

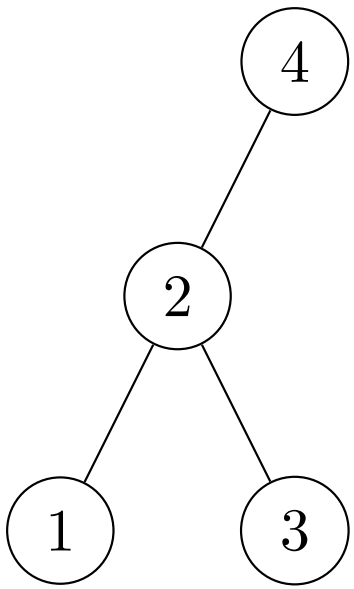
Adding a Node



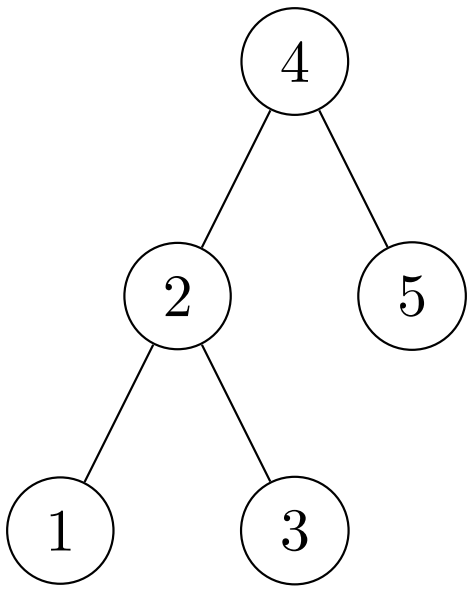
Adding a Node



Adding a Node



Adding a Node



Adding a 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 a Node — Iterative Approach

It shows up 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 tested 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` pointer to a pointer (5th line), otherwise the address of the pointer field that points to the right child of the node is assigned to the `root` parameter (7th line). If one of the fields has a `NULL` value then the loop terminates. Next, the new node is created (line no. 9 and 10) and its address is assigned to field pointer pointed by the `root` parameter. Should the `while` loop terminate without performing a single iteration, the address of the new node would be assigned to the root pointer — the node would become the root of the BST. The fields of the new node are initialized (lines no. 12–14).

Traversing the Binary Tree

There are three recursive algorithms that make it possible to traverse a binary (in fact any) tree:

1. *in–order*,
2. *pre–order*,
3. *post–order*.

In all the algorithms it is assumed that the left subtree is traversed first and the right subtree is traversed next.

Traversing a Binary Tree

In–order Algorithm

The in–order algorithm for traversing a binary tree is defined as follows:

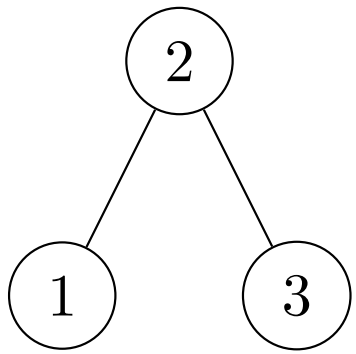
1. traverse the left subtree,
2. visit the root,
3. traverse the right subtree.

The algorithm is usually implemented in a form of a recursive function. The next slide contains a definition of a function that applies the 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 the algorithm.

Traversing a Binary Tree

In–order Algorithm

```
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 a Binary Tree

Pre–order Algorithm

The pre–order algorithm for traversing a binary tree is defined as follows:

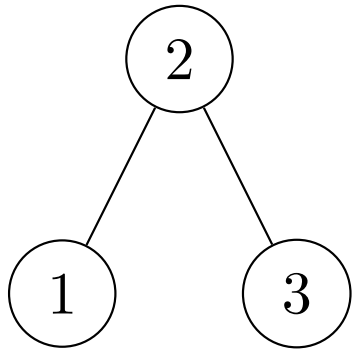
1. visit the root,
2. traverse the left subtree,
3. traverse the right subtree.

This algorithm is also usually implemented in a from of recursive function. The difference between the in–order and pre–order algorithm is that in the latter the root is visited first and then both of the subtrees. In the next slide is defined a function that applied the algorithm for printing the keys stored in a BST. There is also a part of the slide that illustrates how the algorithm is applied for an example BST and what is the result of traversing the tree with the use of the pre–order algorithm — just like in the case of the in–order algorithm.

Traversing a Binary Tree

Pre–order Algorithm

```
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 a Binary Tree

Post–order Algorithm

The post–order algorithm for traversing a binary tree is defined as follows:

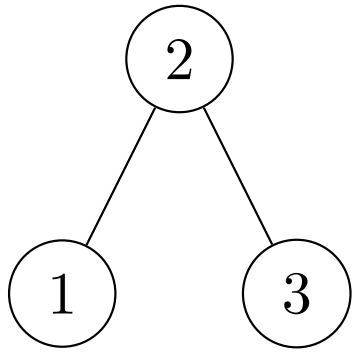
1. traverse the left subtree,
2. traverse the right subtree,
3. visit the root.

It is usually implemented in a form of a recursive function, just like the two previously discusser algorithms. In the 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 tree — just like in cases of previously discussed algorithms.

Traversing a Binary Tree

Post–order Algorithm

```
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 a Binary Tree

The functions presented in the previous slides display on the screen only the keys stored in the BST, using a given binary tree traversing algorithm. The next slides contain 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 algorithm. Each key - value pair is printed in a separate line of the screen.

Printing the Content of a BST — In–order Algorithm

```
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 a BST — Pre–order Algorithm

```
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  }
```

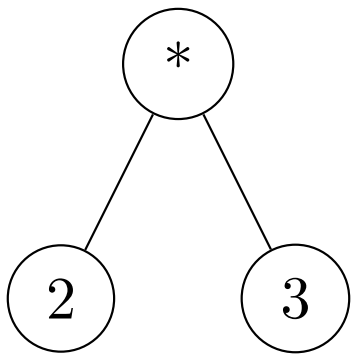
Printing the Content of a BST — Pre–order Algorithm

```
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 traversing algorithms have many others applications than just printing the content of BST or other binary tree. For example, a binary tree may represent an arithmetic expression, like $2 \cdot 3$. The operator of such an expression could be stored in the root of the tree, and the leaves may represent the arguments. In result the construction 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 algorithm then the result will be the arithmetic expression in the Infix Notation. If however, the pre–order algorithm is applied then the resulting expression will be in Polish Notation. Finally, if the post–order algorithm is applied to this tree, then the expression will be in Reversed Polish Notation. The next slides contain images that show the described operations. The arithmetic expressions represented by the arithmetic expression tree can be much more complex than the one presented in the lecture.

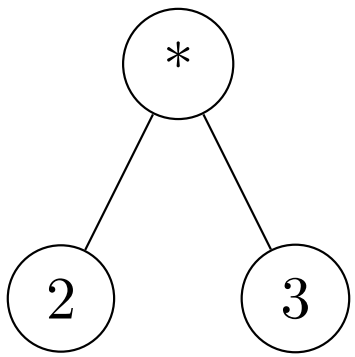
Arithmetic Expression Tree



In–order Algorithm Result

2 * 3

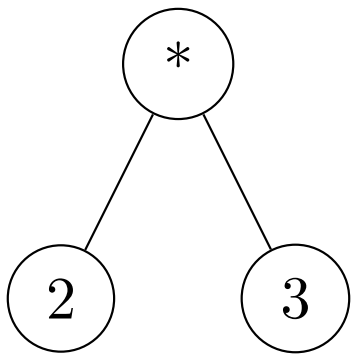
Arithmetic Expression Tree



Pre–order Algorithm Result

* 2 3

Arithmetic Expression Tree



Post-order Algorithm Result

```
2 3 *
```

Showing the Construction of the BST

It would be interesting to print the content of the BST in a way that would show the construction of the tree. To this end the capabilities enabled by the `curses` library are applied. The subsequent levels of the BST[2] are separated from each other by two lines of the screen, to make the construction of the tree more visible. The algorithm applied for this task is the pre-order tree traversing algorithm, because it starts with the root and roots of subtrees, which have to be printed first. Since printing pairs of a key and a value could deteriorate the visibility of the tree construction, 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.

Showing the Construction of the 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  }
```

Showing the Construction of the 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. Next two parameters are used for passing the coordinates where the value of the currently visited node should be displayed. By the last parameter is passed the half of the distance between two sibling nodes. It is taken into account by the function even if only one of the nodes exists. The function checks first 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 coordinates passed to the function. Next, it it calls itself recursively for the left and then the right child of the node. Please note, how the value of the forth parameter is applied in the argument of the recursive calls. For the left child it is subtracted from the horizontal coordinate of the parent of the node, and for the right child it is added to the same coordinate. The vertical coordinate is increased by 2, while the distance between descendants is halved.

Showing the Construction of the 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  }
```

Showing the Construction of the 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 in the statement from 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 algorithm for traversing the tree, so there is no danger that the recursive calls of the function would 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 terminates. 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 stores 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):\n");
2       print_inorder(root);
3       refresh();
4       getch();
5       erase();
6       printw("Data in the tree (post-order):\n");
7       print_postorder(root);
8       refresh();
9       getch();
10      erase();
11      printw("Data in the tree (pre-order):\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 discussed tree traversing algorithms. After each of the functions implementing the aforementioned algorithms is performed the program stops until the user presses any key on the keyboard. After that content of the screen is cleared and another function is performed.

The main() Function

Third Part

```
1       printw("The construction of the BST (values):");
2       print_values(root,COLS/2,1,20);
3       refresh();
4       getch();
5       erase();
6       printw("The construction 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 a way that shows the construction of the tree. Please note the coordinates of the first displayed node, i.e. the root of the tree. The vertical coordinate is `1`, which means that the data from the node are displayed in the second line of the screen. The value of the horizontal coordinate is `COLS/2`, where `COLS` is a constant defining the number of columns in the screen. It means that the value or the key is displayed in the half length of the screen. That way the construction of the BST should be more visible, but the overall effect depends on the actual content of the nodes. Just like in the second part, the program stops after each function terminates and waits for the user to press any key on the keyboard. Then the content of the screen is 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 terminates.

Summary

The properties of the BST will be closely discussed in the next lecture. It is however worth to mention now, that the trees (not only BSTs or binary trees) are flexible data structures which are applied in compilers (the aforementioned 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. For many issues trees are the most effective data structures.

Questions

?

The End

Thank You For Your Attention!