#include <iostream>

#include <strstream>

#include <queue>

using namespace std;

template<class T>

struct node {

 T inf;

 node \* Left;

 node \* Right;

};

template <class T>

class tree

{

public:

 tree();

 ~tree();

 tree(tree const&);

 tree& operator=(tree const&);

 bool empty() const;

 T RootTree() const;

 tree LeftTree() const;

 tree RightTree() const;

 void Create3(T, tree<T>, tree<T>);

 void print() const

 {

 pr(root);

 cout << endl;

 }

 void Create(istream & is = cin)

 {

 CreateTree(root,is);

 }

 private:

 node<T> \*root;

 void DeleteTree(node<T>\* &) const;

 void Copy(node<T> \* &, node<T>\* const&) const;

 void CopyTree(tree<T> const&);

 void pr(const node<T> \*) const;

 void CreateTree(node<T> \* &, istream & is = cin) const;

};

template <class T>

tree<T>::tree()

{

 root = NULL;

}

template <class T>

tree<T>::~tree()

{

 DeleteTree(root);

}

template <class T>

tree<T>::tree(tree<T> const& r)

{

 CopyTree(r);

}

template <class T>

tree<T>& tree<T>::operator=(tree<T> const& r)

{

 if (this != &r)

 {

 DeleteTree(root);

 CopyTree(r);

 }

 return \*this;

}

template <class T>

void tree<T>::DeleteTree(node<T>\* &p)const

{

 if (p)

 {

 DeleteTree(p->Left);

 DeleteTree(p->Right);

 delete p;

 p = NULL;

 }

}

template <class T>

void tree<T>::CopyTree(tree<T> const& r)

{

 Copy(root, r.root);

}

template <class T>

void tree<T>::Copy(node<T> \* & pos, node<T>\* const & r) const

{

 pos = NULL;

 if (r)

 {

 pos = new node<T>;

 pos->inf = r->inf;

 Copy(pos->Left, r->Left);

 Copy(pos->Right, r->Right);

 }

}

template <class T>

bool tree<T>::empty()const

{

 return root == NULL;

}

template <class T>

T tree<T>::RootTree()const

{

 return root->inf;

}

template <class T>

tree<T> tree<T>::LeftTree() const

{

 tree<T> t;

 Copy(t.root, root->Left);

 return t;

}

template <class T>

tree<T> tree<T>::RightTree() const

{

 tree<T> t;

 Copy(t.root, root->Right);

 return t;

}

template <class T>

void tree<T>::pr(const node<T>\*p) const

{

 if (p)

 {

 pr(p->Left);

 cout << p->inf << " " ;

 pr(p->Right);

 }

}

template <class T>

void tree<T>::Create3(T x, tree<T> l, tree<T> r)

{

 root = new node<T>;

 root->inf = x;

 Copy(root->Left, l.root);

 Copy(root->Right, r.root);

}

template <class T>

void tree<T>::CreateTree(node<T> \* & pos, istream & is) const

{

 T x; char c;

 cout << "root: ";

 is >> x;

 pos = new node<T>;

 pos->inf = x;

 pos->Left = NULL;

 pos->Right = NULL;

 cout << "Left Tree of: " << x << " y/n? ";

 is >> c;

 if (c == 'y') CreateTree(pos->Left, is);

 cout << "Right Tree of: " << x << " y/n? ";

 is >> c;

 if (c == 'y') CreateTree(pos->Right, is);

}

template <class T>

struct elem\_link1

{

 T inf;

 elem\_link1<T> \*link;

};

template <class T>

class LList

{

public:

 LList();

 ~LList();

 LList(LList const &);

 LList& operator=(LList const &);

 bool empty() const;

 void IterStart(elem\_link1<T>\* = NULL);

 elem\_link1<T>\* Iter();

 void ToEnd(T const &);

 void InsertAfter(elem\_link1<T> \*, T const &);

 void InsertBefore(elem\_link1<T> \*, T const &);

 void DeleteElem(elem\_link1<T> \*, T &);

 int DeleteAfter(elem\_link1<T> \*, T &);

 int DeleteBefore(elem\_link1<T> \*, T &);

 int length();

 void concat(LList const &);

 void reverse();

 void print();

private:

 elem\_link1<T> \*Start,

 \*End,

 \*Current;

 void DeleteList();

 void CopyList(LList<T> const &);

};

template <class T>

LList<T>::LList()

{

 Start = End = Current = NULL;

}

template <class T>

LList<T>::~LList()

{

 DeleteList();

}

template <class T>

LList<T>::LList(LList<T> const &r)

{

 CopyList(r);

}

template <class T>

LList<T>& LList<T>::operator=(LList<T> const &r)

{

 if (this != &r)

 {

 DeleteList();

 CopyList(r);

 }

 return \*this;

}

template <class T>

void LList<T>::DeleteList()

{

 if (Start)

 {

 elem\_link1<T> \*p;

 while (Start)

 {

 p = Start;

 Start = Start->link;

 delete p;

 }

 End = NULL;

 }

}

template <class T>

void LList<T>::CopyList(LList<T> const &r)

{

 Start = End = NULL;

 elem\_link1<T> \*p = r.Start;

 while (p)

 {

 ToEnd(p->inf);

 p = p->link;

 }

}

template <class T>

bool LList<T>::empty() const

{

 return Start == NULL;

}

template <class T>

void LList<T>::IterStart(elem\_link1<T> \*p)

{

 if (p)

 Current = p;

 else

 Current = Start;

}

template <class T>

elem\_link1<T>\* LList<T>::Iter()

{

 elem\_link1<T> \*p = Current;

 if (Current)

 Current = Current->link;

 return p;

}

template <class T>

int LList<T>::length()

{

 int n = 0;

 IterStart();

 while (Iter())

 n++;

 return n;

}

template <class T>

void LList<T>::ToEnd(T const &x)

{

 Current = End;

 End = new elem\_link1<T>;

 End->inf = x;

 End->link = NULL;

 if (Current)

 Current->link = End;

 else

 Start = End;

}

template <class T>

void LList<T>::InsertAfter(elem\_link1<T> \*p, T const &x)

{

 elem\_link1<T> \*q = new elem\_link1<T>;

 q->inf = x;

 q->link = p->link;

 if (p == End)

 End = q;

 p->link = q;

}

template <class T>

void LList<T>::InsertBefore(elem\_link1<T> \*p, T const &x)

{

 elem\_link1<T> \*q = new elem\_link1<T>;

 \*q = \*p;

 p->inf = x;

 p->link = q;

 if (End == p)

 End = q;

}

template <class T>

int LList<T>::DeleteAfter(elem\_link1<T> \*p, T &x)

{

 if (p == End)

 return 0;

 elem\_link1<T> \*q = p->link;

 x = q->inf;

 p->link = q->link;

 if (End == q)

 End = p;

 delete q;

 return 1;

}

template <class T>

void LList<T>::DeleteElem(elem\_link1<T> \*p, T &x)

{

 if (p == Start)

 {

 x = p->inf;

 if (Start == End)

 {

 Start = NULL;

 End = NULL;

 delete p;

 }

 else

 {

 Start = Start->link;

 delete p;

 }

 }

 else

 {

 elem\_link1<T> \*q = Start;

 while (q->link != p)

 q = q->link;

 DeleteAfter(q,x);

 }

}

template <class T>

int LList<T>::DeleteBefore(elem\_link1<T> \*p, T &x)

{

 if (p == Start)

 return 0;

 elem\_link1<T> \*q = Start;

 while (q->link != p)

 q = q->link;

 DeleteElem(q,x);

 return 1;

}

template <class T>

void LList<T>::concat(LList<T> const &L)

{

 elem\_link1<T> \*p = L.Start;

 while (p)

 {

 ToEnd(p->inf);

 p = p->link;

 }

}

template <class T>

void LList<T>::reverse()

{

 LList<T> L;

 elem\_link1<T> \*p = Start;

 if (p)

 {

 L.ToEnd(p->inf);

 p = p->link;

 while (p)

 {

 L.InsertBefore(L.Start, p->inf);

 p = p->link;

 }

 }

 \*this = L;

}

template <class T>

void LList<T>::print()

{

 elem\_link1<T> \*p = Start;

 while (p)

 {

 cout<< p->inf <<" ";

 p = p->link;

 }

 cout<<"\n";

}

template<typename T>

void printHelper(tree<T> t, ostream & os, size\_t level)

{

 if(t.empty()) return;

 for(size\_t k=0; k<level; k++) os << '\t';

 os << t.RootTree() << endl;

 if( t.LeftTree().empty() && !t.RightTree().empty())

 {

 for(size\_t k=0; k<level+1; k++) os << '\t';

 os << '@' << endl;

 }

 printHelper(t.LeftTree(),os,level+1);

 printHelper(t.RightTree(),os,level+1);

}

template<typename T>

ostream & operator<<( ostream & os, tree<T> t)

{

 printHelper(t,os,0);

 return os;

}

// Задача 1: реализирайте функция, която отпечатва всички

// елементи на дърво, които са по-малки от родителя си

void printAllLessThanParentMembersHelper(tree<int> t, int parent)

{

 if(t.empty())

 return;

 if(t.RootTree() < parent)

 cout << t.RootTree() << endl;

 printAllLessThanParentMembersHelper(t.LeftTree(), t.RootTree());

 printAllLessThanParentMembersHelper(t.RightTree(), t.RootTree());

}

void printAllLessThanParentMembers(tree<int> t)

{

 if(t.empty())

 return;

 printAllLessThanParentMembersHelper(t.LeftTree(), t.RootTree());

 printAllLessThanParentMembersHelper(t.RightTree(), t.RootTree());

}

// Задача 2 : отпечатайте всички думи, които може да се

// "прочетат", започвайки обхождане без връщане на дървото

// от корена му и стигайки до някое от листата.

void printAllWords(tree<int> t, queue<int> q = queue<int>())

{

 if(t.empty())

 return;

 q.push(t.RootTree());

 //if(t.LeftTree().empty() && t.RightTree().empty())

 queue<int> q1 = q;

 {

 while(!q1.empty())

 {

 cout << q1.front();

 q1.pop();

 }

 cout << endl;

 //return;

 }

 printAllWords(t.LeftTree(),q);

 printAllWords(t.RightTree(),q);

}

int main()

{

 char tree1str[] = "1 y 2 n n y 3 y 0 n n y 5 n n \n";

 tree<int> tree1;

 istrstream treestream(tree1str);

 tree1.Create(treestream);

 cout << endl;

 //tree1.print();

 cout << tree1;

 cout << endl;

 cout << " ------------------------------------- " << endl;

 printAllLessThanParentMembers(tree1);

 cout << " ------------------------------------- " << endl;

 printAllWords(tree1);

 cout << " ------------------------------------- " << endl;

 // queue<int> q;

 // genall(tree1,q);

 // cout << endl;

 // allGtParent(tree1).print();

 return 0;

}