```
public class BST <T>
{
    private BSTNode<T> root, current;

    public BST()
    {
        root = current = null;
    }

    public boolean empty()
    {
        return root == null ? true: false;
    }

    public void traverse(Order ord)
    {
        switch (ord)
        {
            case preOrder:  preorder(root);
                            break;

            case inOrder:   inorder(root);
                            break;

            case postOrder: postorder(root);
                            break;
        }
    }

    private void preorder(BSTNode<T> p)
    {
        if (p != null)
        {
            System.out.println(p.key);
            preorder(p.left);
            preorder(p.right);
        }
    }
```

```
private void inorder(BSTNode<T> p)
{
    if (p != null)
    {
        inorder(p.left);
        System.out.println(p.key);
        inorder(p.right);
    }
}

private void postorder(BSTNode<T> p)
{
    if (p != null)
    {
        postorder(p.left);
        postorder(p.right);
        System.out.println(p.key);
    }
}

private BSTNode<T> findparent (BSTNode<T> p)
{
    LinkStack<BSTNode<T>> stack = new LinkStack<BSTNode<T>>();
    BSTNode<T> q = root;

    while (q.right != p && q.left != p)
    {
        if (q.right != null)
           stack.push(q.right);

        if (q.left != null)
            q = q.left;
        else
            q = stack.pop();
    }

    return q;
}

public T retrieve ()
{
    return current.data;
}
```

```java
public boolean findkey(int tkey)
{
    BSTNode<T> p,q;

    p = root; q = root;

    if (empty())
      return false;

    while (p != null)
    {
        q = p;

        if (p.key == tkey)
        {
            current = p;

            return true;
        }
        else if (tkey < p.key)
            p = p.left;
        else
            p = p.right;
    }

    current = q;

    return false;
}
```

```
public boolean insert (int k, T val)
{
    BSTNode<T> p, q = current;

    if (findkey(k))
    {
        current = q;
        return false;
    }

    p = new BSTNode<T>(k, val);

    if (empty())
    {
        root = current = p;

        return true;
    }
    else
    {
        if (k < current.key)
            current.left = p;
        else
            current.right = p;

        current = p;
        return true;
    }
}

public boolean remove_key (int tkey)
{
    Flag removed = new Flag(false);
    BSTNode<T> p;

    p = remove_aux(tkey, root, removed);
    current = root = p;

    return removed.get_value();
}
```

```
private BSTNode<T> remove_aux(int key, BSTNode<T> p, Flag flag)
{
    BSTNode<T> q, child = null;

    if (p == null)
        return null;

    if (key < p.key)
        p.left = remove_aux(key, p.left, flag);
    else if (key > p.key)
        p.right = remove_aux(key, p.right, flag);
    else
    {
        flag.set_value(true);

        if (p.left != null && p.right != null)
        {
            q = find_min(p.right);
            p.key = q.key;
            p.data = q.data;
            p.right = remove_aux(q.key, p.right, flag);
        }
        else
        {
            if (p.right == null)
                child = p.left;
            else if (p.left == null)
                child = p.right;

            return child;
        }
    }

    return p;
}

private BSTNode<T> find_min(BSTNode<T> p)
{
    if (p == null)
      return null;

    while (p.left != null)
        p = p.left;

    return p;
}
```

```
     public boolean update(int key, T data)
    {
        remove_key(current.key);
        return insert(key, data);
    }
////////////////////////////////////////////////////////
    public boolean equal(BST <T> t2)
    {
        return equal(root,t2.root);
    }

    private boolean equal(BSTNode <T> t1,BSTNode <T> t2)
    {
        if (t1 == null && t2 == null)
          return true;
        else if (t1 == null || t2 == null)
          return false;
        else if(t1.key != t2.key)
          return false;

        return equal(t1.left,t2.left) && equal(t1.right,t2.right);
    }

    public boolean isFull()
    {
        return isFull(root);
    }

    private boolean isFull(BSTNode <T> t)
    {
        return countNodes(t) == Math.pow(2,height(t)) - 1;
    }

    public boolean isDegenrate()
    {
        return isDegenrate(root);
    }

    private boolean isDegenrate(BSTNode <T> t)
    {
        return countLeaf(t) == 1;
    }
```

```
public boolean isComplete()
{
    return isComplete(root);
}

private boolean isComplete (BSTNode <T> t)
{
    if (t == null)
      return true;

    int leftHeight = height(t.left);
    int rightHeight = height(t.right);
    int diff = leftHeight - rightHeight;

    if (diff < 0 || diff > 1)
      return false;

    if (! isComplete(t.left))
      return false;

    return isComplete (t.right);
}

public boolean isDegenrate2()
{
 return isDegenrate2(root);
}

private boolean isDegenrate2(BSTNode <T> t)
{
    return countNodes(t) == height(t);
}

public boolean isBST()
{
    if (root == null)
      return true;

    LinkStack <T> s = new LinkStack<T>();
    copyBSTToStack(root,s);

    return isSorted(s);
}
```

```
private void copyBSTToStack(BSTNode <T> t,LinkStack<T> s)
{
    if (root != null)
    {
      copyBSTToStack(t.right,s);
      s.push(t.data);
      copyBSTToStack(t.left,s);
    }
}

public boolean isSorted(LinkStack <T> s)
{
    if (s.empty())
      return true;

    boolean sorted = true;

    T x1,x2;

    x1 = s.pop();

    while(! s.empty() && sorted)
    {
      x2 = s.pop();

      //if (x1.compareTo(x2) > 0)
           //sorted = false;

      x1 = x2;
    }

    return sorted;
}

public boolean isBSTNoStack()
{
    return isBSTNoStack(root);
}
```

```
private boolean isBSTNoStack(BSTNode <T> t)
{
    boolean bst = true;

    if (t != null)
    {
      if (t.left != null)
      {
           if (t.key < t.left.key)
                bst = false;

           bst = bst && isBSTNoStack(t.left);
      }

      if (t.right != null)
      {
           if (t.key > t.right.key)
                bst = false;

           bst = bst && isBSTNoStack(t.right);
      }
    }

    return bst;
}

private boolean isLeaf(BSTNode <T> t)
{
    if (t.left == null && t.right == null)
      return true;
    else
      return false;
}

public boolean isAVL()
{
    if(! isBST())
      return false;

    return AVL(root);
}
```

```
private boolean AVL(BSTNode <T> t)
{
    if (root == null)
      return true;
    else
    {
      if (Math.abs(height(t.left) - height(t.right)) > 1)
           return false;
      else
           return AVL(t.left) && AVL(t.right);
    }
}

public int countNodes()
{
 return countNodes(root);
}

private int countNodes(BSTNode <T> t)
{
    if (t == null)
      return 0;

    return 1 + countNodes(t.left) + countNodes(t.right);
}

public int totalNodes()
{
 return totalNodes(root);
}

private int totalNodes(BSTNode <T> t)
{
    if (t == null)
      return 0;

    return t.key + countNodes(t.left) + countNodes(t.right);
}

public int avg()
{
 if (root == null)
      return 0;
 else
      return totalNodes() / countNodes();
}
```

```
public int countParents()
{
 return countParents(root);
}

private int countParents(BSTNode <T> t)
{
    if (t == null || (t.left == null && t.right == null))
      return 0;

    return 1 + countParents(t.left) + countParents(t.right);
}

public int countLeaf()
{
 return countLeaf(root);
}

private int countLeaf(BSTNode <T> t)
{
    if (t == null)
      return 0;
    else if (t.left == null && t.right == null)
      return 1;

    return countLeaf(t.left) + countLeaf(t.right);
}

public int countOneChild()
{
 return countOneChild(root);
}

private int countOneChild(BSTNode <T> t)
{
    if (t == null)
      return 0;
    else if ((t.left == null && t.right != null)
           || (t.left != null && t.right == null))
      return 1 + countOneChild(t.left) + countOneChild(t.right);

    return countOneChild(t.left) + countOneChild(t.right);
}
```

```
    public int height()
    {
     return height(root);
    }

    private int height(BSTNode <T> t)
    {
        if (t == null)
          return 0;

        return 1 + Math.max(height(t.left),height(t.right));
    }

    public int countLevel(int level)
    {
        return countLevel(root,0,level);
    }

    private int countLevel(BSTNode <T> t,int l,int level)
    {
        if (t == null)
          return 0;

        l++;

        if(l == level)
          return 1 + countLevel(t.left,l,level) +
countLevel(t.right,l,level);
        else
          return countLevel(t.left,l,level) +
countLevel(t.right,l,level);
    }

    public int width()
    {
     return width(root);
    }
```

```
private int width(BSTNode <T> t)
{
 if (root == null)
      return 0;

 int leftD  = width(t.left);
 int rightD = width(t.right);
 int rootD  = width(t.left) + height(t.right) + 1;

 return Math.max(rootD, Math.max(leftD, rightD));
}

public int leafCount()
{
    return leafCount(root);
}

private int leafCount (BSTNode <T> t)
{
    if (t == null)
      return 0;
    else if(isLeaf(t))
      return 1;
    else
      return leafCount(t.left) + leafCount(t.right);
}

public int noneLeafCount()
{
    return noneLeafCount(root);
}

private int noneLeafCount (BSTNode <T> t)
{
    if (t == null)
      return 0;
    else if(! isLeaf(t))
      return 1 + noneLeafCount(t.left) + noneLeafCount(t.right);
    else
      return noneLeafCount(t.left) + noneLeafCount(t.right);
}
```

```
public BST copyBST()
{
    if (root == null)
      return null;

    BST t = new BST();
    copy(root,t);

    return t;
}

private void copy(BSTNode <T> t1,BST<T> t2)
{
    if (t1 != null)
    {
      t2.insert(t1.key,t1.data);
      copy(t1.left,t2);
      copy(t1.right,t2);
    }
}

public BST<T> reverseBST()
{
    if (root == null)
      return null;

    BST<T> t = new BST<T>();
    reverse(root,t);

    return t;
}

private void reverse(BSTNode <T> t1,BST<T> t2)
{
    if (t1 != null)
    {
      t2.root = t2.insertReverse(t2.root,t1.key,t1.data);
      reverse(t1.left,t2);
      reverse(t1.right,t2);
    }
}
```

```
private BSTNode <T> insertReverse(BSTNode <T> t,int key,T data)
{
    if (t == null)
    {
      t = new BSTNode<T>(key,null,null);
      t.data = data;
    }
    else if (key > t.key)
            t.left = insertReverse(t.left,key,data);
    else if (key < t.key)
            t.right = insertReverse(t.right,key,data);
    else
            System.out.println("Duplicates not allowed");

    return t;
}

public void mirror()
{
    mirror(root);
}

private void mirror(BSTNode <T> t)
{
    if (t != null)
    {
      mirror(t.left);
      mirror(t.right);

      BSTNode <T> temp = t.left;
      t.left = t.right;
      t.right = temp;
    }
}
```

```
public boolean findKey(int key)
{
 return findKey(root,key);
}

private boolean findKey(BSTNode <T> t,int k)
{
    if (t == null)
      return false;
    else if(k > t.key)
      return findKey(t.right,k);
    else if (k < t.key)
      return findKey(t.left,k);
    else
      return true;
}

private boolean findKey3(BSTNode<T> b,int tkey)
{
    if (b.key == tkey)
    {
      current = b;
      return true;
    }
    else if (tkey < b.key)
    {
      if (b.left != null)
           return findKey3(b.left,tkey);
      else
      {
           current = b;
           return false;
      }
    }
    else
    {
      if (b.right != null)
           return findKey3(b.right,tkey);
      else
      {
           current = b;
           return false;
      }
    }
}
```

```
    public boolean findkey3(int tkey)
    {
        return findKey3(root,tkey);
    }

    public BSTNode <T> findParent(int key)
    {
        return findParent(root,key);
    }

    private BSTNode <T> findParent(BSTNode <T> t,int k)
    {
        if (t == null || k == t.key)
          return null;
        else if(t.left != null && t.left.key == k)
          return t;
        else if(t.right != null && t.right.key == k)
          return t;
        else if (k > t.key)
          return findParent(t.right,k);
        else
          return findParent(t.left,k);
    }

    public void findAllParents()
    {
        findAllParents(root);
    }

    private void findAllParents(BSTNode <T> t)
    {
        if (t != null)
        {
          BSTNode <T> s = findParent(root,t.key);

          if (s != null)
               System.out.println("Parent of " + t.key + " : " +
s.key);
          else
               System.out.println("No Parent or not found " + t.key);

          findAllParents(t.left);
          findAllParents(t.right);
        }
    }
```

```
public T findMax()
{
    return findMax(root);
}

private T findMax(BSTNode <T> t)
{
    while(t.right != null)
      t = t.right;

    return t.data;
}

public T findMin()
{
    return findMin(root);
}

private T findMin(BSTNode <T> t)
{
    while(t.left != null)
      t = t.left;

    return t.data;
}

public T findMinRec()
{
    return findMinRec(root);
}

private T findMinRec(BSTNode <T> t)
{
    if (t == null)
      return null;
    else if (t.left != null)
      return findMinRec(t.left);
    else
      return t.data;
}

public T findSuccessor(int tkey)
{
    findkey(tkey);
    return findSuccessor(current);
}
```

```
private T findSuccessor(BSTNode <T> t)
{
    return findMin(t.right);
}

public T findPredessor(int tkey)
{
    findkey(tkey);

    return findPredessor(current);
}

private T findPredessor(BSTNode <T> t)
{
    return findMax(t.left);
}

public BSTNode <T> findSmallestKth(int k)
{
    return findKthSmallest(root,k);
}

private BSTNode <T> findKthSmallest(BSTNode <T> root, int k)
{
     if (root == null)
       return null;  // can't find anything, empty

     int numLeft = countNodes(root.left);

     if (numLeft + 1 == k) // current node
        return root;
     else if (numLeft >= k) // in left subtree
        return findKthSmallest(root.left, k);
     else
        return findKthSmallest(root.right, k - (numLeft + 1));
 }

 public BSTNode <T> findLargestKth(int k)
 {
  return findKthLargest(root,k);
 }
```

```
private BSTNode <T> findKthLargest(BSTNode <T> root, int k)
{
    if (root == null)
        return null;  // can't find anything, empty

    int numRight = countNodes(root.right);

    if (numRight + 1 == k) // current node
       return root;
    else if (numRight >= k) // in right subtree
       return findKthLargest(root.right, k);
    else
       return findKthLargest(root.left, k - (numRight + 1));
}

private void inOrderNoRecursive(BSTNode<T> root)
{
    LinkStack <BSTNode <T>> s = new LinkStack<BSTNode <T>>();

    while(root != null || ! s.empty())
    {
        if (root == null)
        {
            root = s.pop();
            System.out.println(root.data);
            root = root.right;
        }

        if (root != null)
        {
            s.push(root);
            root = root.left;
        }
    }
}
```

```
private void preOrderNoRecursive(BTNode <T> root)
{
    LinkStack <BTNode <T>> s = new LinkStack<BTNode <T>>();

    while (root != null || ! s.empty())
    {
      if (root == null)
           root = s.pop();

      System.out.println(root.data);

      if (root.right != null)
           s.push(root.right);

      root = root.left;
     }
}
```

```
private void postOrderNoRecursive(BTNode <T> root)
{
    LinkStack <BTNode <T>> s = new LinkStack<BTNode <T>>();

    while (root != null || ! s.empty())
    {
        if (root == null)
        {
            BTNode <T> temp = s.pop();
            s.push(temp);

            while (! s.empty() && root == temp.right)
            {
                root = s.pop();
                System.out.println(root.data);

                if (! s.empty())
                {
                    temp = s.pop();
                    s.push(temp);
                }
            }

            if (s.empty())
                root = null;
            else
            {
                root = s.pop();
                s.push(root);
                root = root.right;
            }
        }

        if (root != null)
        {
            s.push(root);
            root = root.left;
        }
    }
}
```

```
private void printByLevel(BSTNode <T> t)
{
    if (t != null)
    {
      LinkQueue <BSTNode <T>> q = new LinkQueue<BSTNode <T>>();

      q.enqueue(t);

      while(q.length() != 0)
      {
           t = (BSTNode<T>) q.serve();
           System.out.println(t.data);

           if (t.left != null)
                q.enqueue(t.left);

           if (t.right != null)
                q.enqueue(root.right);
      }
    }
}

private void printLevelRight(BSTNode <T> t)
{
    if (root != null)
    {
      LinkQueue <BSTNode <T>> q = new LinkQueue<BSTNode <T>>();
      q.enqueue(t);

      while(q.length() != 0)
      {
           root = (BSTNode<T>) q.serve();
           System.out.println(t.data);

           if (t.right != null)
                q.enqueue(t.right);

           if (t.left != null)
                q.enqueue(t.left);
      }
    }
}

public void printLevel(int level)
{
    printLevel(root,0,level);
}
```

```
    private void printLevel(BSTNode <T> t, int l, int level)
    {
        if (t !=  null)
        {
          l++;

          if(l == level)
               System.out.print(t.data + " ");
          else if (l < level)
          {
               printLevel(t.left,l,level);
               printLevel(t.right,l,level);
          }
        }
    }

    public void printLevelLines()
    {
        for (int i = 1 ; i <= height(root) ; i++)
        {
          printLevel(i);
          System.out.println();
        }
    }

    private void printDescendents(BSTNode <T> t)
    {
        if (root != null)
        {
                printDescendents(t.left);

                System.out.println("Number of descendebts of node "
                                        + t.key + " : " +
(countNodes(t) - 1));

                printDescendents(t.right);
        }
    }
```

```
public String pathBST(int k)
{
    boolean found = findkey(k);
    String path = null;

    if(found)
    {
      path = "";

      BSTNode <T> p = current;

      while(p != root)
      {
           path += p.data + " ";
           p = findparent(p);
      }

      path += p.data + " ";
    }

    return path + "\n";
}

public String pathBSTStartRoot(int k)
{
    boolean found = findkey(k);
    String path = null;

    if(found)
    {
      path = "";
      BSTNode <T> p = root;

      while(p != current)
      {
           path += p.data + " ";

           if (k < p.key)
                p = p.left;
           else
                p = p.right;
      }
      path += p.data + " ";
    }

    return path + "\n";
}
```

```
    public void printPath(int k)
    {
        if (! findkey(k))
        {
          System.out.println("Not found");
          return;
        }

        if(! empty())
        {
          BSTNode <T> p = root;

          while(p.key != k)
          {
               System.out.print(p.data + " ");

               if (k < p.key)
                    p = p.left;
               else
                    p = p.right;
          }

          System.out.println(p.data + " ");
        }
    }
}
```