avl tree

Divyanshu Nauni

September 2021

1 functions and their logic

1.1 Logic of insertion

For insertion we first take the key value to be inserted in a variable key and pass it as a argument to insert function .

Step1: We first check wether there is any key in the tree or it is empty. We check if the right child of root is null or not as root is the dummy pointer whose right child points to actual root of the tree. If its empty then we store the value in its right child and balance factor of it is 0 as it is only node and return from the function. Else if the root is not empty then we initialize four pointers parImBal,ImBal,iterator and q . Where parImBal is parent of node that can have imbalance, ImBal points to node that can have imbalnce after insertion, iterator is the pointer to iterate through the tree.

First we assign parImBal to root and ImBal and iteratot to right child of root. Then if the key value is less than value of iterator then we set the value of q to left child of iterator else we set it to right child. If the q points to null then we break out of the while loop. Else we check if the balance factor of q is equal to zero . if yes we do nothing and update iterator to q. If the balance of q was not zero then we set parImBal to iterator and ImBal to q then set p to q. We do this until iterator reaches leaf . Then we insert the new node to its correct position and set q to iterators child.

Step2: in step 1 our three pointers are saved at three location first is the ImBal which points to node which may have imbalance and second is its parent pointed by parImBal then we have new node pointed by q. Now in this step we will set the balance factor between ImBal to q .
Here we will declare one variable a which will store 1 or -1 depending on key and value of s, if the key is less than value of s then a will be 1 else -1 . Now we will set a new pointer ImBalChild which will point to child of s depending on value of a.
If a is 1 then both iterator and ImBalChild will point to left child of s else both will point to right child. Now ImBalChild will not move but the iterator will

move and until it reaches q it will update the balance factor of each node in its path. If key value is less than value of iterator then balance factor of iterator will be 1 and it will move to its left child. Else it will set its balance factor to -1 and it will point to its right child.
Now all node from ImBal to q has been restored to correct balance factor.

Step 3: here we will balance the nodes pointed by ImBal , ImBalChild, and iterator. First we have to check whether rotation is needed or not. Suppose the balance factor of ImBal is 0 then whatever be the value of a we must not have made any imbalance as the balance factor will just increase by 1 or decrease by 1 so we set balance factor of ImBal node to a and return.
If the balance factor of ImBal is opposite of a then it must have balanced the balance fator of ImBal and we set balance factor to 0. Else Here we have introduced imbalancing in the tree and now we need to make necessary rotations in the tree .
only three nodes needed to be changed depending on the type of imbalance. We categorized the imbalance based on nature of rotation. That is single rotation and double rotation.
If the value of ImBalChild is same as that of a then we have a single rotation else we have a double rotation.
Now if we have a single rotation we have two cases wither it is LL or RR rotation.

1.1.1 single rotation

1. **LL**
   if we have a = 1 that is LL. Then we will set iterator to ImBalChild. First we will set right child of ImBalChild as a left child of ImBal. Then we will set ImBal as right child of ImBalChild as after rotation ImBalChild becomes parent of ImBal. Now we set balance factor of ImBal to 0. Then we set balance factor of ImBalChild to 0 . these both balance nodes have balance factor as 0 because before rotation s was having a imbalance of 1 but after rotation its left child became its parent and now it will be having same height on its both children.

2. **RR**
   The iterator will point ImBalChild. Right child of ImBal will be left child of ImBalChild . And ImBalChild will have its left child as ImBal. Now the same logic is applied here as In case of LL rotation. That the node ImBal was having a imbalance of -1 before and now after rotation its right child become its parent and its both children are at same height so its balance factor will be 0. Now the balance factor of ImBalChild was previously one less on left side but now its left child is having one more height after rotation and left and right child are at same height so its balance factor is also 0.

1.1.2 double rotation

When balance factor of ImBalChild is not equal to balance factor of a then we have a double rotation

1. **LR**
   When the value of a is 1 then we have LR rotation. Here the pointers will first right toation and then left rotation. Here we will see how balance factor will be changed: if balance factor of iterator is 0 then either iterator has no child or both children, is it has both then ImBalwill receive rchild of iterator and ImBalChild will receive left child of itereator after rotation and thus balance factor of ImBalChild and ImBal becomes 0 else if it is not zero then it can be 1 or -1 . if bal fact (iterator)==1 means after rotation ImBalChild will receive its left child but ImBalwill not receive any child of iterator and thus balance factor of ImBalChild will become 0 and balance factor of ImBalwill become -1 if bal fact(iterator)==-1 then bal factor of ImBalbecomes= 0 but bf of ImBalChild becomes 1 as it receives no right child.

2. **RL**
   When the value of a =-1 Here the logic is opposite to LR. After rotation if earlier iterator had both child or no child then after rotation its left child will become right child of ImBal and its right child will become left child of ImBalChild . and the balance factor of ImBal and ImBalChild will become zero. Now if earlier iterator had only left child then bal factor of ImBal will become 0 but ImBalChild has not received any node on its left so its balance factor will be -1 as it will have one more node on right side. finally we will set the child of parImBal as iterator depending on the value of ImBal.

1.2 logic of search

: Here we pass key as an argument. We initialize a node pointer say searchPtr which points to right child of root. If the searchPtr is null then we return false Else we compare key value with data value of searchPtr node. If key is less then we shift the searchPtr to its left child else we shift it to its right child. If the key value is equals to the value of searchPtr value then we return true . else we continue until searchPtr becomes null. We then break out of while loop and return false

1.3 logic of print tree

This function make a dot gv file with the nodes of the tree along with the links. The structure of the node is —left—key—balance factor—right— . So to the traversal used in this function is level order. Here each node is first put into a queue and then popped out. If the left child and right child of the node are not null then we put first left and then right child into the queue. Now we write the

format of node and key value of node along with its balance factor in a string . In another string we write the links of its child . Then we put both in graphviz file and keep doing until the queue is empty. Then to get the .png file we open command prompt and write the command .

1.4 Logic of deletion

Step 1 : first step in deletion is to search whether the key is present or not so I have called search function with the value key and if the return value is false then I exit from the function . Else if the function return true then the element is present. So we have to store all the node addresses from root till the node in that path so that while returning we can balance each node . So first we create a stack then we push node address in it. We then initialize a node pointer which points to right child of root.https://www.overleaf.com/project/6130e75ffc885061c044480b We compare the key value with this pointer and put address of pointer in stack. We keep on doing this until we find the node which we want do delete.

Step 2: in this part we will delete the node but first we will check whether it is a leaf or a node with only on child or a node with both children.

If the node to be deleted is leaf then we delete it and put its parents left/right child to null depending on which node it is.

If the node to be deleted has only one child then in this case that child can not have any other child as balanced tree property will be voilated. We put value of its child in it , and also copy balance factor of child to node to be deleted. Now we delete child.

If the node to be deleted has both children, First we put address of that node onto the stack. Now we find the inorder successor of the node by checking the smallest elemnt in its right subtree. Until we find the successor we put address of the nodes in the path onto the stack. Now we find the successor , we copy the value of successor to the node to be delete , set the key to be the key of successor because we are now deleting successor node . We initialise a parent pointer of the successor and a temporary pointer pointing to right child of successor since successor cant have a left child. Now if temporary pointer is null it means successor has no child and is leaf. So in this we set the left/right child of parent to null depending on where successor was. And successor pointer points to null. Else if successor has a right child then we set key of successor as key of its child. Copy the balance factor of child to successor node and set left and right child of successor as null or left or right child of temporary . Now we delete temporary pointer node.

Step 3: here we will traverse back and set balance factor of each node which is in stack until stack is empty. Here we define three pointers ImBal-¿ which points to node which we just popped from stack and which can have imbalance. parImBal which is the parent of the node ImBal , and ImBalChild which is child of ImBal.

First we have to check if we require rotation or not.

If the balance factor of the node ImBal is equal to the value of a then no need to do rotation and the new balance factor of the node ImBal becomes zero. The

reason for this is that a can only take +1 or -1 value , but node can have three values for balance factor. Now, if we consider that ImBal had balance factor 1 and the key deleted was less than the value of ImBal then now ImBal will have one less key on the side so its balance factor is set to 0. If the balance factor of ImBal node was previously 0 then after deletion we will not have disturbed balance factor of this node, we can also say about nodes ancesstor of this node that their height will not be disturbed because this node is still balanced and new balance factor will be opposite to value of a because a tells us that deletion is done on left or right child , so if deletion is done on right child we will have more height on left side and vice versa. So new balance factor of ImBal will be negative of a. and then we will exit from the delete function as no more checking is required.

if above two conditions do not satisfy then we need rotation. There can be four types of rotations

Here first we will set the pointer ImBalChild to left or right child of ImBal. If the balance factor of ImBal is 1 then ImBalChild will point to left child of ImBal else it will point to right child.

1.4.1 single rotation

Case 1: If the balance factor of child of ImBal is negative of a value then If the value of a is negative 1 it implies that ImBalChild is having one more node on either side. Now to know which type of rotation it is we will need value of a, If value of a is positive 1 then it means ImBal is having more element on right side because element is deleted on left side of ImBal , so it is a case of RR rotation. And after rotation we balance the pointer and set balance factor of both ImBal and its child to 0. Because now ImBal child will be having same height on both children as its left child height increased by one and right child height was same .

If the value of a is negative 1 then it means ImBal is having more height on left side after deletion so we now do LL rotation. And after rotation we set the balance factor to 0.

Case 2: The balance factor of ImBalChild is 0. it implies that the child of ImBal is balanced and having same height on both children. If the value of a is positive 1 then we have deleted element on left side of ImBal. So we need RR rotation to balance it. We will do necessary pointer movement and now to set the new balance factor of ImBal and ImBalChild, earlier ImBalChild was having balance factor of 0 but now it is having one more height on the side where ImBal is present that is left side so its balance factor is changed to that of value of a.

If the value of a was negative 1 then we have deleted node from right side of ImBal and now we need LL rotation so after doing necessary rotations we set the balance factor of ImBal and ImBalChild as follows : earlier ImBalChild was having balance factor of 0 but after rotation its right child becomes left child of Imbal and its new right child has one more node so balance factor is now negative of value of a

1.4.2 double rotation

when the value of balance factor of ImBalChild is equal to value of a means we have deleted

Case 3: here two types of rotation are there .

1. **RL**
   RL rotation: if the value of a is +1 then the itrPoint will point to left child of ImBalChild and now rotation will happen in these three nodes that are ImBal , ImBalChild and itrPoint. Here itrPoint will become new parent and left child is ImBal and right child is ImBalChild . Now to balance the factors of itrPoint, ImBal, ImBalChild. If the balance factor of itrPoint is 0 then ImBal and ImBalChild both will have new balance factor as 0. If the balance factor of itrPoint is 1 then the new balance factor of ImBal will be 0 and balance factor of ImBalChild will be -1. If the balance factor of itrpoint is -1 then the new balance factor of ImBal will be 1 and that of ImBalChild will be 0. Now we set the new balance factor of itrPoint to 0.

2. **LR**
   LR rotation: If the value of a is -1 then itrPoint will point to right child of ImBalChild . After rotations the parent is itrPoint,left child is ImBalChild and right child is ImBal. Now to balance the factors of itrPoint, ImBal, ImBalChild. If the balance factor of itrPoint is 0 then ImBal and ImBalChild both will have new balance factor as 0. If the balance factor of itrPoint is 1 then the new balance factor of ImBal will be -1 and balance factor of ImBalChild will be 0. If the balance factor of itrpoint is -1 then the new balance factor of ImBal will be 0 and that of ImBalChild will be -1. Now we set the new balance factor of itrPoint to 0.

2 outputs

2.1 after insertion

the insertion sequence is 30,20,15,45,50,60,70,10,5,35,75,25,17,23

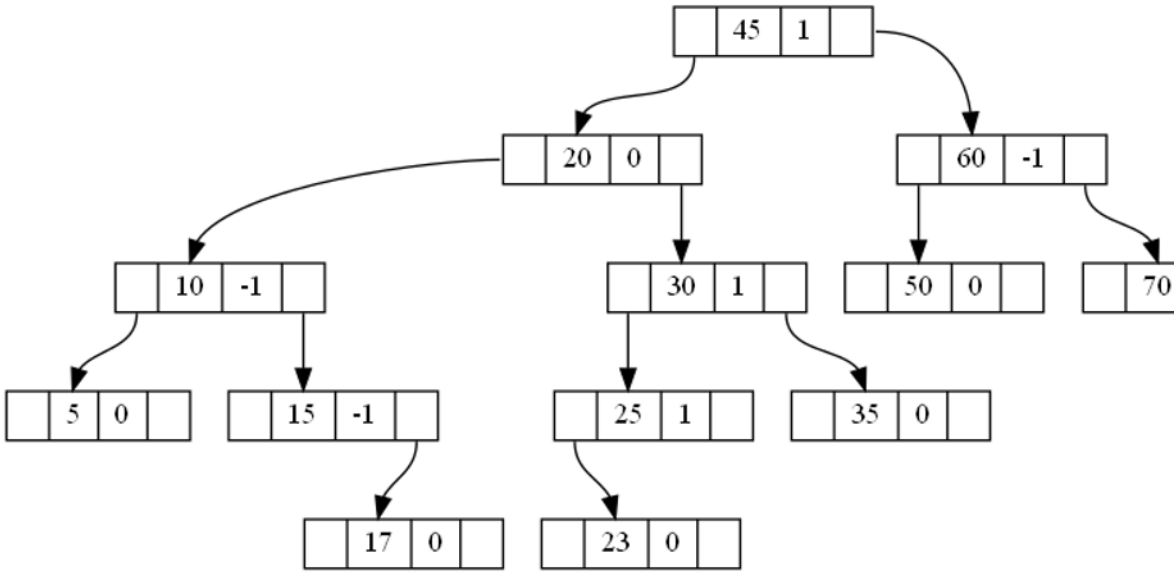

fig:inserting all keys

2.2 after deleting 17

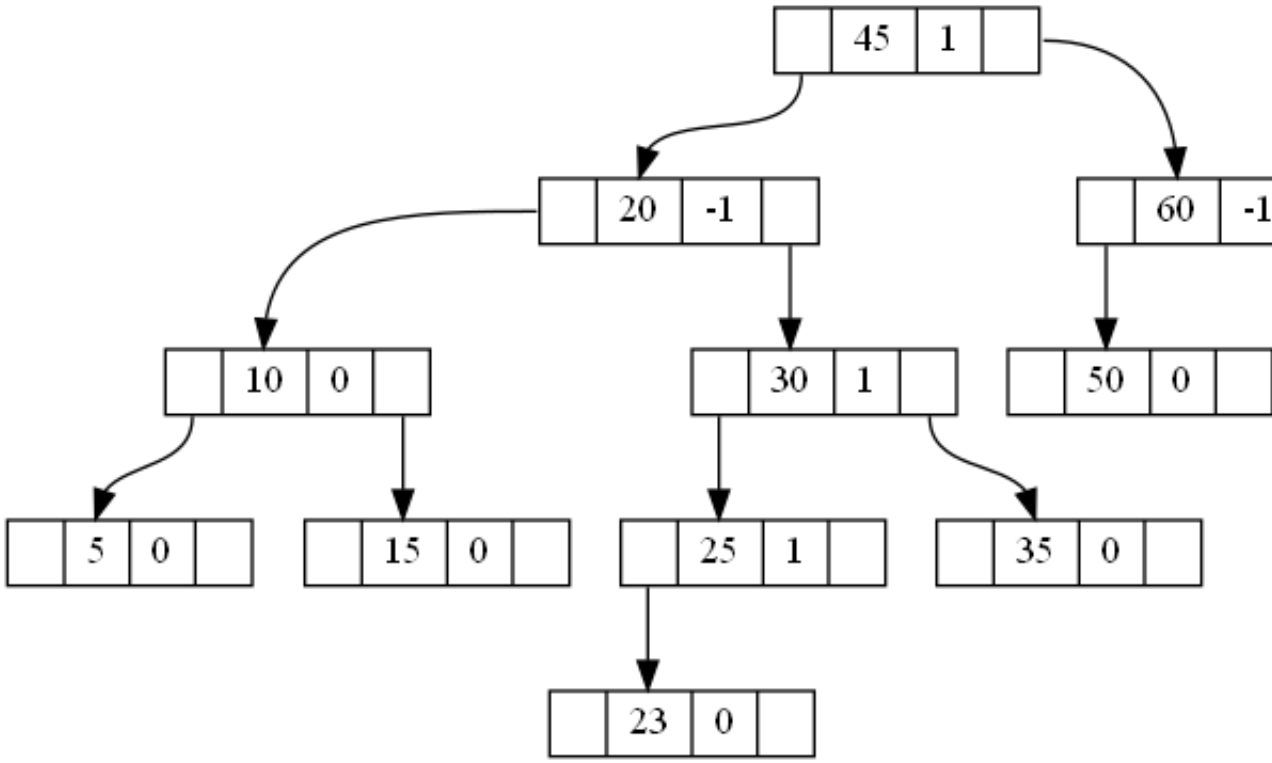

fig:deleting 17

2.3 after deleting 25

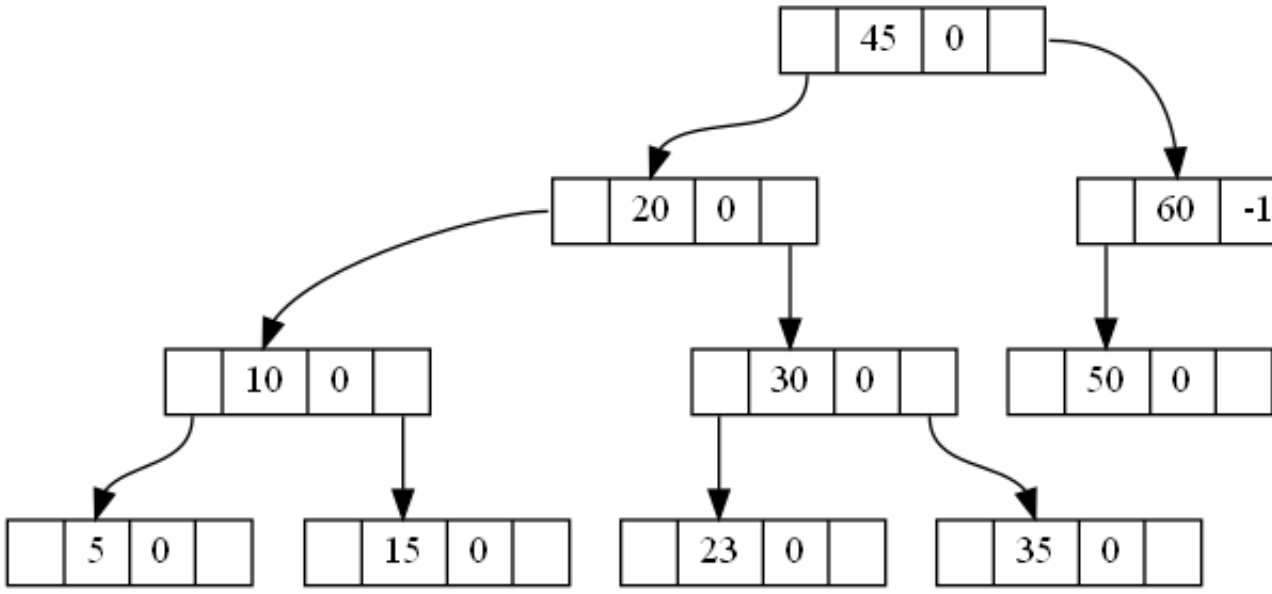

fig:deleting 25

2.4 after deleting 30

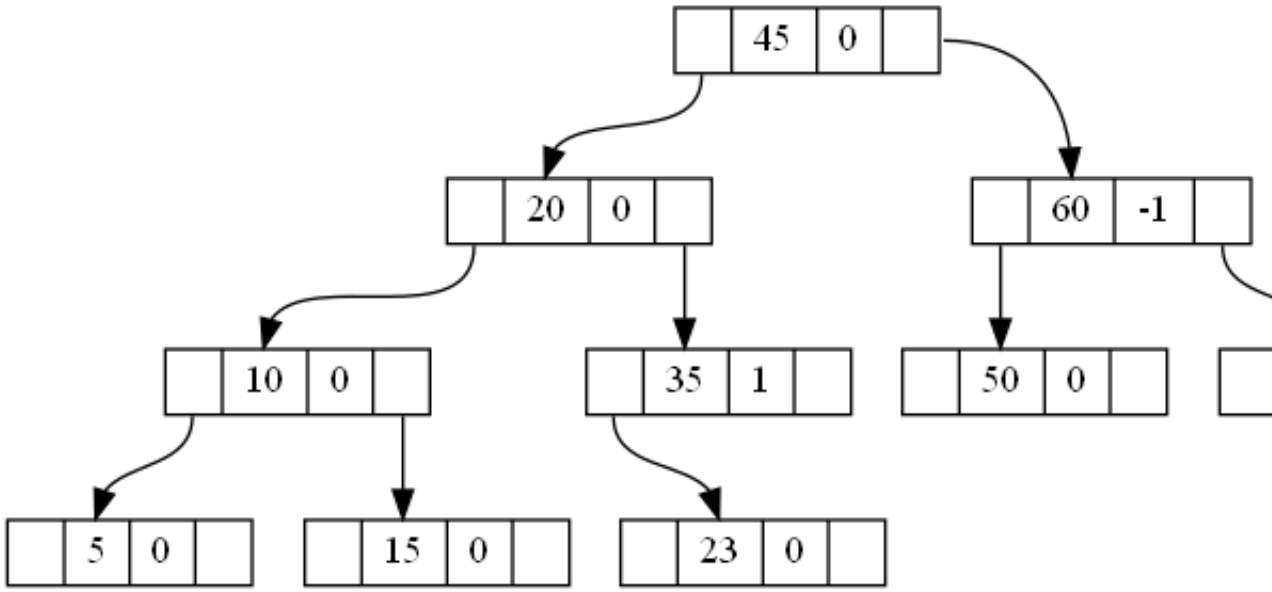

fig:deleting 30

2.5 after searching 17

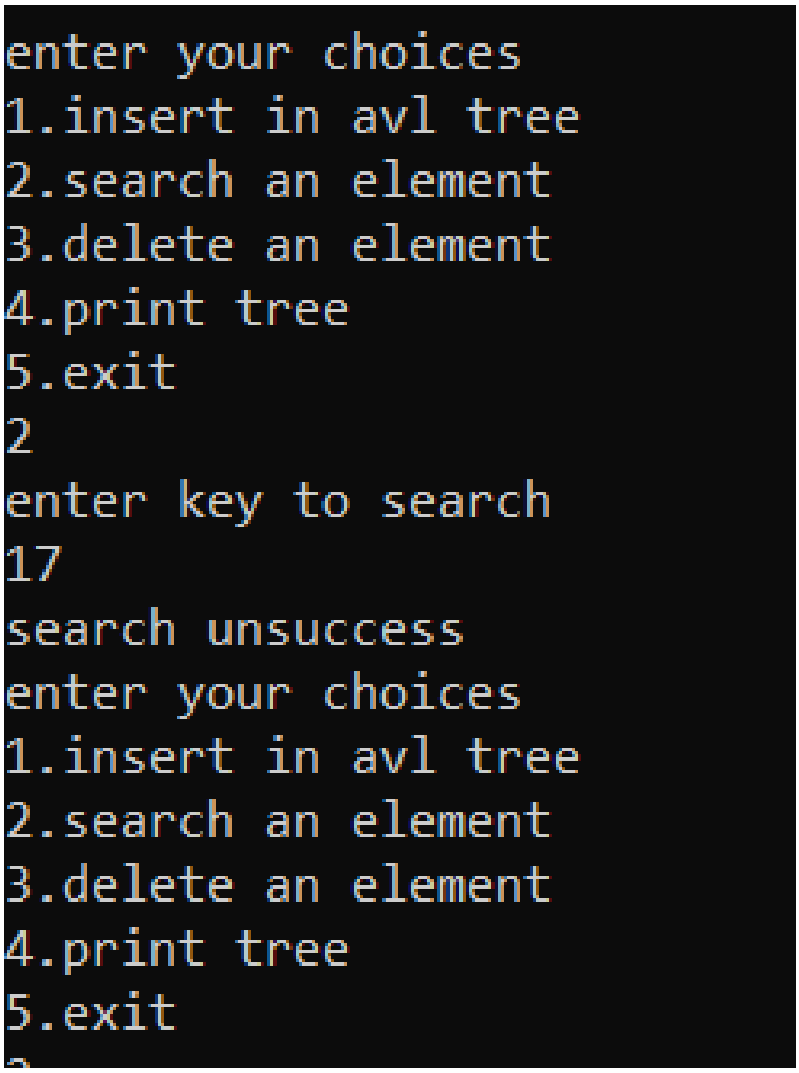

fig:searching deleted node 17

here we can see that the search results in a failure.

2.6 after searching 23

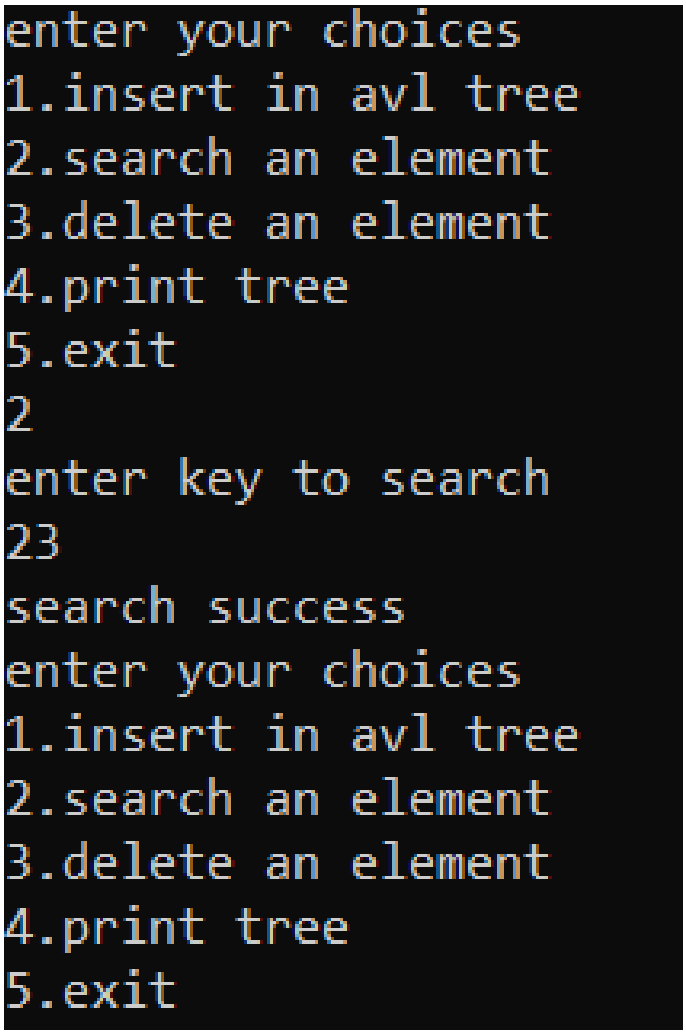

fig:searching 23