

Breadth-first traversal

- Top-down enumeration: starting at root, visit all nodes on each level in turn, from left to right
- Use an initially-empty queue, TQ, of type BinaryNode

```
BFT(BinaryNode T)
{
    TQ.enqueue(T);
    while(!TQ.isEmpty())
    {
        temp = TQ.dequeue();
        temp.displayNode();
        if(temp.left != null)
            TQ.enqueue(temp.left);
        if(temp.right != null)
            TQ.enqueue(temp.right);
    }
}
```

Binary Preorder Traversal

- Visit root, traverse left subtree, traverse right subtree

```
preOrderTraverse(node T)
{
    if(T == null)
        return;
    else
    {
        T.displayNode();
        preOrderTraverse(T.left);
        preOrderTraverse(T.right);
    }
}
```

Binary Postorder Traversal

- Traverse left subtree, traverse right subtree, visit root

```
postOrderTraverse(BinaryNode T)
{
    if(T == null)
        return;
    else
    {
        postOrderTraverse(T.left);
        postOrderTraverse(T.right);
        T.displayNode();
    }
}
```

Inorder Traversal

- Traverse left subtree, visit root, traverse right subtree

```
inOrderTraverse(BinaryNode T)
{
    if(T == null)
        return;
    else
    {
        inOrderTraverse(T.left);
        T.displayNode();
        inOrderTraverse(T.right);
    }
}
```

Height of a Tree

Idea: use a postorder traversal, since we first need the heights of the subtrees

```
int height(BinaryNode T)
{
    if(T == null)
        return -1;
    else
        return 1 +
            max(height(T.left),
                height(T.right));
}
```

Preorder Iterative Traversal

- Use an initially-empty stack, TS, of type BinaryNode

```
preOrderIterativeTraverse(BinaryNode T)
{
    while(T != null)
    {
        T.displayNode();
        /* after left subtree, need to go to right
           subtree: so save it on stack */
        if(T.right != null)
            TS.push(T.right);
        /* traverse left subtree if it exists */
        if(T.left != null)
            T = T.left;
        /* else go to next node in preorder, which is
           at top of stack */
        else if(!TS.isEmpty())
            T = TS.pop();
        else // traversal is finished
            T = null;
    }
}
```

Binary Search Trees – Search (Iterative version)

```
BinaryNode find(BinaryNode T, int key)
// find the node with the given key
{
    curr = T;
    while(curr.info != key)
    {
        if(key < curr.info)
            curr = curr.left;
        else
            curr = curr.right;
        if(curr == null) // not found
            return null;
    }
    return curr;
}
```

Binary Search Trees – findMin (Iterative version)

```
BinaryNode findMin(BinaryNode T)
// find smallest element
{
    if(T != null)
    {
        while(T.left != null)
            T = T.left;
    }
    return T;
}
```


Binary Search Trees – findMax (Recursive version)

```
BinaryNode findMax(BinaryNode T)
// find largest element
{
    if(T == null)
        return null;
    else if(T.right == null)
        return T;
    return findMax(T.right);
}
```

Binary Search Trees – Insert (iterative)

```
void insert(BinaryNode T, BinaryNode newNode)
// insert newNode into tree with root T
{
    if(T == null)
        T = newNode;
    else
    {
        curr = T;
        while(true)
        {
            parent = curr;
            if(newNode.info < curr.info)    // insert in left subtree
            {
                curr = curr.left;
                if(curr == null)            // insert here
                {
                    parent.left = newNode;
                    return;
                }
            }
            else                            // insert in right subtree
            {
                curr = curr.right;
                if(curr == null)            // insert here
                {
                    parent.right = newNode;
                    return;
                }
            }
        }
    }
}
}
}
}
}
```

Binary Search Trees – Delete

```
BinaryNode delete(int key)
// delete and return node with given key
{
    Search for node with given key;
    If no children
        just delete
    Else if no right child
        replace with left subtree
    Else if no left child
        replace with right subtree
    Else (2 children)
        replace with inorder successor
}
```

Representation of a Threaded Tree

```
class ThreadedNode
{
    <declarations for info stored in node, e.g.
    int info;>
    ThreadedNode left;
    ThreadedNode right;
    boolean lThread; // true if left is a thread
    boolean rThread; // true if right is a thread

    public void displayNode(){...}
    // display info stored in node
}
```

