Things to keep in mind

- A Binary Search Tree maintains the property that any element in the left subtree will be less than the root and any element in the right subtree is greater than the root (BST property).
- Any kind of update (Insertion, deletion etc) to the tree should make sure the property is held intact.
- If we delete a node, we should still make sure that the rest of the elements are still there in the tree.

```
struct Node{
    int key;
    Node* left ;
    Node* right;
};

class BST{
    private:
        Node* root;
```

```
**/

Node* BST:: createNode(int data)
{
    Node* newNode = new Node;
    newNode->key = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

BST::BST()
{

}

/**
parameterized constructor. It will create the root and put the data in the root.
**/

BST::BST(int data)
{
    root = createNode(data);
    cout<< "New tree created with "<<data<<endl;
}
```

Search a key in BST

```
Node* BST::searchKeyHelper(Node* currNode, int data){
    if(currNode == NULL)
        return NULL;

    if(currNode->key == data)
        return currNode;

    if(currNode->key > data)
        return searchKeyHelper(currNode->left, data);

    return searchKeyHelper (currNode->right, data);
}

// This function will return whether a key is in the tree
bool BST::searchKey(int key){
    Node* tree = searchKeyHelper(root, key);
    if(tree != NULL) {
        return true;
    }
    cout<<"Key not present in the tree"<<endl;
    return false;
}
```

Search a key in BST

```cpp
Node* BST::searchKeyHelper(Node* currNode, int data){
    if(currNode == NULL)
        return NULL;

    if(currNode->key == data)
        return currNode;

    if(currNode->key > data)
        return searchKeyHelper(currNode->left, data);

    return searchKeyHelper (currNode->right, data);
}

// This function will return whether a key is in the tree
bool BST::searchKey(int key){
    Node* tree = searchKeyHelper(root, key);
    if(tree != NULL) {
        return true;
    }
    cout<<"Key not present in the tree"<<endl;
    return false;
}
```

Insert a key in BST

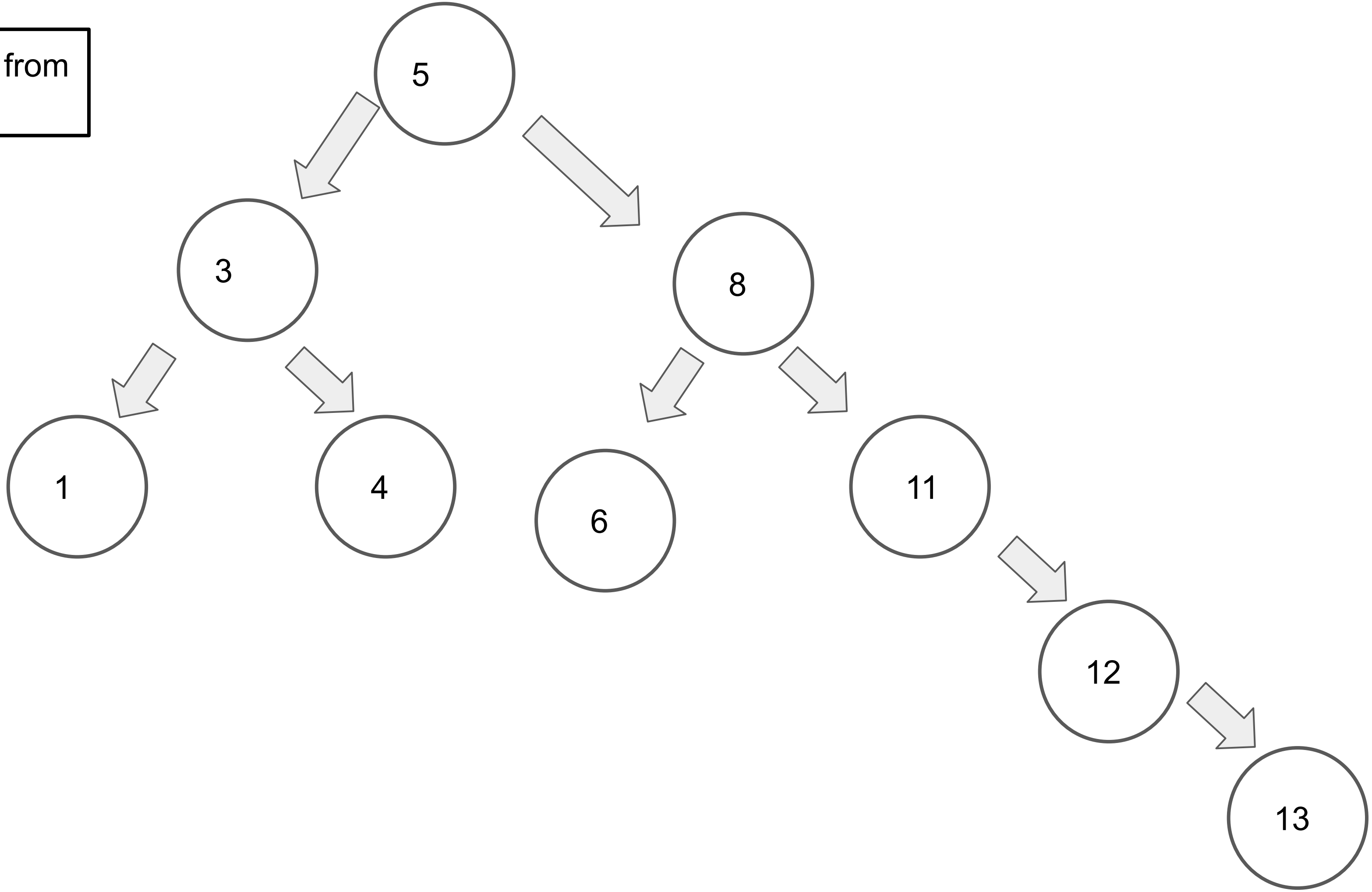
```
Node* BST:: addNodeHelper(Node* currNode, int data)
{
    if(currNode == NULL){
        return createNode(data);
    }
    else if(currNode->key < data){
        currNode->right = addNodeHelper(currNode->right,data);
    }
    else if(currNode->key > data){
        currNode->left = addNodeHelper(currNode->left,data);
    }
    return currNode;

}

void BST:: addNode(int data)
{
    root = addNodeHelper(root, data);
    cout<<data<<" has been added"<<endl;
}
```

Insert a key in BST

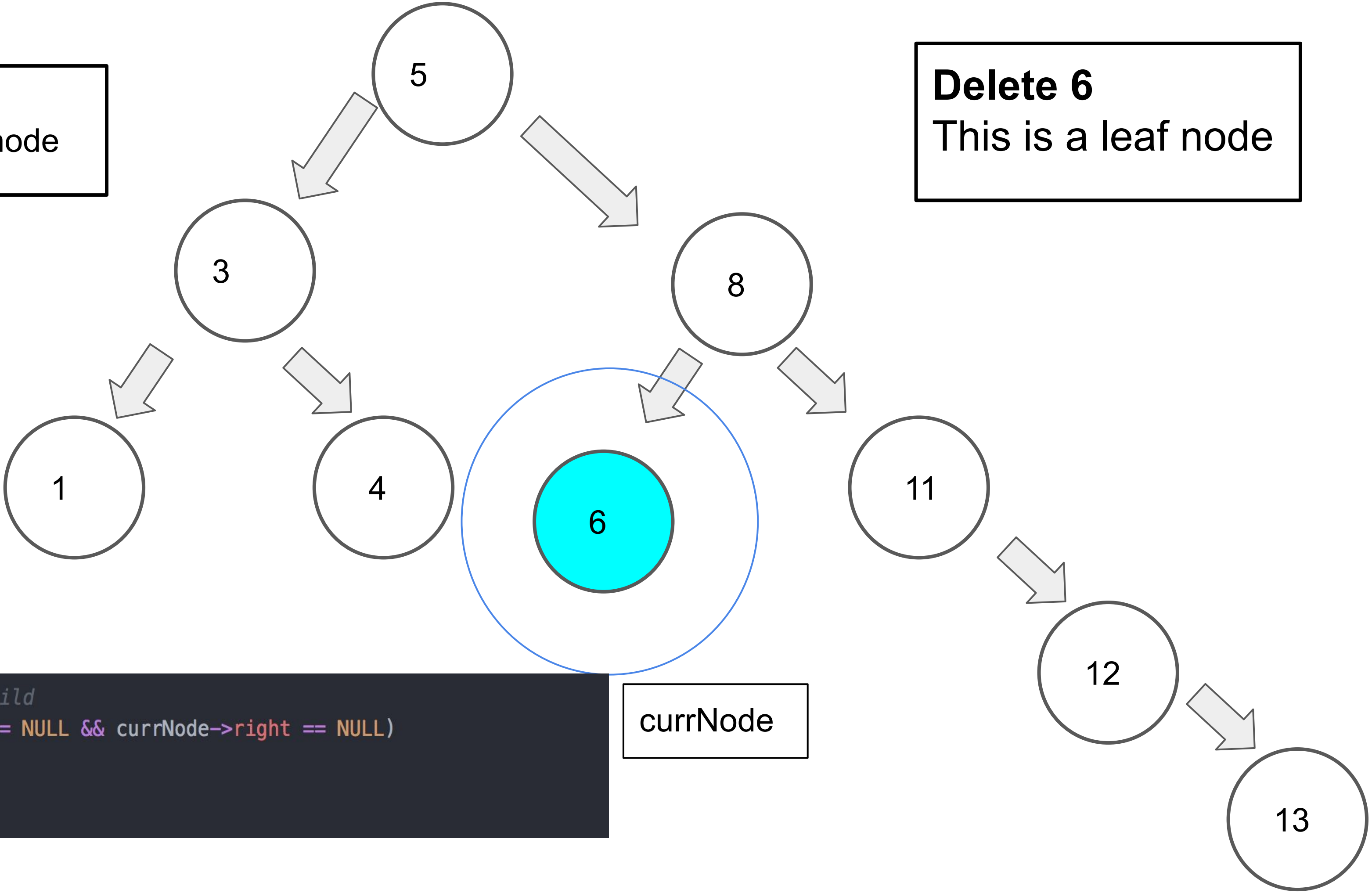
```
Node* BST:: addNodeHelper(Node* currNode, int data)
{
    if(currNode == NULL){
        return createNode(data);
    }
    else if(currNode->key < data){
        currNode->right = addNodeHelper(currNode->right,data);
    }
    else if(currNode->key > data){
        currNode->left = addNodeHelper(currNode->left,data);
    }
    return currNode;

}


void BST:: addNode(int data)
{
    root = addNodeHelper(root, data);
    cout<<data<<" has been added"<<endl;
}
```

```
//                    Get Max and Min Value Node

Node* BST::getMaxValueNode(Node* currNode){
    if(currNode->right == NULL){
        return currNode;
    }
    return getMaxValueNode(currNode->right);
}

Node* BST::getMinValueNode(Node* currNode){

    if(currNode->left == NULL){
      return currNode;
    }
    return getMinValueNode(currNode->left);
}
```

Delete a key from BST



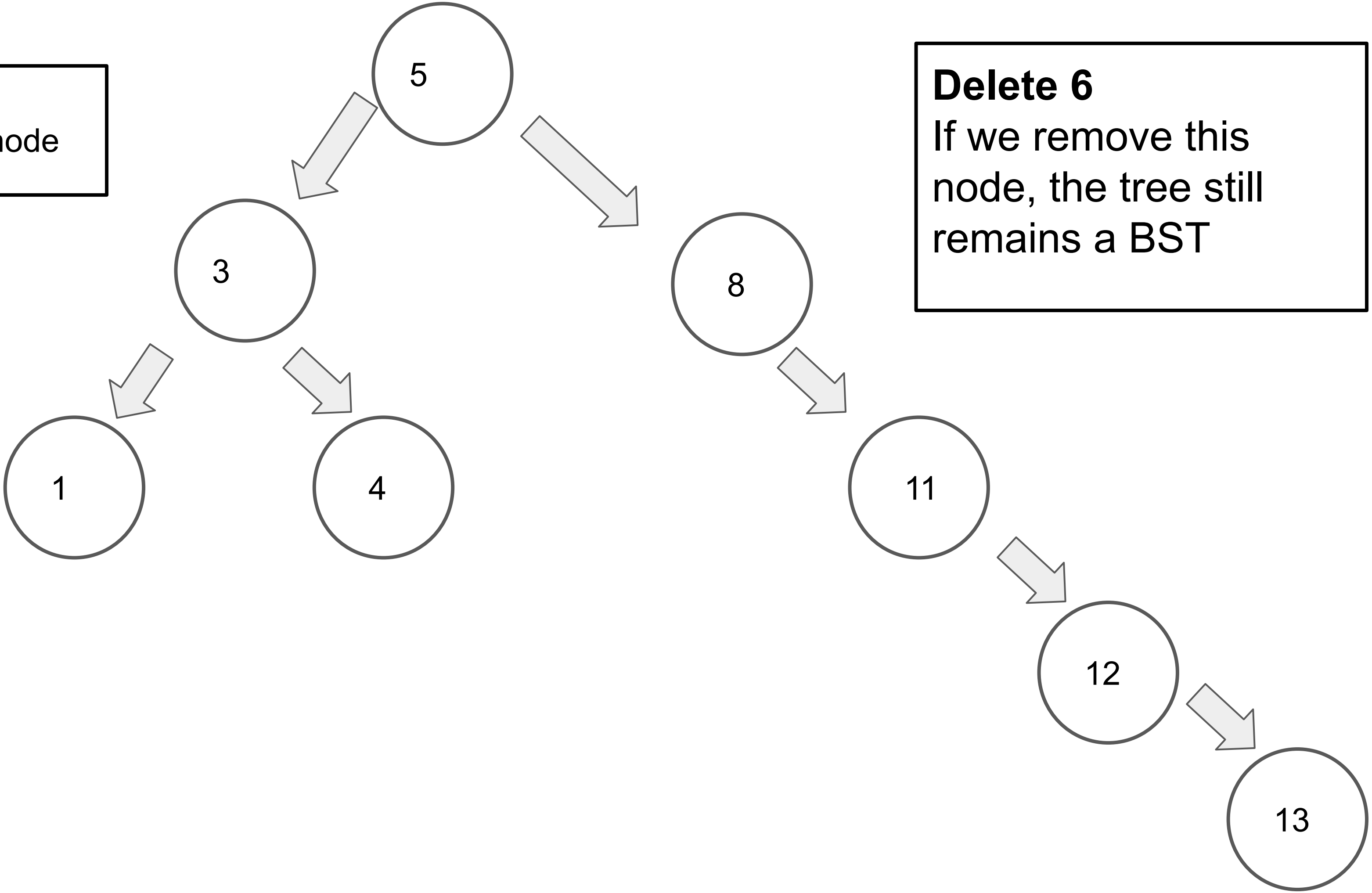
Case 1:
Deleting leaf node



Delete 6
This is a leaf node

```
//TODO Case : No child
if(currNode->left == NULL && currNode->right == NULL)
{

}
```

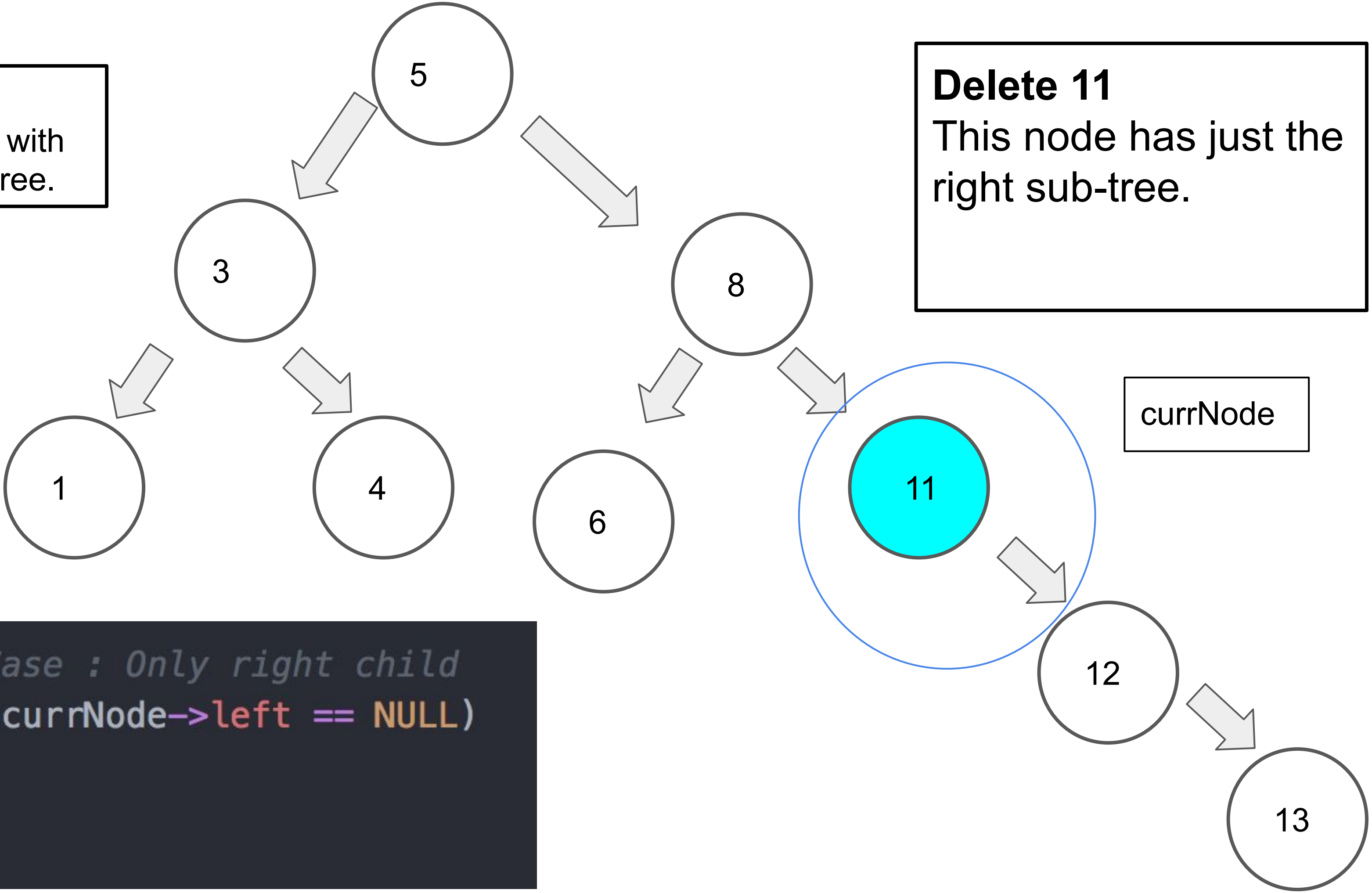
Case 1:
Deleting leaf node



Delete 6
If we remove this node, the tree still remains a BST

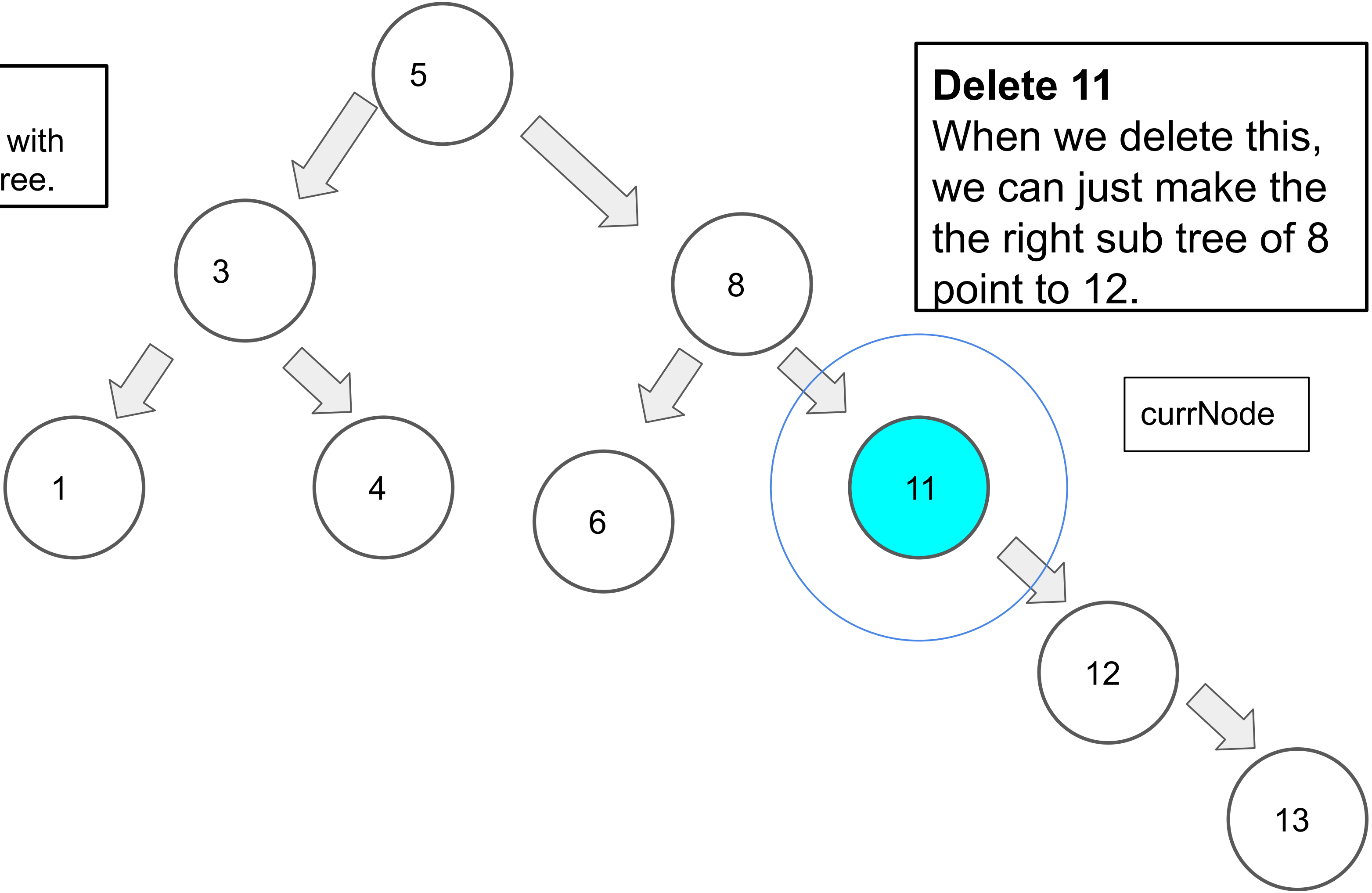
Case 2:
Deleting node with just right sub tree.

Delete 11
This node has just the right sub-tree.




```
//TODO Case : Only right child
else if(currNode->left == NULL)
{

}
```

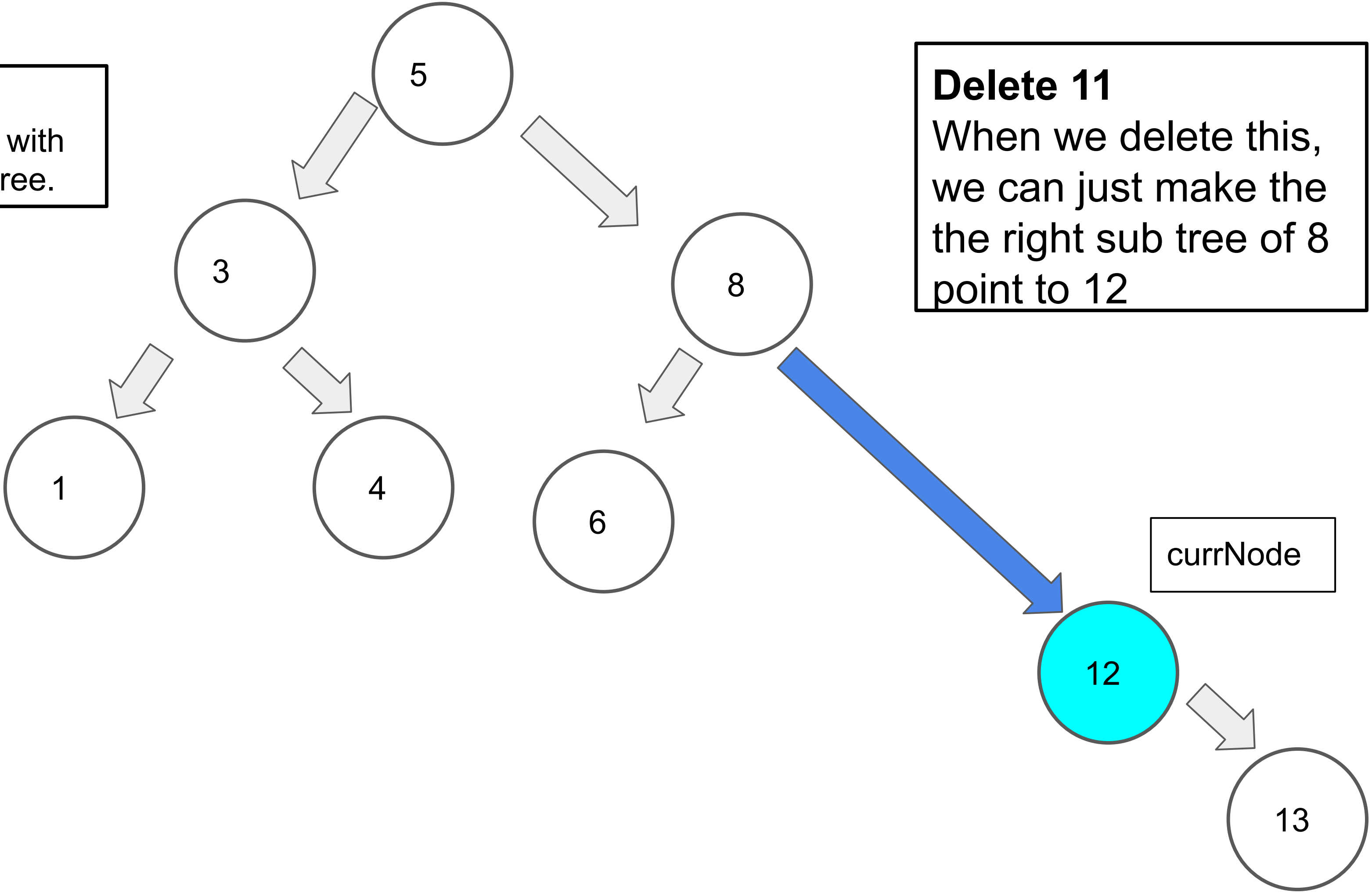
Case 2:
Deleting node with just right sub tree.



Delete 11
When we delete this, we can just make the the right sub tree of 8 point to 12.

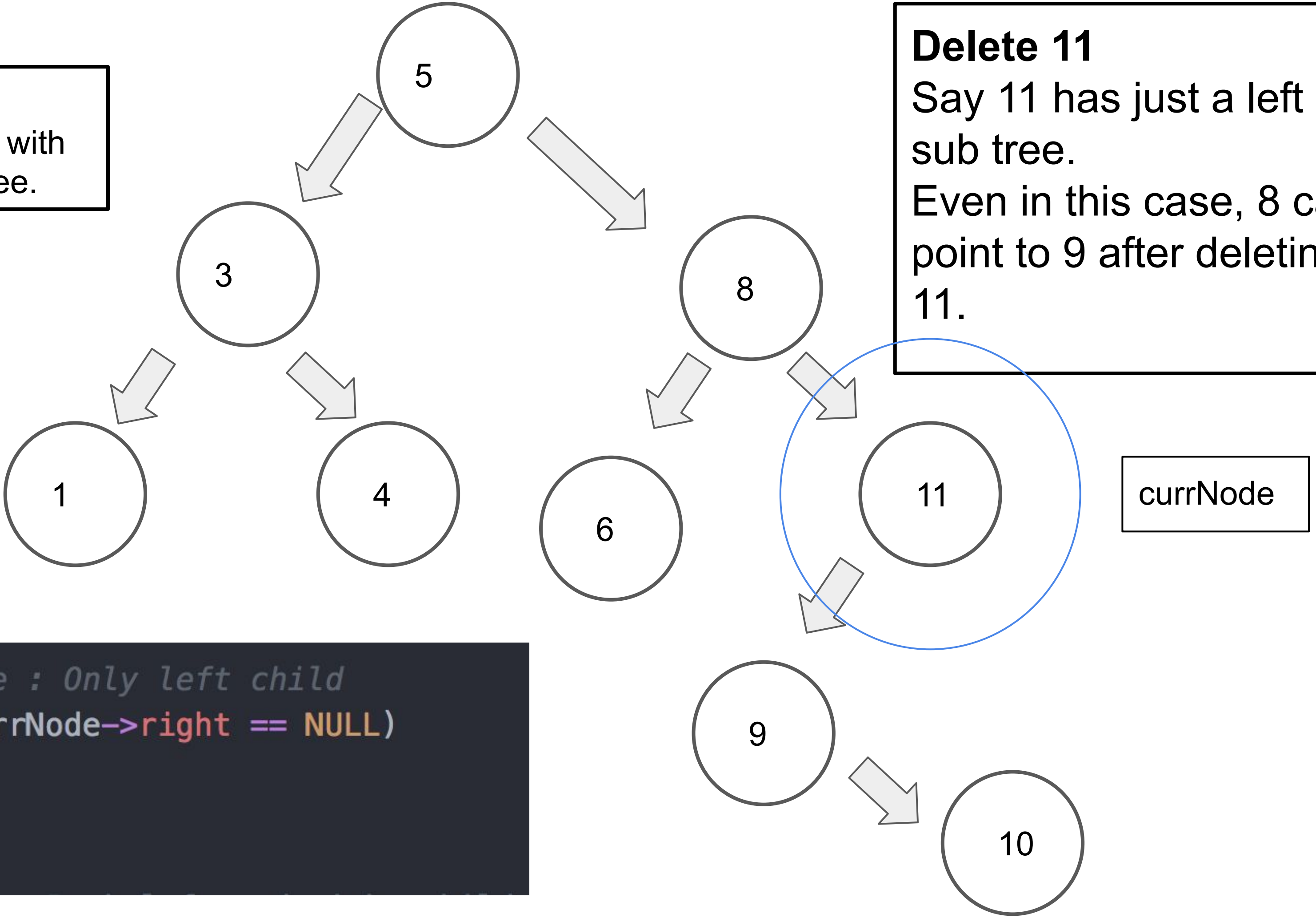
Case 2:
Deleting node with just right sub tree.

Delete 11
When we delete this, we can just make the the right sub tree of 8 point to 12



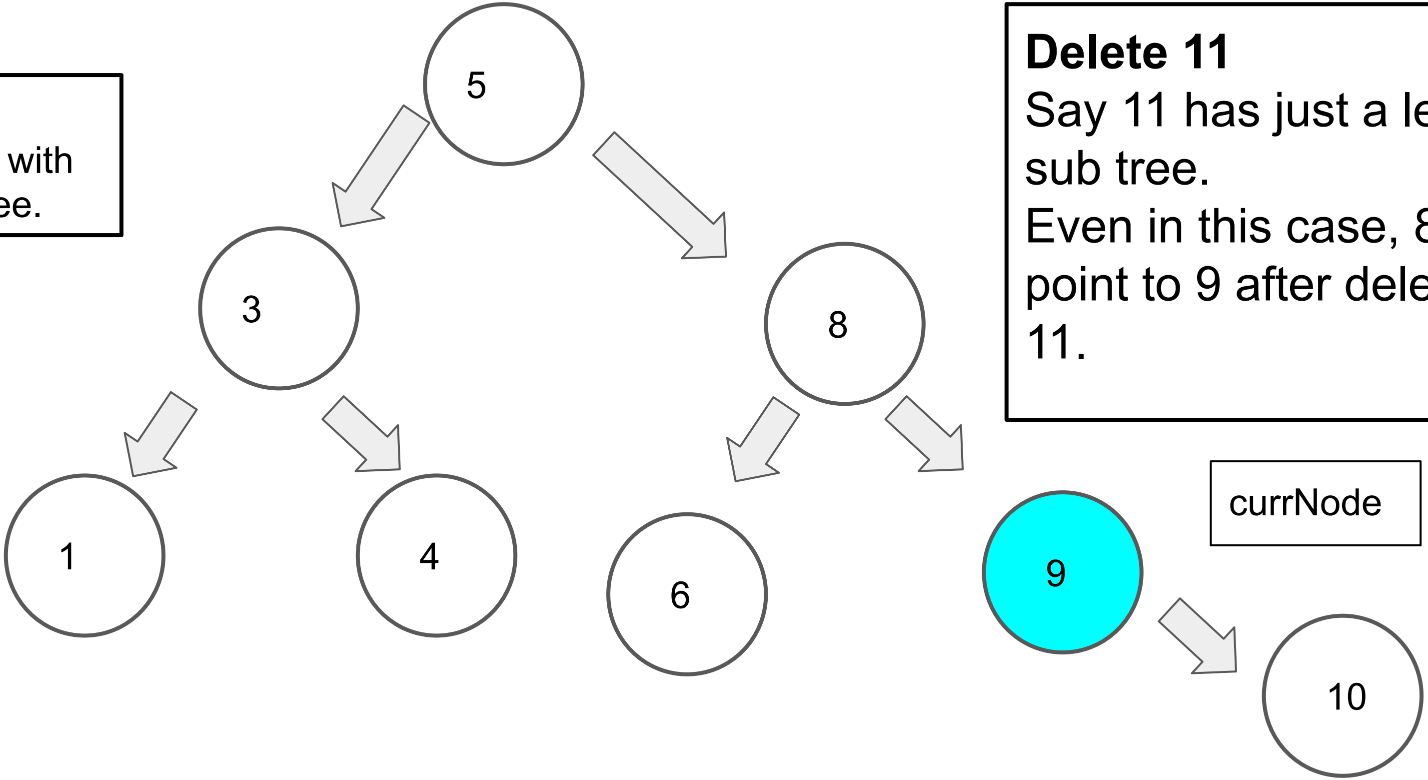
Case 3:
Deleting node with just left sub-tree.

Delete 11
Say 11 has just a left sub tree.
Even in this case, 8 can point to 9 after deleting 11.



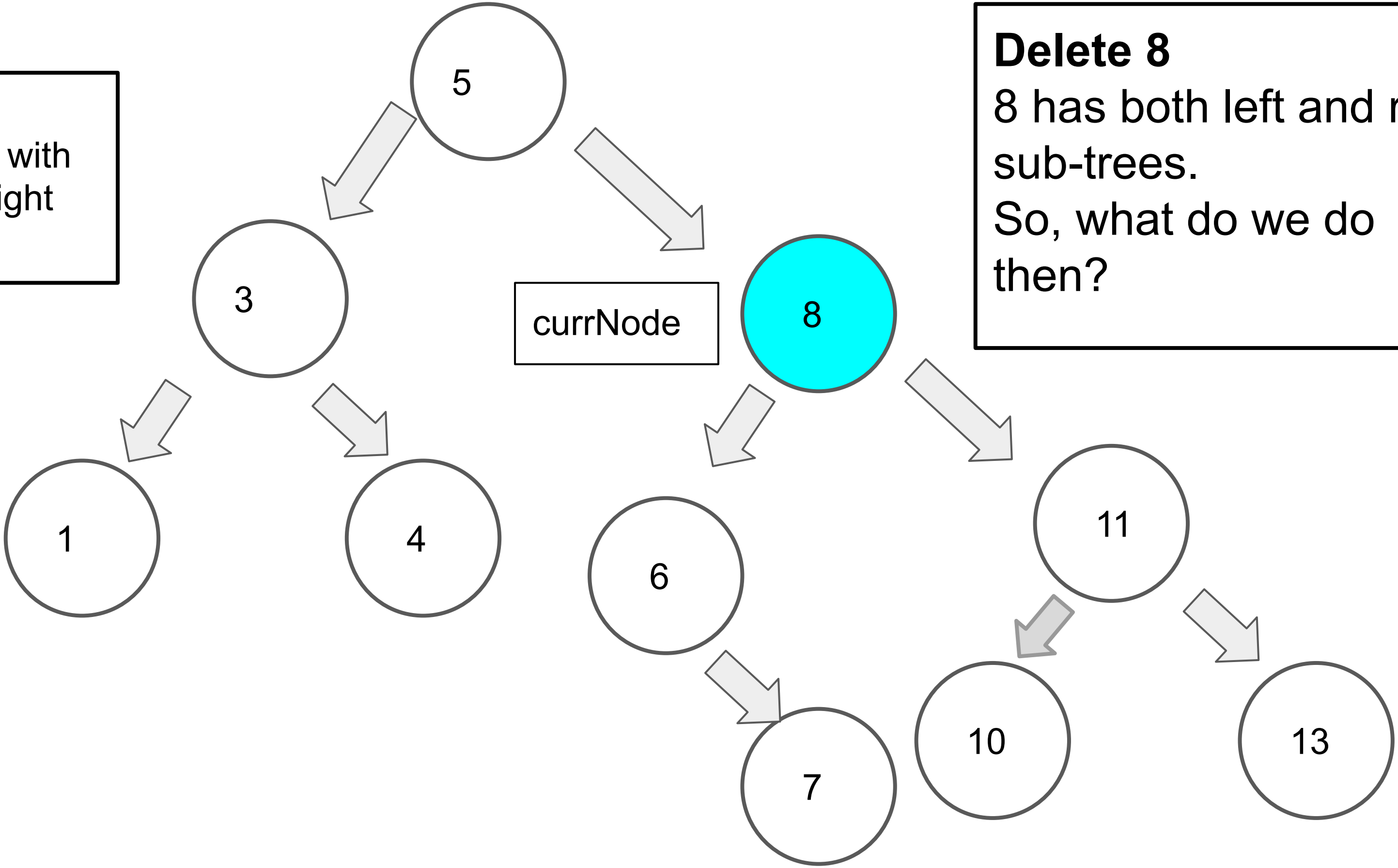

```
//TODO Case : Only left child
else if(currNode->right == NULL)
{

}
```

Case 3:
Deleting node with just left sub-tree.



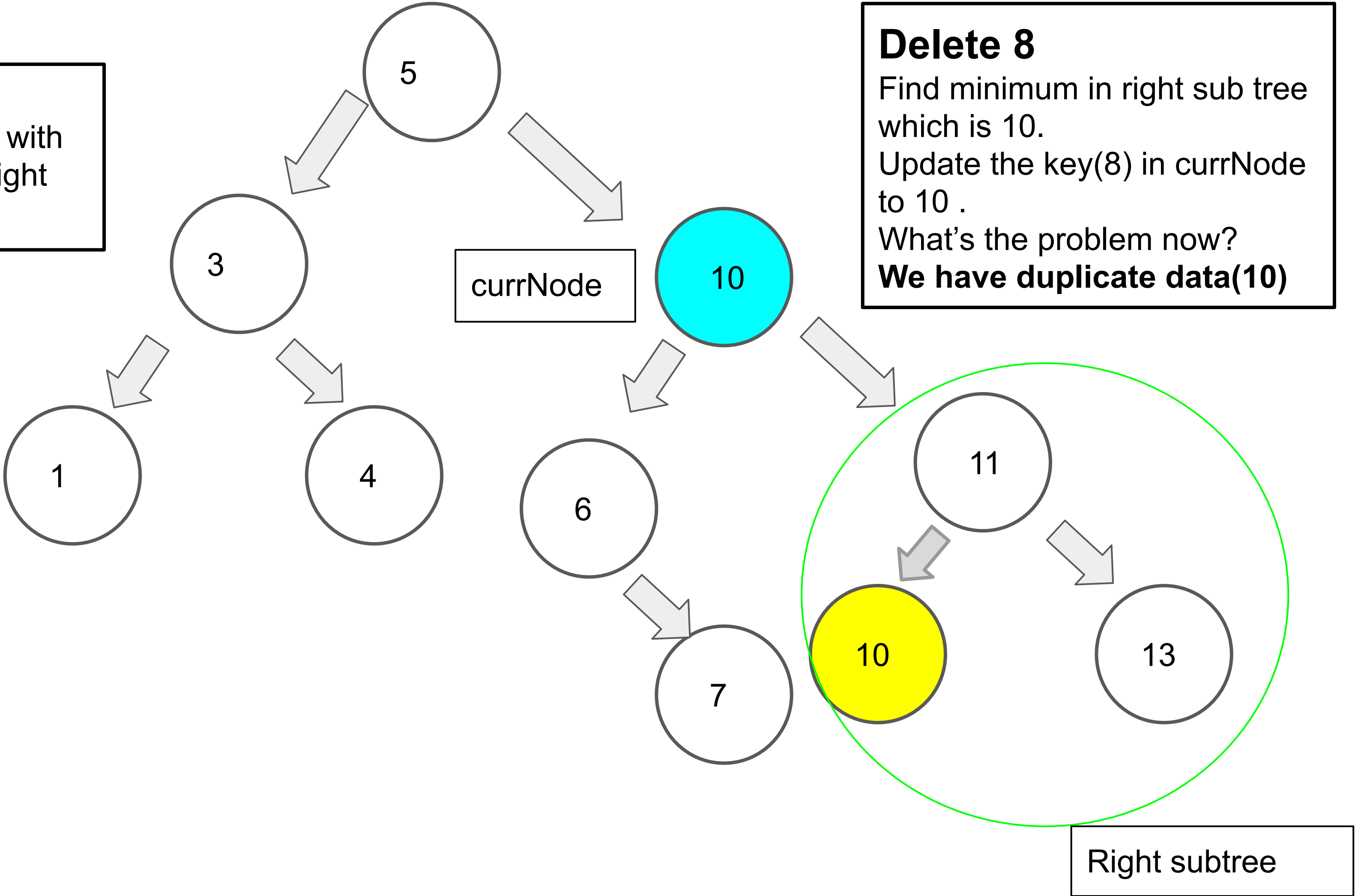
Delete 11
Say 11 has just a left sub tree.
Even in this case, 8 can point to 9 after deleting 11.

Case 4:
Deleting node with both left and right sub-trees.



Delete 8
8 has both left and right sub-trees.
So, what do we do then?

Case 4:
Deleting node with both left and right sub-trees.

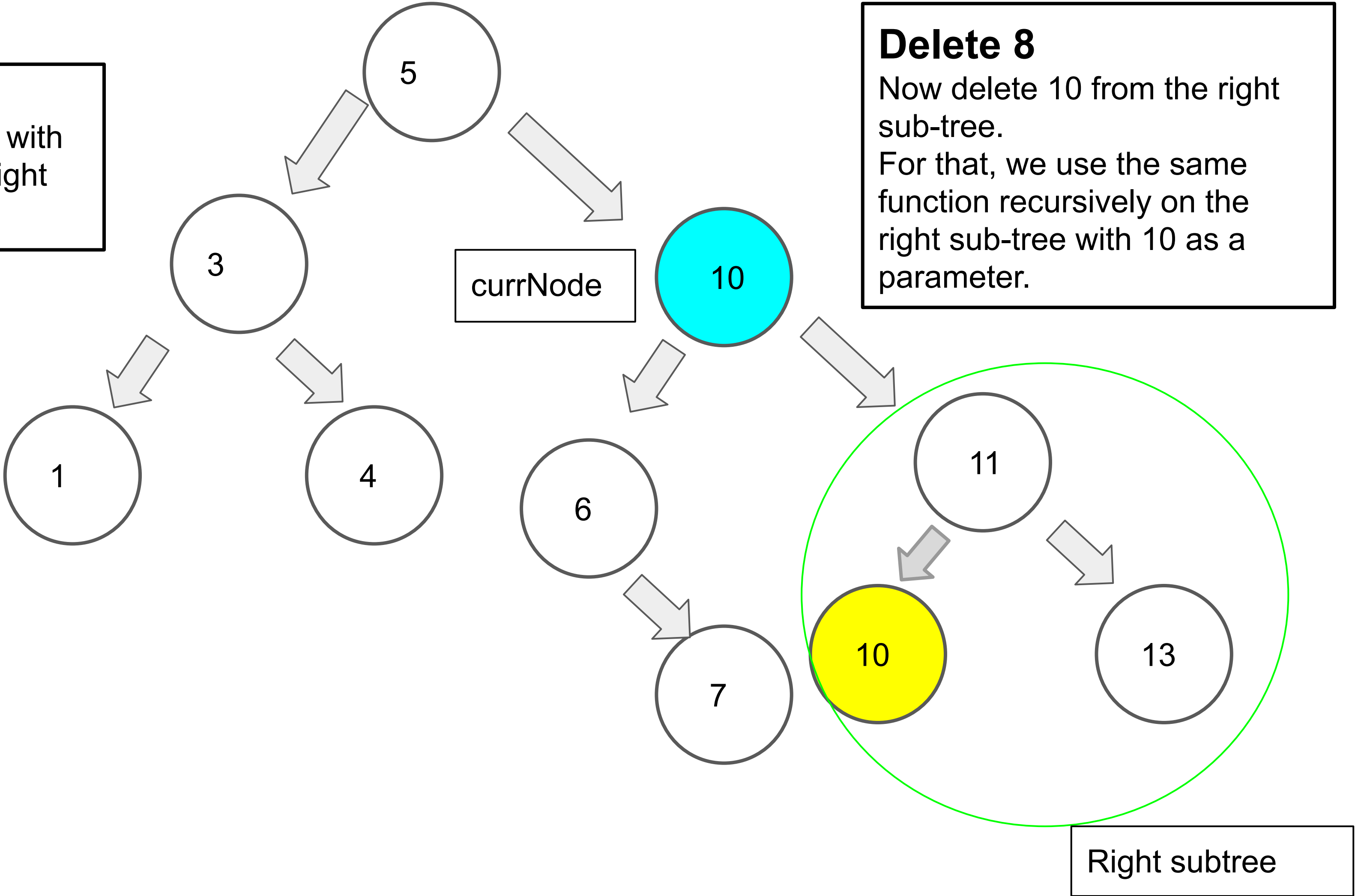


Delete 8
Find minimum in right sub tree which is 10.
Update the key(8) in currNode to 10 .
What’s the problem now?
We have duplicate data(10)

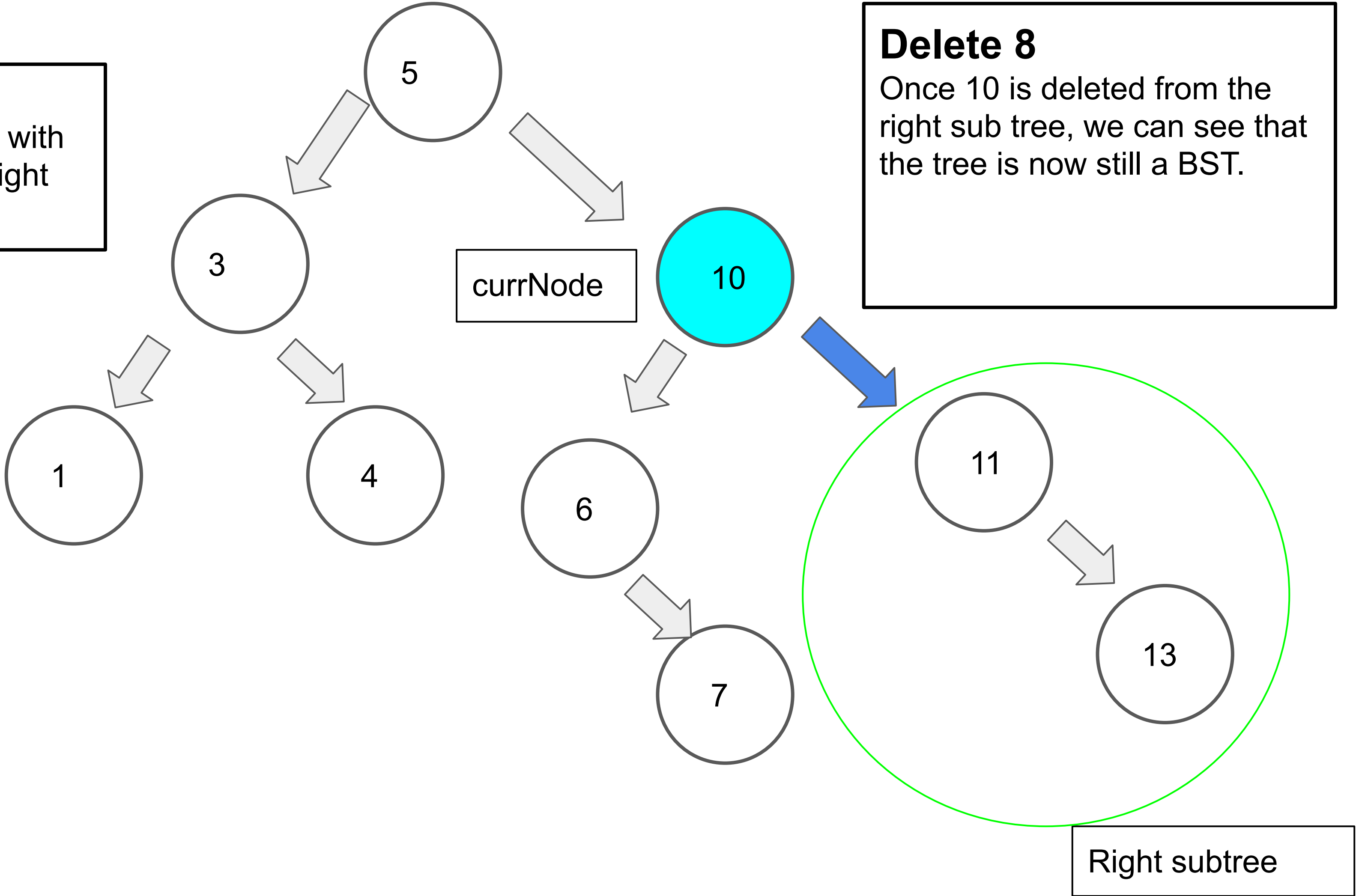
Case 4:
Deleting node with both left and right sub-trees.

Delete 8

Now delete 10 from the right sub-tree.
For that, we use the same function recursively on the right sub-tree with 10 as a parameter.



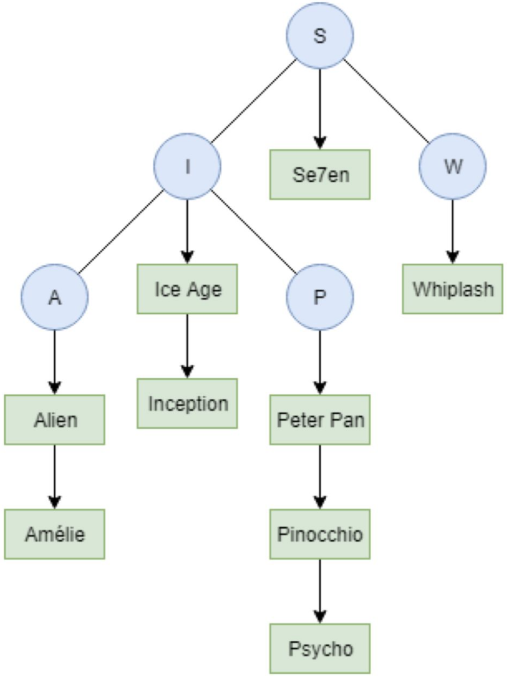
Case 4:
Deleting node with both left and right sub-trees.



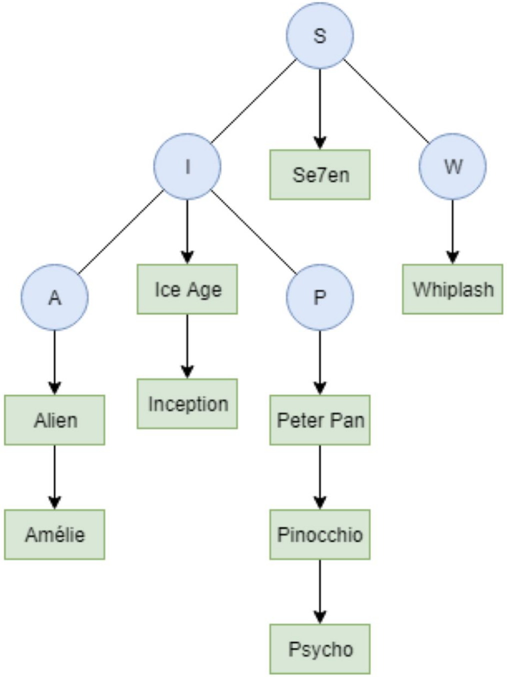
Delete 8

Once 10 is deleted from the right sub tree, we can see that the tree is now still a BST.

Assignment


S
I
Se7en
W
A
Ice Age
P
Whiplash
Alien
Inception
Peter Pan
Amélie
Pinocchio
Psycho

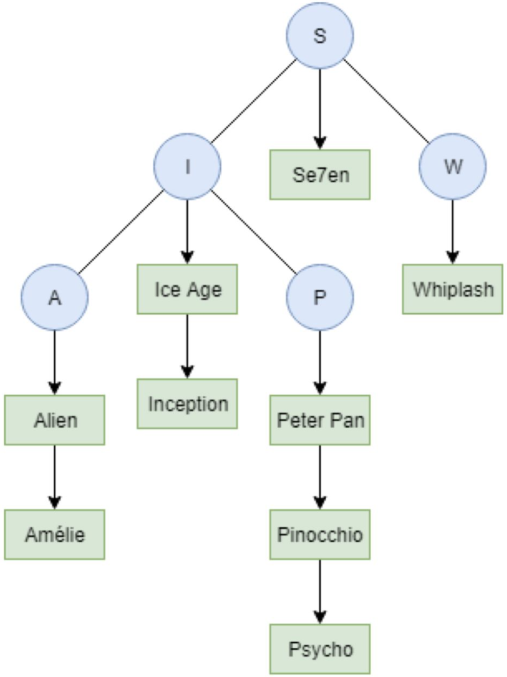
Assignment



addMovie

- Use a **recursive helper** function
- **Base case:**
  When tree node is null, then create a tree node and a linked list node and insert it
- **Recursively solve the problem** by navigating into the left or the right sub-trees.
- If the first character matches one of the nodes, then insert in the **sorted linked list**

Assignment



printMovieInventory

- Use a **recursive helper** function
- Recurse in an **inorder** traversal (Since it is alphabetically sorted).
- For each tree node iterate over all elements and print them