```
#include <iostream>
#include <cassert>

using namespace std;

template<class T>
class Queue{
public:
  Queue(int aSize):mSize(aSize),mHead(0),mTail(0){mStorage=new T[mSize];}
  ~Queue(){delete[] mStorage;}
  void Push(T aValue)
  {
    if (mTail==mSize) mTail=0;//make circular
    mStorage[mTail]=aValue;
    mTail+=1;
  }
  T Pop()
  {
    assert(mTail!=mHead);
    T value=mStorage[mHead];
    mHead+=1;
    if (mHead==mSize) mHead=0;//make circular
    return value;
  }
private:
  int mSize;
  int mHead;//points to current head element
  int mTail;//points to element after last inserted (first free element)
  T* mStorage;
};

template<class T>
class Node{
public:
  Node(T aKey):mKey(aKey),mpLeft(0),mpRight(0){}
  Node* Clone(){return new Node(mKey);}
public:
  T mKey;
  Node* mpLeft;
  Node* mpRight;
};

template<class T>
class Tree{
public:
  Tree();
  ~Tree();
  Tree(const Tree& aTreeToCopy);
  Tree& operator=(const Tree& aTreeToCopy);

  int Size()const;
  void Input(const T aValue);
  void Output(ostream& out)const;
  void OutputPostOrder(ostream& out)const;
  bool operator==(const Tree& aTree)const;
private:
  void mCopyConstructor(Node<T>* apNodeCurrent, Node<T>* apNodeToCopy);
  void mDistructor(Node<T>* apNode);
  int mSize(Node<T>* apNode)const;
  void mOutputPostOrder(Node<T>* apNode,ostream& out)const;
  bool mTestForEquality(const Node<T>* apNodeX, const Node<T>* apNodeY)const;
private:
  Node<T>* mpRoot;
};

template<class T>
Tree<T>::Tree():mpRoot(0){}

template<class T>
Tree<T>::Tree(const Tree<T>& aTreeToCopy)
{
  *this=aTreeToCopy;
}

template<class T>
Tree<T>& Tree<T>::operator=(const Tree<T>& aTreeToCopy)
{
  if (aTreeToCopy.mpRoot!=0)
    {
      mpRoot=aTreeToCopy.mpRoot->Clone();
      mCopyConstructor(mpRoot,aTreeToCopy.mpRoot);
    }
  else {}
  return *this;
}

template<class T>
void Tree<T>::mCopyConstructor(Node<T>* apNodeCurrent, Node<T>* apNodeToCopy)
{
  if (apNodeToCopy->mpLeft!=0)
    {
      apNodeCurrent->mpLeft=apNodeToCopy->mpLeft->Clone();
      mCopyConstructor(apNodeCurrent->mpLeft, apNodeToCopy->mpLeft);
    }
  if (apNodeToCopy->mpRight!=0)
    {
      apNodeCurrent->mpRight=apNodeToCopy->mpRight->Clone();
      mCopyConstructor(apNodeCurrent->mpRight, apNodeToCopy->mpRight);
    }
}

template<class T>
Tree<T>::~Tree()
{
  mDistructor(mpRoot);
}

template<class T>
void Tree<T>::mDistructor(Node<T>* apNode)
{
  if (apNode !=0)
    {
      mDistructor(apNode->mpLeft);
      mDistructor(apNode->mpRight);
      delete apNode;
    }
}

template<class T>
int Tree<T>::Size()const
{
  return mSize(mpRoot);
}

template<class T>
int Tree<T>::mSize(Node<T>* apNode)const
{
  if (apNode==0) return 0;
  return 1+mSize(apNode->mpLeft)+mSize(apNode->mpRight);
}

template<class T>
void Tree<T>::Input(const T aValue)
{
  int  size=Size();
  if (size==0) mpRoot=new Node<T>(aValue);
  else {
    Queue<Node<T>*> local_queue(2*size);
    Node<T>* cursor=mpRoot;
    //iterate until we find a node with either a left or right null child
    while (cursor->mpLeft!=0 && cursor->mpRight!=0)
      {
    local_queue.Push(cursor->mpLeft);
    local_queue.Push(cursor->mpRight);
    cursor=local_queue.Pop();
      }
    //insert substituting null child
    if (cursor->mpLeft==0) cursor->mpLeft=new Node<T>(aValue);
    else cursor->mpRight=new Node<T>(aValue);
  }
}

template<class T>
void Tree<T>::Output(ostream& out)const
{
  Queue<Node<T>*> local_queue(2*Size());
  Node<T>* cursor=mpRoot;
  while (cursor!=0)
    {
      out<<cursor->mKey<<" ";
      local_queue.Push(cursor->mpLeft);
      local_queue.Push(cursor->mpRight);
      cursor=local_queue.Pop();
    }
}

template<class T>
void Tree<T>::OutputPostOrder(ostream& out)const
{
  mOutputPostOrder(mpRoot,out);
}
```

```
template<class T>
void Tree<T>::mOutputPostOrder(Node<T>* apNode, ostream& out)const
{
  if (apNode!=0)
    {
      mOutputPostOrder(apNode->mpLeft, out);
      mOutputPostOrder(apNode->mpRight, out);
      out<<apNode->mKey<<" ";
    }
}

template<class T>
bool Tree<T>::operator==(const Tree& aT)const
{
  return mTestForEquality(mpRoot,aT.mpRoot);
}

template<class T>
bool Tree<T>::mTestForEquality(const Node<T>* apNodeX, const Node<T>* apNodeY)const
{
  if (apNodeX==0 && apNodeY!=0) return false;
  else if (apNodeX!=0 && apNodeY==0) return false;
  else if (apNodeX==0 && apNodeY==0) return true;
  else return apNodeX->mKey==apNodeY->mKey && mTestForEquality(apNodeX->mpLeft,apNodeY->
mpLeft) && mTestForEquality(apNodeX->mpRight,apNodeY->mpRight);
}

template<class T>
ostream& operator<<(ostream& out, const Tree<T>& aTree)
{
  aTree.Output(out);
  return out;
}

int main(){

  const int DIM=10;

  Tree<char> tx;
  for (int i=0;i<DIM;i++)
    tx.Input('a'+i);
  cout<<tx<<endl;
  tx.OutputPostOrder(cout);
  cout<<endl;

  Tree<char> ty;
  ty=tx;
  cout<<ty<<endl;

  Tree<char> tz;
  for (int i=9;i>=0;i--)
    tz.Input('a'+i);
  cout<<tz<<endl;

  cout<<"I due alberi ["<<tx<<"] e ["<<ty<<"] sono ...";
  if (tx==ty) cout<<"uguali"<<endl;
  else cout<<"diversi"<<endl;

  cout<<"I due alberi ["<<tx<<"] e ["<<tz<<"] sono ...";
  if (tx==tz) cout<<"uguali"<<endl;
  else cout<<"diversi"<<endl;

  return 0;
}
```