CS302 - Data Structures

using C++

Topic: Left-Leaning Red-Black Trees Implementation

Kostas Alexis

Relying on the implementation by Lee Stanza – consider these slides only for lecture use. Link to the online implementation by Lee Stanza: http://www.teachsolaisgames.com/articles/balanced_left_leaning.html

Left-Leaning Red-Black Trees Implementation

- **Data Structures**
  - For evaluation purposes, a simple key-value pair is defined, where data is sorted with an unsigned 32bit integer.
    - The value is a blind void* pointer.

```
struct VoidRef_t

{

        U32 Key;

        void* pContext;

};
```

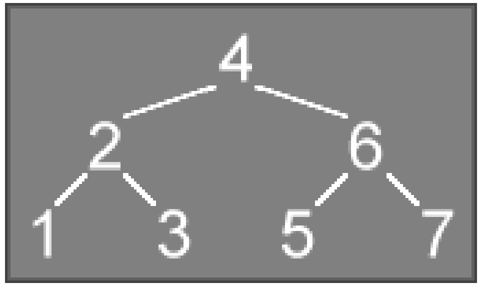
- Each node in the tree uses the following structure

```
struct LLTB_t

{

        VoidRef_t Ref;

        bool IsRed;
        LLTB_t* pLeft;

        LLTB_t* pRight;

};
```

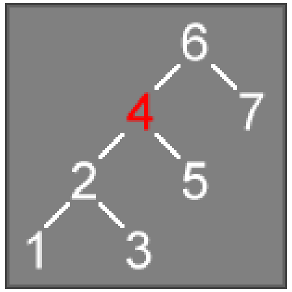
Relying on the implementation by Lee Stanza: http://www.teachsolaisgames.com/articles/balanced_left_leaning.html

Left-Leaning Red-Black Trees Implementation

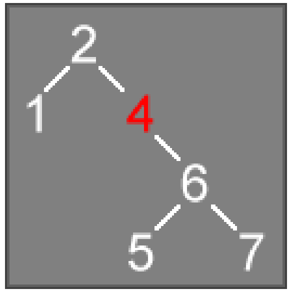
- **Insertion**
  - Consider the following tree



  - One operation used by both insertion and deletion is RotateLeft.
    - This function will rotate the nodes to the left – "6" will take on the red/black color that "4" had before and "4" will change to be a red node.



  - Likewise, the RotateRight function will rotate the nodes to the right.
    - This function will make "2" take on the red/black color that "4" had previously and then change "4" to be a red node.

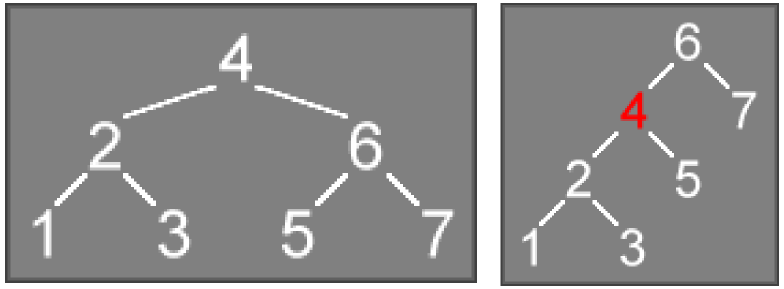


Relying on the implementation by Lee Stanza: http://www.teachsolaisgames.com/articles/balanced_left_leaning.html

Left-Leaning Red-Black Trees Implementation

- **Insertion**
    - **`RotateLeft`**

```
static QzLLTB_t* RotateLeft(QzLLTB_t *pNode)
{
        QzLLTB_t *pTemp = pNode->pRight;
        pNode->pRight = pTemp->pLeft;
        pTemp->pLeft = pNode;
        pTemp->IsRed = pNode->IsRed;
        pNode->IsRed = true;
        return pTemp;
}
```



Relying on the implementation by Lee Stanza: http://www.teachsolaisgames.com/articles/balanced_left_leaning.html

Left-Leaning Red-Black Trees Implementation

- **Insertion**
    - **`RotateRight`**

```
static QzLLTB_t* RotateRight(QzLLTB_t *pNode)
{
        QzLLTB_t *pTemp = pNode->pLeft;
        pNode->pLeft = pTemp->pRight;
        pTemp->pRight = pNode;
        pTemp->IsRed = pNode->IsRed;
        pNode->IsRed = true;
        return pTemp;
}
```

Relying on the implementation by Lee Stanza: http://www.teachsolaisgames.com/articles/balanced_left_leaning.html

Left-Leaning Red-Black Trees Implementation

- **Insertion**
  - `ColorFlip` – Given any node, this method will toggle the red/black color of that node and both of its children

```
static void ColorFlip(QzLLTB_t *pNode)
{
        pNode->IsRed = !pNode->IsRed;

        if (NULL != pNode->pLeft) {
                pNode->pLeft->IsRed = !pNode->pLeft->IsRed;
        }

        if (NULL != pNode->pRight) {
                pNode->pRight->IsRed = !pNode->pRight->IsRed;
        }
}
```

  - **This operation may introduce a color violation. We will need to do fix-up.**

Relying on the implementation by Lee Stanza: http://www.teachsolaisgames.com/articles/balanced_left_leaning.html

Left-Leaning Red-Black Trees Implementation

- **Insertion**
  - **`InsertRec`** – Recursively traverse the tree to find the location of new key insertion.

```
QzLLTB_t* LeftLeaningRedBlack::InsertRec(QzLLTB_t* pNode, VoidRef_t ref)
{
           // Special case for inserting a leaf. Just return the pointer;
           // the caller will insert the new node into the parent node.
           if (NULL == pNode) {
                      pNode = NewNode();
                      pNode->Ref = ref;
                      return pNode;
           }

           // If we perform the color flip here, the tree is assembled as a
           // mapping of a 2-3-4 tree.
           #if defined(USE_234_TREE)
                      // This color flip will effectively split 4-nodes on the way down
                      // the tree (since 4-nodes must be represented by a node with two
                      // red children). By performing the color flip here, the 4-nodes
                      // will remain in the tree after the insertion.
                      if (IsRed(pNode->pLeft) && IsRed(pNode->pRight)) {
                                 ColorFlip(pNode);
           }
#endif
```

Relying on the implementation by Lee Stanza: http://www.teachsolaisgames.com/articles/balanced_left_leaning.html

Left-Leaning Red-Black Trees Implementation

- **Insertion**
  - **`InsertRec`** – Recursively traverse the tree to find the location of new key insertion.

```
// Check to see if the value is already in the tree. If so, we
// simply replace the value of the key, since duplicate keys are
// not allowed.
if (ref.Key == pNode->Ref.Key) {
           pNode->Ref = ref;
}
// Otherwise recurse left or right depending on key value.
// Note: pLeft or pRight may be a NULL pointer before recursing.
// This indicates that pNode is a leaf (or only has one child),
// so the new node will be inserted using the return value.
//
// The other reason for pass-by-value, followed by an assignment,
// is that the recursive call may perform a rotation, so the
// pointer that gets passed in may end up not being the root of
// the subtree once the recursion returns.
else {
           if (ref.Key < pNode->Ref.Key) {
                      pNode->pLeft = InsertRec(pNode->pLeft, ref);
           }
else {
           pNode->pRight = InsertRec(pNode->pRight, ref);
           }
}
```

Relying on the implementation by Lee Stanza: http://www.teachsolaisgames.com/articles/balanced_left_leaning.html

Left-Leaning Red-Black Trees Implementation

- **Insertion**
  - **`InsertRec`** – Recursively traverse the tree to find the location of new key insertion.

```
// If necessary, apply a rotation to get the correct representation
// in the parent node as we're backing out of the recursion. This
// places the tree in a state where the parent can safely apply a
// rotation to restore the required black/red balance of the tree.

// Fix a right-leaning red node: this will assure that a 3-node is
// the left child.
if (IsRed(pNode->pRight) && (false == IsRed(pNode->pLeft)))
{
           pNode = RotateLeft(pNode);
}
```

Relying on the implementation by Lee Stanza: http://www.teachsolaisgames.com/articles/balanced_left_leaning.html

Left-Leaning Red-Black Trees Implementation

- **Insertion**
    - `InsertRec` – Recursively traverse the tree to find the location of new key insertion.

```
// Fix two reds in a row: this will rebalance a 4-node.
if (IsRed(pNode->pLeft) && IsRed(pNode->pLeft->pLeft))
{
           pNode = RotateRight(pNode);
}
```

Relying on the implementation by Lee Stanza: http://www.teachsolaisgames.com/articles/balanced_left_leaning.html

Left-Leaning Red-Black Trees Implementation

- **Insertion**
  - **`InsertRec`** – Recursively traverse the tree to find the location of new key insertion.

```
        // If we perform the color flip here, the tree is assembled as a
        // mapping of a 2-3 tree.
        #if !defined(USE_234_TREE)
        // This color flip will effectively split 4-nodes on the way back
        // out of the tree. By doing this here, there will be no 4-nodes
        // left in the tree after the insertion is complete.
        if (IsRed(pNode->pLeft) && IsRed(pNode->pRight)) {
                ColorFlip(pNode);
        }
        #endif

        // Return the new root of the subtree that was just updated,
        // since rotations may have changed the value of this pointer.
        return pNode;
}
```

  - Note: USE_234_TREE - will control where the color flip is performed.
    - If done near the beginning of the insertion, this will produce a tree that maps to a 2-3-4 tree.
    - But if the color flip is near the end of the function, the tree will be equivalent to a 2-3 tree.

Relying on the implementation by Lee Stanza: http://www.teachsolaisgames.com/articles/balanced_left_leaning.html

Left-Leaning Red-Black Trees Implementation

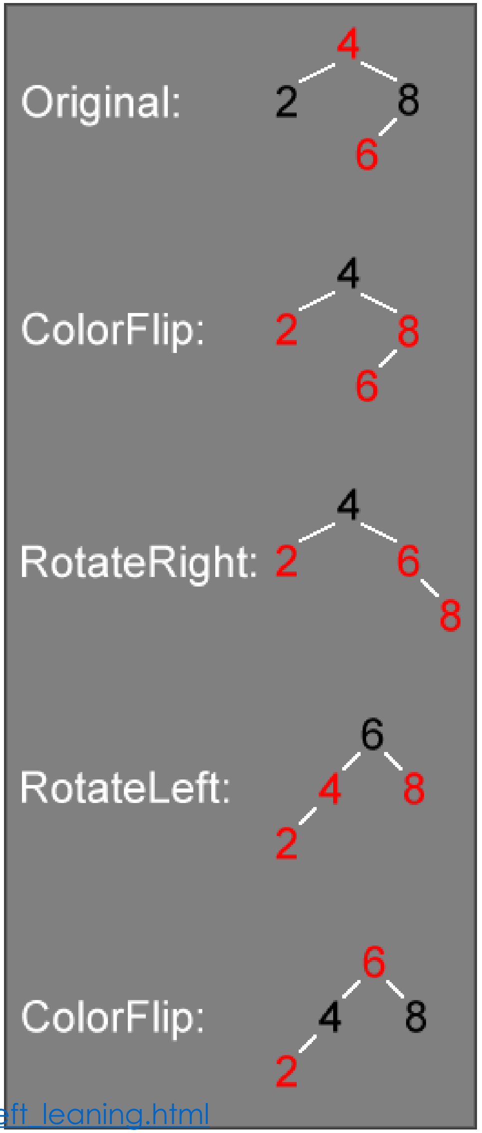
- **Insertion**
  - **`Insert`** – Use InsertRec and ensure that the root of the tree remains black.

```
bool LeftLeaningRedBlack::Insert(VoidRef_t ref)
{
        m_pRoot = InsertRec(m_pRoot, ref);

        // The root node of a red-black tree must be black.
        m_pRoot->IsRed = false;

        return true;
}
```

Relying on the implementation by Lee Stanza: http://www.teachsolaisgames.com/articles/balanced_left_leaning.html

Left-Leaning Red-Black Trees Implementation

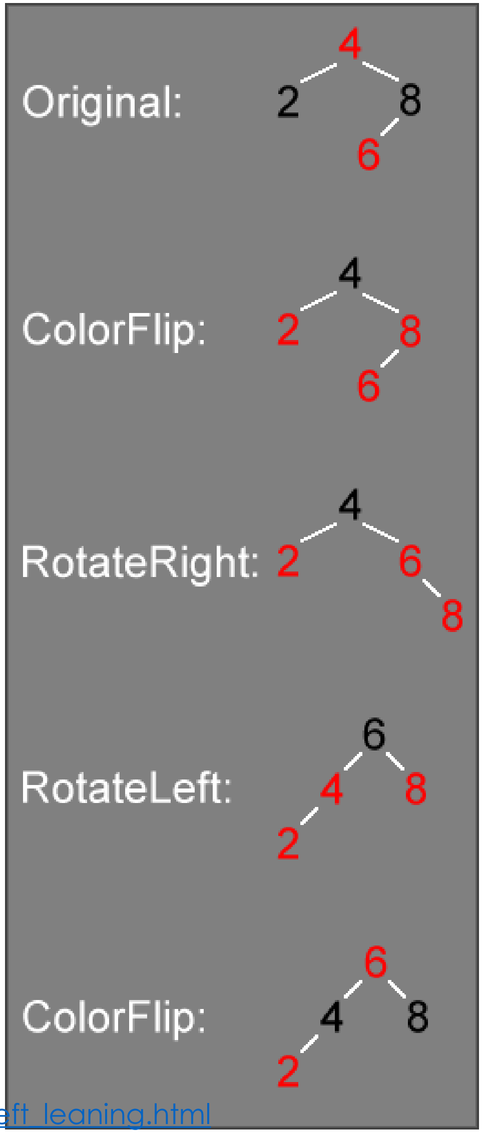
- **Deletion**
  - **`MoveRedLeft`** – when given a sub-tree that starts with a red node, it applies the following transformations:



Relying on the implementation by Lee Stanza: http://www.teachsolaisgames.com/articles/balanced_left_leaning.html

Left-Leaning Red-Black Trees Implementation

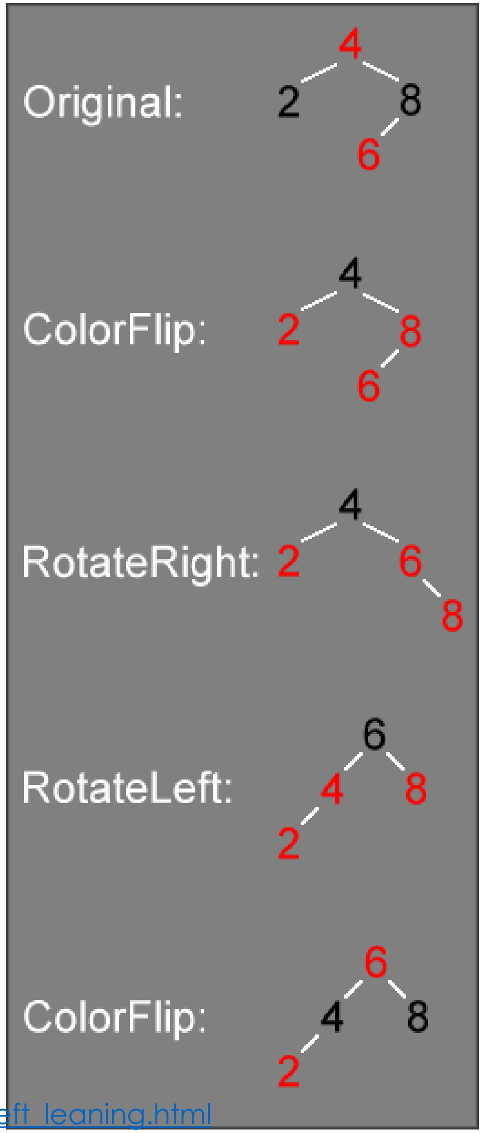
- **Deletion**
  - **`MoveRedLeft`** – when given a sub-tree that starts with a red node, it applies the following transformations:
    - Goal to change the 3-node (the node with the single red child) from being the right child to being the left child.



Relying on the implementation by Lee Stanza: http://www.teachsolaisgames.com/articles/balanced_left_leaning.html

Left-Leaning Red-Black Trees Implementation

- **Deletion**
  - `MoveRedLeft` – when given a sub-tree that starts with a red node, it applies the following transformations:
    - Goal to change the 3-node (the node with the single red child) from being the right child to being the left child.
    - Once complete, the new root of the subtree is still a red node, but the 3-node is now the left child which preserves “left-leaningness”.



Relying on the implementation by Lee Stanza: http://www.teachsolaisgames.com/articles/balanced_left_leaning.html

Left-Leaning Red-Black Trees Implementation

- **Deletion**
  - **MoveRedLeft**

```
static QzLLTB_t* MoveRedLeft(QzLLTB_t *pNode)
{
        // If both children are black, we turn these three nodes into a
        // 4-node by applying a color flip.
        ColorFlip(pNode);

        // But we may end up with a case where pRight has a red child.
        // Apply a pair of rotations and a color flip to make pNode a
        // red node, both of its children become black nodes, and pLeft
        // becomes a 3-node.
        if ((NULL != pNode->pRight) && IsRed(pNode->pRight->pLeft))
        {
                pNode->pRight = RotateRight(pNode->pRight);
                pNode = RotateLeft(pNode);
                ColorFlip(pNode);
        }
        return pNode;
}
```

Relying on the implementation by Lee Stanza: http://www.teachsolaisgames.com/articles/balanced_left_leaning.html

Left-Leaning Red-Black Trees Implementation

- **Deletion**
  - **`ModeRedRight`** – similar / use when pNode has two red children and its left child is a 3 node. **This re-arranges the subtree so that the root is a red node & left-child is still a 3-node**

```
static QzLLTB_t* MoveRedRight(QzLLTB_t *pNode)
{
        // Applying a color flip may turn pNode into a 4-node,
        // with both of its children being red.
        ColorFlip(pNode);

        // However, this may cause a situation where both of pNode's
        // children are red, along with pNode->pLeft->pLeft. Applying a
        // rotation and a color flip will fix this special case, since
        // it makes pNode red and pNode's children black.
        if ((NULL != pNode->pLeft) && IsRed(pNode->pLeft->pLeft))
        {
                pNode = RotateRight(pNode);
                ColorFlip(pNode);
        }
        return pNode;
}
```

Relying on the implementation by Lee Stanza: http://www.teachsolaisgames.com/articles/balanced_left_leaning.html

Left-Leaning Red-Black Trees Implementation

- **Deletion**
  - **`FindMin`** – deleting internal nodes has to deal with the fact that often re-arranging a significant part of the tree is required.
  - **Instead we find the smallest key that is larger than the key being deleted.**

Relying on the implementation by Lee Stanza: http://www.teachsolaisgames.com/articles/balanced_left_leaning.html

Left-Leaning Red-Black Trees Implementation

- **Deletion**
  - **`FindMin`** – deleting internal nodes has to deal with the fact that often re-arranging a significant part of the tree is required.
  - **Instead we find the smallest key that is larger than the key being deleted.**
  - Done by starting at the node's right child, then traversing left until we reach a leaf node.
  - The contents of that leaf node can then be used to replace the value being deleted (replacement key).

Relying on the implementation by Lee Stanza: http://www.teachsolaisgames.com/articles/balanced_left_leaning.html

Left-Leaning Red-Black Trees Implementation

- **Deletion**
    - **`FindMin`** – Find replacement key

```
static QzLLTB_t* FindMin(QzLLTB_t *pNode)
{
        while (NULL != pNode->pLeft) {
                pNode = pNode->pLeft;
        }

        return pNode;
}
```

Relying on the implementation by Lee Stanza: http://www.teachsolaisgames.com/articles/balanced_left_leaning.html

Left-Leaning Red-Black Trees Implementation

- **Deletion**
  - **`FindMin`** – Find replacement key

```
static QzLLTB_t* FindMin(QzLLTB_t *pNode)
{
        while (NULL != pNode->pLeft) {
                pNode = pNode->pLeft;
        }

        return pNode;
}
```

  - **Why is this the solution?**

Relying on the implementation by Lee Stanza: http://www.teachsolaisgames.com/articles/balanced_left_leaning.html

Left-Leaning Red-Black Trees Implementation

- **Deletion**
  - **`DeleteMin`** – Delete the empty leaf node found by FindMin

```
QzLLTB_t* LeftLeaningRedBlack::DeleteMin(QzLLTB_t *pNode)
{
           // If this node has no children, we're done.
           // Due to the arrangement of an LLRB tree, the node cannot have a
           // right child.
           if (NULL == pNode->pLeft) {
                      Free(pNode);
                      return NULL;
           }
           // If these nodes are black, we need to rearrange this subtree to
           // force the left child to be red.
           if ((false == IsRed(pNode->pLeft)) && (false == IsRed(pNode->pLeft->pLeft)))
           {
                      pNode = MoveRedLeft(pNode);
           }
           // Continue recursing to locate the node to delete.
           pNode->pLeft = DeleteMin(pNode->pLeft);
           // Fix right-leaning red nodes and eliminate 4-nodes on the way up.
           // Need to avoid allowing search operations to terminate on 4-nodes,
           // or searching may not locate intended key.
           return FixUp(pNode);
}
```

Relying on the implementation by Lee Stanza: http://www.teachsolaisgames.com/articles/balanced_left_leaning.html

Left-Leaning Red-Black Trees Implementation

- **Deletion**
  - **`FixUp`** – As we recurse down the tree, **the code will leave right-leaning red nodes as unbalanced 4-nodes**. **Recover using FixUp.**

```
static QzLLTB_t* FixUp(QzLLTB_t *pNode)
{
     // Fix right-leaning red nodes.
           if (IsRed(pNode->pRight)) {
           pNode = RotateLeft(pNode);
           }
           // Detect if there is a 4-node that traverses down the left.
           // This is fixed by a right rotation, making both of the red
           // nodes the children of pNode.
           if (IsRed(pNode->pLeft) & IsRed(pNode->pLeft->pLeft))
           {
                      pNode = RotateRight(pNode);
           }
           // Split 4-nodes.
           if (IsRed(pNode->pLeft) && IsRed(pNode->pRight))
           {
                      ColorFlip(pNode);
           }
           return pNode;
}
```

Relying on the implementation by Lee Stanza: http://www.teachsolaisgames.com/articles/balanced_left_leaning.html

Left-Leaning Red-Black Trees Implementation

- **Deletion**
  - `DeleteRec` – Recursive traversal. This code will leave right-leaning red nodes and unbalanced 4-nodes as it works down the tree. **Will require FixUp.**

```
QzLLTB_t* LeftLeaningRedBlack::DeleteRec(QzLLTB_t *pNode, const U32 key)
{
           if (key < pNode->Ref.Key) {
                      if (NULL != pNode->pLeft) {
                                 // If pNode and pNode->pLeft are black, we may need to
                                 // move pRight to become the left child if a deletion
                                 // would produce a red node.
                                 if ((false == IsRed(pNode->pLeft)) && (false == IsRed(pNode->pLeft->pLeft)))
                                 {
                                            pNode = MoveRedLeft(pNode);
                                 }

                                 pNode->pLeft = DeleteRec(pNode->pLeft, key);
                      }
           }
```

Relying on the implementation by Lee Stanza: http://www.teachsolaisgames.com/articles/balanced_left_leaning.html

Left-Leaning Red-Black Trees Implementation

- **Deletion**
  - **`DeleteRec`** – Recursive traversal. This code will leave right-leaning red nodes and unbalanced 4-nodes as it works down the tree. **Will require FixUp.**

```
else {
           // If the left child is red, apply a rotation so we make
           // the right child red.
           if (IsRed(pNode->pLeft)) {
                      pNode = RotateRight(pNode);
           }

           // Special case for deletion of a leaf node.
           // The arrangement logic of LLRBs assures that in this case,
           // pNode cannot have a left child.
           if ((key == pNode->Ref.Key) && (NULL == pNode->pRight))
           {
                      Free(pNode);
                      return NULL;
           }
```

Relying on the implementation by Lee Stanza: http://www.teachsolaisgames.com/articles/balanced_left_leaning.html

Relying on the implementation by Lee Stanza: http://www.teachsolaisgames.com/articles/balanced_left_leaning.html

Left-Leaning Red-Black Trees Implementation

- **Deletion**
  - `DeleteRec` – Recursive traversal. This code will leave right-leaning red nodes and unbalanced 4-nodes as it works down the tree. **Will require FixUp.**

```
// If we get here, we need to traverse down the right node.
// However, if there is no right node, then the target key is
// not in the tree, so we can break out of the recursion.
if (NULL != pNode->pRight) {
	if ((false == IsRed(pNode->pRight)) && (false == IsRed(pNode->pRight->pLeft)))
	{
		pNode = MoveRedRight(pNode);
	}
	// Deletion of an internal node: We cannot delete this node
	// from the tree, so we have to find the node containing
	// the smallest key value that is larger than the key we're
	// deleting. This other key will replace the value we're
	// deleting, then we can delete the node that previously
	// held the key/value pair we just moved.
	if (key == pNode->Ref.Key) {
		pNode->Ref = FindMin(pNode->pRight)->Ref;
		pNode->pRight = DeleteMin(pNode->pRight);
	}
	else {
		pNode->pRight = DeleteRec(pNode->pRight, key);
	}
}
```

Left-Leaning Red-Black Trees Implementation

- **Deletion**
  - **`DeleteRec`** – Recursive traversal. This code will leave right-leaning red nodes and unbalanced 4-nodes as it works down the tree. **Will require FixUp.**

```
        }

        // Fix right-leaning red nodes and eliminate 4-nodes on the way up.
        // Need to avoid allowing search operations to terminate on 4-nodes,
        // or searching may not locate intended key.
        return FixUp(pNode);
}
```

Relying on the implementation by Lee Stanza: http://www.teachsolaisgames.com/articles/balanced_left_leaning.html

Left-Leaning Red-Black Trees Implementation

- **Deletion**
  - **`Delete`** – Rely on DeleteRec and also ensure that the root remains black.

```
void LeftLeaningRedBlack::Delete(const U32 key)
{
		if (NULL != m_pRoot) {
			m_pRoot = DeleteRec(m_pRoot, key);

			// Assuming we have not deleted the last node from the tree, we
			// need to force the root to be a black node to conform with the
			// the rules of a red-black tree.
			if (NULL != m_pRoot) {
				m_pRoot->IsRed = false;
			}
		}
}
```

Relying on the implementation by Lee Stanza: http://www.teachsolaisgames.com/articles/balanced_left_leaning.html

Thank you