1. Introduction

The main goal of this project is to simulate a real world problem about forwarding packets through a network. The program developed receives as input a file that contains a prefix table.

First of all, the table has to be translated to a prefix tree, in order to be managed by users who can perform many tasks, such as: printing the prefix table to the screen, inserting a new prefix and the associated next-hop, looking for the next-hop of an address or deleting a prefix from the tree.

The program is divided in four files (prefixTree.c, trees.c, binaryTrees.c and utils.c), plus main, and the respective headers. The prefixTree.c contains functions that perform all the options that the user has. Trees.c has general functions to deal with trees and utils.c has functions that protect the program against incorrect inputs and one that represents the menu to users. Note: the root representation in this program is an E.

1. trees.c

It’s opportune to start with a description of the functions included in trees.c because they will be used in the other files once they’re the way to manipulate the basic operations of the prefix tree.

It’s not necessary to do the pseudo-codes of these functions once they are really intuitive.

The *newNode(num)* function allocates memory for a new node calling the *allocNode()* function, fills with NULL the left and right pointer and with the parameter num, the value field.

The *setLeft(node, num)* and *setRigth(node, num)* functions check if the nodes already exist and if not, create them, otherwise the *setValue(node, num)* function is called and the value of the nodes, which corresponds to their the next-hop, is assigned.

Both *getLeft(node)* and *getRight(node)* functions are the ones that allow to get the left and the right nodes of its parent node, respectively.

The *getValue(node)* function returns the next-hop of a node.

Finally, the function *freeTree(node)* is used to free all memory allocated in the tree bellow the node parameter. **Note:** the binarytree.c file has the same type of functions but adapted to a two-bit prefix tree.

1. prefixTree.c

This file has the most important functions of the program. All of them will be explained with the respective pseudocode and a brief discussion that includes the complexity of the algorithms.

***PrefixTree: PrintTable:***

**PrintTable(node, str1, str2)**

if node = null then

return;

end if

Allocate memory for an auxiliar string named aux

for i := 0 to lenght(str1) + length(str2)

aux[i] = ‘/0’

end for

Concatenate both str1 and str2 into aux

PrintTable(getLeft(node), aux, “0”)

PrintTable(getRight(node), aux, “1”)

if next-hop of node is from -1 and  -2

print prefix and the correspondent next-hop

end if

free(aux)

**PrefixTree(file)**

ptr :=open(file)

nextHop:= 0

root:= newNode(-2)

if ptr = null then

print No file

return null

end if

while line  null

Assign the prefix read from the file and the correspondent next-hop

InsertPrefix(prefix, nextHop, strlen(prefix), root, 0)

end while

close(ptr)

return root

***DeletePrefix:***

This function calls another one, the deleteP() that takes care of all situations that could happen when the user wants to delete a prefix. For instance: if the node to delete is a leaf of the tree, it has to be checked if its parents have prefixes associated and, in that case, they won’t be erased from the tree, otherwise they could also be deleted. That decision is taken at Erase() function that is called in deleteP().

**DeletePrefix(root, prefix)**

if prefix[0] = DFLT\_CHAR

setValue(root, NO\_HOP)

return

end if

for i := 0 to size of prefix

if prefix[i]  0 or 1 then

print Invalid Prefix

return

end if

end for

if prefix[0] = ‘0’

if deleteP(root->left, prefix, 0, prefixLength) = ERASE) then

root->left = null

end if

else

if deleteP(root->right, prefix, 0, prefixLength) = ERASE) then

root->right = null

end if

end if

Note: DFLT\_CHAR is what defines the root next-hop; NO\_HOP means that theres’s not a next-hop for that node

**DeleteP(node, prefix, index, prefixLength)**

ret:= 0

if node = null then

print prefix not found

return 0

end if

if next index = prefixLength and value of node value of root then

if Erase(&node) = 1 then

return ERASE

else

return 0

end if

end if

if prefix[next index] = ‘0’ then

left = getLeft(node)

ret = delete(left, prefix, index+1, prefixLength)

if ret = ERASE then

left = null

else if prefix[next index] = ‘1’ then

right = getRight(node)

ret = delete(right, prefix, index+1, prefixLength)

if ret = ERASE then

right = null

else

print Invalid prefix

return 0

end if

if ret = ERASE and next-hop of node = -1 and node->right = null and node->left =null then

free node

return ERASE

else

return 0

end if

***InsertPrefix: LookUp:***

**InsertPrefix(prefix, nextHop, prefixLength, node, index)**

node := root

if prefixLength > 0

if prefix[index] = ‘1’ then

if getRight(node) = null then

setRight(node, -1)

end if

InsertPrefix(prefix, nextHop, --prefixLength, getRight(node), ++index)

else if prefix[index] = ‘0’ then

if getLeft(node) = null then

setLeft(node, -1)

end if

InsertPrefix(prefix, nextHop, --prefixLength, getLeft(node), ++index)

end if

else

setValue(node, nextHop)

end if

**LookUp(root, address)**

aux:= root

nextHop:= -1

i:=0

while aux null and i <= size of address

if getValue(aux)  -1 and getValue(aux)  -2

nextHop := getValue(aux)

if i := size of the address

break

end if

if address[i] = ‘0’ then

aux = getLeft(aux)

else if address[i] = ‘1’ then

aux = getRight(aux)

else

nextHop := NOT\_VALID

break

end if

i++

end while

return nextHop

The last option that user has is to print to the screen the equivalent prefix table of even length prefixes. In order to do that, the binary prefix tree has to be converted into a two-bit prefix tree using the function BinaryToTwoBit().

This function has to have some verifications, once in the prefix tree there could have odd prefixes which don’t match completely in the new two-bit tree. In that case, if the next even level of the tree doesn’t have next-hop associated, then it inherits the next-hop of its odd parent. If the even level has a next-hop associated, then the next-hop of the child in the new two-bit prefix tree will be overwritten with the correspondent value.

So, this function receives as arguments a pointer to the root of the prefix tree and a pointer to the root of the new two-bit prefix tree. First, gets the value of the left node of the prefix tree and checks if that node is null or not, inside the getValue() function. If the node is null, then there’s no need to translate and create that node to the new two-bit prefix tree. Otherwise, gets the left node of the left node of the prefix tree and evaluates if its value is NO\_HOP, that means that there’s no next-hop associated, or if the node is null and, in one case or another, sets de first child of the new tree with value createNumber. Otherwise, there’s a next-hop associated to that node and so it sets the first child with the value of the node.

Then, it’s the same algorithm for the other three children. The difference is just the node of the prefix tree that has to be evaluated. Basically, this algorithm takes in one node from the prefix tree and its children and for each subtree defines a new level of the two-bit prefix tree.

The algorithm of PrintTableEven() function is very similar to the PrintTable(), just need to print more children, once each node of the two-bit prefix tree has now four children.

**BinaryToTwoBit(BinTree, TwoBit)**

if BinTree = null or TwoBit = null then

return

end if

createNumber = getValue(getLeft(BinTree))

if createNumber ≠ NOT\_CREATE then

one = getLeft(getLeft(BinTree))

if one = null or getValue(one) = NO\_HOP then

setOne(TwoBit, createNumber)

else

setOne(TwoBit, getValue(one))

end if

two = getRight(getLeft(BinTree))

if two = null or getValue(two) = NO\_HOP then

setTwo(TwoBit, createNumber)

else

setOne(TwoBit, getValue(two))

end if

end if

createNumber = getValue(getRight(BinTree))

if createNumber ≠ NOT\_CREATE then

three = getLeft(getRight(BinTree))

if one = null or getValue(three) = NO\_HOP then

setThree(TwoBit, createNumber)

else

setThree(TwoBit, getValue(three))

end if

four = getRight(getRight(BinTree))

if four = null or getValue(four) = NO\_HOP then

setFour(TwoBit, createNumber)

else

setFour(TwoBit, getValue(four))

end if

end if

BinaryToTwoBit(one, getOne(TwoBit))

BinaryToTwoBit(two, getTwo(TwoBit))

BinaryToTwoBit(three, getThree(TwoBit))

BinaryToTwoBit(four, getFour(TwoBit))

* **Complexity of the algorithms:** consider N the number of bits of a prefix and M the number of prefixes of the tree. So the worst case in terms of complexity would be having a full tree with nodes.

***PrefixTree:*** this complexity is about performing the InsertPrefix() function M times, so it has complexity *O(M\*N);*

***PrintTable:*** all nodes have to be checked, so the complexity is *O(;*

***DeletePrefix:*** a specific prefix has to be deleted, so the complexity is *O() = O(N) = O(N);*

***LookUp:*** this function looks for a specific address so the complexity is also *O() = O((N+1)) = O(N);*

***InsertPrefix:*** the longest prefix corresponds to the worst case, so the complexity is *O(N);*

The two-bit prefix tree have, if its completely full, nodes.

***BinaryToTwoBit:***

***PrintTableEven:*** all nodes have to be checked, so the complexity is *O(;*

1. utils.c

In this file there are three simple functions that allow users to interact with the program.

The *menu()* function is based on getting an option from the user and, with a switch, select the respective function from the prefixTree.c file. That function also includes another one, the *showmenu()*, that just prints to the screen all the options available to users. Lastly, the *checkPrefix()* is an auxiliary function that protects the program against incorrect inputs, checks if the user introduces a valid prefix or not.