|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  | | --- | | LAB | | **11** | | Implementation of the BINARY SEARCH TREE with the help of algorithms for following functions   * SearchBST() * DeleteBST() | |
|  |  | |
| Objective(s): | Upon completion of this lab session, Student will be able to understand the following concepts | |
| |  |  | | --- | --- | | 1 | Binary Search Tree definition | | 2 | Creation of Binary Search Tree | | 3 | Operation of Binary Search Tree | |  |  | |  |  | | | |
| LabTask(s): |  | |
|  | |
|  |  | |
| |  |  | | --- | --- | | 1 | Write a program to create a BinarySearchTree. | | 2 | Add a function to the BinarySearchTree that find a Node in a Binary Search Tree | | 3 | Add a function to the BinarySearchTree that find Minimum and Maximum Value in a Binary Search Tree | | 4 | Add a function to the BinarySearchTree that delete Node from Binary Search Tree | | | |
|  | |
| LAB ACTIVITIES RELATED THEORY | |
|  | |
| **Binary Search Tree** | |
| First of all, binary search tree (BST) is a dynamic data structure, which means, that its size is only limited by amount of free memory in the operating system and number of elements may vary during the program run. Main advantage of binary search trees is rapid search, while addition is quite cheap. Let us see more formal definition of BST.  Binary search tree is a data structure, which meets the following requirements:   * it is a [binary tree](http://www.algolist.net/Data_structures/Binary_tree); * each *node* contains a value; * a total order is defined on these values (every two values can be *compared* with each other); * left subtree of a node contains only values lesser, than the node's value; * rightsubtree of a node contains only values greater, than the node's value.   Notice, that definition above doesn't allow duplicates.  **Example of a binary search tree**  **binary search tree (BST) example**  **Binary search tree. Internal representation**  Like any other dynamic data structure, BST requires storing of some additional auxiliary data, in order to keep its structure. Each node of binary tree contains the following information:   * a value (user's data); * a link to the left child (auxiliary data); * a link to the right child (auxiliary data).   Depending on the size of user data, memory overhead may vary, but in general it is quite reasonable. In some implementations, node may store a link to the parent, but it depends on algorithm, programmer want to apply to BST. For basic operations, like *addition*, *removal* and *search* a link to the parent is not necessary. It is needed in order to implement iterators.  With a view to internal representation, the sample from the overview changes:  binary search tree internal representation  Leaf nodes have links to the children, but they don't have children. In a programming language it means, that corresponding links are set to *NULL*.  **Binary search tree. Removing a node**  Remove operation on binary search tree is more complicated, than add and search. Basically, in can be divided into two stages:   * search for a node to remove; * if the node is found, run remove algorithm.   **Remove algorithm in detail**  Now, let's see more detailed description of a remove algorithm. First stage is identical to algorithm for lookup, except we should track the parent of the current node. Second part is trickier.  There are three cases, which are described below.   1. Node to be removed has no children.   This case is quite simple. Algorithm sets corresponding link of the parent to NULL and disposes the node.  **Example.** Remove -4 from a BST.  BST remove example, remove -4 from the tree   1. Node to be removed has one child.   It this case, node is cut from the tree and algorithm links single child (with it'ssubtree) directly to the parent of the removed node.**Example.**  Remove 18 from a BST.  BST remove example, remove 18 from the tree, pic. 1  BST remove example, remove 18 from the tree, pic. 2  BST remove example, remove 18 from the tree, pic. 3   1. Node to be removed has two children.   This is the most complex case. To solve it, let us see one useful BST property first. We are going to use the idea, that the same set of values may be represented as different binary-search trees. For example those BSTs:  the same trees, pic. 1  the same trees, pic. 2  contains the same values {5, 19, 21, 25}. To transform first tree into second one, we can do following:   * + choose minimum element from the right subtree (19 in the example);   + replace 5 by 19;   + hang 5 as a left child.   The same approach can be utilized to remove a node, which has two children:   * + find a minimum value in the right subtree;   + replace value of the node to be removed with found minimum. Now, right subtree contains a duplicate!   + apply remove to the right subtree to remove a duplicate.   Notice, that the node with minimum value has no left child and, therefore, it's removal may result in first or second cases only.  **Example.** Remove 12 from a BST.  two children case, pic. 1  Find minimum element in the right subtree of the node to be removed. In current example it is 19.  two children case, pic. 2  Replace 12 with 19. Notice, that only values are replaced, not nodes. Now we have two nodes with the same value.  two children case, pic. 3  Remove 19 from the left subtree.  two children case, pic. 4 | |
|  | |
| -------------------------------------------------------------END------------------------------------------------------------ | |