Fundamentals of Programming 2

Binary Search Trees (BST) — Part Two

Arkadiusz Chrobot

Department of Computer Science

May 28, 2019

Outline

Introduction

In this part of the lecture the defining of operations for the BST is continued. The majority of functions that implement that operations is defined both in recursive and iterative form, except for the functions that implement the operation of removing a single node from the BST. Those ones are defined only in iterative form. In the final part of the lecture changes are discussed that have to be introduced to the program from the previous lecture, to invoke the new functions.

BST Search

Let's start with defining functions which find the nodes in BST storing the minimal and maximal key. Those functions return addresses of the aforementioned nodes. If those nodes don't exists the functions will return the `NULL` value.

Searching In BST

The Minimal and Maximal Key

Ordering of the nodes in a BST helps in locating the node that stores minimal or maximal key. The node with minimal key is the leftmost node of the BST, and the node with maximal key is the rightmost node of the BST. Such nodes don't exist only when the BST is empty. Even a BST with only one node has a node with minimal and maximal key — this is the same node.

Searching In BST

Minimal Key

Finding the node with minimal key in BST consists of traversing the tree starting with the root and directing to the left, until a node which field that points to the left child has a `NULL` value is found. The next slide contains a recursive function that implements that algorithm. If the function is applied to a BST node which is not a root then it will locate a node with minimal key in a subtree where the given node is a root.

Searching in BST — Minimal Key

Recursive Version

```
struct tree_node *find_minimum(struct tree_node *root)
{
    if(root && root->left_child)
        return find_minimum(root->left_child);
    else
        return root;
}
```

Searching In BST — Minimal Key

Recursive Version

The function from the previous slide returns the address of the BST node storing the minimal key or a `NULL` value if such a key doesn't exist. It has only one parameter. The parameter is used for passing the address of the root or, in the case of recursive invocations, the addresses of subsequently visited nodes of BST. In the 3rd line the function checks if the node pointed by the `root` parameter exists and if its left child also exists. If the first expression in the condition evaluates to `false` then it means that the function has been invoked for an empty tree. Therefore, it returns the `NULL` value and terminates. If the second expression in the condition evaluates to `true` then it means that the currently visited node is not the one that the function searches for, and hence it invokes itself for the left child of the node (4th line). Those invocation are repeated until the function finds the leftmost node of the BST. The address of the node is eventually returned by the function.

Searching In BST — Minimal Key

Iterative Function

```c
struct tree_node *find_minimum(struct tree_node *root)
{
    while(root && root->left_child)
        root = root->left_child;
    return root;
}
```

Searching In BST — Minimal Key

Iterative Function

The iterative version of the `find_minimum()` has the same prototype as its recursive counterpart. The body of the function contains a `while` loop in which the `root` pointer parameter is used for traversing of BST nodes, starting with the root and finishing with the one that has no left child. If the function is invoked for an empty tree then the loop won't be performed even once. After the loop terminates the function return the address stored in the `root` variable and also terminates.

Searching In BST — Maximal Key

The operation of searching for the BST node with maximal key is performed similarly to the operation of searching for the node with minimal key. The only difference is that this time the key is stored in the rightmost node of the tree. It means that the node has no right child. Searching for that node fails only when the operation is performed for an empty tree. If the operation is performed for a node other than the root of the BST then it will find a node of the BST that stores a maximal key in the subtree in where the former node is a root. The next two slides present the recursive and iterative versions of a function that implements the operation of searching the BST for the maximal node. Because those functions are very similar to the functions for searching the node with minimal key, they are not described in details.

Searching In BST — Maximal Key

Recursive Version

```
struct tree_node *find_maximum(struct tree_node *root)
{
        if(root && root->right_child)
            return find_maximum(root->right_child);
        else
            return root;
}
```

Searching In BST — Maximal Key

Iterative Version

```c
struct tree_node *find_maximum(struct tree_node *root)
{
    while(root && root->right_child)
        root = root->right_child;
    return root;
}
```

Searching In BST — a Given Key

The algorithm for finding a node with a given key is similar to the algorithm for searching a place in the BST for a node in the operation of adding a new node to the tree. By comparing the given key with the key stored in the currently visited node it can be decided in which subtree (left or right) should be the searched node. The operation is finished when the currently visited vertex has the same key as the given key or when there are no nodes left to visit. In the latter case the node storing a given key does not exists.

Searching In BST — a Given Key

Recursive Version

```
struct tree_node *find_node(struct tree_node *root, int key)
{
        if(root && root->key > key)
            return find_node(root->left_child,key);
        else if(root && root->key < key)
            return find_node(root->right_child,key);
        else
            return root;
}
```

Searching In BST — a Given Key

Recursive Version

The function presented in the previous slide returns the address of the BST node and has two parameters. The first parameter, which is a pointer, is used for passing the address of the root or, when the function is called recursively, the address of the left of the right child of a currently visited node. The second parameter is used for passing the given key. In the 3rd line the function check whether the address passed by the `root` parameter is not `NULL` and if the key stored in the node pointed by the `root` parameter is greater than the key the function searches for. If the first expression in the condition evaluates to false when the function is called then it means that it is invoked for an empty tree. If the same expression evaluates to false when the function is called recursively then it means there is no node in the tree that stores the given key. If both expressions evaluate to true then the function is invoked recursively and the first argument of this call is the address of the left child of the currently visited node (4th line).

Searching In BST — a Given Key

Recursive Version

If the condition in the 3rd line of the function is not satisfied then the function tests condition in the 5th line. Once again it checks if the `root` parameter has a value different then `NULL`. It is necessary, because the function performing the statements in the 5th line has no information whether the same expression in the 3rd line evaluated to true or false. If the node pointed by `root` exists then the function checks if the key that it stores is less than the given key. If so, the function is called for the node's right child. The recursive calls can ceased because of two reasons. The first one is that the `root` parameter becomes an empty pointer. That means that there is no node that stores a given key in the BST and the function returns `NULL` (8th line). The second reason is that the node doesn't have a key that is less than or greater than the given one and it means there is only one possibility — the node stores the given key and the function returns its address and terminates.

Searching In BST — a Given Key

Iterative Version

```
struct tree_node *find_node(struct tree_node *root, int key)
{
    while(root) {
        if(root->key == key)
            return root;
        if(root->key > key)
            root = root->left_child;
        else
            root = root->right_child;
    }
    return root;
}
```

Searching In BST — a Given Key

Iterative Version

The prototype of the iterative version of the `find_node()` function is the same as in the recursive version. The tree is traversed with the use of the `while` loop, which repeats itself as long as the parameter `root` has a value different than `NULL`. The value of the parameter is changed inside the loop. If the key stored in the currently pointed by the `root` node is equal to the given key (4th line) then the function returns the address of that node and terminates (line no. 5). If not, but the key stored by the node is greater than the given key (line no. 6) then the `root` pointer is assigned the address of the left child of the node pointed by the `root` (7th line). If even this condition is not satisfied then the address of the right child of that node is assigned to the `root` pointer. If the `while` loop terminates because the condition in the 3rd line is not satisfied then it means that there is no node in the tree that stores the given key. In such a case the function returns `NULL` value and terminates (11th line).

Removing a Node

The operation of removing a single node for the BST removes a node that has a given key. While implementing such an operation the following cases should be considered:

1. the node storing a given key doesn't exist — no action needs to be taken,
2. the node has no children — the node can be deleted, but a pointer field (`left_child` or `right_child`) of its parent that points to the node has to be assigned the `NULL` value,
3. the node has only one child — before the node is removed, the address of its child has to be assigned to the pointer field of its parent that points to the node, just like in the previous case,
4. the node has two children — it's the most complicated case; the node cannot be simply removed; another node should be found in the BST that could be removed instead.

Removing a Node

In the last case the aforementioned another node can be the predecessor or successor of the node originally to be removed. The predecessor is the node that stores the greatest key from all the keys smaller than or equal to the key stored in the original node. The successor is the node that stores the smallest key from all keys greater than the key stored in the original node. The predecessor is also the rightmost node in the left subtree of the original node and the successor is the leftmost node in the right subtree of the original node. Before the successor or predecessor will be removed the data from that node should be assigned to the original node.

In the operation implemented in this lecture the predecessor of the node that has two children is removed. The description of that implementation starts with the presentation of a function that isolates (i.e. finds and unlinks) the predecessor from the tree. Then the function that handles all four cases introduced in the previous slide is explained.

Removing a Node

The `isolate_predecessor()` Function

```
struct tree_node *isolate_predecessor(struct tree_node **root)
{
        while(*root && (*root)->right_child)
            root = &(*root)->right_child;
        struct tree_node *predecessor = *root;
        if(*root)
            *root = (*root)->left_child;
        return predecessor;
}
```

Removing a Node

The `isolate_predecessor()` Function

The function returns the address of the predecessor of the BST node effectively pointed by the `root` parameter, which is a pointer to a pointer. It should be invoked only from within the function that deletes a BST node and then and only then if the node has two children. The function takes one argument. It is the address of the left child (which is also the root of the left subtree) of the node to be removed. In the `while` loop (lines no. 3 and 4) the function traverses the left subtree taking its rightmost branch until it finds the rightmost node of the subtree. Please note, how the `root` pointer to a pointer is used inside the loop. In each iteration the address of pointer field in which is stored the address of the right child of the currently visited node is assigned to the pointer. The loop terminates when the node is located that has no right child. Testing in the 3rd line if the expression `*root` doesn't evaluate to `NULL` is redundant.

Removing a Node

The `isolate_predecessor()` Function

After the predecessor of the node to be removed is found the function stores its address in a local pointer named `predecessor`. Then, after checking that the predecessor exists (6th line), which is also redundant, the function assigns the address stored in its `left_child` field to the variable pointed by the `root` pointer. It is necessary for two reasons. The predecessor doesn't have a right child, but it still may have a left child. If such a child exists then its address has to be stored in the pointer field of the predecessor's parent that points to the predecessor. Otherwise the left child and possibly the whole subtree associated with that node could be lost. If the left child doesn't exist then in the predecessor's `left_child` pointer field is stored the `NULL` value. It should also be assigned to the pointer field of the predecessor's parent that points to the predecessor, after the latter is unlinked from the tree. The statement in the 7th line handles both cases. After the predecessor is unlinked, the function returns its address and terminates (8th line).

Removing a Node

```
void delete_node(struct tree_node **root, int key)
{
    while(*root && (*root)->key!=key) {
        if((*root)->key>key)
            root = &(*root)->left_child;
        if((*root)->key<key)
            root = &(*root)->right_child;
    }
    if(*root) {
        struct tree_node *node = *root;
        if(!node->left_child)
            *root = (*root)->right_child;
        else if(!node->right_child)
            *root = (*root)->left_child;
        else {
            node = isolate_predecessor(&(*root)->left_child);
            (*root)->key = node->key;
            (*root)->value = node->value;
        }
        free(node);
    }
}
```

Removing a Node

The `delete_node()` function returns no value but it has a pointer to a pointer parameter. It is necessary because the function can change the value stored in the root pointer or the pointer fields of the BST nodes. By the second parameter is passed the key that identifies the node for removal. In the `while` loop (3rd to 8th line) the tree is traversed in order to locate the node. The `find_node()` function cannot be applied for this task because not only the address of the node is required but also the address of the variable or field that stores the address is needed. This is enabled by a pointer to a pointer. Inside the `while` loop it is also checked if the `*root` expression doesn't evaluate to `NULL` (3rd line). The loop can terminate when the searched node is found or when there are no BST nodes left to traverse. The latter also happens when the function is invoked for an empty BST. The next activities depend on whether the `*root` expression points to an existing BST node or not. This is tested in the 9th line of the function.

Removing a Node

If the expression `*root` evaluates to `NULL` it means that there is no node that should be removed. Otherwise the function assigns the address of the node pointed by `*root` to a local pointer called `node` (10th line) and then it checks if its left child *doesn't* exist. If so, then there is a node to be removed with at most one child — the right child may exist. The address of the child is assigned to the variable pointed by the `root` parameter. The variable can be a root pointer or a pointer field of the parent of the removed node. The assignment protect the right child, possibly along with a whole subtree associated with it, from being unlinked from the BST. If the right child doesn't exist then the `right_child` pointer field stores the `NULL` value that should be assigned to the variable pointed by `root` parameter. Hence, the statement in the 12th line is correct even in this case.

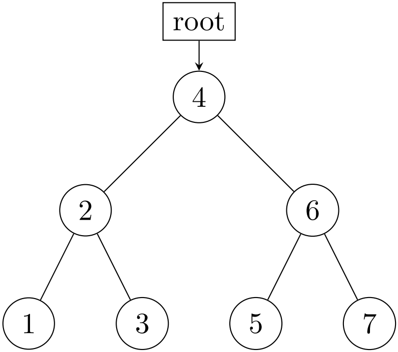
Removing a Node

If the left child exists then the function checks if the right child of the removed node *doesn't* exists. If so it proceeds similarly as when the left child doesn't exist, but this time it is certain that the other child exists (the `left_child` field stores a value different than `NULL`) and its address should be assigned to the variable pointed by the `root` parameter (14th line). Otherwise the left child would be unlinked from the BST. If both children of the node exists (16th line) then the function calls the `isolate_predecessor()` function that finds the predecessor of the node, unlinks it from the tree and returns its address, which is then assigned to the `node` pointer. Next, the `delete_node()` function copies the data from the predecessor to the node that was originally to be removed (17th and 18th lines). Before the function terminates it deallocates the memory allocated to the node pointed by the `node` pointer. In both previously described cases, the address of the node that should be removed is also assigned to the variable (10th line), so in all three cases the correct node is deleted.

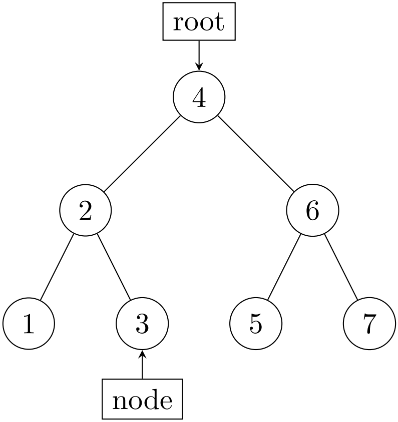
Removing a Node

The next slide contains an animation that shows the operation of removing from the BST a node that has two ancestors (it is also a root of that tree).

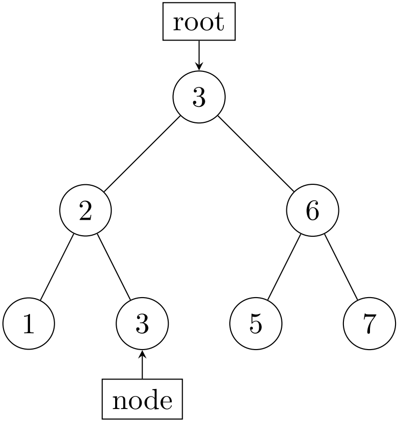
Removing a Node



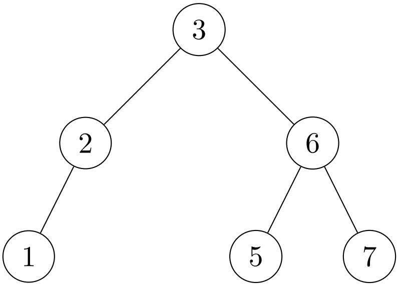
Removing a Node



Removing a Node



Removing a Node



Changes In the `main()` Function

To test the behaviour of functions that have been defined in the lecture it is necessary to invoke them in the `main()` function of the program from the previous lecture, between calling the `add_node()` and the `removing_tree_nodes()` functions. The next slides show an example of a code that can be used for accomplishing the task.

Changes In the `main()` Function

```c
if(root) {
    printw("The value associated with the smallest key: %c\n",
                                            find_minimum(root)->value);
    printw("The value associated with the greatest key: %c\n",
                                            find_maximum(root)->value);
    refresh();
    getch();
}
```

Changes In the `main()` Function

The results of `find_minimum()` and `find_maximum()` functions should be assigned to two different pointers. The content of the nodes should be displayed only after the `main()` function checks if both of the pointer store a value different than `NULL`. But another approach can also be taken, which is demonstrated in the previous slide. The `main()` function checks if the tree is not empty (1st line) before the former functions are called. If so, then the functions will return addresses of existing nodes for sure. Please note, how the functions are called (3rd and 5th lines). Both those functions return addresses of existing nodes that can be directly used for accessing the `value` fields of those nodes.

Changes In the `main()` Function

```c
int key;
scanw("%d",&key);
struct tree_node *result = find_node(root,key);
if(result) {
    printw("The value associated with the key %d is %c\n",
                                                result->key, result->value);
    refresh();
    getch();
    erase();
    delete_node(&root,key);
    print_keys(root,COLS/2,1,20);
    refresh();
    getch();
}
```

Changes In the `main()` Function

The code from the previous slide asks user to input a key which is then searched in the BST (2nd line). If the key is found by the `find_node()` function then the content of the node that stores it is displayed on the screen (lines no. 5 and 6). Next, the node is removed from the tree (10th line) and the keys stored in the BST are displayed in a way that shows the shape of the tree (11th line). The program, that is available on the course's web page, uses a macro called `RECURSIVE`, which is a marker. If it is defined at the beginning of the program code or as a compilation option, then in the compiled program only the recursive functions that implement some of the BST operations will be used. Otherwise the iterative functions will be used.

Summary

Among the operations that are described in the lecture the most time–consuming one is the operation of searching/traversing the BST. The operation is also frequently performed as a part of other operations. The time needed to perform it is proportional to the hight of the tree. In the case of a full binary tree, the hight can be computed with the use of the following formula $log_2(n)$, where n is the number of the nodes in the tree. In most cases the keys in the trees are randomly dispersed and the data structures have shapes close to the shape of the full binary tree. Hence, the BST is usually a very effective data structure. The possibility of implementing a tree with the use of an array has been mentioned in the first part of the lecture. The example of such an implementation in case of the BST is ... a sorted array. Just as in the BST the keys (and possibly values) are in ascending order starting from left to the right. If the binary search algorithm is applied for searching a key in such an array, then the time complexity of such an operation is the same as for the BST searching.

Summary

Using an array allows for implementing other types of trees. An example of such an implementation is presented in the next lecture. Not all BSTs have the same or even close shape as the full binary tree. The corner cases are BSTs in which nodes with already ordered ascending or descending keys are inserted. Those trees have the same shape as linear lists. The time needed for traversing such a tree is directly proportional to the number of its nodes. To avoid such a shape of a tree the operation of *balancing* the BST is applied. The tree that is created with the use of such an operation is called a *balanced tree*. Examples of such trees are the AVL trees and the red–black trees, which however won't be discussed in the lecture.

Questions

?

The End

Thank You For Your Attention!