≡BTree≡ Delete

Overview: Remember that a (non-root) node must have $\geq t-1$ keys and $\leq 2t-1$ keys

Step 0: Start w/ x = root, THEN:

Case 1: Target key is in a leaf node (named "x") w/ $> t-1$ keys: Remove target
↳ this will leave $\geq t-1$ keys

Case 2: Target key "k" is in an internal node "x":

a: Find node "prev" that contains the predecessor of "k" (i.e., the next smaller key). If "prev" has $> t-1$ keys, then steal the predecessor & use it to replace "k".
↳ We will need to recursively delete the predecessor

b: If (a) didn't work out (i.e., "prev" had exactly $t-1$ keys, so we can't remove any (easily)), then find node "next" that contains the successor (i.e., next larger) of "k". If next has $> t-1$ keys, then overwrite "k" w/ the successor, and recursively delete the successor from "next".

c: If neither (a) nor (b) didn't work, it must be the case that both "prev" and "next" nodes BOTH have exactly $t-1$ keys. If we merge them, the resulting node will have $2(t-1) = 2t-2$ keys. This is $\leq 2t-1$, so we merge them, so we merge them, and drop "k" (and the reference to "next") from "x".

Note: if removing the key results in an empty root node, then the node @ the next level down is made into the new root (merging, if needed)

Case 3: Node "x" doesn't contain k_{target} and "x" is internal. We will need to traverse down the tree, searching for "k". As we go, we will first make sure that each child node we search FIRST has $> t-1$ keys (so if we have to run Case 2c, above, we'll be ok) There are two basic strategies to do this:

Step 0: Figure out which node- "child" should/may contain "k"

Case 3a: "child" has only t-1 keys, but a sibling has >t-1 keys. Move a key from the appropriate sibling into "child", then recursively call BTreeDelete on "child"

Case 3b: "child" has only t-1 keys, as do both siblings. Merge "child" w/ a sibling, move appropriate key from "x" into "child" & remove the reference to the now-removed sibling. Recursively call BTreeDelete on "child"

(Case 3c: "child" has >t-1 keys, so we don't need to do anything - just recursively call BTreeDelete("child")

≡BTree≡Delete≡ Pseudocode


```
BTreeRemove ( BTreeNode "x" cur , Key "k" target )
{   if ( cur.fLeaf )
    {   // either cur == tree.root || cur.cKeys > t-1 )
        RemoveKeyFromNode (cur, target);
        return
    } else // internal node
    {   int iTarget = FindNearestKey (cur, target);
iTarget<cur.cKeys && if( cur.rgKeys[iTarget] == target) // Case 2
        {
            if( cur.rgChildren[iTarget].cKeys > t - 1) //Case 2a
            { BTreeNode replacement = RemoveKeyFromNode (MAX, cur.rgChildren
                                                        [iTarget] );

              ReplaceKey( cur, target, replacement );
              return
            }
            else if (cur.rgChildren[iTarget +1].cKeys > t-1) // Case 2b
            {   BTreeNode replacement;
                replacement = RemoveKeyFromNode (MIN, cur.rgChildren[iTarget+1]);
                ReplaceKey (cur, target, replacement)
                return
            } else                                  // Case 2c
            {
                MergeNodes (cur.rgChildren[iTarget], cur.rgChildren[iTarget+1])
                RemoveKeyFromNode (cur, target);
                return
            }
        } // still internal node, but cur doesn't contain target
        // CON'T
        // Case 3
```

```
// Case 3
BTreeNode  child= cur.rgChildren[iTarget];
if(child.cKeys > t-1)          // Case 3c
    BTreeRemove(child, target);
else { BTreeNode sibSmaller=(iTarget>0?.cur.rgChildren[iTarget-1]:null);
    BTreeNode  sibLarger=(iTarget<cur.cKeys?.cur.rgChildren[iTarget+1]:null);
    if(sibSmaller!=null && sibSmaller.cKeys>t-1)        // Case 3a
    { BTreeNode addition;
      addition=RemoveKeyFromNode( MAX , sibSmaller);
InsertKeyNonfull (child, addition);
      return
  }
  else if(sibLarger != null && sibLarger.cKeys > t-1) // Case 3a
    {  BTreeNode addition;                            // (other sibling)
       addition = RemoveKeyFromNode (MIN, sibLarger);
       InsertKeyNonfull (child, addition);
       return
    }else          // Case 3b: need to merge child into a sibling
    { if(sibLarger!=null)
      {            MergeNodes (child, sibLarger);
BTreeNode  key = RemoveKeyFromNode(cur , cur.rgChildren[iTarget]);
            InsertKeyNonfull (child ,key);
            return  BTreeRemove (child, target);
      }else // sibSmaller !=null
      {          MergeNodes (  sibSmaller, child);
          ~~BTreeNode~~ key = RemoveKeyFromNode( cur.rgChildren[iTarget])
            InsertKeyNonfull (sibSmaller, key);
            return  BTreeRemove(sibSmaller, target);
      }
}
```

```
~~Key~~ void    RemoveKeyFromNode( BTreeNode x, Key k)
{
    ~~Key k removed~~; boolean fFound = false;
    for (int i = 0; i <= x.cKeys ; i++)
    {   if( x.rgKeys[i] == k)
        {   fFound = true;
        }
        if( fFound && i < x.cKeys)
        {   x.rgKeys[i] = x.rgKeys[i+1]; }
        <- if(!x.fLeaf && i <= x.cKeys && fFound)
        {   x.rgChildren[i] = x.rgChildren[i+1];
            // <> i+1?
        }
    }
    if (fFound) x.cKeys--;
}
```

```
enum RKFN
{   MIN,
    MAX,
}
```

```
Key     RemoveKeyFromNode( RKFN which, BTreeNode cur)
{   if( which == MAX)
    {                   Key retKey = cur.rgKeys[cur.cKeys-1];
        BTreeRemove(cur, retKey);
        return retKey;
    }else // which == MIN                    // this can be factored out
    {   Key retKey = cur.rgKeys[0];          // MUCH better
        BTreeRemove(cur, retKey);
        return retKey;
    }
}

void    ReplaceKey( BTreeNode cur, Key target, Key replacement)
{   for (int i=0; i<cur.cKeys; i++)
    {   if( cur.rgKeys[i] == target)
        {   cur.rgKeys[i] = replacement;
            return;
        }
    }
}
```

```
void MergeNodes (BTreeNode to, BTreeNode from)
{ // to.cKeys + from.cKeys must be <= 2t - 1
    for(int i = 0; i < from.cKeys; i++)
    {
        to.rgKeys[ to.cKeys + i] = from.rgKeys [i];
        if(!to.fLeaf ) // then from should not be a leaf, either
        {       // copy child references, too


        }
    }
}
```

EC

```
BTreeRemove(BTree tree, Key target)
{   BTreeNode root = tree.root;                    do the merge thing)
    if ( root.cKeys == 1 && root.rgKeys[0] == target )
    {   tree.root = root.rgChildren

    } // root has > t-1 keys, and is not empty

    if( root.fLeaf == true )
    {   return ~~RemoveKeyFromNode(~~ BTreeRemove(root, target) ); // may be left w/ empty node
    }
    else // root.fLeaf == false - internal node
    {   if(root.cKeys > t-1) /*normal*/ || root.rgKeys[0] != target )
            BTreeRemove(root, target) //other fox
        else /* 1 key */ if (root.rgKeys == target)
        {   if (root.rgChildren[0].cKeys > t-1)
        {   replacementKey = Remove~~Node~~KeyFromFullNode(root.rgChildren[0], MAX);
            else if (root.rgChildren[1].cKeys > t-1)
            {   replacementKey = [1], MIN);
            }
            else
            {   MergeNodes (root.rgChildren[0], ~~t~~ [1]);
                treeroot = root.rgChildren[0];
            }
```

ReplaceKey(target, replacement);
STOP

no