Deleting a node from a binary search tree

This operation is rather more complicated. When a node is removed, you must ensure that the tree is still a binary search tree. There are a few ways of doing this. One approach is as follows:

First search down through the tree to located the node to be deleted, x say. Let p be its parent node. Set t = x and then locate the node x that is going to replace t. Once x is found it is linked to p and its left and right subtree are considered.

There are 3 cases to be considered when deleting the node **t**:

1. t has no right hand child node

   ```
   t->r == z
   ```

2. t has a right hand child but its right hand child node has no left subtree

   ```
   t->r->l == z
   ```

3. t has a right hand child node and the right hand child node has a left hand child node

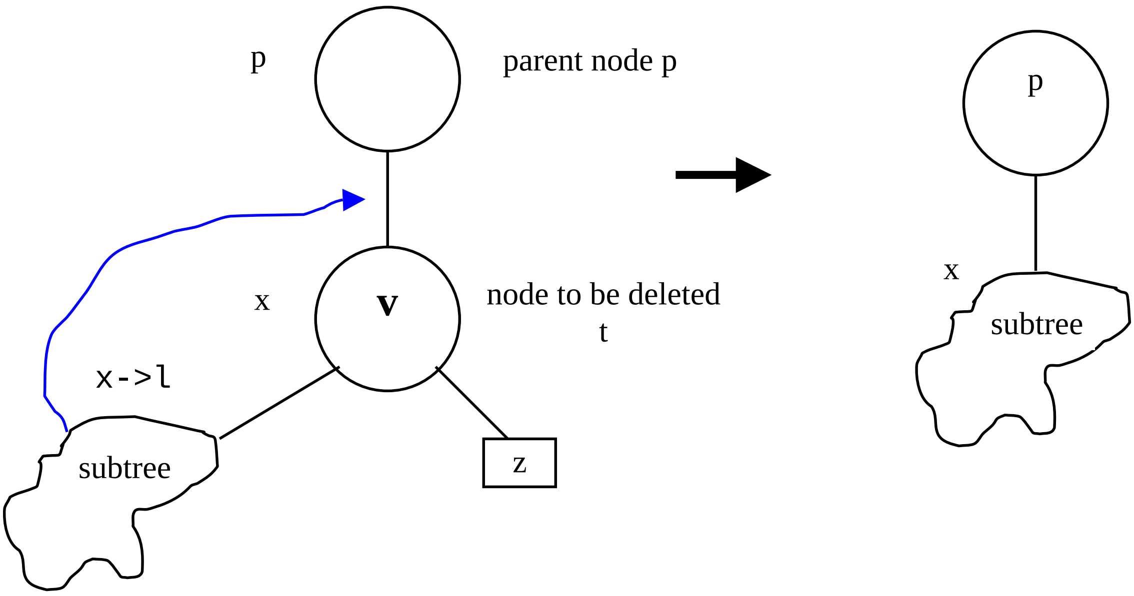
   ```
   t->r->l != z
   ```

First of all we locate the node to be deleted with:

```
node *p, *t, *x, *c;
z->key = v;
p = head;  x = head->r;
while( v != x->key)
     { p = x; x = (v < x->key) ? x->l : x->r; }
t = x;
```

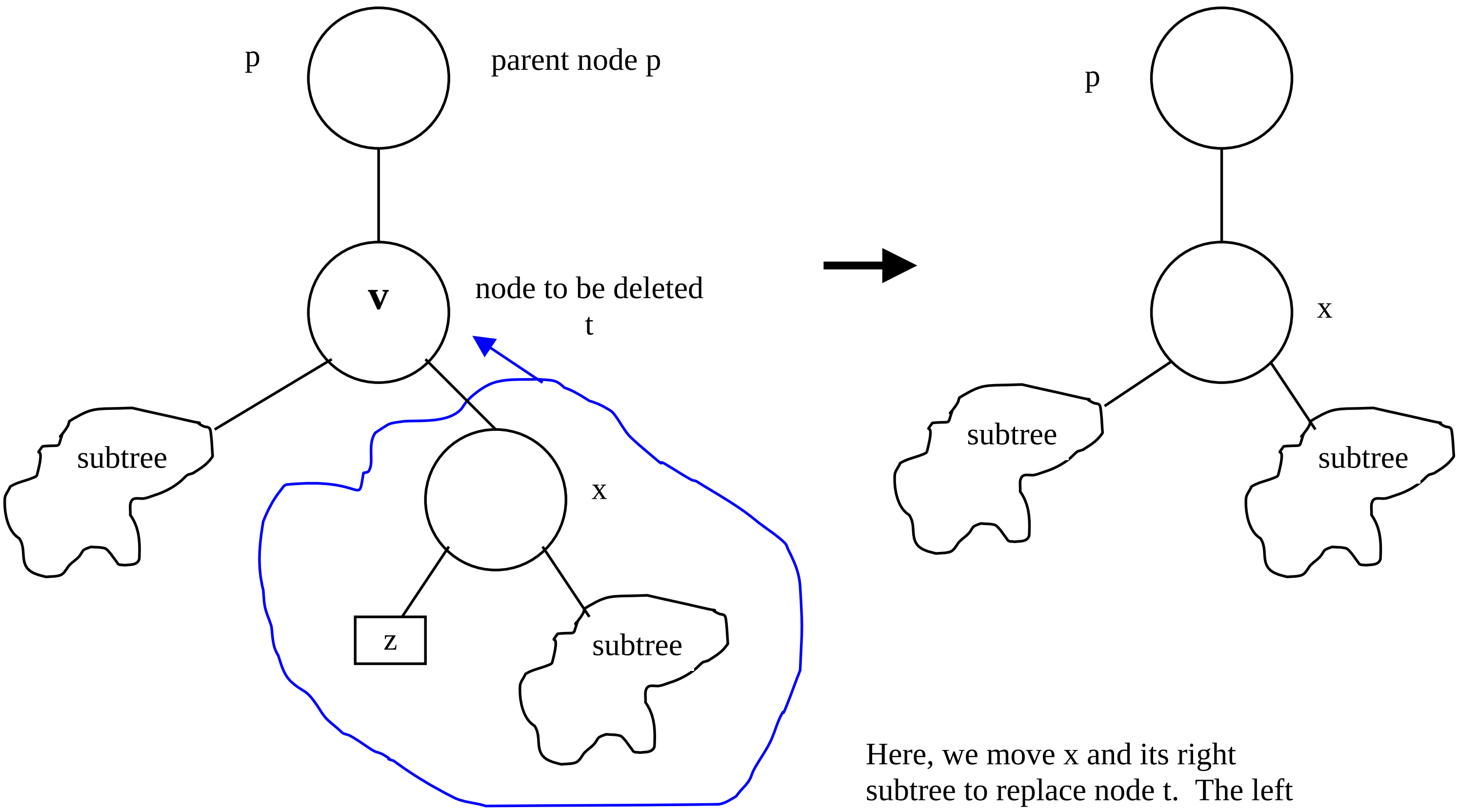
Next we will examine each of the 3 cases in more detail. The first 2 cases are

Case 1




```
if( t->r == z)  x = x->l;
…
…
delete t;
if (v < p->key) p->l = x; else p->r = x;
```

Case 2



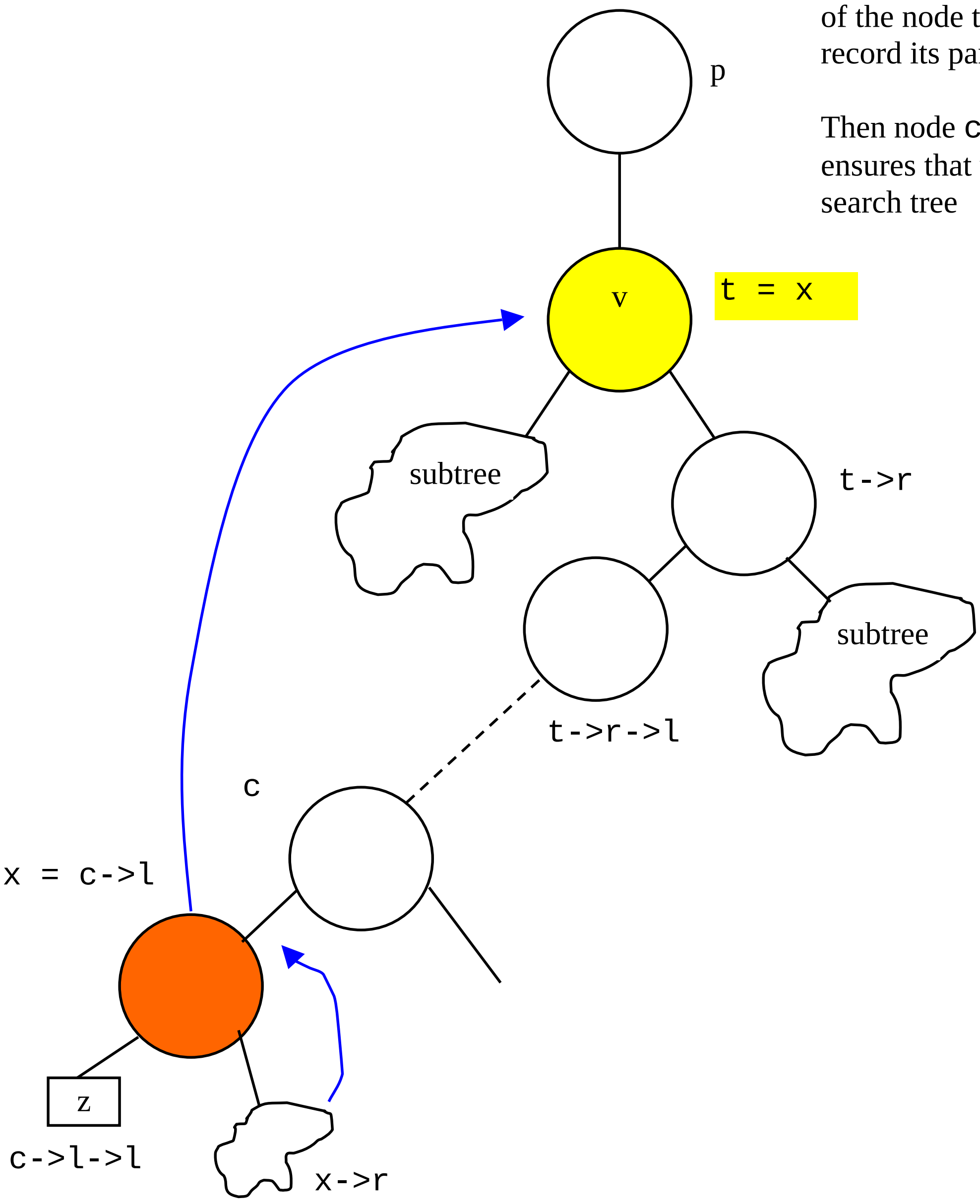
Here, we move x and its right subtree to replace node t. The left subtree of t will become a left subtree of x. This is ok as x has no left subtree to begin with

```
if( t->r == z)  x = x->l;
else if (t->r-l == z) { x = x->r; x->l = t->l;}
…
…
delete t;
if (v < p->key) p->l = x; else p->r = x;
```

Case 3 `t->r->l != z`

In this case we locate the next biggest node in the right subtree of the node to be deleted t. We record its parent with c.

Then node `c->l` replaces t. This ensures that we still have a binary search tree



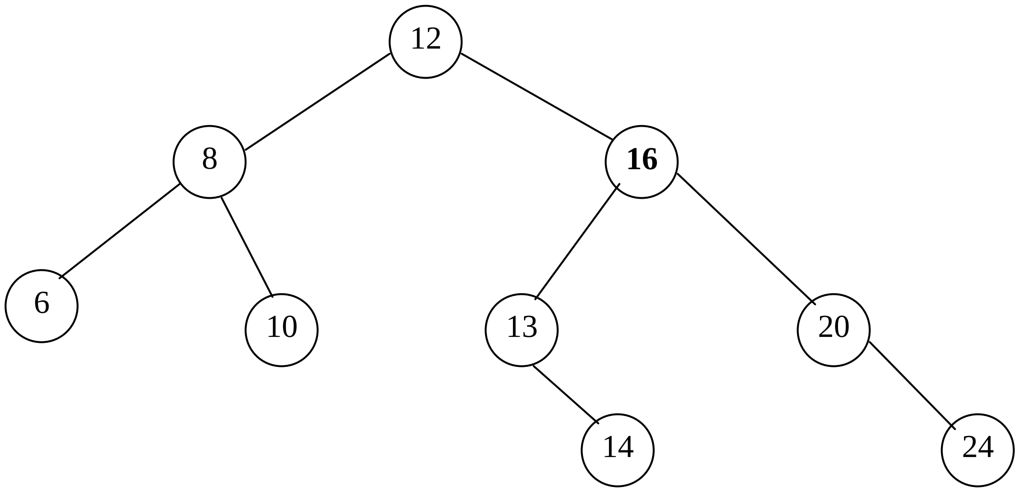

```
if( t->r == z)  x = x->l;
else if (t->r-l == z) { x = x->r; x->l = t->l;}
else {
     c = x->r; while(c->l->l != z) c = c->l;
     x = c->l; c->l = x->r;
     x->l = t->l;  x->r = t->r;
}
delete t;
if (v < p->key) p->l = x; else p->r = x;
```

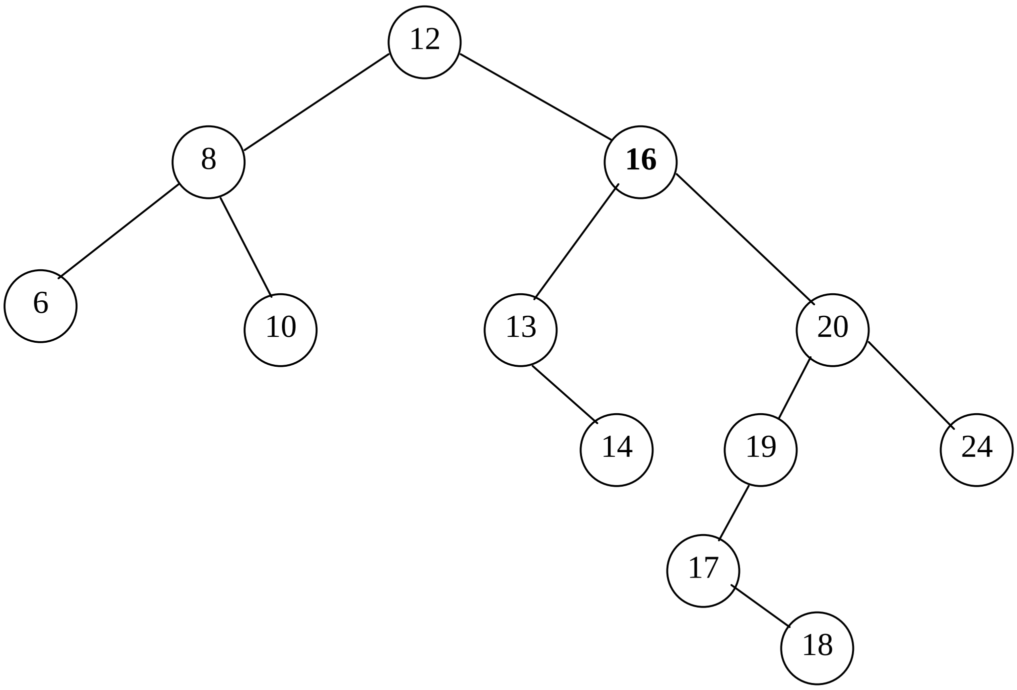
The entire removal method can now be written as:

```
void Dict::remove(itemType v)
{
     node *p, *t, *x, *c;
     z->key = v;
     p = head;  x = head->r;
     while( v != x->key)
          { p = x; x = (v < x->key) ? x->l : x->r; }
     t = x;
     if( t->r == z)  x = x->l;
     else if (t->r-l == z) { x = x->r; x->l = t->l;}
     else {
          c = x->r; while(c->l->l != z) c = c->l;
          x = c->l; c->l = x->r;
          x->l = t->l;  x->r = t->r;
     }
     delete t;
     if (v < p->key) p->l = x; else p->r = x;
}
```

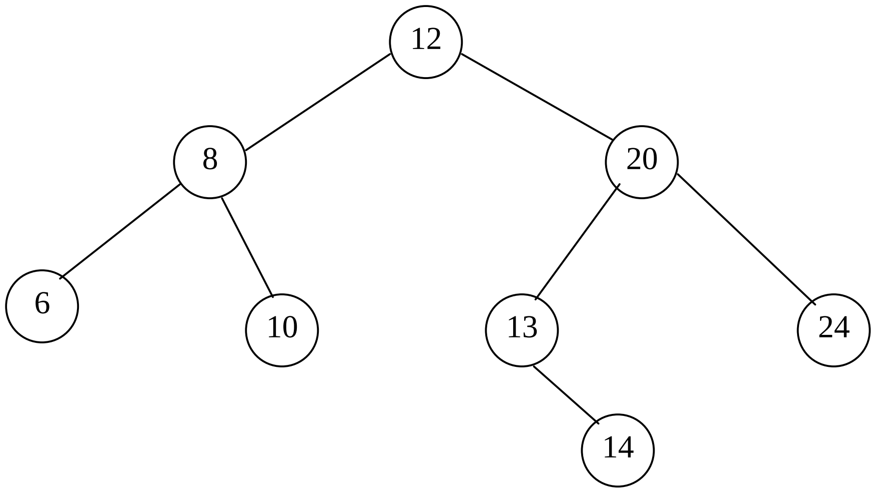
Exercise

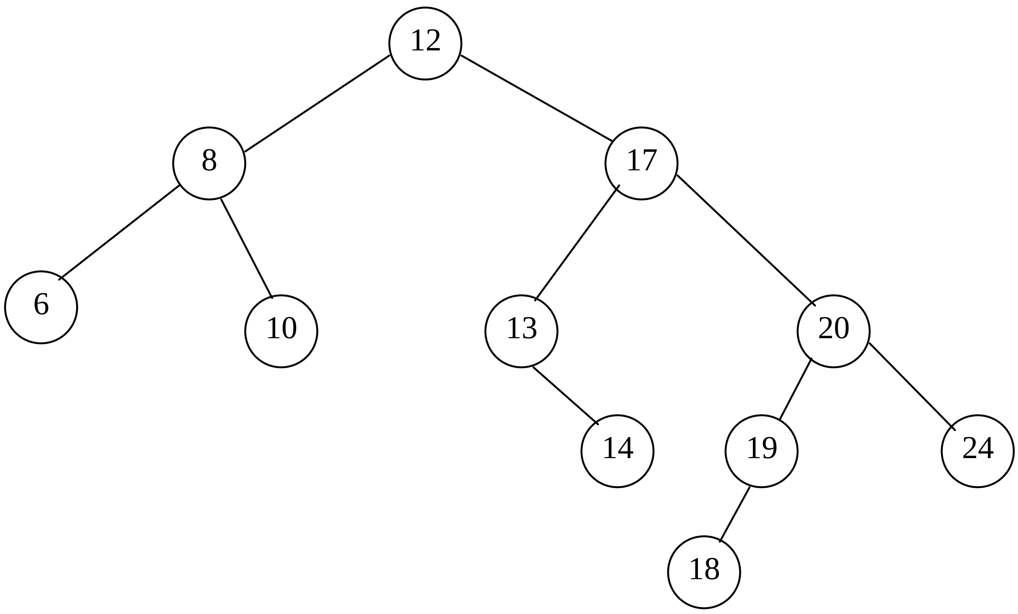
Redraw the following trees after the the node with key value 16 has been deleted.

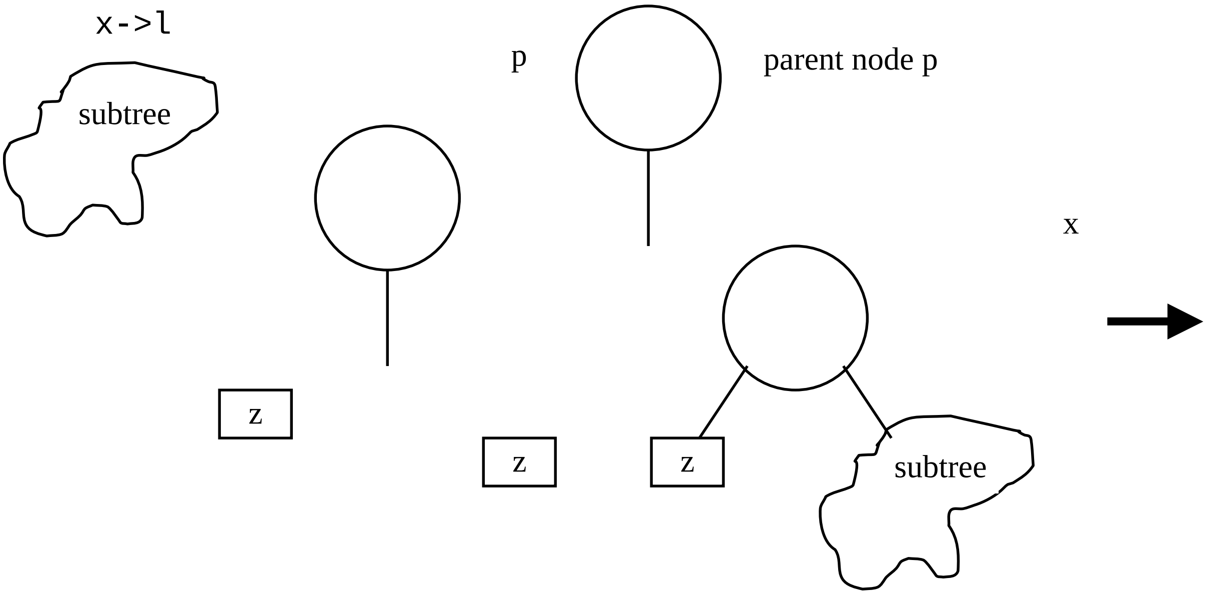




Solution






x->l
subtree
p
parent node p
x
z
z
z
subtree