Threaded Trees – findSuccessor

```
ThreadedNode findSuccessor(ThreadedNode T)
// find successor S of node T
{
  if (T.rThread)
  // thread points to successor
    S = T.right;
  else
  // successor is leftmost node in right subtree
  {
    S = T.right;
    while (!S.lThread)
    S = S.left;
  }
  return S;
}
```

Threaded Trees – Inorder Traversal

```
inorderThreadedTraverse(ThreadedNode T)
{
  S = T;
  // leftmost node is start of traversal
  while (!S.lThread)
    S=S.left;
  while (S != null)
  {
    displayNode(S); // visit S
    S = findSuccessor(S);
  }
}
```

Threaded Trees – thread existing tree

```
ThreadedNode rightThreadTree(ThreadedNode T)
{
  ThreadThis = null;
  if (T.left != null)
  { // thread left subtree
    ThreadThis = rightThreadTree(T.left);
    // ThreadThis is last node visited in T's left subtree
    // so its successor is T: set thread
    ThreadThis.right = T;
  }
  if (T.right != null)            // thread right subtree
    ThreadThis = rightThreadTree(T.right);
  else
  // T.right is null, so must be turned into a thread
  {
    ThreadThis = T;
    T.rThread = true;
  }
  return ThreadThis;   // return node that needs a thread
}
```

Threaded Trees – threadedInsert

```
void threadedInsert(ThreadedNode T, ThreadedNode newNode)
// insert newNode into tree with root T
// T is already threaded - ensure this is maintained
{
  if(T == null)
    T = newNode;
  else if (newNode.info < T.info)  // insert in left subtree
  {
    if(T.lThread)                  // insert here
    {
     newNode.left = T.left;
     newNode.right = T;
     T.left = newNode;
     T.lThread = false;
    }
    else
      T.left = threadedInsert(T.left, newNode);
  }
  else                             // insert in right subtree
  {
    if(T.rThread)                  // insert here
    {
      newNode.left = T;
      newNode.right = T.right;
      T.right = newNode;
      T.rThread = false;
    }
    else
      T.right = threadedInsert(T.right, newNode);
  }
}
```

Height-Balanced Trees

- Search complexity for binary search tree with n nodes:
  - Best case: O(log n) Worst-case: O(n)
- Height-balanced trees attempt to keep subtrees at roughly the same height
- Options for a balance condition:
  - Left and right subtrees of the root have the same height: but height can still be n/2
  - For every node, left and right subtrees have the same height: only possible if $n = 2^k - 1$
  - For every node, left and right subtrees differ in height by at most 1: this is the *AVL property*

Representation of an AVL Node

```
class AVLNode
{ < declarations for info stored in node,
  e.g. int info; >
  AVLnode left;
  AVLnode right;
  int height;

  int height(AVLNode T)
  {
    return T == null? -1 : T.height;
  }
  …
};
```

AVL Insert

```
AVLNode insert(AVLNode T, AVLNode newNode)
{
  if(T == null)
    T = newNode;
  else if(newNode.info < T.info)      // insert in left subtree
  {
    T.left = insert(T.left, newNode);
    if(height(T.left) - height(T.right) == 2)
      if(newNode.info < T.left.info)  //left subtree of T.left
        T = rotateWithLeftChild(T);
      else                            //right subtree of T.left
        T = doubleWithLeftChild(T);
  }
  else                                // insert in right subtree
  {
    T.right = insert(T.right, newNode);
    if(height(T.right) - height(T.left) == 2)
      if(newNode.info > T.right.info) // right subtree of T.right
        T = rotateWithRightChild(T);
      else                            // left subtree of T.right
        T = doubleWithRightChild(T);
  }
  T.height = max(height(T.left), height(T.right)) + 1;
  return T;
}
```

Single rotation – left-left

- Insertion in left subtree of `k2.left` caused an imbalance at `k2`: need to rebalance from `k2` down.

```
AVLNode rotateWithLeftChild(AVLNode k2)
{
  AVLNode k1 = k2.left;
  k2.left = k1.right;
  k1.right = k2;
  k2.height = max(height(k2.left,
                  height(k2.right)) + 1;
  k1.height = max(height(k1.left),
                  k2.height) + 1;
  return k1;
}
```

- Note: right-right case is symmetric to above (left↔right).

Double rotation – right-left

- Insertion in right subtree of left child caused imbalance at `k3`: need to rebalance from `k3` down.

```
AVLNode doubleWithLeftChild(AVLNode k3)
{
  k3.left = rotateWithRightChild(k3.left);
  return rotateWithLeftChild(k3);
}
```

- Note: left-right case is symmetric to above (left↔right).