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
- root(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 and representation, the only difference being the implementation.

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

2. ADT Binary tree representation

<u>BTNode:</u>
info: TElem
left: ↑ BTNode
right: ↑ BTNode

<u>BinaryTree:</u>
root: ↑ BTNode

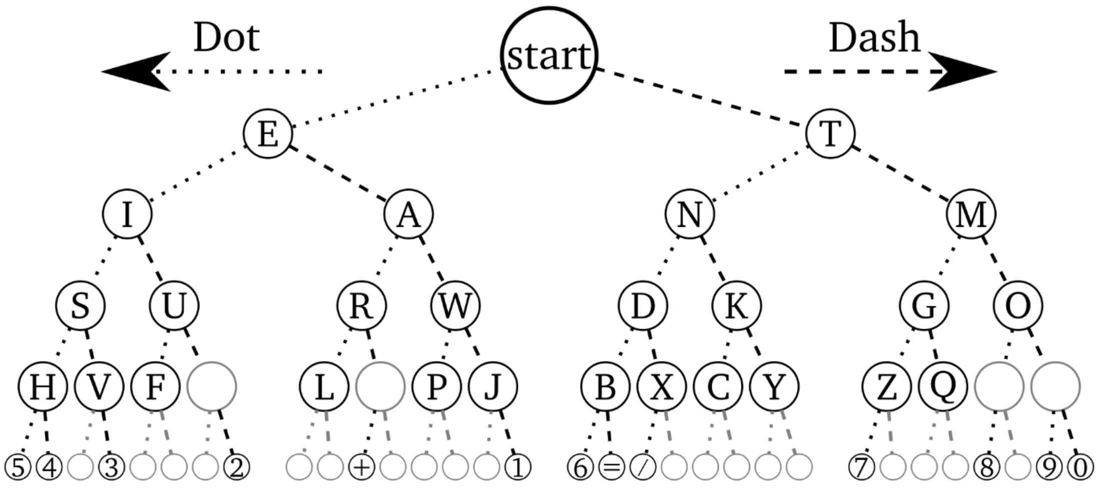
<u>IteratorBT;</u>
bt: BinaryTree
s: Stack
currentNode: ↑ BTNode

3. 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(“_”).

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 given by the user at run-time or on text stored on a file

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.