**Data structures and algorithms Project documentation**

**Naiman Alexandru Nicolae**

**ADT Binary Tree**

**(With iterators to iterate in preorder, inorder, postorder and levelorder)**

**Linked representation with dynamic allocation**

1. **ADT Binary Tree specification and interface + iterator**

A tree is a connected, acyclic graph (usually undirected). An ordered tree is a tree in which the order of the children is well defined and relevant (instead of having a set of children, each node has a list of children). An ordered tree in which each node has at most two children is called binary tree. In a binary tree we call the children of a node the left child and right child.

**Domain of the ADT Binary tree**

BT = {bt | bt binary tree with nodes containing information of type TElem}

**Interface:**

* init (bt):

descr: creates a new, empty binary tree

pre: true

post: bt ∈ BT , bt is an empty binary tree

* initLeaf(bt, e) :

descr: creates a new binary tree, having only the root with a given value

pre: e ∈ TElem

post: bt ∈ BT , bt is a binary tree with only one node (its root) which contains the value e

* initTree(bt, left, e, right) :

descr: creates a new binary tree, having a given information in the root and two given binary trees as children

pre: left,right ∈ BT , e ∈ TElem

post: bt ∈ BT , bt is a binary tree with left child equal to left, right child equal to right and the information from the root is e

* insertLeftSubtree(bt, left) :

descr: sets the left subtree of a binary tree to a given value (if the tree had a left subtree, it will be changed)

pre: bt, left ∈ BT

post: bt’ ∈ BT , the left subtree of bt’ is equal to left

* insertRightSubtree(bt, right):

descr: sets the right subtree of a binary tree to a given value (if the tree had a right subtree, it will be changed)

pre: bt,right ∈ BT

post: bt’ ∈ BT , the right subtree of bt’ is equal to right

* getRootInfo(bt):

descr: returns the information from the root of a binary tree

pre: bt ∈ BT , bt != Φ

post: root ← e, e ∈ TElem, e is the information from the root of bt

throws: an exception if bt is empty

* left(bt):

descr: returns the left subtree of a binary tree

pre: bt ∈ BT , bt != Φ

post: left ← l, l ∈ BT , l is the left subtree of bt

throws: an exception if bt is empty

* right(bt):

descr: returns the right subtree of a binary tree

pre: bt ∈ BT , bt != Φ

post: right ← r, r ∈ BT , r is the right subtree of bt

throws: an exception if bt is empty

* isEmpty(bt):

descr: checks if a binary tree is empty

pre: bt ∈ BT

post: empty ← {True, if bt = Φ

False, otherwise

* iterator (bt, traversal, i):

descr: returns an iterator for a binary tree

pre: bt ∈ BT, traversal represents the order in which the tree has to be traversed

post: i ∈ I, i is an iterator over bt that iterates in the order given by traversal

* destroy(bt):

descr: destroys a binary tree

pre: bt ∈ BT

post: bt was destroyed

**ADT Binary Tree Iterator Interface**

Since the binary trees can be traversed in four different ways (preorder, inorder, postorder, levelorder) we can define four iterators for iterating them in such a manner but all of them will have the same interface, the only difference being the implementation and the representation of those

**Domain**

I={i | i is an iterator over a Binary Tree bt)

**Interface:**

* init( bt, it)

descr: creates a new iterator for a Binary tree (bt)

pre: bt ∈ BT

post: it ∈ I and it points to the root of the tree bt, if bt is neither empty nor valid

* getCurrent(it, k):

descr: returns the current node from the iterator

pre: it ∈ I, it is valid

post: getCurrent←current node from iterator

* valid(it):

descr: verifies if the iterator is valid

pre: it ∈ I

post: valid←True, if the iterator poins to valid node of the binary tree, False,otherwise

* next(it):

descr: moves the current element to the next node according to the way of the traversal, or makes the iterator invalid if there are no more nodes

pre: it ∈ I, it is valid

post: the current element points to the next node of the tree

1. **ADT Binary tree representation**

**BTNode:**

info: TElem

left: ↑ BTNode

right: ↑ BTNode

**BinaryTree:**

root: ↑ BTNode

**PreOrderIteratorBT;**

bt: BinaryTree

s: Stack

currentNode: ↑ BTNode

**PostOrderIteratorBT;**

bt: BinaryTree

s: Stack

currentNode: ↑ BTNode

**InOrderIteratorBT;**

bt: BinaryTree

s: Stack

currentNode: ↑ BTNode

**LevelOrderIteratorBT;**

bt: BinaryTree

q: Queue

currentNode: ↑ BTNode

1. **Problem statement and justification**

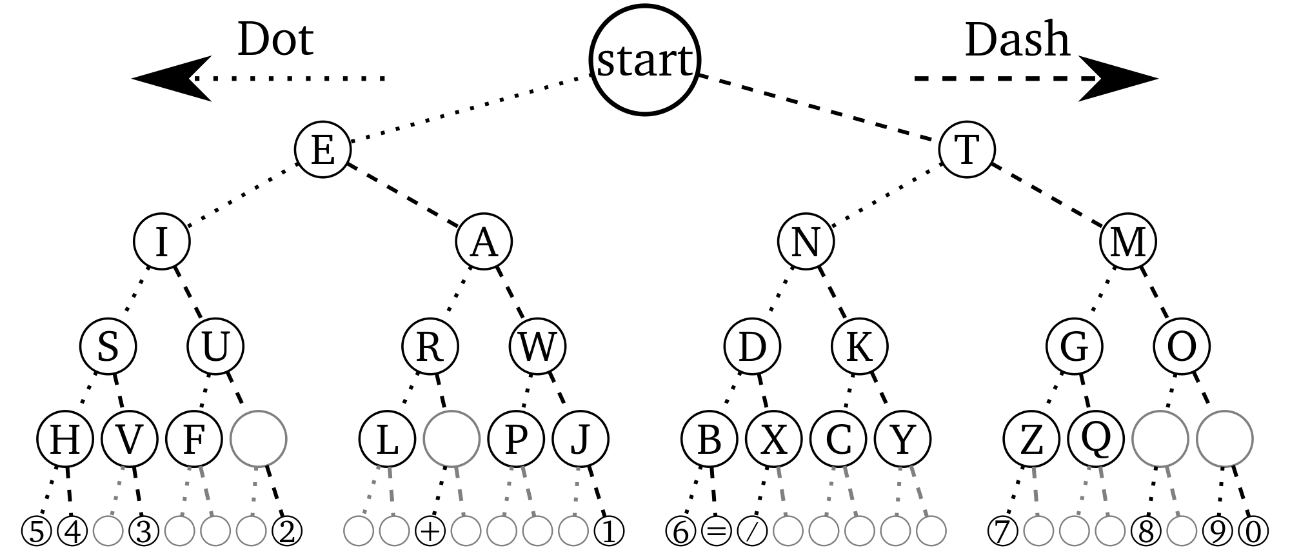
**Morse code** is a method of transmitting text information as a series of on-off tones, lights, or clicks that can be directly understood by a skilled listener or observer without special equipment. Each Morse code symbol represents either a text character (letter or numeral) or a prosign and is represented by a unique sequence of dots(“.”) and dashes(“\_”). The coded letters are usually separated through empty blank spaces (“ “), and the words are separated through a special character (“/”).

**Example:**

-... .. -. .- .-. -.-- / - .-. . . == binary tree

As you are a big fan data encryption you want to create an application that is capable of encrypting and decrypting phrases of unspecified size from and into Morse code. You we’ll need to performs these operations on text stored on a file. As the symbols (dash, dot, slash) appearance have no relevant meaning to the Morse code, you can change them to use any character of your like, with the condition that they are different two by two.

I choose to use an ADT Binary Tree for this application because, one can organize the Morse code as a binary tree, making it easy to retrieve data, using the iterators so we can encrypt/decrypt the letters one by one easily



A graphical representation of the binary tree of the Morse code: Shifting to the left represents a Dit (.), and a shift to the right represents a Dah (-). Where one lands indicates the letter for the code.

**6.Implementation**

1. Implementation for the operation of the binary tree interface

* subalgorithm init(bt) // Theta(1)

//descr: creates a new, empty binary tree

//pre: true

//post: bt ∈ BT , bt is an empty binary tree

@creates a new binary tree

end-subalgorithm

* subalgorithm initLeaf(bt, e): // Theta(1)

//descr: creates a new binary tree, having only the root with a //given value

//pre: e ∈ TElem

//post: bt ∈ BT, bt is a binary tree with only one node (its root) //which contains the value e

bt.root = allocate()

[bt.root].info = e

end-subalgorithm

* subalgorithm initTree(bt, left, e, right):

//descr: creates a new binary tree, having a given information //in the root and two given binary trees as children

//pre: left,right ∈ BT , e ∈ TElem

//post: bt ∈ BT , bt is a binary tree with left child equal to left, //right child equal to right and the information from the root is //e

// Theta(log2(l+r)), l/r – number of nodes of left/right

bt.root <- allocate()

[bt.root].left <-copySubTree(left.root);

[bt.root].right <-copySubTree(right.root);

end-subalgorithm

* subalgorithm insertLeftSubtree(bt, left):// O(h+n), n – //number of nodes of the, h-number of nodes of left subtree

//descr: sets the left subtree of a binary tree to a given value (if //the tree had a left subtree, it will be changed)

//pre: bt, left ∈ BT

//post: bt’ ∈ BT , the left subtree of bt’ is equal to left

deleteSubTrees([bt.root]left)

[bt.root].left = copySubTree(left.root);

end-subalgorithm

* subalgorithm inserRightSubtree(bt, right): ):// O(h + n), h-//number of nodes of left subtree,

//descr: sets the right subtree of a binary tree to a given value //(if the tree had a right subtree, it will be changed)

//pre: bt, right ∈ BT

//post: bt’ ∈ BT , the right subtree of bt’ is equal to right

deleteSubTrees([bt.root].right)

[bt.root]. right = copySubTree(right.root);

end-subalgorithm

* function getRootInfo(bt): // Theta(1)

//descr: returns the information from the root of a binary tree

//pre: bt ∈ BT , bt != Φ

//post: root ← e, e ∈ TElem, e is the information from the root //of bt

//throws: an exception if bt is empty

if bt.root = NIL then

@throw exception("Tree is empty!")

end-if

getRootInfo<-[bt.root].info

end-function

* function getRoot(bt):// Theta(1)

//descr: returns the root of a binary tree

//pre: bt ∈ BT , bt != Φ

//post: root ← r, r ∈ ↑BTNode the root of bt

//throws: an exception if bt is empty

if bt.root = NIL then

@throw exception("Tree is empty!")

end-if

getRoot<-bt.root

end-function

* function left(bt): // Theta(1)

//descr: returns the left subtree of a binary tree

//pre: bt ∈ BT , bt != Φ

//post: left ← l, l ∈ BT, l is the left subtree of bt

//throws: an exception

end-function

if bt.root = NIL then

@throw exception("Tree is empty!")

end-if

left <- [bt.root].left

end-function

* function right(bt): // Theta(1)

//descr: returns the right subtree of a binary tree

//pre: bt ∈ BT , bt != Φ

//post: right ← l, l ∈ BT, l is the right subtree of bt

//throws: an exception

end-function

if bt.root = NIL then

@throw exception("Tree is empty!")

end-if

right <- [bt.root]. right

end-function

* function isEmpty(bt): //Theta(1)

//descr: checks if a binary tree is empty

//pre: bt ∈ BT

//post: empty ← {True, if bt = Φ, False, otherwise

If bt.root = NIL then

isEmpty <- True

else

isEmpty<- False

end-if

end-function

* subalgorithm iterator (bt, traversal, i): // O(log2n)

//descr: returns an iterator for a binary tree

//pre: bt ∈ BT, traversal represents the order in which the tree //has to be traversed

//post: i ∈ I, i is an iterator over bt that iterates in the order //given by traversal

If @traversal = preorder then

i<-preOrderInit(bt,i)

else if @traversal = inorder then

i<-inOrderInit(bt,i)

else if @traversal = postorder then

i<-postOrderInit(bt,i)

else if @traversal = levelorder then

i<-levelOrderInit(bt,i)

end-if

end-subalgorithm

* subalgorithm destroy(bt):

//descr: destroys a binary tree

//pre: bt ∈ BT post:

//bt was destroyed

destroyNodesFromSubTree(bt.root);

end-subalgorithm

1. **Helper functions used to implement the functions of the binary tree’s interface**

* subalgorithm destroySubTrees(node): // Theta(n)

//desc: deletes the subtrees from a given node

//pre: node ∈ ↑BTNode

//post: the subtree is deleted

if node !=NILL then

destroyNodesFromSubTree([node].left)

destroyNodesFromSubTree([node].right)

end-if

end-subalgorithm

* subalgorithm destroyNodesSubTrees(node): // O(n)

//desc: deletes the subtrees and frees its nodes, from a given //node

//pre: node ∈ ↑BTNode

//post: the subtree is deleted

if node!=NIL then

destroyNodesSubTrees([node].left)

destroyNodesSubTrees([node].right)

free(node)

end-if

end-subalgorithm

Complexity:

T(n)={1, if n=1

2T(n/2) + 1, otherwise

T(n)=2T(n/2)+1

T(n/2)=2T(n/4)+1

T(n/4)=2T(n/8)+1

T(n)=(2^3)\*T(n/8) + 4 + 2 + 1

…

T(n)=(2^k)\*T(n/k)

k=log2(n)

T(n) = n \* T(1) + 2^(log2(n)-1)-1 = n + n/2 ∈ O(n)

* function copySubTree(subtree): //O(n)

//desc: copies recursively the nodes from a subtree (node)

//pre: node ∈ ↑BTNode

//post: copySubTree<- the new subTree with all the elements //copied from the given subtree

if subtree != nill then

pNew <- allocate()//allocate new node

[pNew].info = [subtree].info

[pNew].left <-copySubTree([subtree].left)

[pNew].right<-copySubTree([subtree].right)

copySubTree<-pNew

end-if

copySubTree<-NIL

end-function

1. **Implementation of the functions from the iterators’ interface**
2. **PreOrder Iterator**

* subalgorithm initPreorder(bt,it) // Theta(1)

//descr: creates a new preorder iterator for a Binary tree (bt)

//pre: bt ∈ BT

//post: it ∈ I and it points to first node of the tree bt, if bt is //neither empty nor valid

@initialize it.s as a Stack

it.bt <- bt

node <- node.root

if node!=NIL then

push(it.s, node)

end-if

if not empty(it.s) then

it.currentNode <- top(it.s)

else

it.currentNode<-NIL

end-if

end-subalgorithm

* function getCurrent(it): // Theta(1)

//descr: returns the current node from the iterator

//pre : it ∈ I, it is valid

//post: getCurrent←current node from iterator

getCurrent<- [it.currentNode].info

end-function

* function valid(it): // Theta(1)

//descr: verifies if the iterator is valid

//pre: it ∈ I

//post: valid←True, if the iterator poins to valid node of the //binary tree, False,otherwise

If it.currentNode = NIL then

valid<-False

else

valid<-True

end-if

end-function

* subalgorithm next(it): // Theta(1)

//descr: moves the current element to the next node according //to the way of the preorder traversal, or makes the iterator //invalid if there are no more nodes

//pre: it ∈ I, it is valid

//post: the current element points to the next node of the tree

node<- pop(it.s)

if [node].right != NIL then

push(it.s, [node].right)

end-if

if [node].left != NIL then

push(it.s, [node].left)

end-if

if not empty(it.s) then

it.currentNode = top(it.s);

else

it.currentNode = NIL

end-if

end-subalgorithm

1. **InOrder Iterator**

* subalgorithm initInOrder(bt,it) // Theta(log2(n))

//descr: creates a new inorder iterator for a Binary tree (bt)

//pre: bt ∈ BT

//post: it ∈ I and it points to the first elem of the tree bt, if bt is //neither empty nor valid

@initialize it.s as a Stack

it.bt <- bt

node <- node.root

while node!=NIL execute:

push(s,node)

node<-[node].left

end-while

if not empty(it.s) then

it.currentNode <- top(it.s)

else

it.currentNode<-NIL

end-if

end-subalgorithm

* function getCurrent(it): // Theta(1)

//descr: returns the current node from the iterator

//pre : it ∈ I, it is valid

//post: getCurrent←current node from iterator

getCurrent<- [it.currentNode].info

end-function

* function valid(it): // Theta(1)

//descr: verifies if the iterator is valid

//pre: it ∈ I

//post: valid←True, if the iterator points to valid node of the //binary tree, False,otherwise

If it.currentNode = NIL then

valid<-False

else

valid<-True

end-if

end-function

* subalgorithm next(it): // O(log2(n)

//descr: moves the current element to the next node according //to the way of the inorder traversal, or makes the iterator //invalid if there are no more nodes

//pre: it ∈ I, it is valid

//post: the current element points to the next node of the tree

node<- pop(it.s)

if [node].right = NIL then

node<- [node].right

while node!=NIL execute

push(it.s,node)

node<-[node].left

end-while

end-if

if not empty(it.s) then

it.currentNode = top(it.s);

else

it.currentNode = NIL

end-if

end-subalgorithm

1. **PostOrder Iterator**

* subalgorithm initPostOrder(bt,it) // Theta(log2(n))

//descr: creates a new postorder iterator for a Binary tree (bt)

//pre: bt ∈ BT

//post: it ∈ I and it points to the first node of the tree bt, if bt is //neither empty nor valid

@initialize it.s as a Stack

it.bt <- bt

findNextLeaf(it,bt.root)

if not empty(it.s) then

res<-pop(it.s)

if not empty(it.s) then

top<-top(it.s)

if res = [top].left then

findNextLeaf(it,[top].right)

end-if

it.currentNode <- res

end-if

else

it.currentNode <- NIL

end-if

end-subalgorithm

* subalgorithm findNextLeaf(it, current): O(log2(n))

//descr: helper function used to build the stack for the iterator //and to the next leaf we need to put on the stack

//pre : it ∈ I, it is valid, current ∈ ↑BTNode

//post: it.s contains all the nodes from the given to node to the //next leaf we search for

while current!=NIL execute:

push(it.s,current)

if [current].left !=NIL then

current <- [current].left

else

current <- [current].right

end-if

end-while

end-subalgorithm

* function getCurrent(it): // Theta(1)

//descr: returns the current node from the iterator

//pre : it ∈ I, it is valid

//post: getCurrent←current node from iterator

getCurrent<- [it.currentNode].info

end-function

* function valid(it): // Theta(1)

//descr: verifies if the iterator is valid

//pre: it ∈ I

//post: valid←True, if the iterator poins to valid node of the //binary tree, False,otherwise

If it.currentNode = NIL then

valid<-False

else

valid<-True

end-if

end-function

* subalgorithm next(it): // O(log2(n))

//descr: moves the current element to the next node according //to the way of the postorder traversal, or makes the iterator //invalid if there are no more nodes

//pre: it ∈ I, it is valid

//post: the current element points to the next node of the tree

if not empty(it.s) then

res<- pop(it.s)

if res = [top].left then

findNextLeaf(it,[top].right)

end-if

it.currentNode<-res

else

it.currentNode <- NIL

end-if

end-subalgorithm

1. **LevelOrder Iterator**

* subalgorithm initLevelOrder(bt,it): // Theta(1)

//descr: creates a new levelorder iterator for a Binary tree (bt)

//pre: bt ∈ BT

//post: it ∈ I and it points to the root of the tree bt, if bt is neither //empty nor valid

@initialize it.q as a queue

it.bt <- bt

push(it.q,bt.root)

if not empty(it.q) then

it.currentNode <- front(it.q)

else

it.currentNode <-NIL

end-subalgorithm

* function getCurrent(it): // Theta(1)

//descr: returns the current node from the iterator

//pre : it ∈ I, it is valid

//post: getCurrent←current node from iterator

getCurrent<- [it.currentNode].info

end-function

* function valid(it): // Theta(1)

//descr: verifies if the iterator is valid

//pre: it ∈ I

//post: valid←True, if the iterator poins to valid node of the //binary tree, False,otherwise

If it.currentNode = NIL then

valid<-False

else

valid<-True

end-if

end-function

* subalgorithm next(it): // Theta(1)

//descr: moves the current element to the next node according //to the way of the levelorder traversal, or makes the iterator //invalid if there are no more nodes

//pre: it ∈ I, it is valid

//post: the current element points to the next node of the tree

node<-pop(it.q)

if [node].left !=NIL then

push(it.q,[node].left)

end-if

if [node].right !=NIL then

push(it.q,[node].right)

end-if

if not empty(it.q) then

it.currentNode<-front(it.q)

else

it.currentNode<-NIL

end-subalgorithm

1. **Problem solution implementation**

* Additional structures:

morseLetter:

letter: string

morse: string

UI:

dotSymbol: string

dashSymbol: string

separator: string

wordSeparator: string

morseTree: BinaryTree

* Implementation:

subalgorithm initUI(ui):

@creates and initialize the morseTree accordingly with the Morse code diagram

end-subalgorithm

subalgorithm run(ui):

cond<-True

while cond execute

printMenu(ui)

command<- readInput(ui)

if command = 0 then

cond<-False

else if command = 1 then

morseToTextMenu(ui)

else if command = 2 then

textToMorseMenu(ui)

else if command = 3 then changeSymbols(UI)

end-if

end-subalgorithm

subalgorithm morseToTextMenu(ui)

@read the filename from user and validate it

while @read word by word separated by \ui.wordSeparator execute:

while @get first “letter”of the word \execute:

@print morseToText(ui,letter)

end-while

end-while

end-subalgorithm

function morseToText(ui,codedLetter): // O(log2(n))

@initialize decodedWord as empty string

node<-getRoot(ui.morseTree)

while @codedLetter.size() > 0 execute:

@initialize s with the first char form \codedLetter if s!=dashSymbol and s!= dotSymbol then @throw exception

end-if

if s = dotSymbol then

node<-[node].left

@remove first char of codedLetter

else if s = dashSymbol then

node<-[node].right

@remove first char of codedLetter

end-if

end-while

decodedWord<-[node].info.morse

morseToText<-decodedWord

end-function

subalgorithm textToMorseMenu(ui):

@read the filename from user and validate it

while @read word by word separated by \ui.separator execute:

@read the text letter by letter

@check if we can convert the word

@if we cannot throw exceptions

root<- getRoot(ui.morseTree)

letter<-textToMorse(ui,root, letter)

@create the word from the morse symbols

@write the words to the file

end-while

end-subalgorithm

function textToMorse(ui,node, search): // O(n)

if node !=NIL then

if [node].letter = search then

textToMorse <-node

end-if

else

foundNode<- textToMorse([node].left,search)

if foundNode = NIL then

foundNode=textToMorse([node].right,search)

end-if

textToMorse<-foundNode

end-if

else

textToMorse<-NIL

end-if

end-function

subalgorithm printMenu(ui): // Theta(1)

@prints the menu Options:

end-subalgorithm

function read\_input(ui,node, search):

@forces to user to input a valid unsigned integer-n

read\_input<-n

end-function

subalgorithm changeSymbols(ui,node, search):

//Theta(1)

@read dot

@read dash

@read separator

if dot=dash or dash=separator or dot = separator \then

@throw exception/print message error

ui.dotSymbol <-dot

ui.dashSymbol <-dash

ui.wordSeparator<-separator

end-subalgorithm

**7. Tests**

void testConstructors()

{

//test exceptions

BinaryTree<int>o{};

try {

o.getRootValue();

assert(false);

}

catch(std::exception&e){

}

try {

o.left();

assert(false);

}

catch (std::exception&e) {

}

try {

o.right();

assert(false);

}

catch (std::exception&e) {

}

//test adt

BinaryTree<int>t;

assert(t.isEmpty());

t = BinaryTree<int>{ 6 };

assert(t.getRootValue() == 6);

assert(!t.isEmpty());

BinaryTree<int>t1{ 7 };

BinaryTree<int>t2{ 8 };

assert(t.getRoot()->element = t.getRootValue());

t.insertLeftSubtree(t1);

t.insertRightSubtree(t2);

assert(t1.getRootValue() == t.left().getRootValue());

assert(t2.getRootValue() == t.right().getRootValue());

t.insertLeftSubtree(t1);

assert(t.left().getRootValue() == t1.getRootValue());

}

void testIterators()

{

BinaryTree<std::string>ta;

ta = BinaryTree<std::string>{ "A" };

BinaryTree<std::string>tg{ "G" };

BinaryTree<std::string>tt{ "T" };

BinaryTree<std::string>tq{ "Q" };

BinaryTree<std::string>tx{ "X" };

BinaryTree<std::string>tn{ "N" };

BinaryTree<std::string>to{ "O" };

BinaryTree<std::string>ty{ "Y" };

BinaryTree<std::string>tj{ "J" };

BinaryTree<std::string>tk{ "K" };

tj.insertLeftSubtree(tk);

tx.insertRightSubtree(tj);

tx.insertLeftSubtree(ty);

tg.insertLeftSubtree(tq);

tg.insertRightSubtree(tx);

tt.insertLeftSubtree(tn);

tt.insertRightSubtree(to);

ta.insertLeftSubtree(tg);

ta.insertRightSubtree(tt);

std::string checkInOrderTraversal = "Q G Y X K J A N T O ";

std::string auxString = "";

AbstractIterator<std::string>\* inOrder = ta.iterator("inorder");

while (inOrder->valid()) {

auxString += inOrder->getCurrent() + " ";

inOrder->next();

}

assert(checkInOrderTraversal == auxString);

std::string checkPreOrderTraversal = "A G Q X Y J K T N O ";

auxString = "";

AbstractIterator<std::string>\* preOrder = ta.iterator("preorder");

while (preOrder->valid()) {

auxString += preOrder->getCurrent() + " ";

preOrder->next();

}

assert(checkPreOrderTraversal == auxString);

std::string checkPostOrderTraversal = "Q Y K J X G N O T A ";

auxString = "";

AbstractIterator<std::string>\* postOrder = ta.iterator("postorder");

while (postOrder->valid()) {

auxString += postOrder->getCurrent() + " ";

postOrder->next();

}

assert(checkPostOrderTraversal == auxString);

std::string checkLevelOrderTraversal = "A G T Q X N O Y J K ";

auxString = "";

AbstractIterator<std::string>\* levelOrder = ta.iterator("levelorder");

while (levelOrder->valid()) {

auxString += levelOrder->getCurrent() + " ";

levelOrder->next();

}

assert(checkLevelOrderTraversal == auxString);

}