Lab 6
Binary Search Tree (BST)

Quang D. C.
dungcamquang@tdtu.edu.vn

October 23, 2023

Note

In this tutorial, we will continue to practice with a new data structure call Binary Search Tree. We will learn some problems of Binary Search Tree:

- Create a Node of BST
- Insert a Node to BST
- Tree traversal
- Search a key
- Find minimum key and maximum key
- Delete the node containing minimum key
- Delete a Node in BST

Part I
Classwork

In this part, lecturer will:

- Summarize the theory related to this lab.
- Instruct the lesson in this lab to the students.
- Explain the sample implementations.

Responsibility of the students in this part:

- Students practice sample exercises with solutions.
- During these part, students may ask any question that they don't understand or make mistakes. Lecturers can guide students, or do general guidance for the whole class if the errors are common.

1. What is Binary Search Tree?

Binary Search Tree is a data structure which extends from binary tree but it has a rule: "$x.left.key < x.key < x.right.key$". Visualizing this rule:

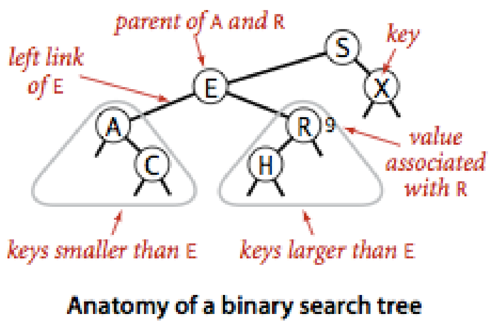


Figure 1: Binary search tree

2. Create a Node of BST

Each node of BST we will define as a class includes: a key, a left node, and a right node. The *left node* is the node that contains a smaller key, the *right node* is the node that contains a larger key.

```java
public class Node {
    Integer key;
    Node left, right;

    public Node(Integer key) {
        this.key = key;
        this.left = this.right = null;
    }
}
```

3. Insert a Node to BST

When we want to insert a node to BST, always remember the rule of BST: **"the key of the right node is larger and the key of the left node is smaller than the key of the current node"**. So when you insert a node to BST not null, it will have the same steps as searching a key in BST.

```java
private Node insert(Node x, Integer key) {
    if (x == null)
        return new Node(key);
    int cmp = key.compareTo(x.key);
    if (cmp < 0)
        x.left = insert(x.left, key);
    else if (cmp > 0)
        x.right = insert(x.right , key);
    else
```

```
        x.key = key;
    return x;
}
```

4. Tree traversal

We have 3 ways to traverse a tree: pre-order, in-order, and post-order. In this section, we will traverse and print the key of each node in the tree. Depend on which way that we have chosen, the output will be different.

4.1. Pre-order

Pre-order (or node-left-right): In this way, the value of the current node will be printed first, then the left subtree of the current node will be traversed next and finally for the right subtree. This process will be performed recursively for all nodes in the tree.

```java
public void NLR(Node x) {
    if (x != null) {
        System.out.print(x.key + " ");
        NLR(x.left);
        NLR(x.right);
    }
}
```

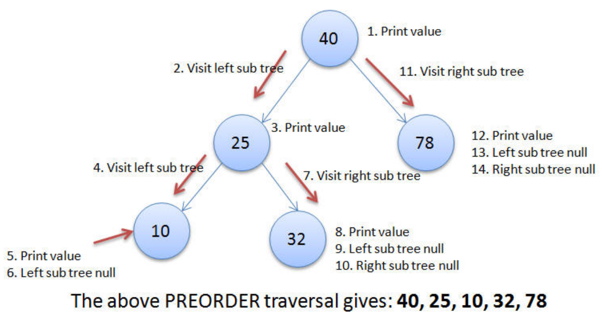


Figure 2: Pre-order

4.2. In-order

In-order (or left-node-right): In this traversing method, the left subtree of the current node is traversed first, then the value of the current node is printed and finally the right subtree is traversed. This process will be performed recursively for all nodes in the tree.

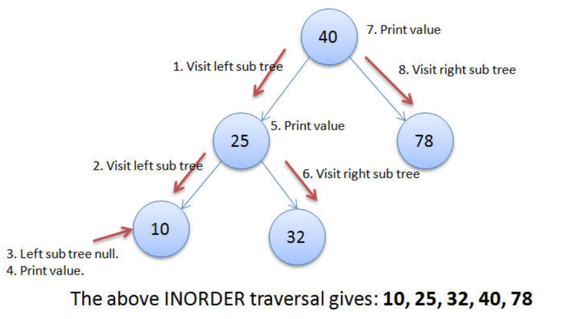


Figure 3: In-order

```java
public void LNR(Node x) {
    //code here
}
```

4.3. Post-order

Post-order (or left-right-node): In this method, the left subtree of the current node will be traversed first, then the right subtree will be traversed next and finally the value of the current node will be printed. This process will be performed recursively for all nodes in the tree.

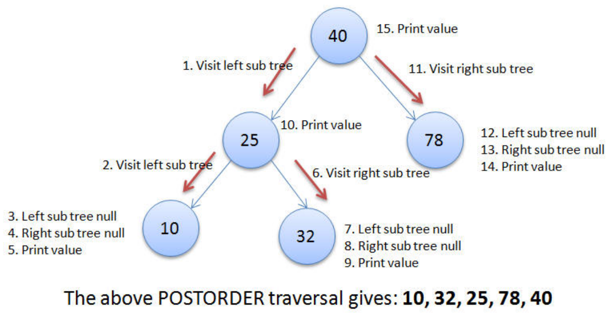


Figure 4: Post-order

```java
public void LRN(Node x) {
    //code here
}
```

5. Search a key

Since the BST tree has a recursive structure, a tree search is easy performed by a recursive algorithm: If the tree is empty, the searching process return null; If the search key is equal to the key at the root node, the searching process ends with the return result as the root node. Otherwise, we will search (recursively) in the next subtree.

```java
private Node search(Node x, Integer key) {
    if (x == null)
        return null;
    int cmp = key.compareTo(x.key);
    if (cmp < 0)
        return search(x.left, key)
    else if (cmp > 0)
        return search(x.right, key)
    else
        return x;
}
```

6. Find minimum key and maximum key

Since the rule of BST, we can find the node with minimum value or the maximum value easily. If we want to find the minimum value, we just need to traverse the node from root to left recursively until left is *NULL*. The node whose left is *NULL* is the node with minimum value. Likewise, traverse the right to find the node with the maximum value.

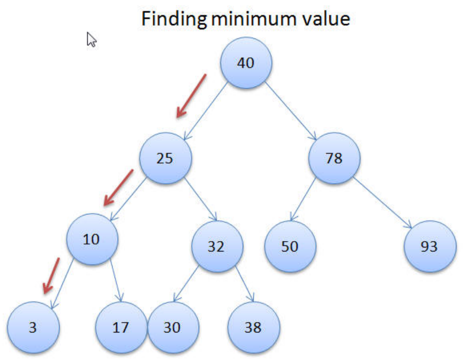


Figure 5: Find minimum value

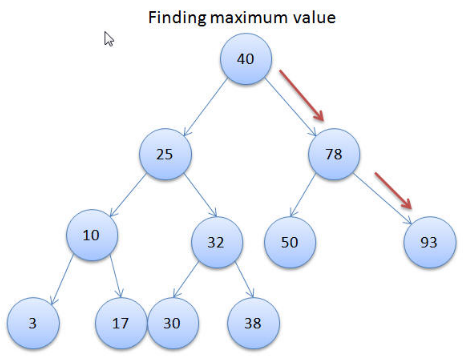


Figure 6: Find maximum value

```
private Node min(Node x) {
    if (x.left == null)
        return x;
    else
        return min(x.left);
}
```

```java

private Node max(Node x) {
        // code here
}
```

7. Delete the node containing minimum key

To delete the node containing the smallest key in the tree, we will continue to traverse the left subtree until we find the smallest node. This smallest node is also the node that has no left child tree. Then replace the link from the parent node to the right child tree (if any).

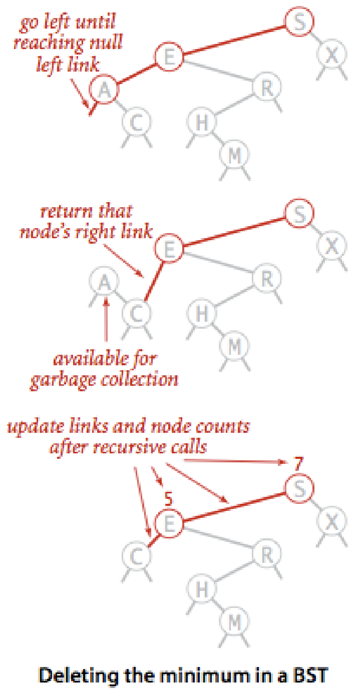

go left until reaching null left link
return that node's right link
available for garbage collection
update links and node counts after recursive calls
7
5
Deleting the minimum in a BST

Figure 7: Delete the node contains mimimum value

```java
private Node deleteMin(Node x) {
    if (x.left == null)
        return x.right;
    x.left = deleteMin(x.left);
    return x;
}
```

8. Delete a Node in BST

We will have 3 cases when we want to delete a node in BST.

1. Node to be deleted is leaf
2. Node to be deleted has only one child

3. Node to be deleted has two children

In case (1) and (2), simply replace the node to be deleted with a left or right child node.

In case (3), we find the successor of the node to be deleted. Use the successor to replace the node which we want to delete and delete the successor. The successor can be obtained by finding the minimum value in the right child of the node to be deleted.

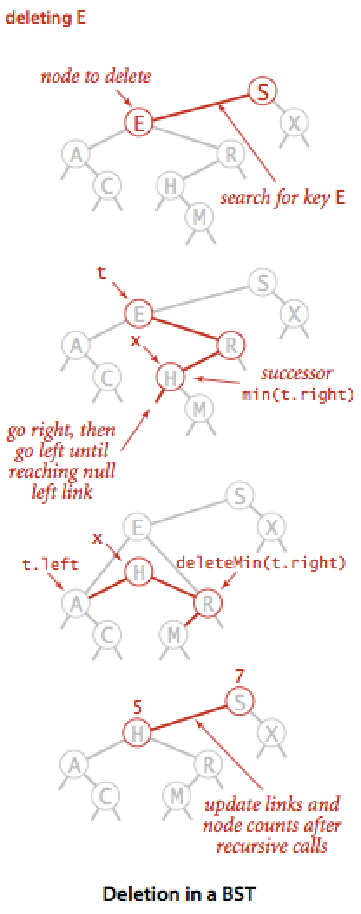


Figure 8: Delete a node in BST

```
private Node delete(Node x, Integer key) {
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if (cmp < 0)
        x.left = delete(x.left, key);
    else if (cmp > 0)
        x.right = delete(x.right , key);
    else {
        // node with only one child or no child
        if (x.right == null)
            return x.left;
        if (x.left == null)
            return x.right;

        // node with two children: Get the successor (smallest in the right subtree)
        x.key = min(x.right).key;
        x.right = deleteMin(x.right);
    }
    return x;
}
```

Part II
Excercise

Responsibility of the students in this part:

- Complete all the exercises with the knowledge from **Part I**.
- Ask your lecturer if you have any question.
- Submit your solutions according to your lecturer requirement.

Exercise 1

Based on the sections above, implement a BST class completely and write the *main* function to check it. *(Notice: the functions which giving in the sections above are **private** so we need to implement the **public** functions to call them with root node)*

Exercise 2

In the class containing the *main* function, define a function call *createTree(String strKey)*. Giving a string of integers (separated by a space character), this function will create a BST tree with the keys of integers following the input string.

Exercise 3

Write the function that prints on the screen the values of the tree in descending order.

Exercise 4

Write *contains* function with input is a **key**, the function returns *true* if the key in the tree. Otherwise, it returns *false*.

```
public boolean contains(Integer key) {
    // your code here
}
```

Exercise 5

Write the function deleteMax() to delete the node containing maximum key in BST.

```java
public void deleteMax() {
    // your code here
}
```

Exercise 6

Write the function to delete a node in BST, but you must use the predecessor instead of the successor.

```java
public void delete_pre(Integer key) {
    // your code here
}
```

Exercise 7

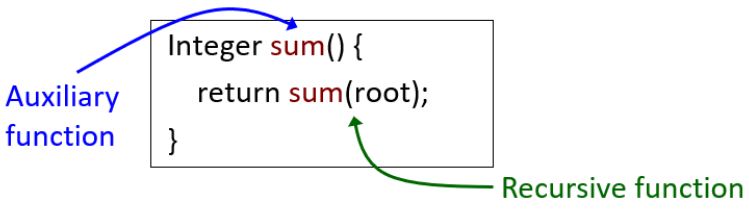
Write the function to calculate the *height* of the BST.

```java
public int height() {
    // your code here
}
```

Exercise 8

Write the function **public Integer sum(Node x)** to calculate the sum of all keys of the subtree **x**. After that, write the auxiliary function *sum()* to calculate the sum of all keys of the BST.

The recursive function is hard to use for the users, so we need an auxiliary function to call the recursive function. Using overloading, the auxiliary function can have the same name as the actual recursive function it calls.



```java
Integer sum() {
    return sum(root);
}
```

Figure 9: Create an auxiliary function

Note: The following exercises only mention the auxiliary functions. Students must write the corresponding recursive functions.

Exercise 9

Write the function *sumEven()* to calculate the sum of even keys in the BST.

```java
public Integer sumEven() {
// your code here
}
```

Exercise 10

Write the function *countLeaves()* to count the leaves of the BST.

```java
public int countLeaves() {
// your code here
}
```

Exercise 11

Write the function *sumEvenKeysAtLeaves()* to sum the even keys stored in the leaves of the BST.

```java
public int sumEvenKeysAtLeaves() {
// your code here
}
```

Exercise 12

Write the function *bfs()* to traverse the BST by level order and print the keys on the screen.

```java
public void bfs() {
// your code here
}
```

THE END