Binary Search Tree(BST)

- A binary search tree, also known as **ordered** or **sorted binary tree**,
- the nodes are arranged in an order.
- The nodes of the tree store a key and each has two distinguished sub-trees
- **binary search property:**
  - the key in each node is **greater than any key** stored in the **left sub-tree**,
  - and **less than or equal to any key stored in the right sub-tree**

Binary Search Tree(BST) Operations

- Binary search trees support three main operations:
  - Lookup (checking whether a key is present)
  - Insertion
  - deletion.

BST Insertion Operation

- New nodes are inserted as leaf nodes in the BST.

BST Insertion Operation

- New nodes are inserted as leaf nodes in the BST.

```
case(key < root->key):
     recurse down left subtree

case(key >= root->key):
     recurse down right subtree

case(root == NULL):
     create new node
```

BST Insertion Operation

```
struct node *insert(struct node *root, int key){
  if (root==NULL)
    root = createNode(key);
  else if (key < root->key)
    root->lch = insert(root->lch, key);
  else  // key >= root->key
    root->rch = insert(root->rch, key);
  return root;
}
```

BST Insertion Operation

```
root = NULL
root = Insert(root, 8)
```

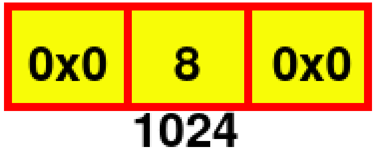
BST Insertion Operation

```
struct node *insert(struct node *root, int key){
  if (root==NULL)
    root = createNode(key);
  else if (key < root->key)
    root->lch = insert(root->lch, key);
  else  // key >= root->key
    root->rch = insert(root->rch, key);
  return root;
}
```

if(r = NULL) r = createNode(8)
return r

root = NULL
root = Insert(root, 8)

BST Insertion Operation

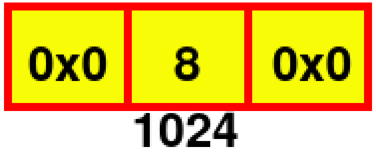



```
struct node *insert(struct node *root, int key){
  if (root==NULL)
    root = createNode(key);
  else if (key < root->key)
    root->lch = insert(root->lch, key);
  else  // key >= root->key
    root->rch = insert(root->rch, key);
  return root;
}
```

if(r = NULL) r = createNode(8)
return r

root = NULL
root = Insert(root, 8)

BST Insertion Operation




```c
struct node *insert(struct node *root, int key){
  if (root==NULL)
    root = createNode(key);
  else if (key < root->key)
    root->lch = insert(root->lch, key);
  else  // key >= root->key
    root->rch = insert(root->rch, key);
  return root;
}
```

If(key < r->key)
r ->lch = Insert(r->lch, 3)
return r

root = NULL
root = Insert(root, 8)
root = Insert(root, 3)

BST Insertion Operation

0x0	8	0x0

1024

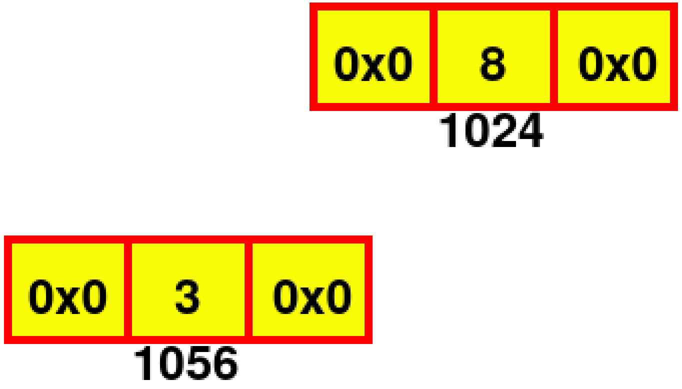
```
struct node *insert(struct node *root, int key){
  if (root==NULL)
    root = createNode(key);
  else if (key < root->key)
    root->lch = insert(root->lch, key);
  else  // key >= root->key
    root->rch = insert(root->rch, key);
  return root;
}
```

```
if(r = NULL) r = createNode(3)
return r
```

```
If( key < r->key)
r ->lch = Insert(r->lch, 3)
return r
```

```
root = NULL
root = Insert(root, 8)
root = Insert(root, 3)
```

BST Insertion Operation

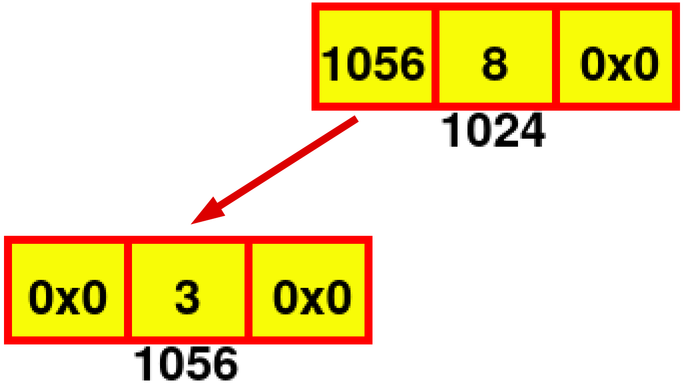



```
if(r = NULL) r = createNode(3)
return r
```

```
If( key < r->key)
r ->lch = Insert(r->lch, 3)
return r
```

```
root = NULL
root = Insert(root, 8)
root = Insert(root, 3)
```

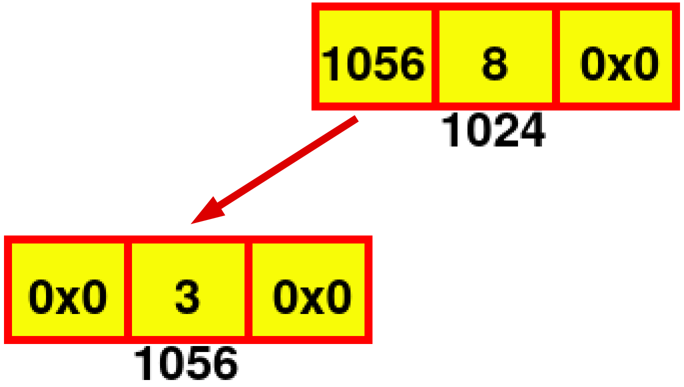
BST Insertion Operation




```
If( key < r->key)
r ->lch = Insert(r->lch, 3)
return r
```

```
root = NULL
root = Insert(root, 8)
root = Insert(root, 3)
```

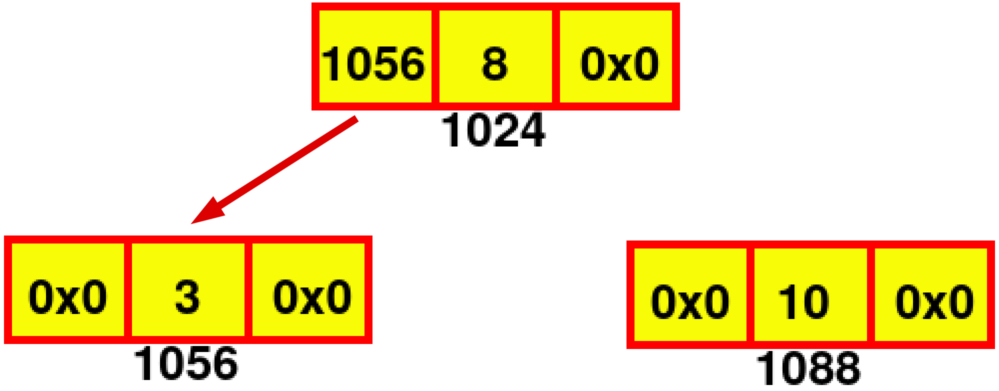
BST Insertion Operation




```
If( key > r->key)
r ->rch = Insert(r->rch, 10)
return r
```

```
root = NULL
root = Insert(root, 8)
root = Insert(root, 3)
root = Insert(root, 10)
```

BST Insertion Operation

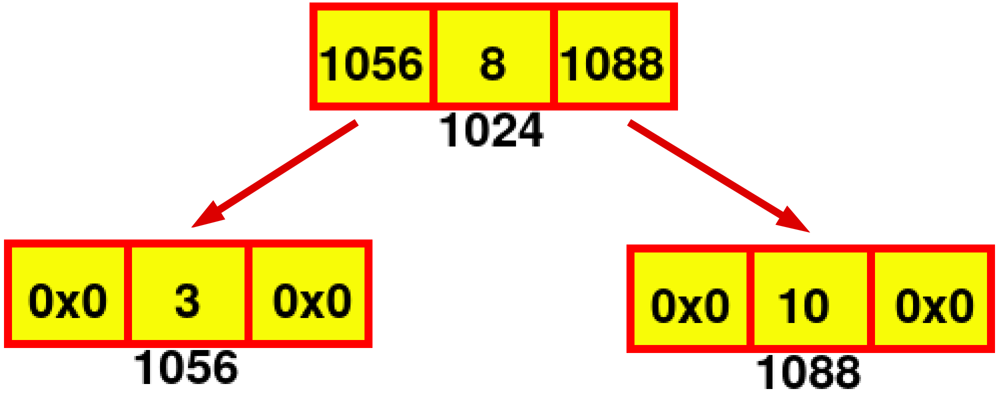



```
if(r = NULL) r = createNode(10)
return r
```

```
If( key > r->key)
r ->rch = Insert(r->rch, 10)
return r
```

```
root = NULL
root = Insert(root, 8)
root = Insert(root, 3)
root = Insert(root, 10)
```

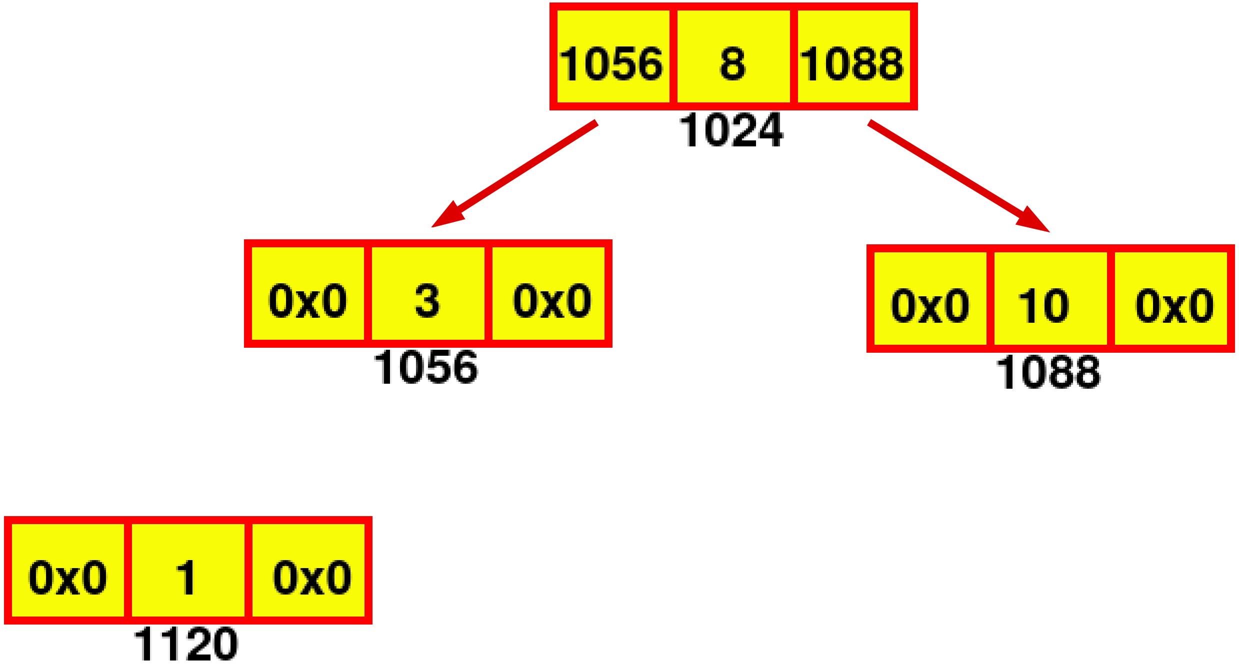
BST Insertion Operation




```
If( key > r->key)
r ->rch = Insert(r->rch, 10)
return r
```

```
root = NULL
root = Insert(root, 8)
root = Insert(root, 3)
root = Insert(root, 10)
```

BST Insertion Operation



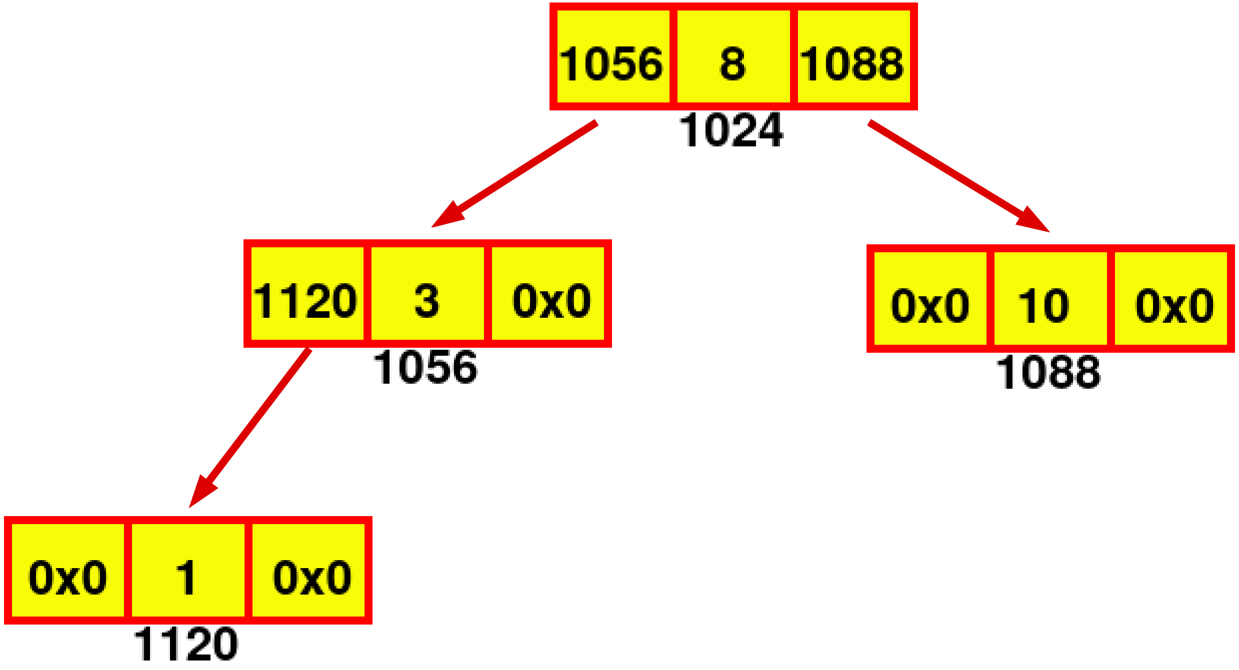

```
if(r = NULL) r = createNode(1)
return r
```

```
If( key < r->key)
r ->lch = Insert(r->lch, 1)
return r
```

```
If( key < r->key)
r ->lch = Insert(r->lch, 1)
return r
```

```
root = NULL
root = Insert(root, 8)
root = Insert(root, 3)
root = Insert(root, 10)
root = Insert(root, 1)
```

BST Insertion Operation

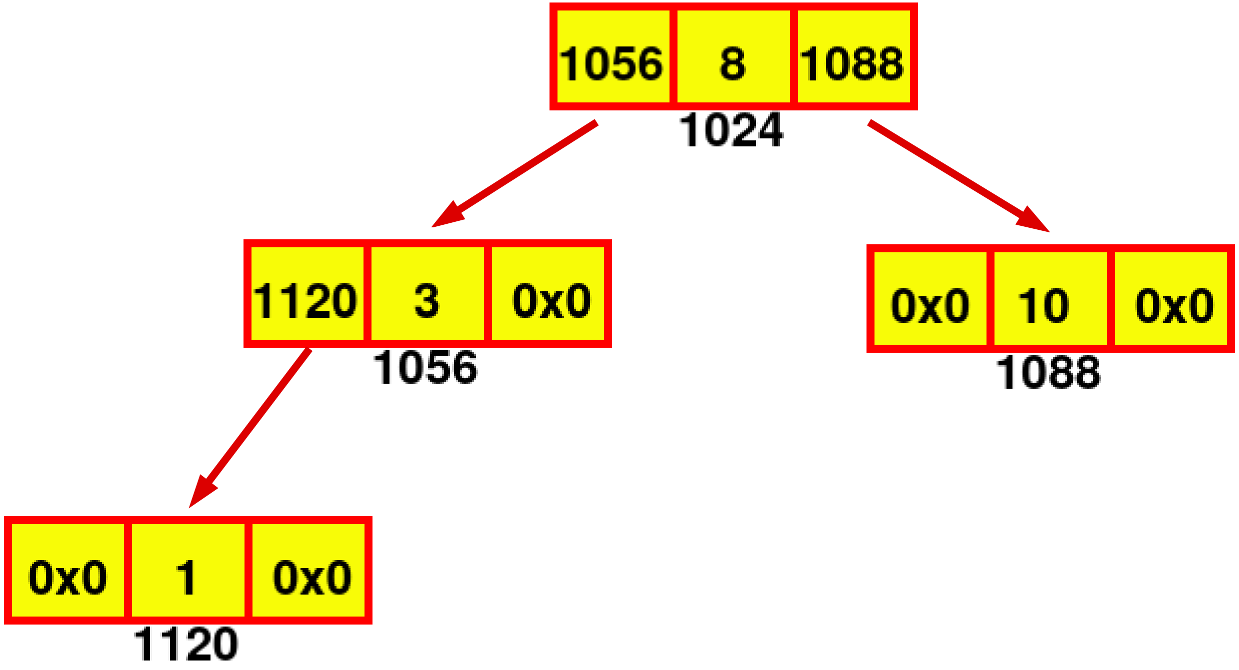



```
If( key < r->key)
r ->lch = Insert(r->lch, 1)
return r
```

```
If( key < r->key)
r ->lch = Insert(r->lch, 1)
return r
```

```
root = NULL
root = Insert(root, 8)
root = Insert(root, 3)
root = Insert(root, 10)
root = Insert(root, 1)
```

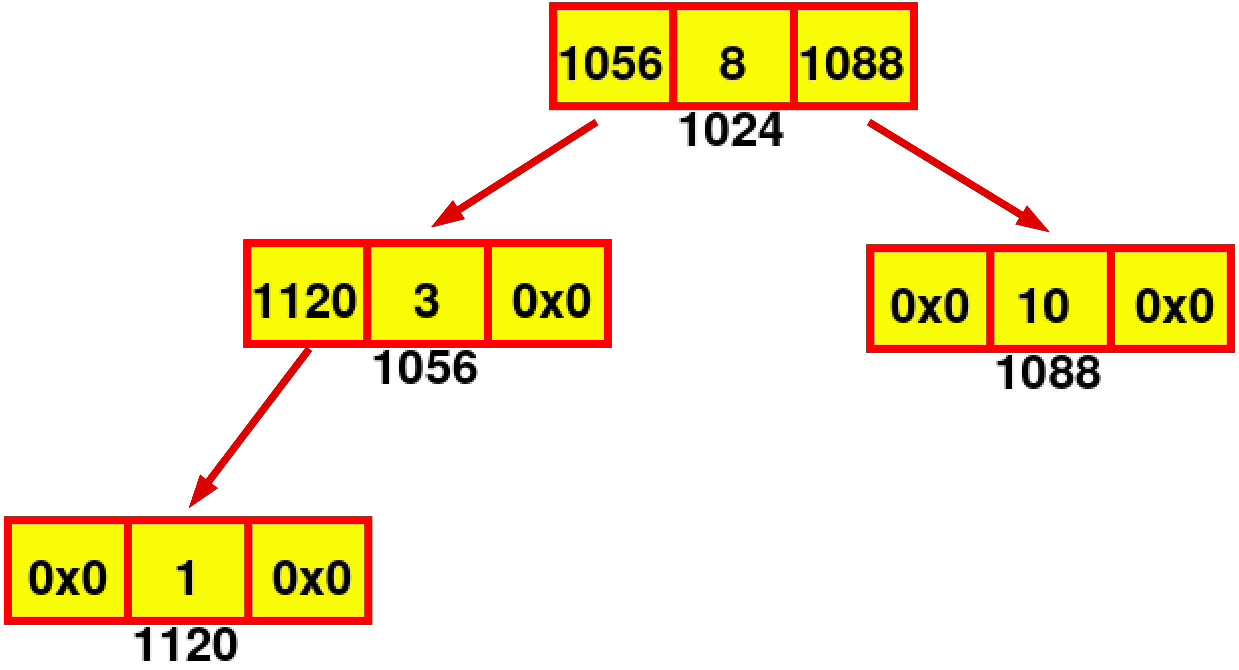
BST Insertion Operation




```
If( key < r->key)
 l ->rch = Insert(r->lch, 6)
return r
```

```
root = NULL
root = Insert(root, 8)
root = Insert(root, 3)
root = Insert(root, 10)
root = Insert(root, 1)
root = Insert(root, 6)
```

BST Insertion Operation

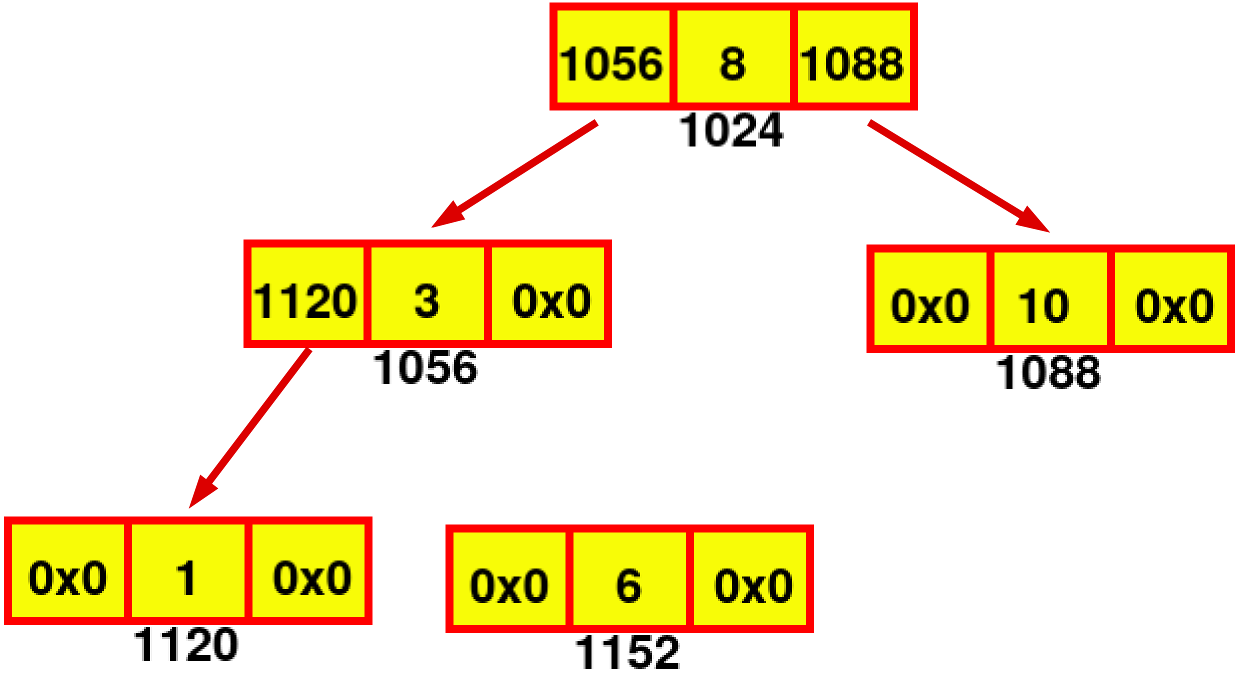



```
If( key > r->key)
r ->rch = Insert(r->rch, 6)
return r
```

```
If( key < r->key)
l ->rch = Insert(r->lch, 6)
return r
```

```
root = NULL
root = Insert(root, 8)
root = Insert(root, 3)
root = Insert(root, 10)
root = Insert(root, 1)
root = Insert(root, 6)
```

BST Insertion Operation



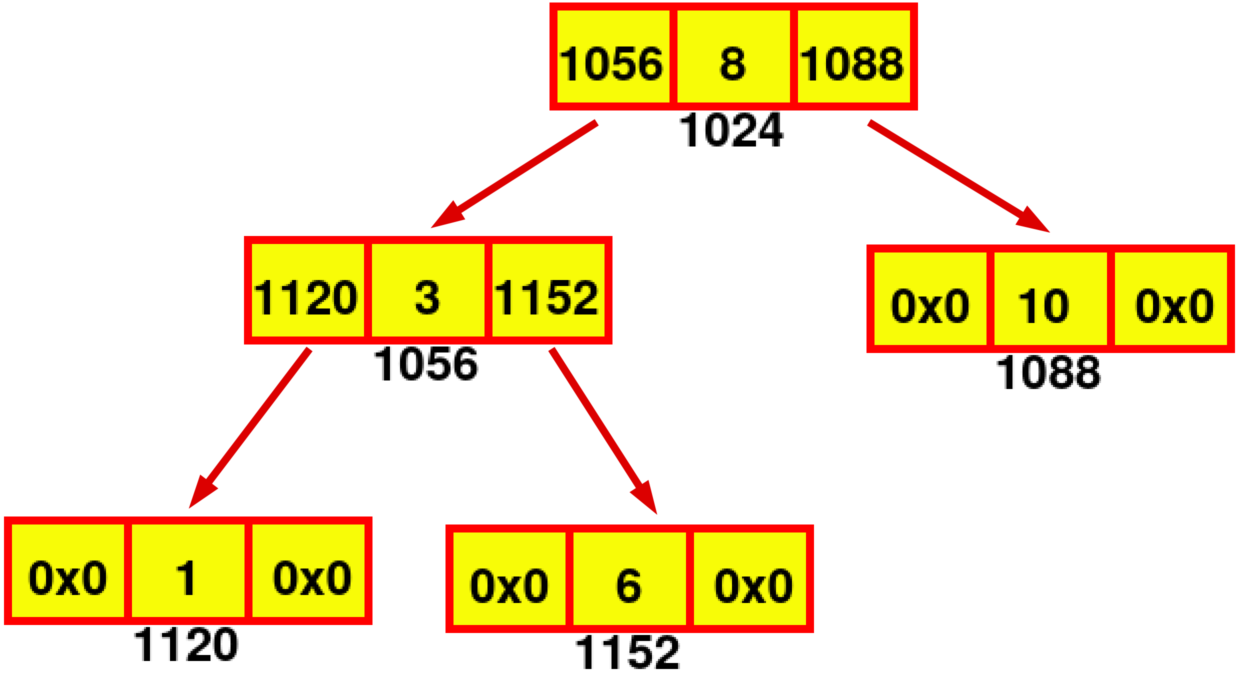

```
if(r = NULL) r = createNode(6)
return r
```

```
If( key > r->key)
r ->rch = Insert(r->rch, 6)
return r
```

```
If( key < r->key)
l ->rch = Insert(r->lch, 6)
return r
```

```
root = NULL
root = Insert(root, 8)
root = Insert(root, 3)
root = Insert(root, 10)
root = Insert(root, 1)
root = Insert(root, 6)
```

BST Insertion Operation

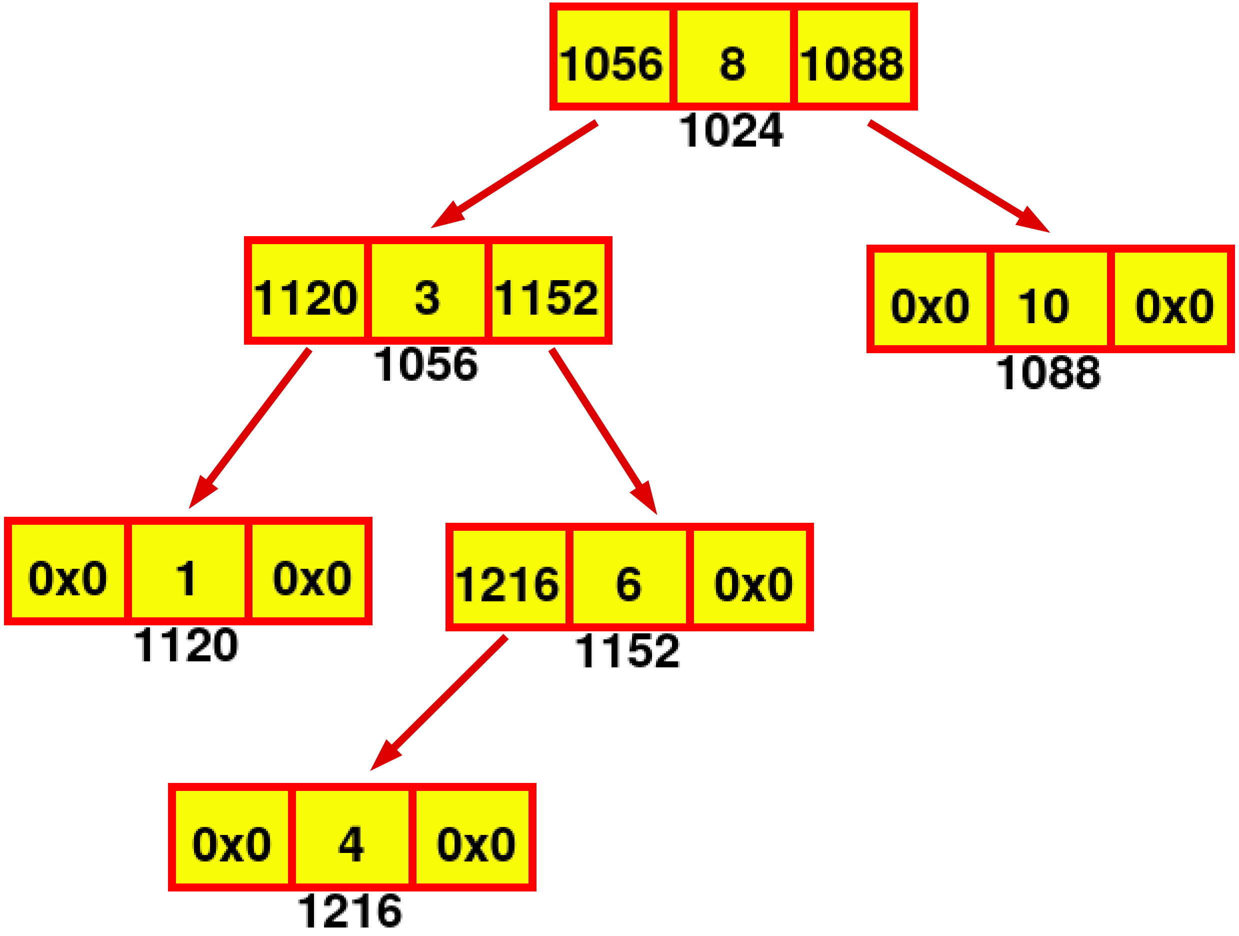



```
If( key > r->key)
r ->rch = Insert(r->rch, 6)
return r
```

```
If( key < r->key)
 l ->rch = Insert(r->lch, 6)
return r
```

```
root = NULL
root = Insert(root, 8)
root = Insert(root, 3)
root = Insert(root, 10)
root = Insert(root, 1)
root = Insert(root, 6)
```

BST Insertion Operation




```
if(r = NULL) r = createNode(4)
return r
```

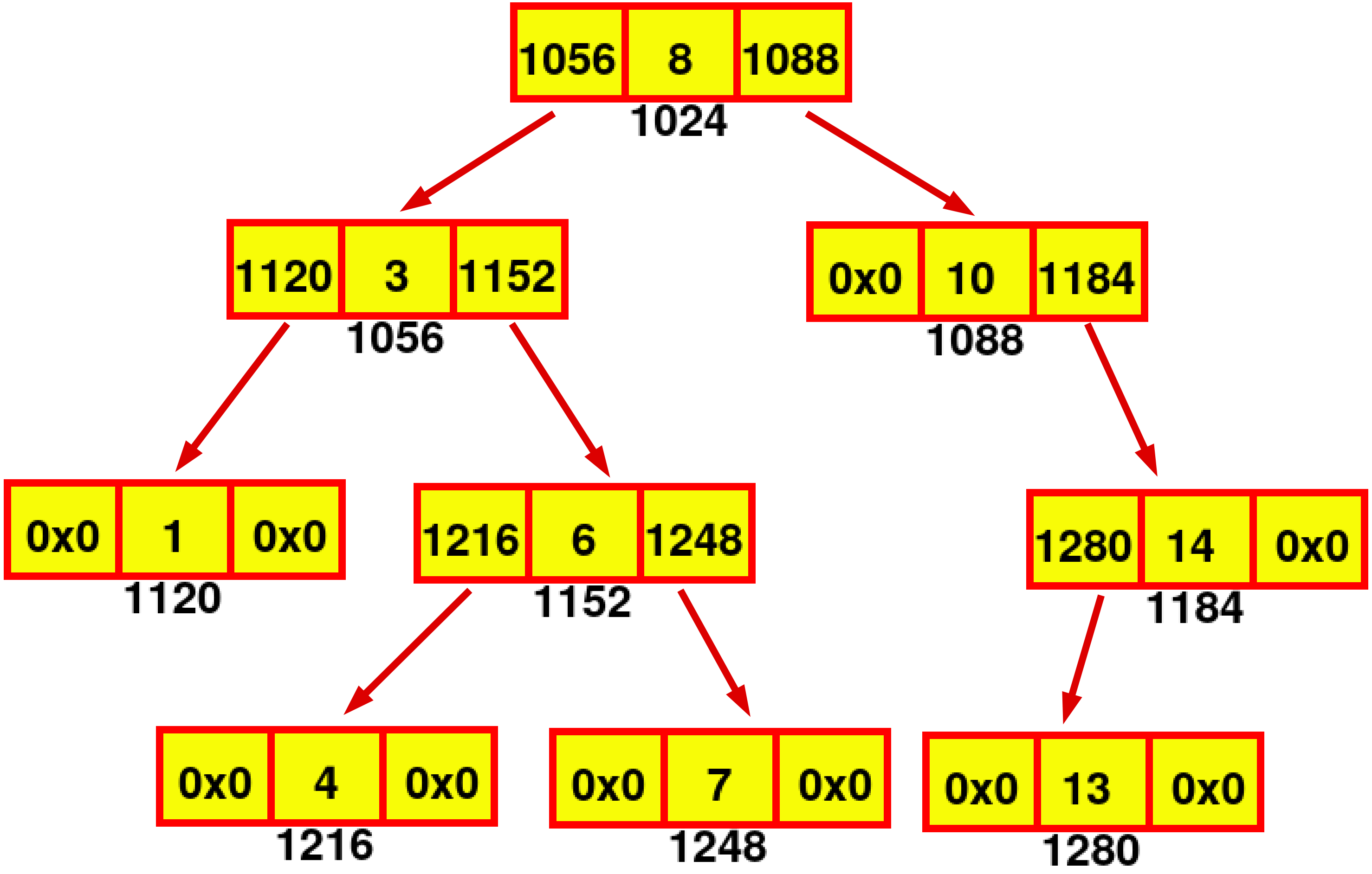
```
If( key < r->key)
r ->lch = Insert(r->lch, 4)
return r
```

```
If( key > r->key)
r ->rch = Insert(r->rch, 4)
return r
```

```
If( key < r->key)
r ->lch = Insert(r->lch, 4)
return r
```

```
root = NULL
root = Insert(root, 8)
root = Insert(root, 3)
root = Insert(root, 10)
root = Insert(root, 1)
root = Insert(root, 6)
root = Insert(root, 4)
```

BST Insertion Operation




```
root = NULL
root = Insert(root, 8)
root = Insert(root, 3)
root = Insert(root, 10)
root = Insert(root, 1)
root = Insert(root, 6)
root = Insert(root, 4)
root = Insert(root, 7)
root = Insert(root, 14)
root = Insert(root, 13)
```

Binary Search Tree(BST)

- Since in insertion we need to traverse to till some leaf node,
- So the running time complexity of BST insertion is O (h) here h is the height of the tree.
- However, the worst case for BST insertion is O (n) here n is the total number of nodes in the BST, because an unbalanced BST may degenerate to a linked list.
- If the BST is height-balanced the height is O (log n)
- So insertion will take O(log n) time

Binary Search Tree(BST)

- The time complexity of operations on the binary search tree is directly proportional to the height of the tree.
- the nodes in a BST are laid out in such a way that each comparison skips about half of the remaining tree, the lookup performance is proportional to that of binary logarithm.
- The performance of a binary search tree is dependent on the order of insertion of the nodes into the tree;
- The complexity analysis of BST shows that, on average, the insert, delete and search takes O(log n) for n nodes.
- In the worst case, they degrade to that of a singly linked list: O(n).

BST Search

```
struct node* search(struct node *root, int key){
    if(root == NULL || root->key == key)
            return root;
    if(key < root->key)
          return search(root->lch,key);
    if(key > root->key)
          return search(root->rch,key);
}
```

BST search time complexity

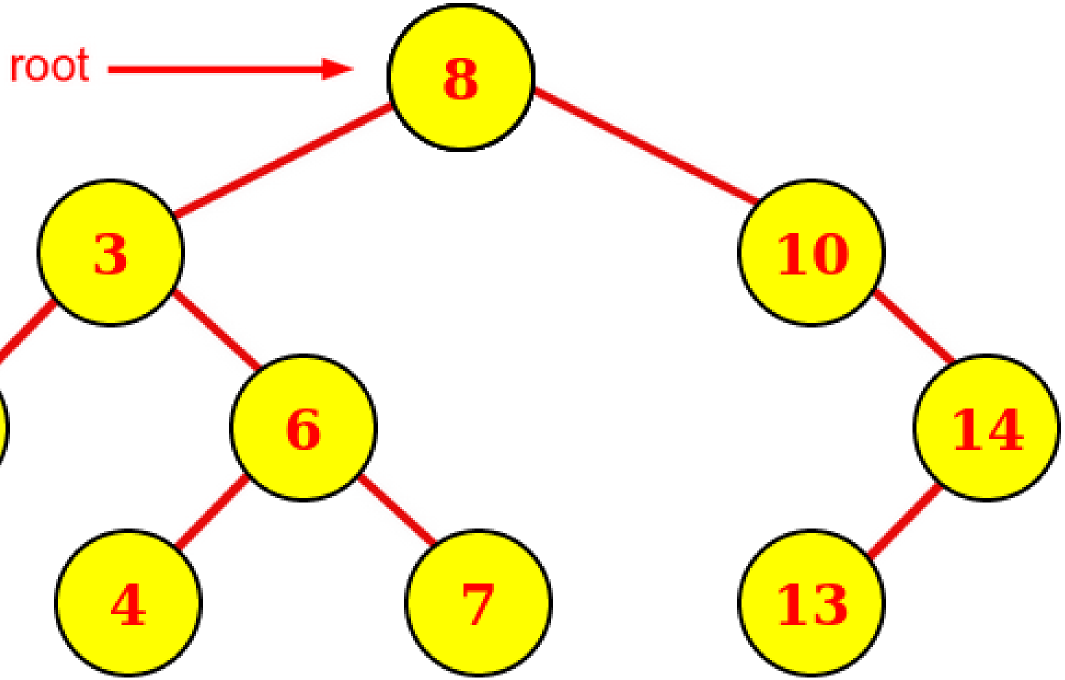
- In BST search we need to traverse to till some leaf node
- So the running time complexity of BST search is O (h) here h is the height of the tree.
- However, the worst case for BST search is O (n) here n is the total number of nodes in the BST, because an unbalanced BST may degenerate to a linked list.
- If the BST is height-balanced the height is O (log n)
- So search will take O(log n) time
- The time complexity of operations on the binary search tree is directly proportional to the height of the tree.

BST Deletion Operation

- First we need to find the node to be deleted
- Replace node to be deleted with its successor

BST Deletion Operation

- First we need to find the node to be deleted
- Replace node to be deleted with its successor



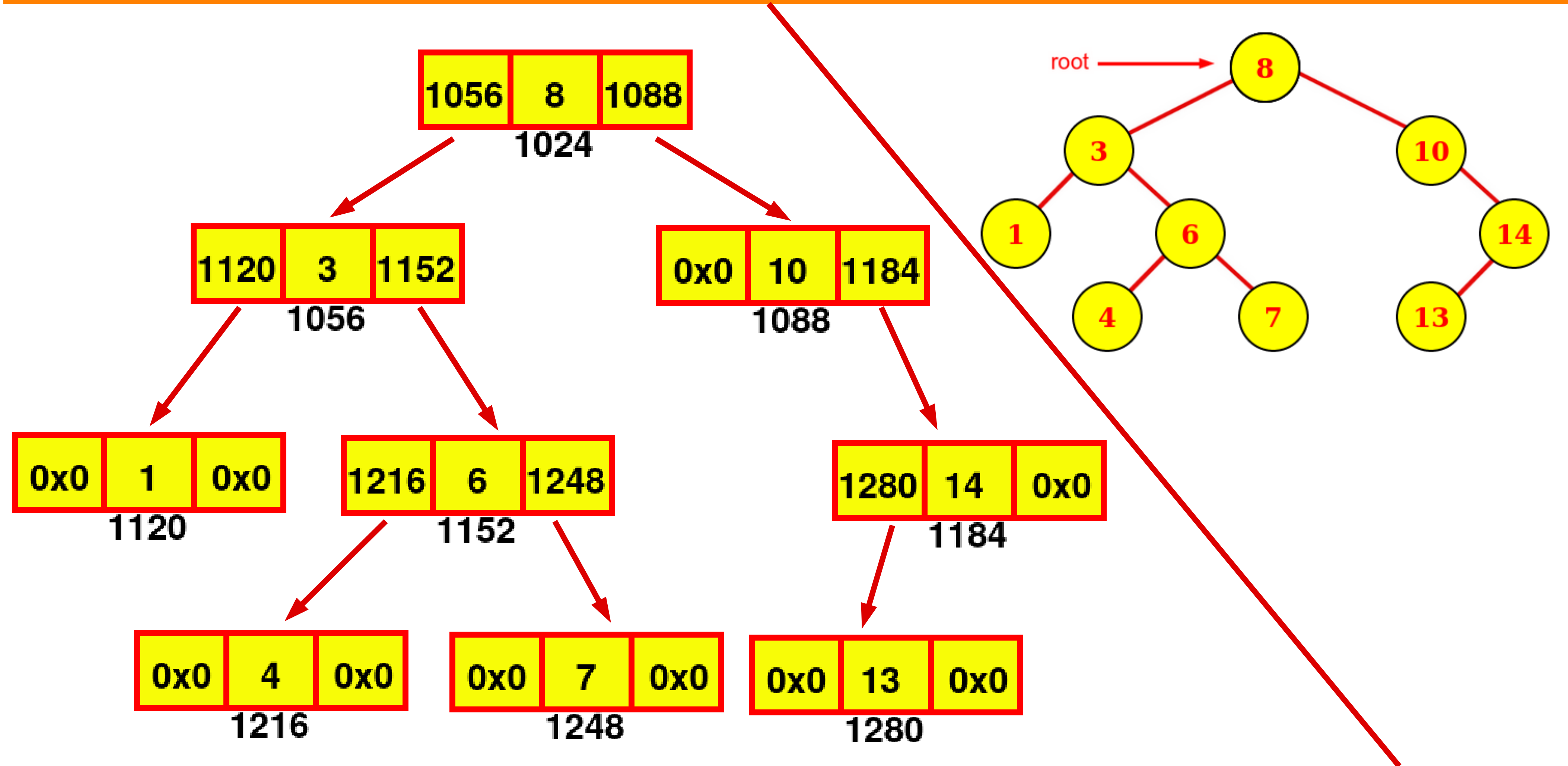
BST Deletion Operation

- If node to be deleted(z) is a **leaf node**,
  - the **parent node's pointer to the z** replaced with **null**

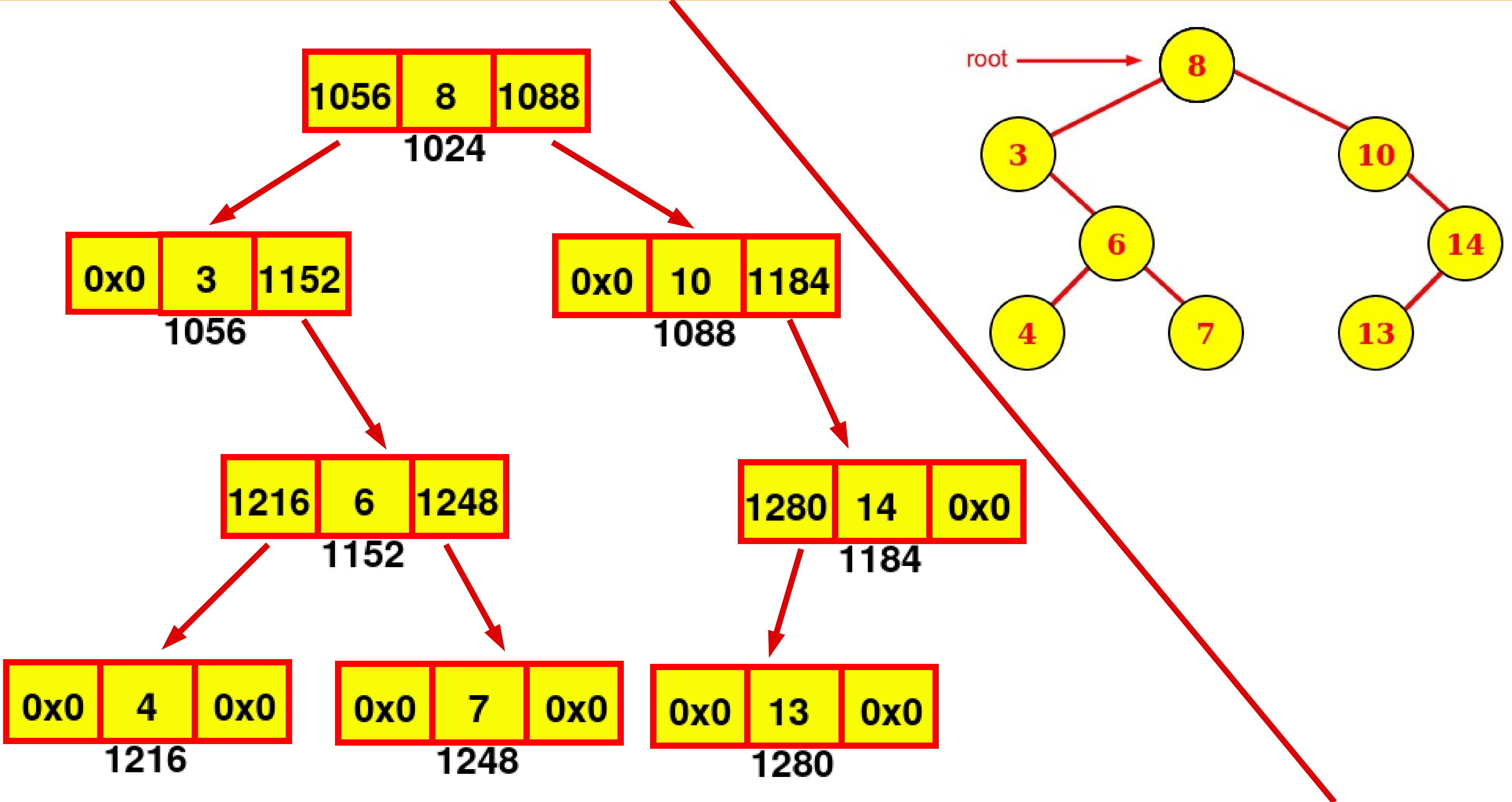
- If node to be deleted(z) has a **single child node**
  - the **parent node's pointer to the z** replaced with **z child**

- If z has both a **left and right child**
  - the successor of z (let it be y) takes the position of z in the tree(Replace node with successor).
  - y will be deleted

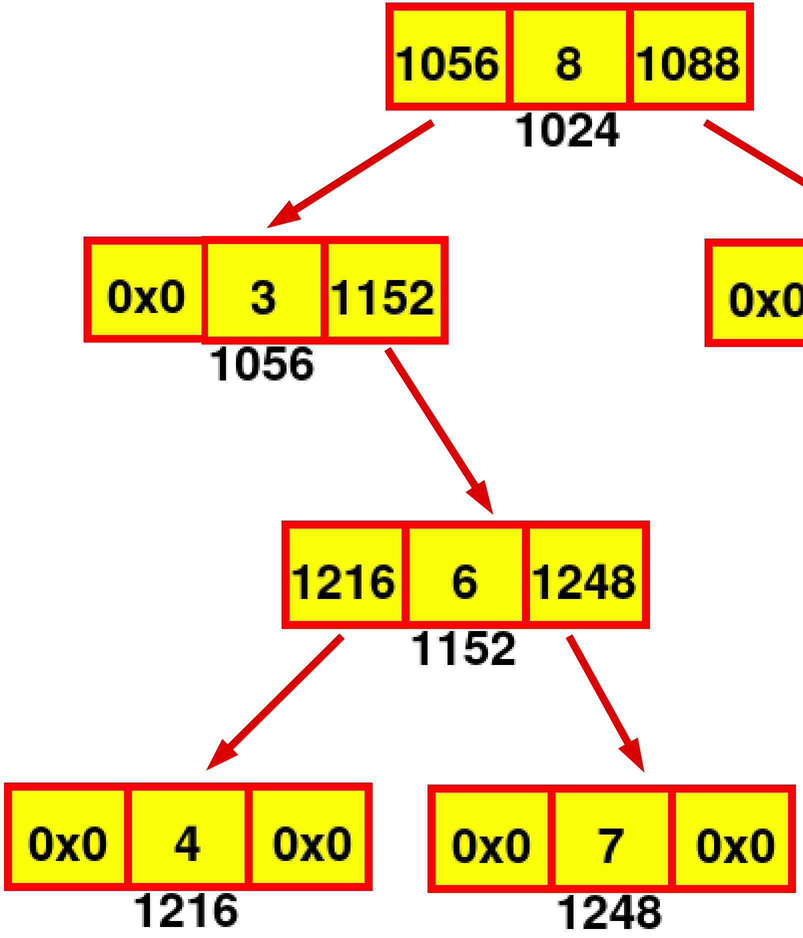
BST Deletion Operation

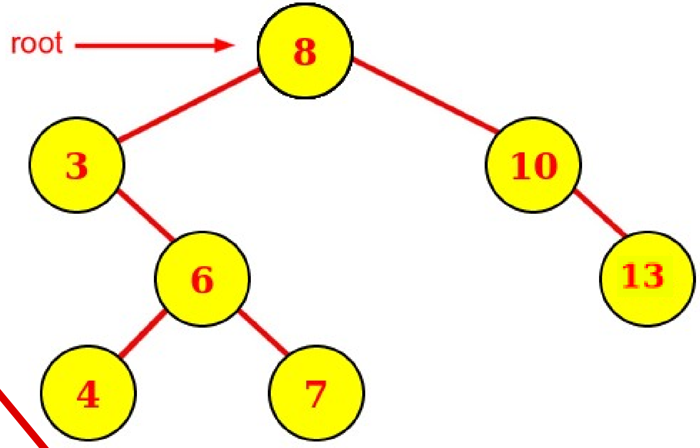


BST Deletion Operation



BST Deletion Operation





BST Search

```
struct node* search(struct node *root, int key){
    if(root == NULL || root->key == key)
        return root;
    if(key < root->key)
        return search(root->lch,key);
    if(key > root->key)
        return search(root->rch,key);
}
```

BST Deletion

```
struct node* deletion(struct node* root, int key){
   if (root == NULL) return root;
   else if (key > root->key) root->rch = deletion(root->rch, key);
   else if(key < root->key) root->lch = deletion(root->lch, key);
   else{
      if(root->lch && root->rch){       /*if node has both right and left childs*/
         struct node* sucessor = root->rch;
         while (sucessor->lch != NULL)
            sucessor = sucessor->lch;
         root->key = sucessor->key;
         root->rch = deletion(root->rch, sucessor->key);
      }else {
         if(root->rch)   return root->rch;       /*if node has right child*/
         else   return root->lch;       /*if node has left child or no childs*/
         free(root);
      }
   }
   return root;
}
```