Threaded binary search trees

Lab 4, Grade Option “A”

Rutger Yager

Dr. Burris COSC 3319.01

**OUTPUT**

The following output was generated with the command: binarysearchtreetest > Lab4out.txt

Lab4out.txt

Inserted [Furbee ,295-1492 ]

Inserted [Lee ,291-1864 ]

Inserted [Guo ,295-1601 ]

Inserted [Hall ,293-6122 ]

Inserted [Carazo ,295-1882 ]

Inserted [Rios ,291-7890 ]

Inserted [Sido ,294-8075 ]

Inserted [Furbee ,584-3622 ]

Patil # Iterative: Not in tree.

Patil # Recrusive: Not in tree.

Carazo # Iterative: 295-1882

Carazo # Recursive: 295-1882

Iterative InOrder Traversal From 'Guo':

[Guo ,295-1601 ]

[Hall ,293-6122 ]

[Lee ,291-1864 ]

[Rios ,291-7890 ]

[Sido ,294-8075 ]

[Carazo ,295-1882 ]

[Furbee ,295-1492 ]

[Furbee ,584-3622 ]

Inserted [Sodmann ,294-1568 ]

Inserted [Carazo ,294-1882 ]

Inserted [Montes ,295-6622 ]

Iterative InOrder Traversal:

[Carazo ,295-1882 ]

[Carazo ,294-1882 ]

[Furbee ,295-1492 ]

[Furbee ,584-3622 ]

[Guo ,295-1601 ]

[Hall ,293-6122 ]

[Lee ,291-1864 ]

[Montes ,295-6622 ]

[Rios ,291-7890 ]

[Sido ,294-8075 ]

[Sodmann ,294-1568 ]

Deleting entry: [Furbee ,295-1492 ]

Deleting entry: [Hall ,293-6122 ]

Deleting entry: [Furbee ,584-3622 ]

Deleting entry: [Sodmann ,294-1568 ]

Inserted [Caldwell ,294-1666 ]

Inserted [Houck ,295-1882 ]

Iterative InOrder Traversal:

[Caldwell ,294-1666 ]

[Carazo ,295-1882 ]

[Carazo ,294-1882 ]

[Guo ,295-1601 ]

[Houck ,295-1882 ]

[Lee ,291-1864 ]

[Montes ,295-6622 ]

[Rios ,291-7890 ]

[Sido ,294-8075 ]

Recursive Reverse InOrder Traversal:

[Sido ,294-8075 ]

[Rios ,291-7890 ]

[Montes ,295-6622 ]

[Lee ,291-1864 ]

[Houck ,295-1882 ]

[Guo ,295-1601 ]

[Carazo ,294-1882 ]

[Carazo ,295-1882 ]

[Caldwell ,294-1666 ]

Iterative PreOrder Traversal Using Threads:

[Guo ,295-1601 ]

[Carazo ,295-1882 ]

[Caldwell ,294-1666 ]

[Carazo ,294-1882 ]

[Lee ,291-1864 ]

[Houck ,295-1882 ]

[Rios ,291-7890 ]

[Montes ,295-6622 ]

[Sido ,294-8075 ]

Iterative PostOrder Traversal Using Threads:

[Caldwell ,294-1666 ]

[Carazo ,294-1882 ]

[Carazo ,295-1882 ]

[Houck ,295-1882 ]

[Montes ,295-6622 ]

[Sido ,294-8075 ]

[Rios ,291-7890 ]

[Lee ,291-1864 ]

[Guo ,295-1601 ]

Recursive PostOrder Traversal:

[Caldwell ,294-1666 ]

[Carazo ,294-1882 ]

[Carazo ,295-1882 ]

[Houck ,295-1882 ]

[Montes ,295-6622 ]

[Sido ,294-8075 ]

[Rios ,291-7890 ]

[Lee ,291-1864 ]

[Guo ,295-1601 ]

**SOURCE CODE**

BinarySearchTreeTest.adb

with Ada.Text\_IO;

use Ada.Text\_IO;

with Ada.Strings.Fixed;

use Ada.Strings.Fixed;

with Ada.Strings.Maps.Constants;

use Ada.Strings.Maps.Constants;

with BinarySearchTree;

procedure BinarySearchTreeTest is

type String10 is new String

(1 .. 10);

type Customer is

record

Name : String10;

PhoneNumber : String10;

end record;

function "<" (

TheKey : in String10;

ARecord : in Customer)

return Boolean is

begin

return Translate(String(TheKey), Ada.Strings.Maps.Constants.Lower\_Case\_Map) < Translate(String(ARecord.Name), Ada.Strings.Maps.Constants.Lower\_Case\_Map);

end;

function ">" (

TheKey : in String10;

ARecord : in Customer)

return Boolean is

begin

return Translate(String(TheKey), Ada.Strings.Maps.Constants.Lower\_Case\_Map) > Translate(String(ARecord.Name), Ada.Strings.Maps.Constants.Lower\_Case\_Map);

end;

function "=" (

TheKey : in String10;

ARecord : in Customer)

return Boolean is

begin

return Translate(String(TheKey), Ada.Strings.Maps.Constants.Lower\_Case\_Map) = Translate(String(ARecord.Name), Ada.Strings.Maps.Constants.Lower\_Case\_Map);

end;

function GetName (

ARecord : in Customer)

return String10 is

begin

return ARecord.Name;

end;

function GetPhone (

ARecord : in Customer)

return String10 is

begin

return ARecord.PhoneNumber;

end;

function NewCustomer (

NewName : in String10;

NewPhone : in String10)

return Customer is

begin

return (

Name => NewName,

PhoneNumber => NewPhone);

end;

procedure PutString10 (

Input : in String10) is

begin

Put(String(Input));

end PutString10;

package MySearchTree is new BinarySearchTree(String10, Customer, "<", ">", "=", GetName, GetPhone, NewCustomer, PutString10);

use MySearchTree;

procedure Insert (

Tree : in out BinarySearchTreePoint;

Name : in String10;

Phone : in String10) is

begin

MySearchTree.InsertBinarySearchTree(Tree, Name, Phone);

Put("Inserted [");

Put(String(Name));

Put(",");

Put(String(Phone));

Put\_Line("]");

end Insert;

TestTree : MySearchTree.BinarySearchTreePoint;

TestNode : MySearchTree.BinarySearchTreePoint;

begin

--C OPTION STEPS

--1

MySearchTree.MakeEmptyBinarySearchTree(TestTree, "Head ", "Node ");

Insert(TestTree, "Furbee ", "295-1492 ");

Insert(TestTree, "Lee ", "291-1864 ");

Insert(TestTree, "Guo ", "295-1601 ");

Insert(TestTree, "Hall ", "293-6122 ");

Insert(TestTree, "Carazo ", "295-1882 ");

Insert(TestTree, "Rios ", "291-7890 ");

Insert(TestTree, "Sido ", "294-8075 ");

Insert(TestTree, "Furbee ", "584-3622 ");

--2

New\_Line;

MySearchTree.FindCustomerIterative(TestTree, "Patil ", TestNode);

Put("Patil # Iterative: ");

if not IsNull(TestNode) then

Put\_Line(String(MySearchTree.CustomerPhone(TestNode)));

else

Put\_Line("Not in tree.");

end if;

--3

MySearchTree.FindCustomerRecursive(TestTree, "Patil ", TestNode);

Put("Patil # Recrusive: ");

if not IsNull(TestNode) then

Put\_Line(String(MySearchTree.CustomerPhone(TestNode)));

else

Put\_Line("Not in tree.");

end if;

--4

MySearchTree.FindCustomerIterative(TestTree, "Carazo ", TestNode);

Put("Carazo # Iterative: ");

if not IsNull(TestNode) then

Put\_Line(String(MySearchTree.CustomerPhone(TestNode)));

else

Put\_Line("Not in tree.");

end if;

--5

MySearchTree.FindCustomerRecursive(TestTree, "Carazo ", TestNode);

Put("Carazo # Recursive: ");

if not IsNull(TestNode) then

Put\_Line(String(MySearchTree.CustomerPhone(TestNode)));

else

Put\_Line("Not in tree.");

end if;

--6

New\_Line;

Put\_Line("Iterative InOrder Traversal From 'Guo':");

MySearchTree.FindCustomerIterative(TestTree, "Guo ", TestNode);

Put("[");

Put(String(MySearchTree.CustomerName(TestNode)));

Put(",");

Put(String(MySearchTree.CustomerPhone(TestNode)));

Put\_Line("]");

TestNode := MySearchTree.InOrderSuccessor(TestNode);

while MySearchTree.CustomerName(TestNode) /= "Guo " loop

Put("[");

Put(String(MySearchTree.CustomerName(TestNode)));

Put(",");

Put(String(MySearchTree.CustomerPhone(TestNode)));

Put\_Line("]");

TestNode := MySearchTree.InOrderSuccessor(TestNode);

end loop;

--7

New\_Line;

Insert(TestTree, "Sodmann ", "294-1568 ");

Insert(TestTree, "Carazo ", "294-1882 ");

Insert(TestTree, "Montes ", "295-6622 ");

--8

New\_Line;

Put\_Line("Iterative InOrder Traversal:");

TestNode := MysearchTree.InOrderSuccessor(TestTree);

Put("[");

Put(String(MySearchTree.CustomerName(TestNode)));

Put(",");

Put(String(MySearchTree.CustomerPhone(TestNode)));

Put\_Line("]");

TestNode := MySearchTree.InOrderSuccessor(TestNode);

while TestNode /= MysearchTree.InOrderSuccessor(TestTree) loop

Put("[");

Put(String(MySearchTree.CustomerName(TestNode)));

Put(",");

Put(String(MySearchTree.CustomerPhone(TestNode)));

Put\_Line("]");

TestNode := MySearchTree.InOrderSuccessor(TestNode);

end loop;

--B OPTION STEPS

--9

New\_Line;

MySearchTree.FindCustomerIterative(TestTree, "Furbee ", TestNode);

MySearchTree.DeleteRandomNode(TestTree, TestNode);

MySearchTree.FindCustomerIterative(TestTree, "Hall ", TestNode);

MySearchTree.DeleteRandomNode(TestTree, TestNode);

MySearchTree.FindCustomerIterative(TestTree, "Furbee ", TestNode);

MySearchTree.DeleteRandomNode(TestTree, TestNode);

MySearchTree.FindCustomerIterative(TestTree, "Sodmann ", TestNode);

MySearchTree.DeleteRandomNode(TestTree, TestNode);

MySearchTree.FindCustomerIterative(TestTree, "Burris ", TestNode);

MySearchTree.DeleteRandomNode(TestTree, TestNode);

MySearchTree.FindCustomerIterative(TestTree, "Furbee ", TestNode);

MySearchTree.DeleteRandomNode(TestTree, TestNode);

--10

New\_Line;

Insert(TestTree, "Caldwell ", "294-1666 ");

Insert(TestTree, "Houck ", "295-1882 ");

--11

New\_Line;

Put\_Line("Iterative InOrder Traversal:");

TestNode := MysearchTree.InOrderSuccessor(TestTree);

Put("[");

Put(String(MySearchTree.CustomerName(TestNode)));

Put(",");

Put(String(MySearchTree.CustomerPhone(TestNode)));

Put\_Line("]");

TestNode := MySearchTree.InOrderSuccessor(TestNode);

while TestNode /= MysearchTree.InOrderSuccessor(TestTree) loop

Put("[");

Put(String(MySearchTree.CustomerName(TestNode)));

Put(",");

Put(String(MySearchTree.CustomerPhone(TestNode)));

Put\_Line("]");

TestNode := MySearchTree.InOrderSuccessor(TestNode);

end loop;

--12

New\_Line;

Put\_Line("Recursive Reverse InOrder Traversal:");

MySearchTree.ReverseInOrder(TestTree);

--13

New\_Line;

Put\_Line("Iterative PreOrder Traversal Using Threads:");

MySearchTree.PreOrder(TestTree);

--A OPTION STEPS

--14

New\_Line;

Put\_Line("Iterative PostOrder Traversal Using Threads:");

MySearchTree.PostOrderIterative(TestTree);

--15

New\_Line;

Put\_Line("Recursive PostOrder Traversal:");

MySearchTree.PostOrderRecursive(TestTree);

end BinarySearchTreeTest;

BinarySearchTree.ads

generic

type Akey is private;

type BinarySearchTreeRecord is private;

with function "<" (

TheKey : in Akey;

ARecord : in BinarySearchTreeRecord)

return Boolean;-- Is TheKey less than the key of ARecord?

with function ">" (

TheKey : in Akey;

ARecord : in BinarySearchTreeRecord)

return Boolean;

with function "=" (

TheKey : in Akey;

ARecord : in BinarySearchTreeRecord)

return Boolean;

--needed if I want to manipulate tree record data fields since this is generic

with function GetName (

ARecord : in BinarySearchTreeRecord)

return AKey;

with function GetPhone (

ARecord : in BinarySearchTreeRecord)

return AKey;

with function NewCustomer (

Name : in AKey;

Phone : in AKey)

return BinarySearchTreeRecord;

--needed to print keys

with procedure Put (

Key : in AKey);

package BinarySearchTree is

type BinarySearchTreePoint is private;

--Constructor

procedure MakeEmptyBinarySearchTree (

NewTree : in out BinarySearchTreePoint;

Name : in AKey;

Phone : in AKey);

--Check if a tree pointer is null

function IsNull (

Tree : in BinarySearchTreePoint)

return Boolean;

--Returns the size of the tree.

function TreeSize (

Tree : in BinarySearchTreePoint)

return Integer;

procedure InsertBinarySearchTree (

Root : in out BinarySearchTreePoint;

CustName : in Akey;

CustPhone : in Akey);

procedure FindCustomerIterative (

Root : in BinarySearchTreePoint;

CustomerName : in Akey;

CustomerPoint : out BinarySearchTreePoint);

procedure FindCustomerRecursive (

Root : in BinarySearchTreePoint;

CustomerName : in Akey;

CustomerPoint : out BinarySearchTreePoint);

function InOrderSuccessor (

TreePoint : in BinarySearchTreePoint)

return BinarySearchTreePoint;

function CustomerName (

TreePoint : in BinarySearchTreePoint)

return Akey;

function CustomerPhone (

TreePoint : in BinarySearchTreePoint)

return Akey;

--Deletes a random node given the pointer to the node and the tree the node is in.

--The pointer to the delete node will be null after execution.

procedure DeleteRandomNode (

Tree : in BinarySearchTreePoint;

DeletePoint : in out BinarySearchTreePoint);

--Recursively prints the tree in reverse inorder

procedure ReverseInOrder (

Start : in BinarySearchTreePoint);

procedure PreOrder (

TreePoint : in BinarySearchTreePoint);

procedure PostOrderIterative (

TreePoint : in out BinarySearchTreePoint);

procedure PostOrderRecursive (

TreePoint : in out BinarySearchTreePoint);

private

type Node;

type BinarySearchTreePoint is access Node;

type Node is

record

Llink,

Rlink : BinarySearchTreePoint;

Ltag,

Rtag : Boolean;

Info : BinarySearchTreeRecord;

end record;

end BinarySearchTree;

BinarySearchTree.adb

with Ada.Text\_IO;

use Ada.Text\_IO;

with Ada.Unchecked\_Deallocation;

package body BinarySearchTree is

--constructor

procedure MakeEmptyBinarySearchTree (

Newtree : in out BinarySearchTreePoint;

Name : in AKey;

Phone : in AKey) is

begin

NewTree := new Node'(LTag => False, RTag => True, LLink => null, RLink => null, Info => NewCustomer(Name, Phone));

NewTree.LLink := NewTree;

NewTree.RLink := Newtree;

end;

function IsNull (

Tree : in BinarySearchTreePoint)

return Boolean is

begin

return Tree = null;

end IsNull;

function TreeSize (

Tree : in BinarySearchTreePoint)

return Integer is

LChild : Integer := 0;

RChild : Integer := 0;

P : BinarySearchTreePoint := Tree;

begin

if Tree.RTag = True and Tree.RLink = Tree then

if Tree.LTag = True then

P := P.LLink;

else

return 0;

end if;

end if;

if P.LTag = True then

LChild := TreeSize(P.LLink);

end if;

if P.RTag = True then

RChild := TreeSize(P.RLink);

end if;

return 1 + LChild + RChild;

end TreeSize;

--C

procedure InsertBinarySearchTree (

Root : in out BinarySearchTreePoint;

CustName : in Akey;

CustPhone : in Akey) is

P : BinarySearchTreePoint;

Q : BinarySearchTreePoint;

begin

if Root.Ltag = False then

Q := new Node'(LTag => False, RTag => False, LLink => Root, RLink => Root, Info => NewCustomer(CustName, CustPhone));

Root.LLink := Q;

Root.LTag := True;

else

P := Root.LLink;

loop

if CustName < P.Info then

if P.LTag = True then

P := P.LLink;

else

Q := new Node'(LTag => False, RTag => False, LLink => null, RLink => null, Info => NewCustomer(CustName, CustPhone));

Q.LLink := P.LLink;

P.LLink := Q;

P.LTag := True;

Q.RLink := P;

exit;

end if;

elsif CustName > P.Info or CustName = P.Info then

if P.RTag = True then

P := P.RLink;

else

Q := new Node'(LTag => False, RTag => False, LLink => null, RLink => null, Info => NewCustomer(CustName, CustPhone));

Q.RLink := P.RLink;

P.RLink := Q;

P.RTag := True;

Q.LLink := P;

exit;

end if;

end if;

end loop;

end if;

end InsertBinarySearchTree;

procedure FindCustomerIterative (

Root : in BinarySearchTreePoint;

CustomerName : in Akey;

CustomerPoint : out BinarySearchTreePoint) is

P : BinarySearchTreePoint;

begin

if Root.LTag = False then

CustomerPoint := null;

return;

end if;

P := Root.LLink;

loop

if CustomerName = P.Info then

CustomerPoint := P;

exit;

else

if CustomerName < P.Info then

if P.LTag = True then

P := P.LLink;

else

CustomerPoint := null;

exit;

end if;

elsif CustomerName > P.Info then

if P.RTag = True then

P := P.RLink;

else

CustomerPoint := null;

exit;

end if;

end if;

end if;

end loop;

end FindCustomerIterative;

procedure FindCustomerRecursive (

Root : in BinarySearchTreePoint;

CustomerName : in Akey;

CustomerPoint : out BinarySearchTreePoint) is

P : BinarySearchTreePoint;

begin

P := Root;

if P.RTag = True and P.RLink = P then

if P.LTag = False then

CustomerPoint := null;

return;

end if;

P := P.LLink;

end if;

if CustomerName < P.Info then

if P.LTag = True then

FindcustomerRecursive(P.LLink, CustomerName, CustomerPoint);

else

CustomerPoint := null;

return;

end if;

elsif CustomerName > P.Info then

if P.RTag = True then

FindcustomerRecursive(P.RLink, CustomerName, CustomerPoint);

else

CustomerPoint := null;

return;

end if;

else

CustomerPoint := P;

end if;

end FindCustomerRecursive;

function InOrderSuccessor (

TreePoint : in BinarySearchTreePoint)

return BinarySearchTreePoint is

Q : BinarySearchTreePoint;

begin

Q := TreePoint.RLink;

if TreePoint.RTag = False then

if Q.RLink = Q then

while Q.LTag = True loop

Q := Q.LLink;

end loop;

end if;

else

while Q.LTag = True loop

Q := Q.LLink;

end loop;

end if;

return Q;

end InOrderSuccessor;

function CustomerName (

TreePoint : in BinarySearchTreePoint)

return Akey is

begin

return GetName(TreePoint.Info);

end CustomerName;

function CustomerPhone (

TreePoint : in BinarySearchTreePoint)

return Akey is

begin

return GetPhone(TreePoint.Info);

end CustomerPhone;

--B

procedure DeleteRandomNode (

Tree : in BinarySearchTreePoint;

DeletePoint : in out BinarySearchTreePoint) is

P : BinarySearchTreePoint;

Q : BinarySearchTreePoint;

R : BinarySearchTreePoint;

S : BinarySearchTreePoint;

T : BinarySearchTreePoint;

procedure FreeNode is

new Ada.Unchecked\_Deallocation(Node, BinarySearchTreePoint);

type DeleteType is

(Left,

Right,

Root);

DT : DeleteType;

begin

--Do nothing if either pointer is null or tree is empty.

if (Tree = null or DeletePoint = null) or else Tree.LTag = False then

return;

end if;

Put("Deleting entry: [");

Put(CustomerName(DeletePoint));

Put(",");

Put(CustomerPhone(DeletePoint));

Put\_Line("]");

--Setup P and Q.

P := Tree;

Q := Tree.LLink;

DT := Left;

--If setup is immediately correct, set to root deletion

if CustomerName(DeletePoint) = Q.Info then

DT := Root;

else

--Otherwise move P and Q until they are correct.

loop

--PreStep: Move P down to Q. Q will be a child of P by the end of the

--loop.

P := Q;

--Two steps:

--1. Move Q in the direction of the node to delete.

--2. Check if it has landed on the node to delete. If so, change direction

-- accordingly and exit the loop.

if CustomerName(DeletePoint) < Q.Info then

Q := Q.LLink;

if CustomerName(DeletePoint) = Q.Info then

exit;

end if;

else

Q := Q.RLink;

if CustomerName(DeletePoint) = Q.Info then

DT := Right;

exit;

end if;

end if;

end loop;

end if;

--Begin Deletion

T := Q;

if T.RTag = False then

if T.LTag = True then

--T has left children but no right child

Q := T.LLink;

while Q.RTag = True loop

Q := Q.LLink;

end loop;

Q.RLink := T.RLink;

Q := T.LLink;

FreeNode(DeletePoint);

else

--T has no left or right children

if DT = Root or DT = Left then

Q := Q.LLink;

P.LTag := False;

else

Q := Q.RLink;

P.RTag := False;

end if;

FreeNode(DeletePoint);

end if;

else

if T.LTag = False then

--T has no left children and a right child

R := T.RLink;

if R.LTag = False then

R.LLink := T.LLink;

Q := R;

FreeNode(DeletePoint);

else

S := R.LLink;

while S.LTag /= False loop

R := S;

S := R.LLink;

end loop;

if S.RTag = True then

R.LLink := S.RLink;

else

R.LTag := False;

S.RTag := True;

end if;

S.LLink := T.LLink;

S.LTag := True;

S.RLink := T.RLink;

Q := S;

FreeNode(DeletePoint);

end if;

else

--T has left children and a right child

R := T.RLink;

if R.LTag = False then

R.LLink := T.LLink;

R.LTag := True;

Q := T.LLink;

while Q.RTag = True loop

Q := Q.RLink;

end loop;

Q.RLink := R;

Q := R;

FreeNode(DeletePoint);

else

S := R.LLink;

while S.LTag /= False loop

R := S;

S := R.LLink;

end loop;

if S.RTag = True then

R.LLink := S.RLink;

else

R.LTag := False;

S.RTag := True;

end if;

S.LLink := T.LLink;

S.LTag := True;

S.RLink := T.RLink;

Q := T.LLink;

while Q.RTag = True loop

Q := Q.RLink;

end loop;

Q.RLink := S;

Q := S;

FreeNode(DeletePoint);

end if;

end if;

end if;

if DT = Root then

P.LLink := Q;

else

if DT = Left then

P.LLink := Q;

else

P.RLink := Q;

end if;

end if;

end DeleteRandomNode;

procedure ReverseInOrder (

Start : in BinarySearchTreePoint) is

begin

if Start = null then

return;

end if;

if Start.RTag = True and Start.RLink = Start then

if Start.LTag = True then

ReverseInOrder(Start.LLink);

return;

else

return;

end if;

end if;

if Start.RTag = True then

ReverseInOrder(Start.RLink);

end if;

Put("[");

Put(CustomerName(Start));

Put(",");

Put(CustomerPhone(Start));

Put\_Line("]");

if Start.LTag = True then

ReverseInOrder(Start.LLink);

end if;

end ReverseInOrder;

procedure PreOrder (

TreePoint : in BinarySearchTreePoint) is

P : BinarySearchTreePoint;

Stop : BinarySearchTreePoint;

begin

P := TreePoint;

if TreePoint.RTag = True and TreePoint.RLink = TreePoint then

P := TreePoint.LLink;

end if;

Stop := P;

Put("[");

Put(CustomerName(P));

Put(",");

Put(CustomerPhone(P));

Put\_Line("]");

loop

if P.LTag = True then

P := P.LLink;

else

while P.RTag = False loop

P := P.RLink;

end loop;

if P.RLink = P then

P := P.LLink;

else

P := P.RLink;

end if;

end if;

exit when P = Stop;

Put("[");

Put(CustomerName(P));

Put(",");

Put(CustomerPhone(P));

Put\_Line("]");

end loop;

end PreOrder;

--A

procedure PostOrderIterative (

TreePoint : in out BinarySearchTreePoint) is

P : BinarySearchTreePoint;

Stop : BinarySearchTreePoint;

Top : Integer;

begin

Top := TreeSize(TreePoint);

P := TreePoint;

if P.RTag = True and P.RLink = P then

P := P.LLink;

end if;

Stop := P;

declare

Stack : array (1 .. Top) of BinarySearchTreePoint;

begin

Top := 1;

Stack(Top) := P;

loop

if P.RTag = True then

P := P.RLink;

else

while P.LTag = False loop

P := P.LLink;

end loop;

P := P.LLink;

end if;

exit when P = Stop;

Top := Top + 1;

Stack(Top) := P;

end loop;

while Top > 0 loop

Put("[");

Put(CustomerName(Stack(Top)));

Put(",");

Put(CustomerPhone(Stack(Top)));

Put\_Line("]");

Top := Top - 1;

end loop;

end;

end PostOrderIterative;

procedure PostOrderRecursive (

TreePoint : in out BinarySearchTreePoint) is

begin

if TreePoint = null then

return;

end if;

if TreePoint.RTag = True and TreePoint.RLink = TreePoint then

if TreePoint.LTag = True then

PostOrderRecursive(TreePoint.LLink);

return;

else

return;

end if;

end if;

if TreePoint.LTag = True then

PostOrderRecursive(TreePoint.LLink);

end if;

if TreePoint.RTag = True then

PostOrderRecursive(TreePoint.RLink);

end if;

Put("[");

Put(CustomerName(TreePoint));

Put(",");

Put(CustomerPhone(TreePoint));

Put\_Line("]");

end PostOrderRecursive;

end BinarySearchTree;