Binary Trees

A binary tree has the following fundamental qualities:

- Each node has at most 2 children
- Every left node should be less than its parent and every right node should be greater than its parent.

Traversal of Trees

Breadth First

This is printing every node of certain level before moving onto the next level.

```
public void breadthFirst()
{
    BSTNode<T> p = root;
    Queue<BSDNode<T>> queue = new Queue<BSTNode<T>>();
    if (q != null)
    {
        queue.enqueue(p);
        while (!queue.isEmpty())
        {
            p = queue.dequeue();
            visit(p);      //any processing we wish to do
            if (p.left != null)
                queue.enqueue(p.left);
            if (p.right != null)
                queue.enqueue(p.right);
        }
    }
}
```

Depth First

Traversal goes as far as possible one way, until an end is found, and then backtacks to go down another path.

Pre-Order

```
protected void preorder(BSTNode<T> p)
{
    if (p != null)
    {
        visit(p);
        preorder(p.left);
        preorder(p.right);
    }
}
```

```
       [x]          [x]        [x]
     /   \        /   \      /   \
   [ ]   [ ]    [x]   [ ]   [x]   [x]
```

In-order

```
protected void inorder(BSTNode<T> p)
```

```
{
    if (p != null)
    {
        inorder(p.left);
        visit(p);
        inoder(p.right);
    }
}
```

```
      [ ]          [x]        [x]
     /   \        /   \      /   \
   [x]   [ ]    [x]   [ ]  [x]   [x]
```

Post-order

```
protected void postoder(BSTNode<T> p)
{
    if (p != null)
    {
        postorder(p.left);
        postorder(p.right);
        visit(p);
    }
}
```

```
      [ ]          [ ]        [x]
     /   \        /   \      /   \
   [x]   [ ]    [x]   [x]  [x]   [x]
```

Stack-Less Depth-First

Threaded Trees.

These are trees where the null right children point to the immediate node above them to the right.

Morris Traversal

This algorithm linearizes any tree, traverses it and then puts it back together.

Non-reversible

```
1. Initlialize p as root
2. While p is not NULL
    I If p does not have left child:
        a) Visit p
        b) Go to p.right
    II Else
        a) Make p the right child of the rightmost node in  p's left subtree.
        b) Go to p.left
```

Reversible

```
1. Initialize p as root
2. While p is not NULL
    I If p.left is NULL
        a) Visit p
        b) Go to p.right
    II Else
        a) tmp <- p.left, keep right unit conditions:
        b) If tmp.right == NULL
            i Make p the right child of tmp.
            ii. Go to p.left.
        c) Else if (tmp.right == p)
            i Visit p
            ii tmp.right <- NULL
            iii Go to p.right
```

Deletion from Trees

Deletion by Merging

This is a simple method of deletion where we maintain the fundamental rules of the binary tree.

```
     [D]                          [D]
    /   \         DELETE B       /   \
  [B]   [E]          ->        [A]   [E]
  / \                            \
[A]  [C]                          [C]
```

```
public void deleteByMerging(T el)
{
    BSTNode<T> tmp,node, p = root, prev = null;
    while (p != null && !p.el.equals(el))
    {
        prev = p;
        if (el.compareTo(p.el) < 0)
            p = p.left;
        else p = p.right;
    }
    node = p;
}
```

Inserting into Trees

```
public void insert(T el)
{
    BSTNode<T> p = root, prev = null;
    while (p != null)
    {
        prev = p;
        if (el.compareTp(p.el) < 0) //if element is smaller than cu
rrent
            p = p.left;
        else p = p.right;
    }
    if (root == null)
        root = new BSTNode<T>(el);
    else if (el.compareTo(prev.el) < 0)  //if element is smaller than current
        prev.left = new BSTNode<T>(el);
    else
        prev.right = new BSTNode<T>(el);
}
```

Regan Koopmans 2016