BINARY SEARCH TREE

presented by : Dr/Lamiaa AlRefaai
Eng/Shaimaa Yousry

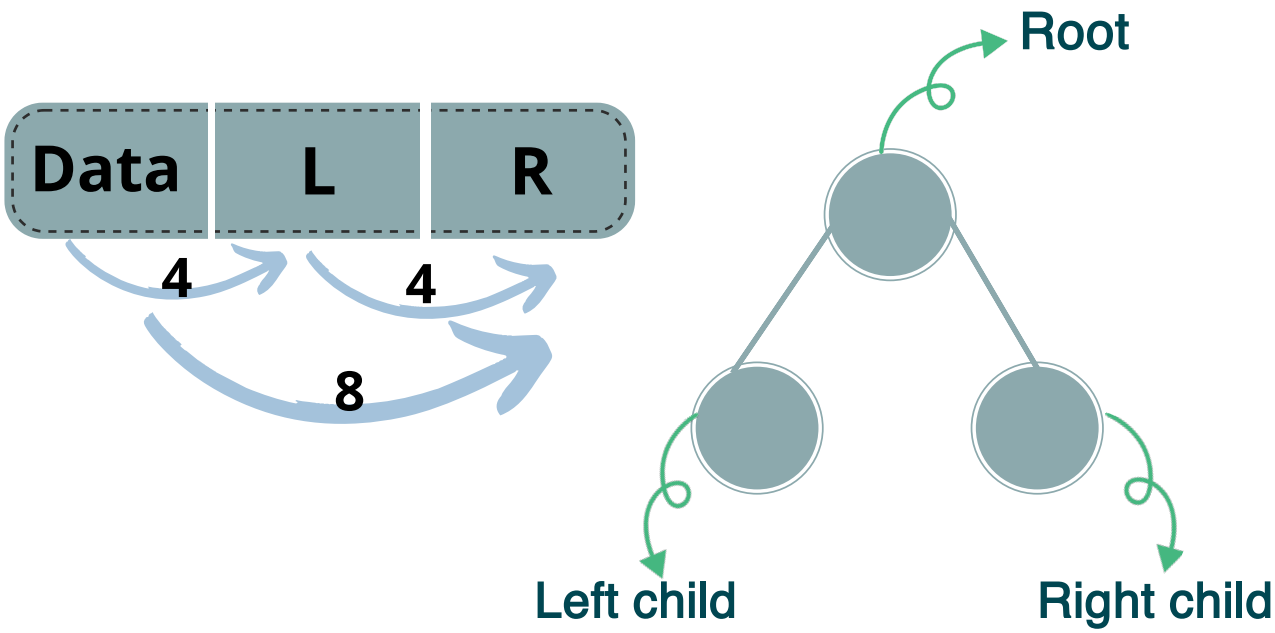
Team Members

1. Abd-ElRahman Osama (select, depth)
2. Adbullah Khaled (delete)
3. Abd-Elhamid Mohamed (insert)
4. Asmaa Mohamed (create, search)
5. Manar Tarek (traverse, user interface)
6. Shrouk Sayed (size, clear, empty)

Content

01 Overview
02 Create
03 Select
04 Insert
05 PreOrder
06 InOrder
07 PostOrder
08 Size
09 Depth
10 Search
11 Delete
12 Clear
13 Empty
14 User Interface

OVERVIEW:

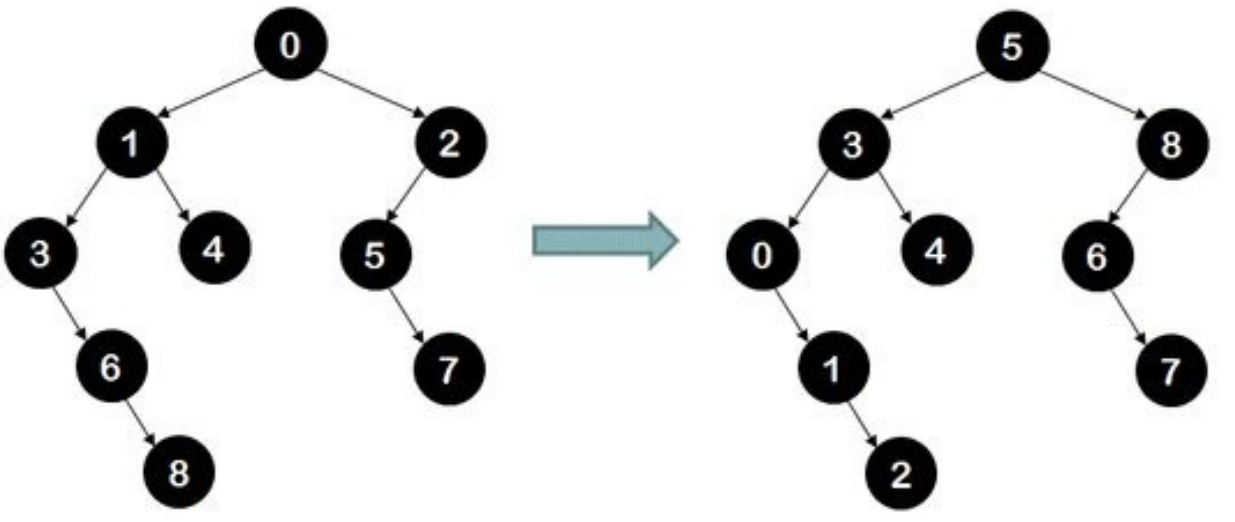


A binary search tree (BST) is a data structure used for organizing a collection of elements. It's a type of binary tree where each node has at most two children, often referred to as the left child and the right child. The key property that distinguishes a binary search tree from an ordinary binary tree is the ordering of elements:

- Binary Tree Structure: Each node in a binary search tree contains a key (or value) along with pointers to its left and right children.
- Ordering Property: For any node in the binary search tree, all elements in its left subtree are less than or equal to the node's key, and all elements in its right subtree are greater than the node's key. This property ensures efficient searching, insertion, and deletion operations.

- **Searching: Binary search trees support efficient searching operations. To find a particular key in the tree, the search algorithm compares the target key with the current node's key. If they are equal, the search is successful. If the target key is less than the current node's key, the algorithm continues searching in the left subtree. If the target key is greater, the algorithm continues searching in the right subtree.**

- **Insertion and Deletion: Insertion and deletion operations in a binary search tree maintain the ordering property. When inserting a new key, the algorithm traverses the tree according to the ordering property to find the appropriate position for the new node. Similarly, when deleting a node, the algorithm maintains the ordering property by rearranging the tree as necessary.**



- **Balanced vs. Unbalanced Trees: In a balanced binary search tree, the heights of the left and right subtrees of any node differ by at most one. This property ensures that the tree remains relatively balanced, which helps maintain efficient searching, insertion, and deletion operations. However, unbalanced trees can occur, especially after a series of insertions or deletions, which may degrade the performance of these operations.**
- **Applications: Binary search trees are commonly used in various applications, including database systems, symbol tables, and implementing various data structures and algorithms like priority queues and sets.**

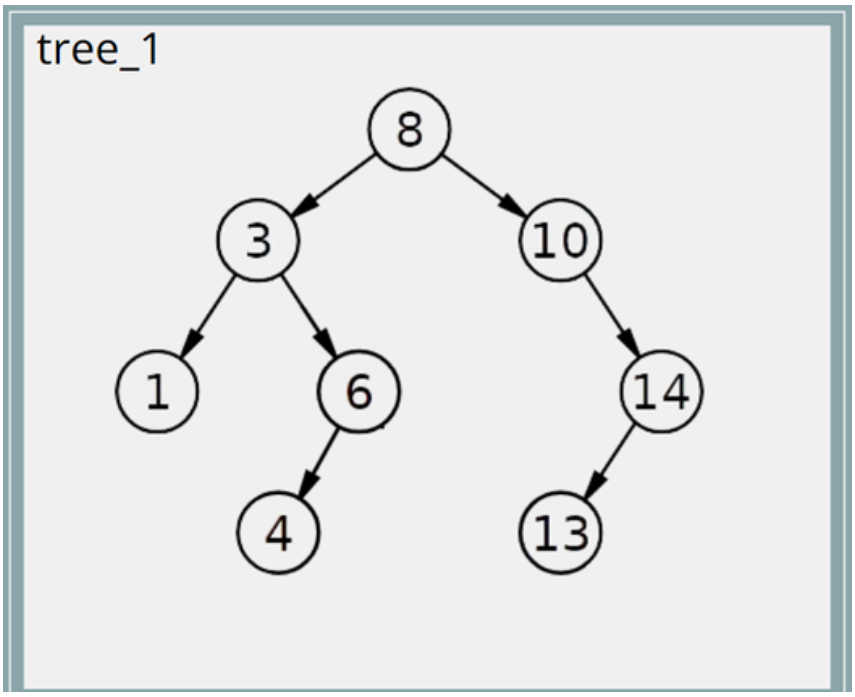
Overall, binary search trees provide an efficient way to organize and search for data, making them a fundamental data structure in computer science and software development. However, ensuring the tree remains balanced is crucial for maintaining optimal performance.

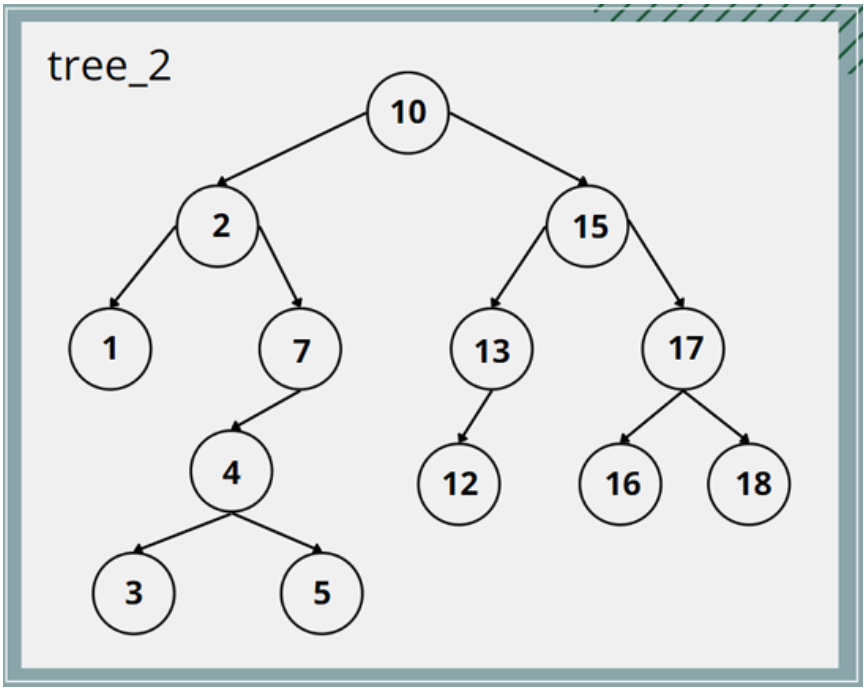
02 - CREATE THE TREE:

- **We can Create more than One tree ^_^**

To create a new node in a data structure and make it the root of a new subtree.

1. **Allocate memory for the new node.**
2. **Store the value of the element to be inserted in the node.**





ASSEMPLY CODE:

```
803  CreateNewTreeRoot:
804          lw $t2 , NumOfTrees
805          addi $t2 , $t2 , 1
806          sw $t2 , NumOfTrees
807          li $v0 , 9
808          li $a0 , 0
809          syscall
810          move $s7 , $v0
```

1. Increment Tree Count:
   - The code starts by loading the current number of trees (NumOfTrees) into $t2.
   - It then increments $t2 by 1 to reflect the creation of a new tree.
   - The updated value is stored back in NumOfTrees.
2. Allocate Memory:
   - Loads the system call code into register $v0.
   - Sets the argument $a0 to 0, which might indicate requesting the minimum amount of memory depending on the system implementation.
   - syscall: Executes the system call to allocate memory.
   - Moves the address of the allocated memory from $v0 to register $s7. This address now points to the new root node.

```
812         sw $s7 , 0($s4)
813         addi $s4 , $s4 , 4
814         addi $t2 , $zero , 1
815         sw $t2 , 0($s4)
816         addi $s4 , $s4 , 4
817         addi $sp , $sp , -4
818         sw $s6 , 0($sp)
819         lw $s6 , FlagTreeRoot
820         addi $s6 , $zero , 1
821         sw $s6 , FlagTreeRoot
822         lw $s6 , 0($sp)
823         addi $sp , $sp , 4
824         jr $ra
```

3.

- **The allocated memory address ($s7) is stored as the root pointer of the new tree.**
- **$s4 is a pointer assumed to keep track of the current memory location within the allocated block. It's incremented by 4 bytes (assuming pointer size) to move to the next memory location.**
- **The value 1 is loaded into $t2 and stored as the initial child count of the new tree.**
- **$s4 is incremented by 4 bytes again.**

4. **Update Tree Root Flag:**

- **Space is allocated on the stack to temporarily store the value of $s6 (potentially a temporary variable).**

- The flag for tree root existence (FlagTreeRoot) is loaded into $s6.
- The flag is set to 1 to indicate that a root now exists.
- The updated flag value is stored back in FlagTreeRoot.
- The saved value of $s6 is restored from the stack.
- Stack space is deallocated.
- The select function is used to select a specific subtree before performing other operations on it.

03 - SELECT FUNCTION:

```
Please enter a command (A,I,P,O,E,S,C,D,Z,Q,R): r
r
Chosse Tree Number : 1
```

ASSEMPLY CODE:

```
834  SelectTreeRoot:
835          addi $sp , $sp , -8
836          sw $t1 , 0($sp)
837          sw $a1 , 4($sp)
838          la $t1 , StartTreeRoot
839          addi $a1 , $a1 , -1
840          mul $a1 , $a1 , 8
841          add $t1 , $t1 , $a1
842          sw $t1 , AboAlaaWeAbdo
843          lw $s7 , 0($t1)
844          lw $t1 , 0($sp)
845          lw $a1 , 4($sp)
846          addi $sp , $sp , 8
847          jr $ra
```

1.Allocates space on the stack for two registers, $t1 and $a1.

Saves the current value of register $t1 on the stack.

Saves the current value of register $a1 on the stack.

2. Calculate Root Pointer Address:

- Loads the address of a variable named StartTreeRoot into register $t1.
- Subtracts 1 from the value in register $a1. This assumes the input index for the tree starts from 1, and subtracting 1 adjusts it to a 0-based index for array access.
- Multiplies the adjusted index ($a1) by 8.
- Adds the calculated offset ($a1) to the base address ($t1) to get the memory address of the desired tree root.

3. Store Address and Restore Registers:

- Stores the calculated root pointer address in a register named AboAlaaWeAbdo.
- l Loads the value from the calculated root pointer address into register $s7.
- lw $t1 , 0($sp): Restores the saved value of $t1 from the stack.
- lw $a1 , 4($sp): Restores the saved value of a1 from the stack.

04 - INSERT

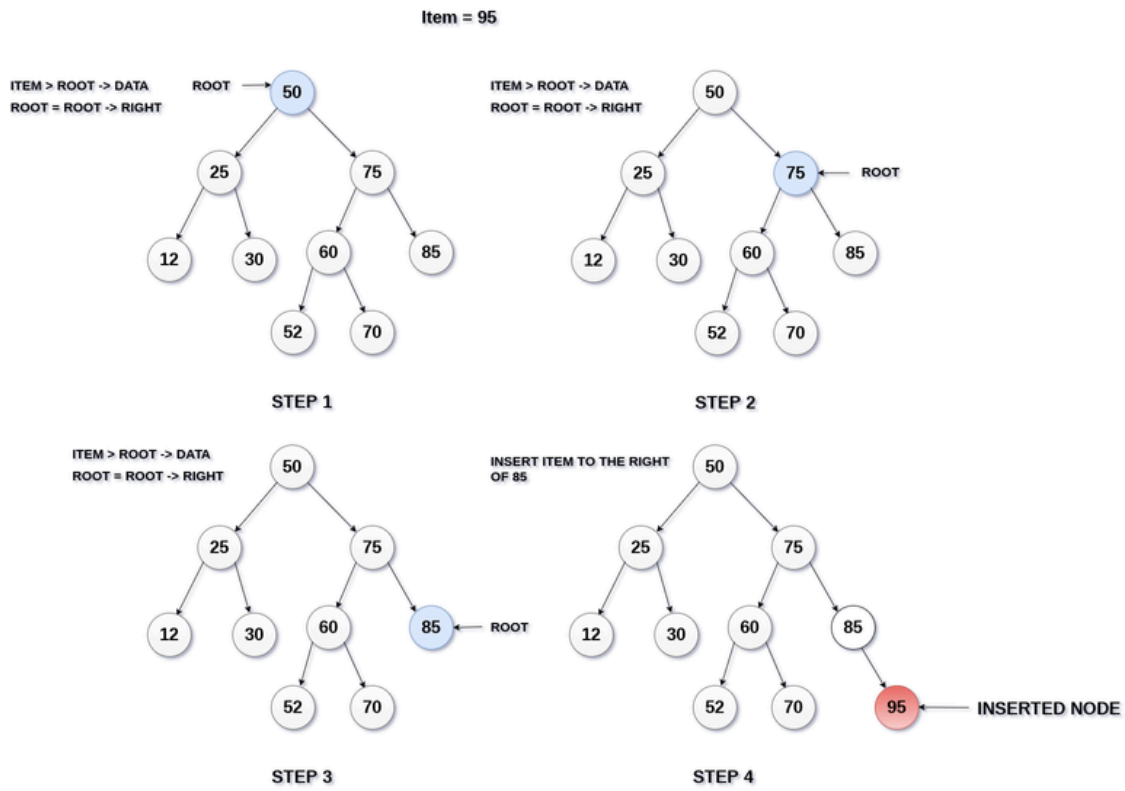
DEFINITION

Implementing the insert function for a binary search tree in MIPS assembly language involves recursively traversing the tree to find the appropriate position for insertion and then adding a new node containing the value at that position. Below is a simplified explanation of how you could approach implementing the insert function in MIPS assembly

CODE OF INSERT FUNCTION AT [C] :

```c
#include <stdio.h>
#include <stdlib.h>

#include "definition.h"



void InsertTree(Tree *pt, TreeEntry *pe){
    if (!*pt){
        *pt=(Tree)malloc(sizeof(TreeNode));
        (*pt)->entry=*pe;
        (*pt)->left=NULL;
        (*pt)->right=NULL;
    }else if ( (*pt)->entry > (*pe) )
        InsertTree(&(*pt)->left, pe);
    else
        InsertTree(&(*pt)->right, pe);
}


```

SIMPLE DIAGRAME OF INSERT :



HERE STEPS SHOW TO MAKE INSERT FUNCTION

1.Start at the Root:

begin at the root node of the BST.
If the root is empty (null), create a new node with the given value and make it the root.

2.Compare Values:
Compare the value to be inserted with the current node’s value.
If the value is less than the current node’s value, move to the left child; otherwise, move to the right child.

3.Traverse Left or Right:

>>If moving left:
> Check if the left child exists.
> If it does, repeat step 2 for the left child.§ If it doesn't exist, create a new node with the given value and attach it as the left child of the current node.

>>If moving right:
> Check if the right child exists.
> If it does, repeat step 2 for the right child.
> If it doesn't exist, create a new node with the given value and attach it as the right child of the current node.

HERE STEPS SHOW TO MAKE INSERT FUNCTION

4-Repeat Until Empty Spot:
>>Continue this process recursively until you find an empty spot (i.e., a null pointer).
>>At that point, insert the new node with the given value.

5-BST Property Maintenance:
>>Ensure that the BST property is maintained:
>>All nodes in the left subtree have values less than the current node's value.
>> All nodes in the right subtree have values greater than the current node's value.

6-Handling Duplicates:
>>Decide how to handle duplicate values (e.g., skip duplicates or update existing nodes).

7-Memory Allocation:
>>Allocate memory dynamically for new nodes (e.g., using syscall to allocate memory).

EXAMPLE:TO INSERT THE VALUE 10 INTO THE BST:

- Start at the root:
  - If the root is empty, create a new node with value 10.
  - Otherwise, compare 10 with the root's value.
  - Since 10 is greater than the root's value, move to the right child (if it exists).
  - Continue this process until an empty spot is found, and insert the new node there.

HERE STEPS SHOW TO MAKE INSERT FUNCTION

1. Node Structure:
   - Each node in the BST contains three components:
     - An integer value (often referred to as the key).
     - A pointer to the left child node (if it exists).
     - A pointer to the right child node (if it exists).

2. Memory Layout:
In memory, each node occupies a contiguous block of space.
The layout typically looks like this

```
+------------------+
|      value       |  # 4 bytes
+------------------+
|      left        |  # 4 bytes (pointer to left child)
+------------------+
|      right       |  # 4 bytes (pointer to right child)
+------------------+
```

THIS IS FIRST PART TO DEFINE THE ROOT OF TREE AND HEAP

```
28      NULL: .word 0              # Representation of NULL pointer.
29     FlagTreeRoot: .word 0         #check if there is root ot not
30     Heap_Start: .word 268697600   #it allocates memory for a single word
```

HERE IS THE IMPLEMENTATION OF INSERT FUNCTION

- insert code :

```
1                  # -----Insert Function Implementation -----#
2   insert:
3
4       # Load address of AboAlaaWeAbdo and move 4 bytes ahead
5       lw $a0, AboAlaaWeAbdo
6       addi $a0, $a0, 4
7       # Load value at the updated address
8       lw $v0, 0($a0)
9       # If value is not zero, jump to insertOp
10      bne $v0, $zero, insertOp
11      # Store $a2 in the memory address pointed by $a3
12      sw $a2, 0($a3)
13      # Set $v0 to 1
14      addi $v0, $zero, 1
```

```
    # Store 1 in the memory address pointed by $a0
    sw $v0, 0($a0)
    # Jump back to the return address
    jr $ra
```

>>Here in this part use to store all pointer and value in Stack
>>ALLOcate memory for a new node

```
                # ----Save used pointer-reg in the stack-----#
insertOp:
    subu $sp, $sp, 28
    # Store values in the stack
    sw $a3, 0($sp)
    sw $s2, 4($sp)
    sw $s3, 8($sp)
    sw $a2, 12($sp)
    sw $t2, 16($sp)
    sw $t3, 20($sp)
    sw $s6, 24($sp)
    # Allocate memory for a new node
    li $v0, 9
    li $a0, 12
```

1. Allocate Memory for a New Node:
    - The instruction li $v0, 9 sets the system call code for memory allocation (specifically, allocating memory on the heap).
    - The value 12 in $a0 specifies the size of memory to allocate for a new node (assuming each node occupies 12 bytes).
    - The syscall instruction allocates memory and returns the address of the allocated block in register $v0.
    - The address of the new node is then stored in register $s2.
2.

Check if Tree Is Empty:
The value of the flag indicating whether the tree is empty is loaded into register $s6 (presumably from some global variable or memory location).
If the flag is equal to 1, it means the tree is empty (no root node exists).
In this case, the program branches to the label insertRoot to create the root node.

```
31      # Allocate memory for a new node
32      li $v0, 9
33      li $a0, 12
34      syscall
35      move $s2, $v0       # $s2 points to the new node
36
37      # Check if tree is empty, make the Root Node
38      lw $s6, FlagTreeRoot
39      beq $s6, 1, insertRoot
40
41      # Start $s3 points to the Root Node
42      move $s3, $a3
```

1. insert Loop: This label marks the beginning of a loop (presumably within an insertion function for a binary search tree).
2. · lw $t2, 0($s3):
3. This instruction loads the value of the current node (retrieved from memory) into register $t2.
4. $s3 likely holds the address of the current node.
5. · Conditional Branches:
6. The following instructions involve conditional branches based on the input data ($a2) and the current node’s value ($t2).

7·beq $a2, $t2, Repeat:
8.If the value of the input data is equal to the current node’s value, the program branches to the label Repeat.
9.This suggests that duplicate values are handled by repeating the insertion process (e.g., inserting the same value again).
10.blt $a2, $t2, CheckLeft:
11.If the input data value is less than the current node’s value, the program branches to the label CheckLeft.
12.This indicates that the program should traverse to the left child of the current node.
13.j CheckRight

```
45      insertLoop:
46  # $t2 holds the value of Data of the current node
47          lw $t2, 0($s3)
48  # If the value of the Input Data is less than the current node, Traverse
49          beq $a2, $t2, Repeat
50          blt $a2, $t2, CheckLeft
51  # Else Traverse Right
52          j CheckRight
```

1. Load Word (lw):
2. The instruction lw $t2, 4($s3) loads a word from memory into register $t2.
3. It fetches the value stored at the memory address $s3 + 4 and places it in $t2.
4. Branch if Equal (beq):
5. The next line, beq $t2, $zero, insertLeft, checks if the value in $t2 is equal to zero.
6. If it is, the program jumps to the insertLeft label (presumably to insert a new node as the left child).

7.Move (move):
8.If the value in $t2 is not zero (i.e., the left child is not null), the program executes the move $s3, $t2 instruction.
9.This instruction copies the value from $t2 to $s3, effectively moving to the left child.
10.Jump (j):
11.Finally, the j insertLoop instruction unconditionally jumps to the insertLoop label.
12.This label likely represents the continuation of the insertion process

```
63      CheckRight:
64  # If the right child is null, insert the new node as the right child
65          lw $t2, 8($s3)
66          beq $t2, $zero, insertRight
67  # If Node is not NULL, move to the left child
68          move $s3, $t2
69  # Jump to behave this Right Child as the Root of sub tree now
70          j insertLoop
```

1. Store Word (sw):
2. The instruction sw $s2, 4($s3) stores the value in register $s2 into memory.
3. It saves the value at the memory address $s3 + 4.
4. ·Store Word (sw) (again):
5. The next line, sw $a2, 0($s2), stores the value in register $a2 into memory.
6. It places the value at the memory address $s2.
7. ·Move (move):
8. The instruction move $s7, $s2 copies the value from $s2 to $s7.
9. ·Restore Values:

10.The subsequent lines restore various registers from the stack.
11.These registers were likely saved earlier (e.g., during a function call) and are now being restored.
12·Adjust Stack Pointer (addu):
13.The instruction addu $sp, $sp, 28 adjusts the stack pointer by adding 28 bytes.
14.This typically deallocates space on the stack that was previously reserved for local variables.
15·Jump to Return Address (jr):
16.Finally, the jr $ra instruction jumps to the return address stored in register $ra.
This is used to exit the current function and return control to the calling function

```
72      insertLeft:
73          sw $s2, 4($s3)      # Make $s2 now point to Left Address
74          sw $a2, 0($s2)      # Store the Input Data to the Data Place
75          move $s7, $s2
76          # Restore Values
77          lw $a3, 0($sp)
78          lw $s2, 4($sp)
79          lw $s3, 8($sp)
80          lw $a2, 12($sp)
81          lw $t2, 16($sp)
82          lw $t3, 20($sp)
83          lw $s6, 24($sp)
84          addu $sp, $sp, 28
85          jr $ra
```

1. ·
2. Store Word (sw):
3. The instruction sw $s2, 8($s3) stores the value in register $s2 into memory.
4. It saves the value at the memory address $s3 + 8.
5. ·Store Word (sw) (again):

6.The next line, sw $a2, 0($s2), stores the value in register $a2 into memory.
7.It places the value at the memory address $s2.
8·Move (move):
9.The instruction move $s7, $s2 copies the value from $s2 to $s7.
10·Restore Values:
11.The subsequent lines restore various registers from the stack.
12.These registers were likely saved earlier (e.g., during a function call) and are now being restored.
13·Adjust Stack Pointer (addu):
14.The instruction addu $sp, $sp, 28 adjusts the stack pointer by adding 28 bytes.
15.This typically deallocates space on the stack that was previously reserved for local variables.
16·Jump to Return Address (jr):
17.Finally, the jr $ra instruction jumps to the return address stored in register $ra.
This is used to exit the current function and return control to the calling function.

```
87      insertRight:
88          sw $s2, 8($s3)      # Make $s2 now point to Right Address
89          sw $a2, 0($s2)      # Store the Input Data to the Data Place
90          move $s7, $s2
91          # Restore Values
92          lw $a3, 0($sp)
93          lw $s2, 4($sp)
94          lw $s3, 8($sp)
95          lw $a2, 12($sp)
96          lw $t2, 16($sp)
97          lw $t3, 20($sp)
98          lw $s6, 24($sp)
99          addu $sp, $sp, 28
100         jr $ra
```

05 - PREORDER

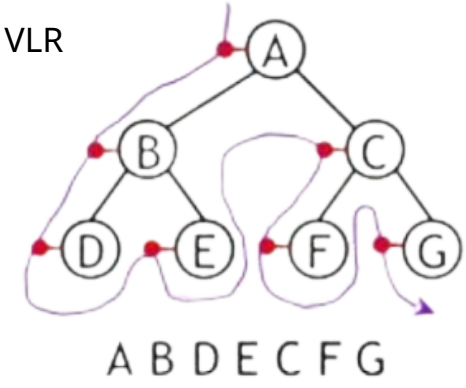
C CODE:

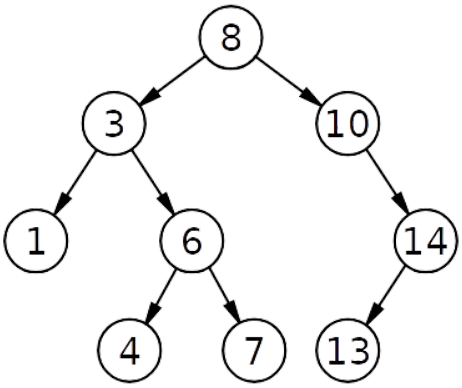
```c
#include <stdio.h>
#include <stdlib.h>

#include "definition.h"

void Preorder(Tree *pt, void(*pvisit)(TreeEntry)){
    if (*pt){
        (*pvisit)((*pt)->entry);
        Preorder(&(*pt)->left, pvisit);
        Preorder(&(*pt)->right, pvisit);
    }
}
```

1.void Preorder(Tree *pt, void(*pvisit)(TreeEntry)): This line defines a function named Preorder. It takes two parameters:

- **pt: A pointer to a pointer to a Tree structure. This is likely the root of the binary tree or a subtree.**
- **pvisit: A pointer to a function that takes a TreeEntry parameter and returns void. This function is called on each node during the traversal.**
- **if (*pt) { ... }: This conditional statement checks if the pointer *pt is not null, indicating that there is a node at the current position in the tree.**
- **(*pvisit)((*pt)->entry);: This line calls the function pointed to by pvisit, passing the entry of the current node as an argument. This is typically where the visiting action is performed on the current node, such as printing its value.**
- **Preorder(&(*pt)->left, pvisit);: This line recursively calls the Preorder function on the left child of the current node, if it exists.**
- **Preorder(&(*pt)->right, pvisit);: Similarly, this line recursively calls the Preorder function on the right child of the current node, if it exists.**

VLR





PreOrder: 8 3 1 6 4 7 10 14 13

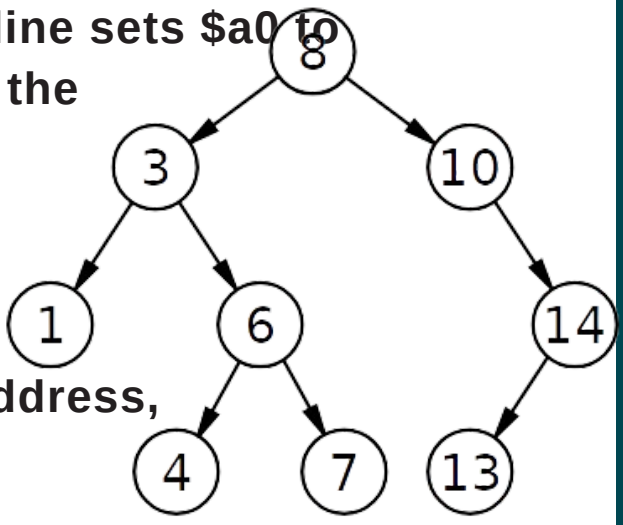
ASSEMPLY CODE:

```
616 preOrder:
617     lw $a0 , AboAlaaWeAbdo
618     addi $a0 , $a0 , 4
619     lw $a0 , 0($a0)
620     bne $a0 , $zero  , preOrderOp
621     move $a0, $zero
622     jr $ra
623 preOrderOp:
624                     move $a0, $a3
625                     # if(bPtr == NULL)
626                     bne $a0, $0, preOrderRecurse
627                     jr $ra
628 preOrderRecurse:
629                     addi, $sp, $sp, -8                    # Allocate 2 re
630                     sw $ra, 0($sp)
631                     sw $a0, 4($sp)
632
633                      jal printSmall
634
635                     lw $a3,4($a3)                                 # curr
636                     jal preOrderOp
637
638                     lw $a0, 4($sp)
639                     move $a3, $a0                        # $s7 = $a0
640                     lw $a3, 8($a3)                                # curr
641                     jal preOrderOp
642                     lw $ra, 0($sp)
643                     addi $sp, $sp, 8                      # Free the 2 re
644                     jr $ra
645 # End of preOrder traversal
```

```
688 printSmall:
689
690                      lw $a0, 0($a3)
691                      li $v0, 1
692                      syscall
693
694                      li $v0, 4
695                      la $a0,format
696                      syscall
697
698                      jr $ra
699
700 # End of printSmall
```

Let's break down the code into sections and explain what each part does:

1. **preOrder:**
   - **This appears to be the starting label for the preorder traversal subroutine.**
2. **Loading Current Node:**
   - **lw $a0, AboAlaaWeAbdo: This line loads the address of the current node into register $a0.**
   - **addi $a0, $a0, 4: This line adds an offset of 4 bytes to $a0, likely to access a field in the node structure.**
   - **lw $a0, 0($a0): This line loads the value of the current node into $a0.**
3. **Check for NULL:**
   - **bne $a0, $zero, preOrderOp: This branch instruction checks if $a0 is not equal to zero, which would indicate that the current node is not NULL. If it's not NULL, it jumps to the preOrderOp label.**
4. **Handle NULL Node:**
   - **move $a0, $zero: This line sets $a0 to zero, likely indicating that the current node is NULL.**



- **jr $ra: This instruction jumpsback to the return address, indicating the end of the subroutine.**

5.preOrderOp:

This label seems to handle the case where the current node is not NULL.

6.**Recursive Preorder Traversal:**

addi $sp, $sp, -8: This line allocates space on the stack for two registers.

- **sw $ra, 0($sp): This stores the return address ($ra) onto the stack.**
- **sw $a0, 4($sp): This stores the current node pointer ($a0) onto the stack.**
- **jal printSmall: This jumps to the printSmall subroutine to print the value of the current node.**
- **lw $a3, 4($a3): This loads the left child of the current node into $a3.**
- **jal preOrderOp: This recursively calls preOrderOp to traverse the left subtree.**
- **lw $a0, 4($sp): This reloads the original value of $a0 (the parent node).**
- **move $a3, $a0: This moves the parent node pointer into $a3.**
- **lw $a3, 8($a3): This loads the right child of the parent node into $a3.**
- **jal preOrderOp: This recursively calls preOrderOp to traverse the right subtree.**
- **lw $ra, 0($sp): This loads the return address back from the stack.**
- **addi $sp, $sp, 8: This frees the space allocated on the stack.**
- **jr $ra: This returns to the caller.**

7.**printSmall:**

- **This subroutine appears to print the value of the current node.**
- **This code recursively traverses the binary tree in a preorder fashion, printing the value of each node.**

06 - INORDER

C CODE:

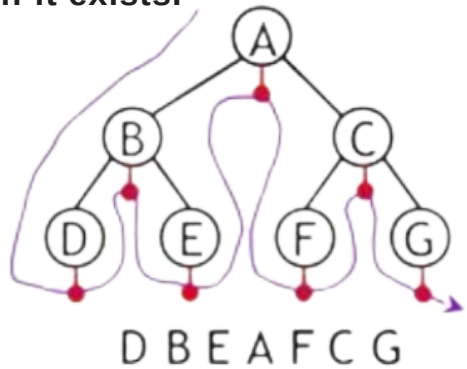
```c
#include <stdio.h>
#include <stdlib.h>
#include "definition.h"

void Inorder(Tree *pt, void(*pvisit)(TreeEntry)) {
    if (*pt) {
        Inorder(&((*pt)->left), pvisit);
        (*pvisit)((*pt)->entry);
        Inorder(&((*pt)->right), pvisit);
    }
}
```

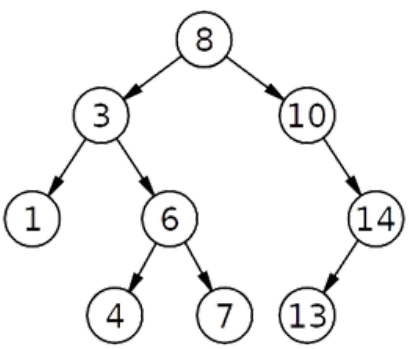
1.Function Definition:

- void Inorder(Tree *pt, void(*pvisit)(TreeEntry)): This line defines the Inorder function. It takes two parameters:
- pt: A pointer to a pointer to a Tree structure, likely the root of the binary tree or a subtree.
- pvisit: A pointer to a function that takes a TreeEntry parameter and returns void. This function is called on each node during the traversal.

2.Traversal Logic:

- if (*pt) { ... }: This conditional statement checks if the pointer *pt is not null, indicating that there is a node at the current position in the tree.
- Inorder(&((*pt)->left), pvisit);: This line recursively calls the Inorder function on the left child of the current node, if it exists.
- (*pvisit)((*pt)->entry);: This line calls the function pointed to by pvisit, passing the entry of the current node as an argument. This is typically where the visiting action is performed on the current node, such as printing its value.
- Inorder(&((*pt)->right), pvisit);: Similarly, this line recursively calls the Inorder function on the right child of the current node, if it exists.





InOrder: 1 3 4 6 7 8 10 13 14

ASSEMPLY CODE:

```
#InOrder
InOrder:
                    lw $a0 , AboAlaaWeAbdo
    addi $a0 , $a0 , 4
    lw $a0 , 0($a0)
    bne $a0 , $zero  , InOrderOp
    move $a0, $zero
    jr $ra
InOrderOp:
                    move $a0, $a3

                    # if(bPtr == NULL)
                    bne $a0, $0, InOrderRecurse
                    jr $ra
InOrderRecurse:
                    addi, $sp, $sp, -8                      # Allocate 2
                    sw $ra, 0($sp)
                    sw $a0, 4($sp)
                    lw $a3,4($a3)                                   # curr
                    jal InOrderOp

                    lw $a0, 4($sp)
                    move $a3, $a0                       # $s7 = $a0
                    jal printSmall                              # Call
                    lw $a3, 8($a3)                              # curr
                    jal InOrderOp
                    lw $ra, 0($sp)
                    addi $sp, $sp, 8                        # Free the 2
                    jr $ra
# End of InOrder traversal
```

1. **Initialization:**
    - **The code begins by loading the address of the root node of the binary tree into register $a0.**
    - **It then moves to the left child of the root node by adding an offset of 4 to the address stored in $a0.**
    - **The value of the left child node is loaded into register $a0.**
2. **Checking for NULL:**
    - **After loading the left child node's value into $a0, the code checks if the value is not zero. This check is to see if there actually is a left child node.**
    - **If $a0 is not zero (meaning there is a left child), the code jumps to the label InOrderOp.**
    - **If $a0 is zero (indicating there is no left child), the code sets $a0 to zero and returns to the caller using jr $ra instruction.**
3. **InOrderOp:**
    - **In this section, the code copies the value of register $a3 (which presumably holds the value of the current node) into $a0. This is likely done for further processing or printing.**

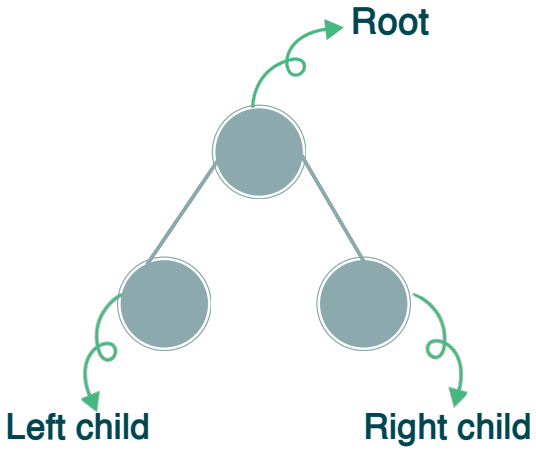
4.Recursion for Traversal:

- **The code enters into a recursive function InOrderRecurse if there's a left child node.**
- **InOrderRecurse allocates space on the stack for two registers ($ra and $a0) using addi $sp, $sp, -8. This is done to preserve their values during recursion.**
- **It saves the return address and the value of the current node on the stack.**

- It then traverses to the left child node by loading the address of the left child node into $a3 using lw $a3, 4($a3), presumably from the current node's left pointer.
- The code then recursively calls InOrderOp, effectively traversing the left subtree.
- After processing the left subtree, it restores the original value of the current node from the stack into $a0.
- It then likely performs some operation with the current node, such as printing its value using a function called printSmall.
- Next, it traverses to the right child node by loading the address of the right child node into $a3 using lw $a3, 8($a3), presumably from the current node's right pointer.
- Again, it recursively calls InOrderOp to traverse the right subtree.
- Finally, it restores the return address and frees the space allocated on the stack for the registers.

5. Conclusion:

- The code effectively traverses a binary tree in an inorder manner, processing each node and its subtrees recursively. This approach ensures that nodes are visited in non-decreasing order (hence the name "inorder" traversal), making it useful for tasks such as printing the contents of a binary search tree in sorted order.



07- POSTORDER

C CODE:

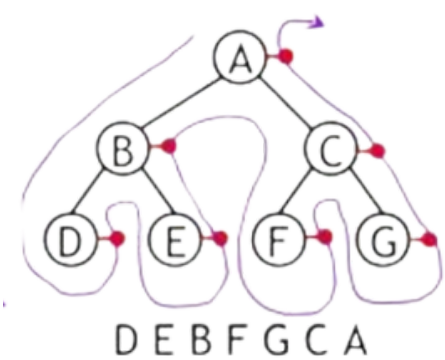
```c
#include <stdio.h>
#include <stdlib.h>
#include "definition.h"

void Postorder(Tree *pt, void(*pvisit)(TreeEntry)){
    if (*pt){
        Postorder(&((*pt)->left), pvisit);
        Postorder(&((*pt)->right), pvisit);
        (*pvisit)((*pt)->entry);
    }
}
```

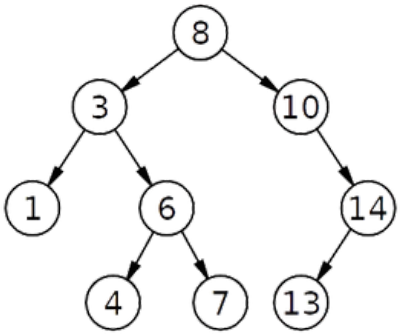
1.Function Definition:

- **void Postorder(Tree *pt, void(*pvisit)(TreeEntry)): This line defines the Postorder function. It takes two parameters:**
- **pt: A pointer to a pointer to a Tree structure, likely the root of the binary tree or a subtree.**
- **pvisit: A pointer to a function that takes a TreeEntry parameter and returns void. This function is called on each node during the traversal.**

2.Traversal Logic:

- **if (*pt) { ... }: This conditional statement checks if the pointer *pt is not null, indicating that there is a node at the current position in the tree.**
- **Postorder(&((*pt)->left), pvisit);: This line recursively calls the Postorder function on the left child of the current node, if it exists.**
- **Postorder(&((*pt)->right), pvisit);: Similarly, this line recursively calls the Postorder function on the right child of the current node, if it exists.**
- **(*pvisit)((*pt)->entry);: This line calls the function pointed to by pvisit, passing the entry of the current node as an argument. This is typically where the visiting action is performed on the current node, such as printing its value.**





PostOrder: 1 4 7 6 3 13 14 10 8

ASSEMPLY CODE:

```
postOrder:
    lw $a0 , AboAlaaWeAbdo
    addi $a0 , $a0 , 4
    lw $a0 , 0($a0)
    bne $a0 , $zero  , postOrderOp
    move $a0, $zero
    jr $ra
postOrderOp:
    move $a0, $a3           # a0 = $s0
    # if (bPtr == NULL)
    bne $a0, $0, postOrderRecurse
    jr $ra                  # Return to caller
postOrderRecurse:
    addi $sp, $sp, -8       # Allocate 2 registers to stack
    sw $ra, 0($sp)          # $ra is the 1st
    sw $a0, 4($sp)          # $a0 is the 2nd
    lw $a3, 4($a3)        # curr = curr->left
    jal postOrderOp           # Call postOrder
    lw $a0, 4($sp)          # Retrieve original $a0
    move $a3, $a0           # $s0 = $a0
    lw $a3, 8($a3)        # curr = curr->right
    jal postOrderOp           # Call postOrder
    lw $a0, 4($sp)          # Retrieve original $a0
    move $a3, $a0           # $s0 = $a0
    jal printSmall          # Call printSmall
    lw $ra, 0($sp)          # Retrieve original $ra
    addi $sp, $sp, 8        # Free the 2 register stack spaces
    jr $ra
 #End of postOrder traversal
```

1. Initialization:
   - It loads the address of the root node of the binary tree into register $a0.
     - Then, it moves to the left child of the root node by adding an offset of 4 to the address stored in $a0.
   - The value of the left child node is loaded into register $a0.
2. Checking for NULL:
   - After loading the left child node's value into $a0, the code checks if the value is not zero. This check is to see if there actually is a left child.
   - If $a0 is not zero (indicating there is a left child), the code jumps to the label postOrderOp.
   - If $a0 is zero (indicating there is no left child), the code sets $a0 to zero and returns to the caller using jr $ra instruction.

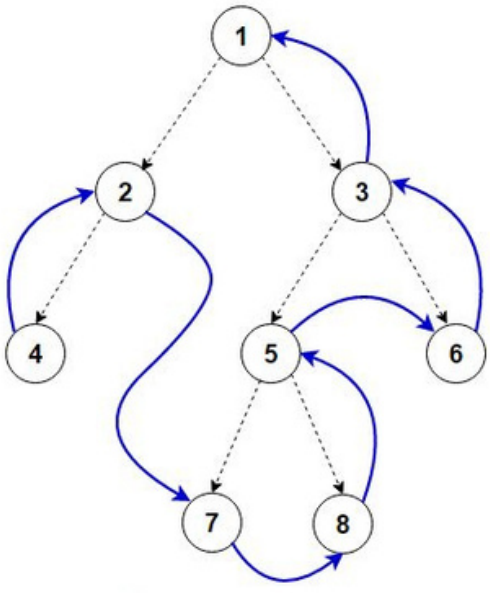
ASSEMPLY CODE:

3.postOrderOp:

- In this section, the code copies the value of register $a3 (presumably holding the value of the current node) into $a0.

4.Recursion for Traversal:

- The code enters into a recursive function postOrderRecurse if there's a left child node.
- postOrderRecurse allocates space on the stack for two registers ($ra and $a0) using addi $sp, $sp, -8. This is done to preserve their values during recursion.
- It saves the return address and the value of the current node on the stack.
- It then traverses to the left child node by loading the address of the left child node into $a3 using lw $a3, 4($a3), presumably from the current node's left pointer.
- The code then recursively calls postOrderOp, effectively traversing the left subtree.
- After processing the left subtree, it restores the original value of the current node from the stack into **$a0**.
- It then traverses to the right child node by loading the address of the right child node into **$a3** using **lw $a3, 8($a3)**, presumably from the current node's right pointer.
- Again, it recursively calls **postOrderOp** to traverse the right subtree.
- After processing both left and right subtrees, it prints the value of the current node using **printSmall**.
- Finally, it restores the return address and frees the space allocated on the stack for the registers.



Postorder: 4, 2, 7, 8, 5, 6, 3, 1

08- SIZE

C CODE:

```
21
22  int TreeSize(Tree *pt){
23      if (!*pt)
24          return 0;
25      return (1+TreeSize(&(*pt)->left)+TreeSize(&(*pt)->right));
26  }
27
```

This function, ‘TreeSize’ is likely intended to compute the number of nodes in a binary tree.
1.The function takes a pointer to a Tree structure as input.
2.It checks if the pointer is null (!*pt). If it is, it means the tree is empty,so it returns 0.
3. If the pointer is not null, it recursively calculates the size of the left sub tree (TreeSize(&(*pt)->left)) and the size of the right sub tree (TreeSize(&(*pt)->right)), and adds 1 to account for the current node.
4. The sizes of the left and right sub trees are calculated recursively using the same function.
5. Finally, the function returns the sum of 1 (for the current node) and the sizes of its left and right sub trees.

ASSEMBLY CODE:

This function follows the pre-order traverse.

```
size:
        lw $a0 , AboAlaaWeAbdo
    addi $a0 , $a0 , 4
    lw $a0 , 0($a0)
    bne $a0 , $zero  , sizeop
    move $v0,$zero
    move $a0, $zero
    jr $ra
```

This ‘size’ function is to calculate the number of nodes of a binary tree. And this is by adding offset of 4 bytes to the address of the root to access the left child pointer, Then It checks if the address of the left child node is not null and It branches to the ‘sizeop’ function if It’s not null, but if It’s null, This means that the flag is 0 so the tree is not activated and then the tree is empty so the size is 0, Then It returns to the caller.

```
sizeop:
    move $v0,$zero
    move $a0, $a3              # $a0 = $a3
    # if(bPtr == NULL)
    bne $a0, $0, sizeRecurse

    jr $ra                  # Return to caller
```

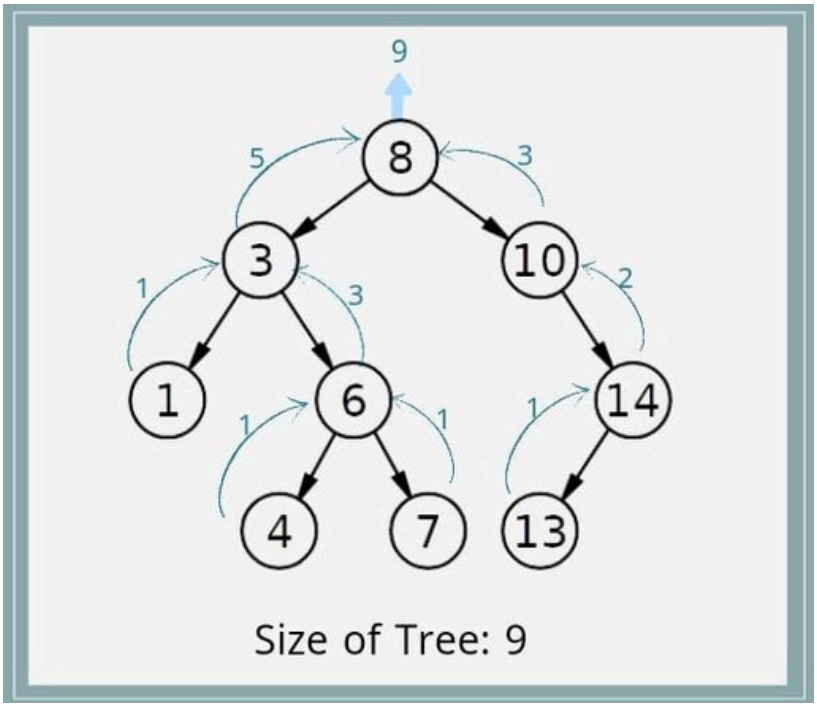
This ‘sizeop’ function sets up for the recursive traversal of a binary tree’s left sub tree by checking if the left child is null. If the left child is not null, it will continue the traversal by branching to ‘sizeRecurse’ function. Otherwise, it returns to the caller.

```
sizeRecurse:
    addi $sp, $sp, -8        # Allocate 2 registers to stack
    sw $ra, 0($sp)           # $ra is the first register
    sw $a0, 4($sp)           # $a0 is the second register

    jal increaseSize             # Call printSize

    lw $a3, 4($a3)           # curr = curr->left
    jal sizeop             # Call size

    lw $a0, 4($sp)           # Retrieve original value of $a0
    move $a3, $a0            # $s0 = $a0
    lw $a3, 8($a3)           # curr = curr->right
    jal sizeop             # Call size

    move $v0, $s5
    lw $ra, 0($sp)           # Retrieve original return address
    addi $sp, $sp, 8         # Free the 2 register stack spaces
    jr $ra               # Return to caller
```

This ‘sizeRecurse’ function is to recursively traverse the binary tree to calculate its size. First, We allocate space on the stack for storing two registers and save the return address and the current node pointer, Then We increase the size counter by calling the ‘increaseSize’ function, Then We traverse left sub tree by loading the left child pointer of the current node and recursively calling the ‘sizeop’ function. We restore the address of the current node from the stack. Then We traverse right sub tree by loading the left child pointer of the current node and recursively calling the ‘sizeop’ function. We free the space allocated on the stack and return to the caller of the function.

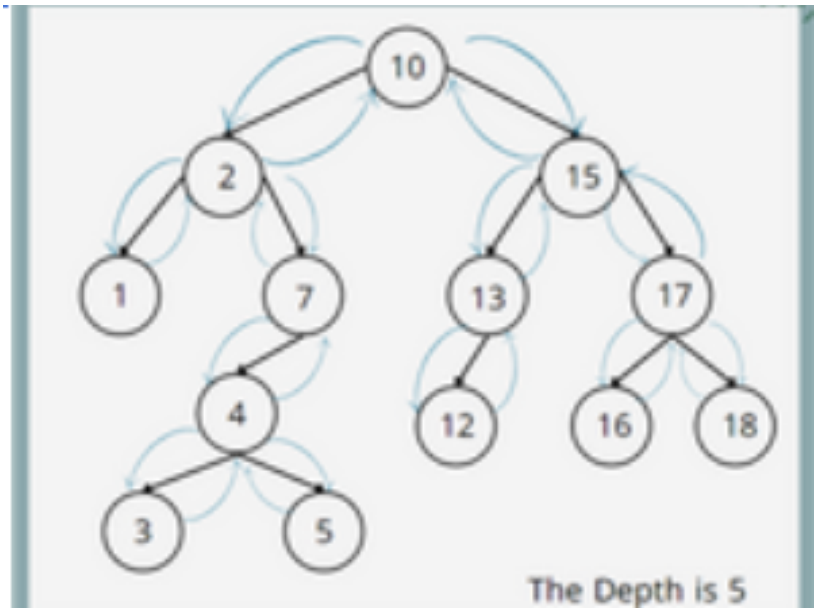
```
increaseSize:

    addi $s5, $s5, 1

    jr $ra                  # Return to caller
```

The ‘increaseSize’ function is to increment a counter used to keep track of the size of the binary tree.



09-DEPTH

- The Depth is the number of nodes along the longest path from the root node down to the farthest leaf node.



C CODE:

```c
int TreeDepth(Tree *pt){
    if (!*pt)
        return 0;
     int a=TreeDepth(&(*pt)->left);
     int b=TreeDepth(&(*pt)->right);
     return (a>b)? 1+a : 1+b;
}

```

ASSEMPLY CODE:

```
TreeDepth:

        # Store in Stack
        addi $sp , $sp , -16
        sw $ra , 0($sp)    # Return Call
        sw $a0 , 4($sp)  # Need in Workable Pointer
        sw $s1 , 8($sp)  # Need in Result : A
        sw $s1 , 12($sp) # Need in Result : B
        beq $a0 , 0 , Base_Case
        j Rec_Depth

Base_Case:
        # Base Case
        addi $v0 , $zero , 0
        addi $sp  $sp , 16
        jr $ra

```

1. Allocates space on the stack for 4 registers: $ra (return address), $a0 (workable pointer), $s1 (result A), and $s2 (result B).

2. Checks if the $a0 pointer is null (indicating an empty tree).

- If true, it jumps to the Base_Case label.

```
867 Rec_Depth:
868         # Move Left
869         lw $a0 , 4($a0)
870         # Call TreeDepth
871         jal TreeDepth
872         # Load Workable Again
873         lw $a0 , 4($sp)
874         # Work on Result : A
875         move $s1 , $v0
876         # Move Right
877         lw $a0 , 8($a0)
878         # Call TreeDepth
879         jal TreeDepth
880         # Work on Result : B
881         move $s2 , $v0
882
883         # Operation
884
885         ble $s1 , $s2 , Increase_B       # If A <= B , Increase_B
886         bgt $s1 , $s2 , Increase_A # If A > B , Increase_A
887
887
888           Restore:
889           # Restore from Stack
890           lw $ra , 0($sp)    # Return Call
891           lw $a0 , 4($sp)  # Need in Workable Pointer
892           lw $s1 , 8($sp)  # Need in Result : A
893           lw $s1 , 12($sp) # Need in Result : B
894           addi $sp , $sp , 16
895           jr $ra
896
897           Increase_A:
898                   addi $s1 , $s1 , 1
899                   move $v0 , $s1
900                   j Restore
901           Increase_B:
902                   addi $s2 , $s2 , 1
903                   move $v0 , $s2
904                   j Restore
905
```

- Sets the return value ($v0) to 0 (depth of an empty tree is 0).
- Deallocates space for the registers on the stack.
- Jumps back to the calling function with the calculated depth (0).

3. Else, it jumps to the Rec_Depth function.

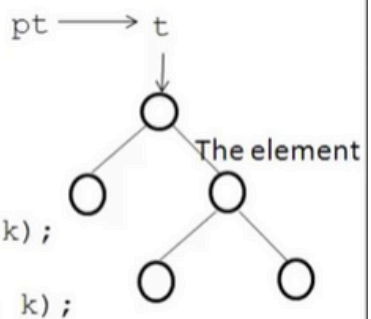
4. Loads the address of the left child from the current node ($a0) into $a0.
jal TreeDepth: Recursively calls the TreeDepth function with the left child pointer as the new argument.
This effectively calculates the depth of the left subtree.
Loads the saved initial value of $a0 (root pointer) back from the stack.
5. Restore Workable Pointer and Work on Result A:
move $s1 , $v0: Moves the returned depth of the left subtree (from the recursive call) to $s1 (result A).
6. Loads the address of the right child from the current node ($a0) into $a0.
jal TreeDepth: Recursively calls the TreeDepth function with the right child pointer as the new argument.
This effectively calculates the depth of the right subtree.
7. Restore Workable Pointer and Work on Result B:
move $s2 , $v0: Moves the returned depth of the right subtree (from the recursive call) to $s2 (result B).
8. compares the depths from the left and right subtrees and chooses the larger one and print it.

10 - SEARCH

Searching begins by examining the root node. If the tree is null, the key being searched for does not exist in the tree. Otherwise, if the key equals that of the root, the search is successful and the node is returned. If the key is less than that of the root, the search proceeds by examining the left subtree. Similarly, if the key is greater than that of the root, the search proceeds by examining the right subtree. This process is repeated until the key is found or the remaining subtree is null. If the searched key is not found after a null subtree is reached, then the key is not present in the tree.

C CODE:

```
int FindItemTreeRec(Tree *pt, TreeEntry *pe, KeyType k){
   if (!*pt)
      return 0;
   if (EQ((*pt)->entry.key, k)){
      *pe=(*pt)->entry;
       return 1;
   }
   if (LT(k, (*pt)->entry.key)))
      return FindItemTreeRec(&(*pt)->left, pe, k);
   else
      return FindItemTreeRec(&(*pt)->right, pe, k);
}
```



ASSEMPLY CODE:

```
1   Search:
2       # Stack Store
3        subu $sp, $sp, 24
4        sw $a2, 0($sp) # root
5        sw $a3, 4($sp) # data
6        sw $s0, 8($sp)
7        sw $t0, 12($sp)
8        sw $t1, 16($sp)
9        sw $t7, 20($sp)
10
11      # Conditions
12      # Check if Root is not NULL
13        beq $a2 , 0 , end
14        beq $s0 , 1 , end
15
16
17       # Operations
18       # Load Value of Data in $t0
19          lw $t0 , 0($a2)
20          beq $t0 , $a3 , Found
```

Stack Store

- subu $sp, $sp, 24: This instruction subtracts 24 from the stack pointer ($sp). This reserves space on the stack for storing temporary variables used during the function's execution.
- sw $a2, 0($sp) to sw $t7, 20($sp): These lines save the values of registers $a2, $a3, $s0, $t0, $t1, and $t7 onto the stack. This is done because the code might overwrite these registers, and we need to restore them after the function call.

First check if $a2 (the address of the root node) is equal to zero . If it is, it means the root is NULL, so the code jumps to the end label and checks if $s0 () is equal to 1. If it is, it also jumps to the end label.

```
end:
        addi $v0 , $zero , -1
        lw $a2, 0($sp)
        lw $a3, 4($sp)
        lw $s0, 8($sp)
        lw $t0, 12($sp)
        lw $t1, 16($sp)
        lw $t7, 20($sp)
        addi $sp, $sp, 24
        jr $ra
```

After checking, loads the value stored at the memory location pointed to by $a2 (assumed to be the data field of the current node) into register $t0.

```
Found:
        beq $a0 , 0, wwexit      #$v1=1
        move $v0 , $a2 # address of element
        move $v1, $t9# address of parent
        j wwwexit
wwexit:
        addi $v1, $zero ,1
wwwexit:
        lw $a2, 0($sp)
        lw $a3, 4($sp)
        lw $s0, 8($sp)
        lw $t0, 12($sp)
        lw $t1, 16($sp)
```

then, compares $t0 with $a3 (which is assumed to hold the value we're looking for). If they're equal, it jumps to the **Found** label.

```
# Load Value of Data in $t0
    lw $t0 , 0($a2)
    beq $t0 , $a3 , Found
    slt $t1 , $a3 , $t0
    beq $t1 , 1 , GoLeft
    addi $t9, $a2, 0
    lw $t0 , 8($a2)
    move $a2 , $t0
    j Search
```

if $t0 not equal $a3, We performs a signed less-than comparison between $a3 and $t0. The result (0 or 1) is stored in $t1. checks if the value in $t1 is equal to 1. If it is, it means $a3 is less than $t0 (the current node's data), so the code jumps to the GoLeft label to explore the left subtree.

```
45  GoLeft:
46          addi $t9, $a2, 0
47          lw $t0 , 4($a2)
48          move $a2 , $t0
49          j Search
50
51  end:
52          addi $v0 , $zero , -1
53          lw $a2, 0($sp)
54          lw $a3, 4($sp)
55          lw $s0, 8($sp)
56          lw $t0, 12($sp)
57          lw $t1, 16($sp)
58          lw $t7, 20($sp)
59          addi $sp, $sp, 24
60          jr $ra
```

ASSEMPLY CODE:

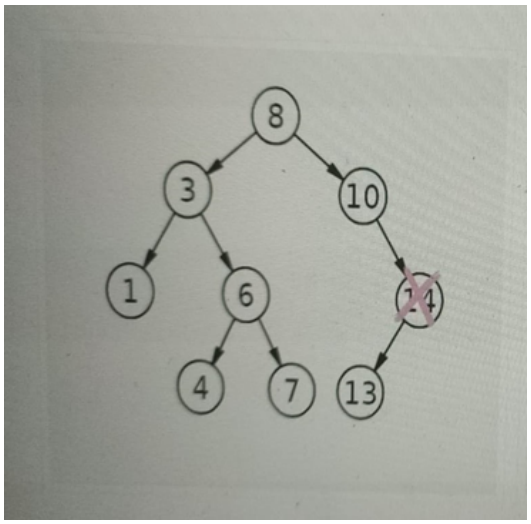
if $t0 not equal $a3, We performs a signed less-than comparison between $a3 and $t0. The result (0 or 1) is stored in $t1. checks if the value in $t1 is equal to 1. If it is, it means $a3 is less than $t0 (the current node's data), so the code jumps to the GoLeft label to explore the left subtree.

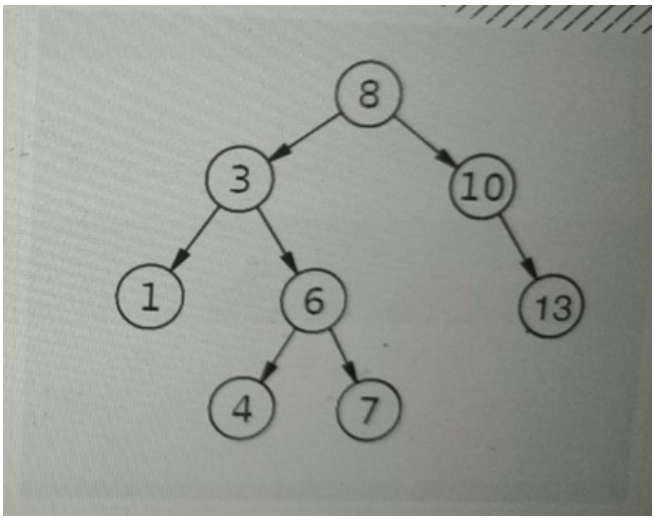
```
45  GoLeft:
46          addi $t9, $a2, 0
47          lw $t0 , 4($a2)
48          move $a2 , $t0
49          j Search
50
51  end:
52          addi $v0 , $zero , -1
53          lw $a2, 0($sp)
54          lw $a3, 4($sp)
55          lw $s0, 8($sp)
56          lw $t0, 12($sp)
57          lw $t1, 16($sp)
58          lw $t7, 20($sp)
59          addi $sp, $sp, 24
60          jr $ra
```

11 - DELETE

OVERVIEW:

Delete function uses Search to get two pointers, one for the node we want to delete and the other one for its parent. then, we determine wither its a leaf node or internal node with left child or right child or both, then we can delete it as shown in the next slides





C CODE:

```c
void DeleteNodeTree(Tree *pt)
{
    TreeNode *q = *pt, *r;
    if (!(*pt)->left) // First case
        *pt = (*pt)->right;
    else if (!(*pt)->right) // Second case
        *pt = (*pt)->left;  // Also, both account for a leaf node
    else
    { // third case
        q = (*pt)->left;
        if (!q->right)
        {
            (*pt)->entry = q->entry;
            (*pt)->left = q->left;
        }
        else
        {
            do
            {
                r = q;
                q = q->right;
            } while (q->right);
            (*pt)->entry = q->entry;
            r->right = q->left;
        }
    }
    free(q);
}
```

ASSEMPLY CODE:

```
922  Delete:
923          move $s5,$ra
924          addi $a0,$zero,1
925          DeleteTwo:
926          jal Search # a2 is the address of the root
927                     #and $a3 is the value to delete
928          move $s0,$v0  # address of element
929          move $s1,$v1  # address of parent
930          addi $t7,$zero ,-1
931          beq $v0, $t7,EEWXIT
932          beq $v1,$0,Root
933          Abdo:
```

First, we call the function Search to get the location of the element we want to delete so this function returns two pointers one for the address of the element and the other for his parent.

If the element we want to delete does not exist in the tree, Search function will return -1 in v0, so we check if v0 is -1 then end the delete function and inform the user that this element does not exist in the tree.

Another check if the v1 is null pointer, this means that we want to delete the root element.
Else we continue with s0 has the address of the element and s1 has the address of his parent.

```
934          lw $t1,4($s0) # address of left of the element
935          lw $t2,8($s0) # address of right of the element
936          sne $t3,$t2,$0 # if there is right
937          sne $t4,$t1,$0 # if there is left
938          add $t5,$t3,$t4 #
939          beq $t5,$0,CaseOne # there is no left neither right
940          beq $t4,1,CaseTwoI
941          beq $t3,1,CaseTwoII
```

After checking that we do not want to delete the non-exist element or the root, we store the address of the left element in t1 and the address of the right element in t2.

Then we check if the right element is not null, make t3 equals one.

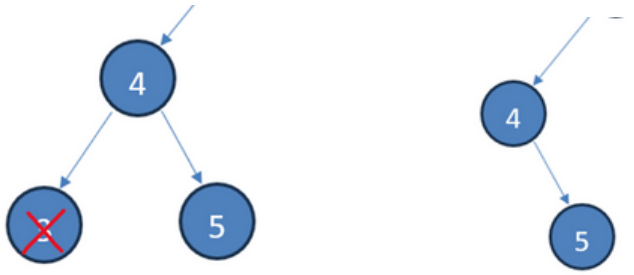
The same thing for the left element, if he is not null, make t4 equals one.

We store the sum of t3 and t4 into t5 to check if it is null element, then this case one
(we want to delete leaf element with no right or left).

After that we check if the element has the left child, so it is case two I.

After that we check if the element has the right child, so it is case two II.

```
942  CaseOne:
943          addi $s2,$s1,4
944          lw $s2,0($s2)
945          beq $s2,$s0,Null
946          addi $s2,$s1,8
947          lw $s2,0($s2)
948          beq $s2,$s0,Null
949          j RA
950  Null:
951          sw $0,0($s2)
952          sw $0,0($s0)
```



Back to case one (leaf element):

To delete it we want to know if he is right of left his parent to update the Childrens of the parent.

First, we load the address of the left child in s2 and check if it is null, if so, we make the left of the parent null pointer and make the element null.

Same as the left, we check if the element is the right child of his parent, if so, we make the right of the parent null pointer and make the element null.

After that we jump to label RA that ends the function and return to the main function.

```
728  CaseTwoI:
729          beq $t3,1,FinalCase
730          lw $t8,4($s1) # left of the parent
731          beq $t8,$s0,CaseThreeII # if the element in left of the parent
732          lw $t6,4($s0)
733          sw $t6,8($s1)
734          j RA
735
736  CaseTwoII:
737          beq $t4,1,FinalCase
738          lw $t8,4($s1) # left of the parent
739          beq $t8,$s0,CaseThreeI # if the element in left of the parent
740          # else the element in right of the parent
741          lw $t8, 8($s0)
742          sw $t8, 8($s1)
743          j RA
```

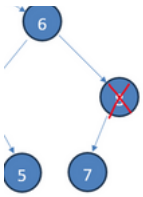
Back to case two I (the element has left child):

First, we check if the element has also right a child, then it is the final case (the element has both left and right childrens).

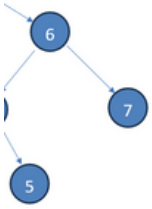
Else we made sure that the element only has left child, so to delete it we need to know his location according to his parent.

If he is on the left of his parent, we make the left branch of his parent points to the left child of this element.

Else, if he is on the right of his parent, we make the right branch of his parent points to the left child of this element.



We make this by loading the address of the left pointer of the parent in t8 and comparing it with the element address, if they are equal, this will be case three II (the element is the left child of his parent)



Else, the element will be the right child of his parent, so we load the address of the left child of the element in t6 and store it in the right child of the parent (make the right branch of the parent points to the child of element).

Then jump to label RA to make the element null pointer return to the main function.

Back to case two II (the element has right child):

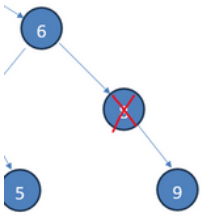
First, we check if the element has also left a child, then it is the final case (the element has both left and right childrens).

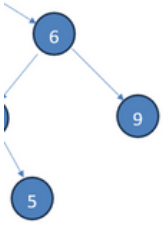
Else we made sure that the element only has right child, so to delete it we need to know his location according to his parent.

If he is on the left of his parent, we make the left branch of his parent points to the right child of this element.

Else, if he is on the right of his parent, we make the right branch of his parent points to the right child of this element.

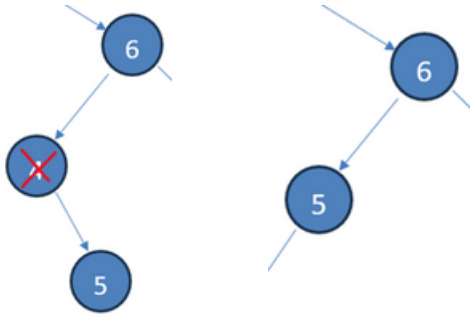
We make this by loading the address of the left pointer of the parent in t8 and comparing it with the element address, if they are equal, this will be case three I (the element is the left child of his parent)
Else, the element will be the right child of his parent, so we load the address of the right child of the element in t6 and store it in the right child of the parent (make the right branch of the parent points to the child of element).



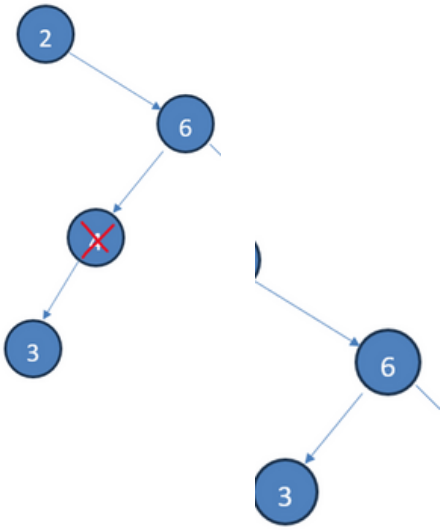


Then jump to label RA to make the element null pointer and return to the main function.

```
744  CaseThreeI:
745          lw $t6,8($s0)
746          sw $t6,4($s1)
747          j RA
748  CaseThreeII:
749          lw $t6,4($s0)
750          sw $t6,4($s1)
751          j RA
```



Back to case three I (the element is the left child of his parent and has right child):



We load the address of the right child of the element in t6 and store it as the left child of the parent instead of the element.

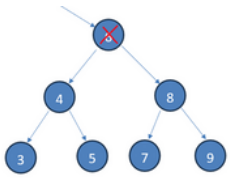
Then jump to the label RA to make the element null.

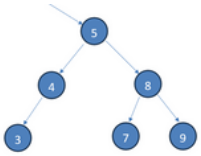
Back to case three II (the element is the left child of his parent and has left child):

We load the address of the left child of the element in t6 and store it as the left child of the parent instead of the element.

Then jump to the label RA to make the element null.

```
FinalCase:
        # else the element has left and right
        move $t0,$t1 # both t1 and t0 has the address of left of the element
        Loop:
                lw $t2,8($t0) # right of left of the element
                beq $t2,$0,FinalCaseII
                move $t1,$t0
                move $t0,$t2
                j Loop
        FinalCaseII:
        lw $t7,4($s0)
        beq $t7,$t0,LeftWithoutRight
        lw $t6,0($t0)
        sw $t6,0($s0)
        move $a2,$t1
        move $a3,$t6
        j DeleteTwo
```





Back to the final case (the element has both left and right childrens):

To delete an element that has left and right nodes we need to get the rightest node in his left branch and replace it with the element we want to delete then delete the rightest node as any case of the previous.

So, we loop until we reach the rightest node and exit the loop with the address of the rightest element in t0 and its parent in t1.

After the loop we make sure that there is rightest element or not, if there is no, then it is case left without right.

Else, we load the value of the rightest element in t6 and store it in the element we want to delete.

Then we delete the rightest element by store its value in a3 and its parent address in a2 as the root of the subtree and begin the function again with the subtree.

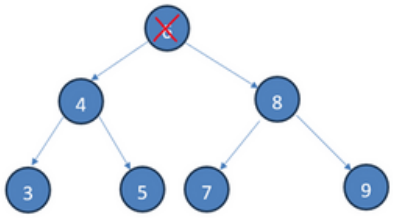
```
769  LeftWithoutRight:
770          sw $t0,4($s1)
771          lw $t8,8($s0)
772          sw $t8,8($t0)
773          j RA
774  Root:
775          lw $t2,4($s0)
776          lw $t3,8($s0)
777          add $t1,$t2,$t3
778          beq $t1,$0,RA
779          beq $t2,$0,RootNoLeft
780
781          j Abdo
```

Back to case left without right:

We store the address of the left of the element in the left of the parent.

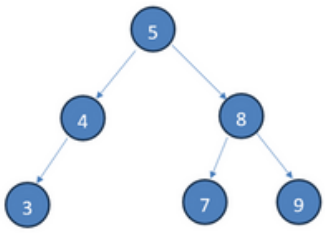
Then we get the address of the right of the element in t8 and store it in the right of the left of the element.

Then we jump to the label RA to make the element null.



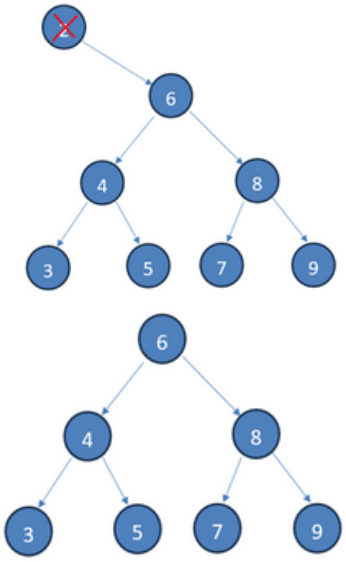
Back to root element:

First, we check if the root has both right and left childrens, if so, this means that it is any case of the previous, so we jump to the delete function again.



If the root has neither right nor left childrens, then we jump to the label RA to just make it null pointer.

```
782   RootNoLeft:
783           lw $t7,0($t3)
784           lw $t8,4($t3)
785           lw $t9,8($t3)
786           sw $t7,0($s0)
787           sw $t8,4($s0)
788           sw $t9,8($s0)
789           sw $0, 0($t3)
790           sw $0, 4($t3)
791           sw $0, 8($t3)
792           move $ra,$s5
793           jr $ra
```



Back to case root no left:

To delete the root that has no left child, we want to make the right children the new root and modify the root left branch.

So, we load the values of the right child of the root and the addresses of his left and right nodes in t7, t8, t9.

Then, we store the value of the right child of the root in the root itself and store t8 as the left branch and t9 as the right branch to the new root.

Then, we delete the right child of the old root by make it null and return to the main function.

Back to RA label:
This label makes the final element null and return to the main function.

Back to end delete label:
This label to show a message if the element we want to delete not found in the tree then return to the main function.

```
794   RA:
795             move $ra,$s5
796             sw $0,0($s0)
797             sw $0,4($s0)
798             sw $0,8($s0)
799             jr $ra
800   endDelete:
801             move $ra,$s5
802             li $v0,4
803             la $a0,notDeleteMessage
804             syscall
805             jr $ra
806
```

12 - CLEAR

C CODE:

```c
void ClearTree(Tree *pt){
    ClearTreeRecAux(pt);
    *pt=NULL;
}


void ClearTreeRecAux(Tree *pt){
    if (*pt){
        ClearTreeRecAux(&(*pt)->left);
        ClearTreeRecAux(&(*pt)->right);
        free(*pt);
    }
}
```

These two functions are likely used to deallocate the memory allocated for a binary tree.

1.ClearTree(Tree *pt): This function serves as the entry point to clear the tree. It first calls the auxiliary recursive function ‘ClearTreeRecAux’, passing the pointer to the root of the tree. Then, it sets the root pointer (*pt) to NULL, effectively marking the tree as empty.

2.ClearTreeRecAux(Tree *pt): This is the recursive auxiliary function responsible for traversing and deallocating the nodes of the tree. It takes a pointer to a tree node (*pt) as input

- If the pointer is not null (*pt), it recursively calls itself on the left subtree (ClearTreeRecAux(&(*pt)->left)) and then on the right subtree (ClearTreeRecAux(&(*pt)->right)).

- Once both subtrees are cleared, it deallocates the memory for the current node using free(*pt). This assumes that the nodes were allocated dynamically (e.g., with malloc).

- This function essentially performs a post-order traversal of the tree, ensuring that child nodes are cleared before their parent.

ASSEMPLY CODE:

```
clear:
    move $a0,$s7          # a0 = $s0
    # if (bPtr == NULL)
    bne $a0, $0, clearRecurse
    lw $a0 , AboAlaaWeAbdo
    addi $a0 , $a0 , 4
    sw $zero , 0($a0)
    move $a0 , $zero
    jr $ra                # Return to caller
```

This‘clear’ function is to clear all nodes in a binary tree. First, It checks if the address of the current node is not null and It branches to the ‘clearRecurse’ function If it’s not null, If the address is null that means that there is no node so We don’t need to clear it (tree is not activated) and We will jump back to the address saved in the ‘$ra’ register.

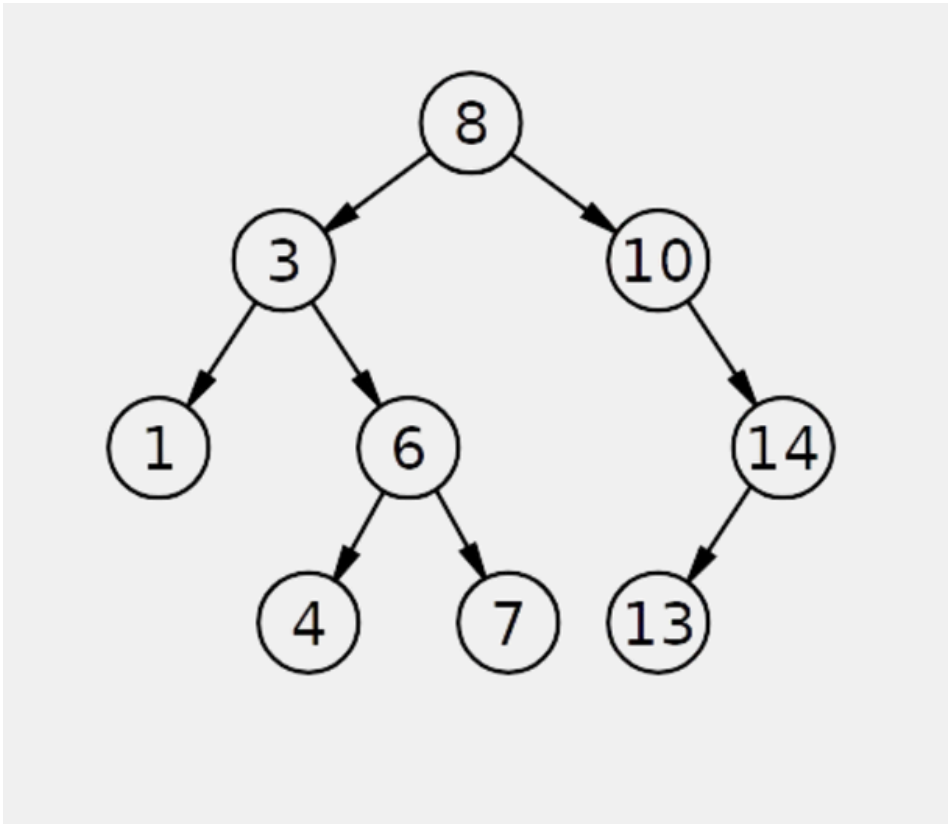
```
clearRecurse:
    addi $sp, $sp, -8       # Allocate 2 registers to stack
    sw $ra, 0($sp)          # $ra is the 1st
    sw $a0, 4($sp)          # $a0 is the 2nd

    lw $s7 , 4($s7)        # curr = curr->left
    jal clear           # Call postOrder

    lw $a0, 4($sp)          # Retrieve original $a0
    move $s7, $a0           # $s0 = $a0

    lw $s7, 8($s7)         # curr = curr->right
    jal clear           # Call postOrder

    lw $a0, 4($sp)          # Retrieve original $a0
    move $s7, $a0           # $s0 = $a0
    jal finalclear          # Call printSmall

    lw $ra, 0($sp)          # Retrieve original $ra
    addi $sp, $sp, 8        # Free the 2 register stack spaces
    jr $ra
```

This ‘clearRecurse’ function is to clear each node in a binary tree. First, We allocate space on the stack for storing two registers and save the return address ‘$ra’ and the current node pointer ‘$a0’. Then We traverse the left sub tree by loading the left child pointer of the current node and recursively calling the ‘clear’ function to clear the left sub tree. Then We traverse the right sub tree by restoring the address of the current node from the stack , loading the right child pointer of the current node and recursively calling the ‘clear’ function to clear the right sub tree. Then We clear the current node by restoring the address of the current node from the stack , calling the ‘finalclear’ function which clears the current node, restoring the return address from the stack, freeing the space allocated on the stack and returning to the caller.

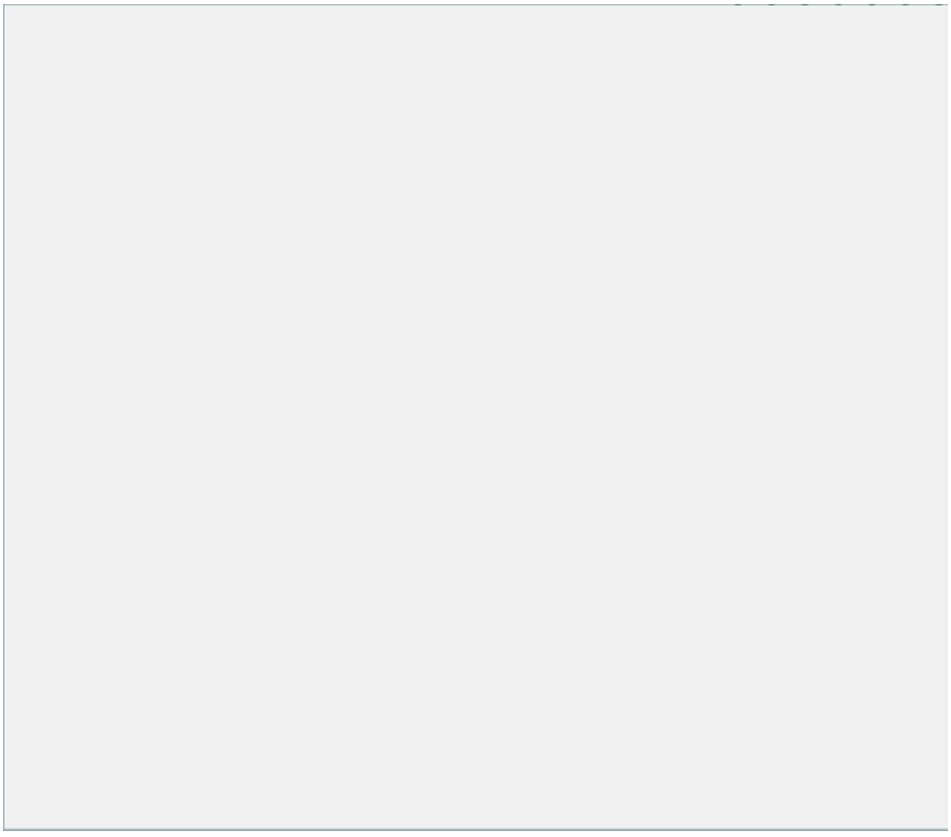
```
732 finalclear:
733     sw $zero,0($s7)
734     sw $zero , 4($s7)        # Set left pointer  to null
735     sw $zero , 8($s7)        # Set right pointer  to null
736     li $s7 , 0              # Set $s7 to null
737     jr $ra                  # Return to caller
```

This ‘finalclear’ function is to clear a single node in a binary tree by setting the current node, left child pointer and the right child pointer to null then returning to the caller.

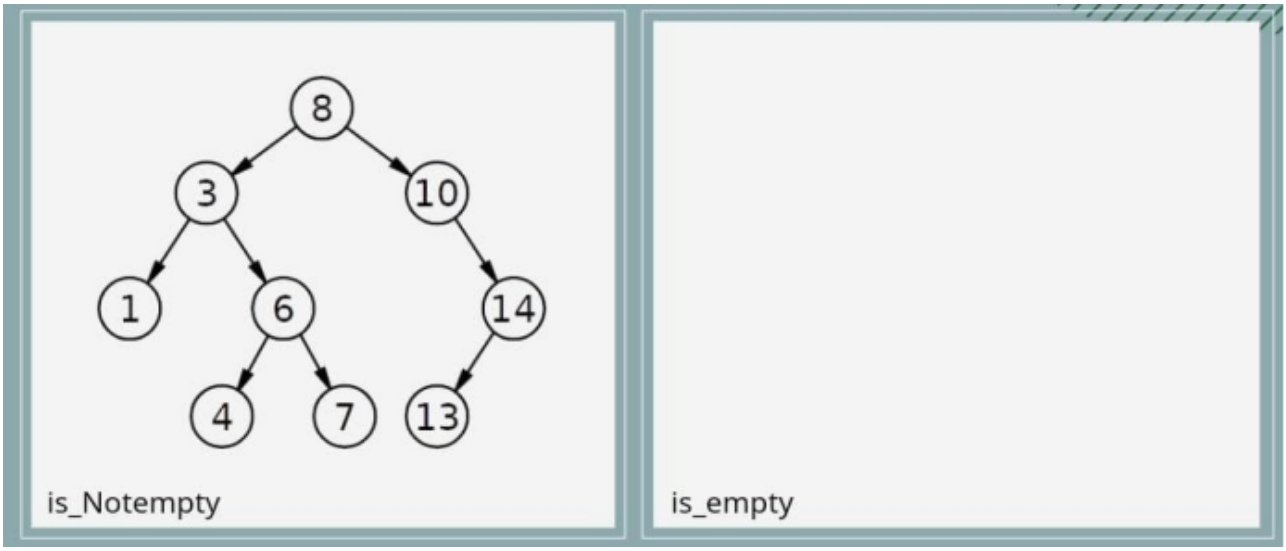
Before clearing the tree:



After clearing the tree:



13 - EMPTY



The ‘TreeEmpty’ function is a simple utility function that checks whether a binary tree is empty.

1.It takes a pointer to a Tree structure (Tree *pt) as input.

2.It checks if the pointer is null (!*pt). If it is, it means that the tree is empty, so it returns 1 (true), indicating that the tree is indeed empty.

3.If the pointer is not null, it means that the tree is not empty, so it returns 0 (false), indicating that the tree contains at least one node.

C CODE:

```
10
11    int TreeEmpty(Tree *pt){
12        return (!*pt);
13    }
14
```

ASSEMPLY CODE:

```
820  # Empty function
821  empty:
822          addi $sp, $sp, -4
823          sw $ra, 0($sp)
824          jal size
825          move $t8, $v0
826          li $v0, 0
827          beq $t8, $zero, isempty
828          lw $ra, 0($sp)
829          addi $sp, $sp, 4
830          jr $ra
831
832  isempty:
833          addi $v0, $v0, 1
834          lw $ra, 0($sp)
835          addi $sp, $sp, 4
836          jr $ra
837  # End of empty function
```

This function checks if the tree is empty by calling the ‘size’ function and checking if It returns 0 that means the tree is empty so the ‘isempty’ function will return 1. Otherwise ‘empty’ function will return 0. And this is by allocating space on the stack for storing one register and saving the return address, Then It calls the ‘size’ function to calculate the size of the tree and checks if the size is 0, It branches to ‘isempty’ function and returns 1 to the caller after restoring the return address from the stack and freeing the space allocated on the stack.

14 - USER INTERFACE

ASSEMPLY CODE:

- Print Commands:

```
printWelcome: .asciiz "Welcome to Exhausted Lions Tree!!\n"
create_message: .asciiz "please press 'R' to create the tree first of all...\n"
printComExplain: .asciiz "\tA - Add a new record\n \tP - Perform a preorder trave
printCommand: .asciiz "Please enter a command (A,I,P,O,T,E,S,C,D,Z,R,Q): "
printACommand: .asciiz " - Add a new record...\n"
printPCommand: .asciiz " - Perform a preorder traversal...\n"
printICommand: .asciiz " - Perform an inorder traversal...\n"
printOCommand :  .asciiz " - Perform an postorder traversal...\n"
printCCommand: .asciiz " - clear the tree ...\n"
printZCommand:  .asciiz " - get the size of the tree ...\n"
printECommand: .asciiz " -check whether the tree is empty or not ...\n"
printSCommand: .asciiz " -Search about an element from the tree ...\n"
prinDCommand :.asciiz " -Delete an element from the tree...\n"
printCRCommand_1:  .asciiz "\n-Tree Created , You have Now "
prinQCommand :.asciiz " -program end...\n"
message: .asciiz "enter an element to search\n"
prinDepthCommand : .asciiz "Check depth of tree\n"
DeleteMessage : .asciiz "enter the element you want to delete\n"
notDeleteMessage : .asciiz "element not found\n"
ChooseTreeNum: .asciiz "\nChoose Tree Number : "
printNotEmpty: .asciiz "The Tree is Not Empty....\n"
printempty: .asciiz "The Tree is Empty....\n"
printnotFound:.asciiz "Element is Not Found....\n"
printFound:.asciiz "Element is Found....\n"
```

Conditions:

```
li $v0, 4                                   # Prompt for the command ch
la $a0, printCommand
syscall

li $v0, 12                                  # Enter the command charact
syscall
move $s0, $v0                               # Store in $s0

li $v0, 4                                   # Print nextline
la $a0, printNextline
syscall

beq $s0, 'A', isA                  # If command character = A (call isA)
beq $s0, 'a', isA                  # If command character = a (call isA)
beq $s0, 'P', isP                  # If command character = P (call isP)
beq $s0, 'p', isP                  # If command character = p (call isP)
beq $s0, 'I', isI                  # If command character = I (call isI)
beq $s0, 'i', isI                  # If command character = i (call isI)
beq $s0, 'O', isO                   # If command character = O (call isO)
beq $s0, 'o', isO                   # If command character = i (call isO)
beq $s0, 'Z', isZ                   # If command character = Z (call isZ)
beq $s0, 'z', isZ                   # If command character = z (call isZ)
beq $s0, 'C', isC                   # If command character = C (call isC)
beq $s0, 'c', isC                   # If command character = c (call isC)
beq $s0, 'E', isE                   # If command character = E (call isE)
beq $s0, 'e', isE                   # If command character = e (call isE)
beq $s0, 'S', isS                   # If command character = S (call isS)
beq $s0, 's', isS                    # If command character = s (call isS)
beq $s0, 'T', isDepth                # If command character = T (call isDepth)
beq $s0, 't', isDepth                # If command character = t (call isDepth)
beq $s0, 'D', isD                     # If command character = D (call isD)
beq $s0, 'd', isD                    # If command character = d (call isD)
beq $s0, 'Q', isQ                    # If command character = Q (call isQ)
beq $s0, 'q', isQ                    # If command character = q (call isQ)
beq $s0, 'R', isCreate               # If command character = R (call isR)
beq $s0, 'r', isCreate               # If command character = r (call isR)

li $v0 , 4
la $a0 , WrongInput
syscall
jal after
```

Branches:

```
124 isA:
125                     EnterAgain:
126                     li $v0, 4
127                     la $a0, ChooseTreeNum
128                     syscall
129
130                     li $v0 , 5
131                     syscall
132                     move $a1 , $v0
133                     lw $a2 , NumOfTrees
134                     bgt $a1 , $a2 , EnterAgain
135                     jal SelectTreeRoot
136
137                     li $v0 , 5    #enter an integer
138                     syscall
139                     move $a2 , $v0
140                     move $a3 , $s7
141                     jal insert
142                     jal after
```

```
145 isP:
146                     li $v0, 4
147                     la $a0, ChooseTreeNum
148                     syscall
149
150                     li $v0 , 5
151                     syscall
152
153                     move $a1 , $v0
154                     lw $a2 , NumOfTrees
155                     bgt $a1 , $a2 , EnterAgain
156                     jal SelectTreeRoot
157
158                     li $v0, 11
159                     move $a0, $s0
160                     syscall
161
162                     li $v0, 4
163                     la $a0, printPCommand
164                     syscall
165
166                     #lw $s7, root
167                     move $a3 , $s7
168                     jal preOrder
169                     jal after
```

```
isI:
                li $v0, 4
                la $a0, ChooseTreeNum
                syscall

                li $v0 , 5
                syscall
                move $a1 , $v0
                lw $a2 , NumOfTrees
                bgt $a1 , $a2 , EnterAgain
                jal SelectTreeRoot

                li $v0, 11
                move $a0, $s0
                syscall
                li $v0, 4
                la $a0, printICommand
                syscall

                #lw $s0, root
                move $a3 , $s7
                jal InOrder
isO:
                li $v0, 4
                la $a0, ChooseTreeNum
                syscall

                li $v0 , 5
                syscall
                move $a1 , $v0
                lw $a2 , NumOfTrees
                bgt $a1 , $a2 , EnterAgain
                jal SelectTreeRoot

                li $v0, 11
                move $a0, $s0
                syscall
                li $v0, 4
                la $a0, printOCommand
                syscall

                #lw $s0, root
                move $a3 , $s7
                jal postOrder
                jal after
```

```
isC:

                    li $v0, 4
                    la $a0, ChooseTreeNum
                    syscall

                    li $v0 , 5
                    syscall
                    move $a1 , $v0
                    lw $a2 , NumOfTrees
                    bgt $a1 , $a2 , EnterAgain
                    jal SelectTreeRoot
                    li $v0, 11
                    move $a0, $s0
                    syscall
                    li $v0, 4
                    la $a0, printCCommand
                    syscall

                    #lw $s0, root

                    jal clear
                    jal after
```

```
isZ:
                    li $v0, 4
                    la $a0, ChooseTreeNum
                    syscall

                    li $v0 , 5
                    syscall
                    move $a1 , $v0
                    lw $a2 , NumOfTrees
                    bgt $a1 , $a2 , EnterAgain
                    jal SelectTreeRoot
                    li $v0, 11
                    move $a0, $s0
                    syscall
                    li $v0, 4
                    la $a0, printZCommand
                    syscall

                    #lw $s0, root
                    move $a3 , $s7
                    li $s5,0
                    jal size
                    move $a0, $v0
                    li $v0, 1
                    syscall
                    jal after
```

```
isE:
                    li $v0, 4
                    la $a0, ChooseTreeNum
                    syscall
                    li $v0 , 5
                    sysc
                    move $a1 , $v0
                    lw $a2 , NumOfTrees
                    bgt $a1 , $a2 , EnterAgain
                    jal SelectTreeRoot
                     li $v0, 11
                    move $a0, $s0
                    syscall
                    li $v0, 4
                    la $a0, printECommand
                    syscall

                    #lw $s0, root
                    move $a3 , $s7
                    li $s5,0
                    jal empty
                    move $a0, $v0
                    move $v1,$t8
                    beq $v1,0,EmptyTree
                    li $v0,4
                    la $a0,printNotEmpty
                    syscall
                    jal after

isS:

                    li $v0, 4
                    la $a0, ChooseTreeNum
                    syscall

                    li $v0 , 5
                    syscall
                    move $a1 , $v0
                    lw $a2 , NumOfTrees
                    bgt $a1 , $a2 , EnterAgain
                    jal SelectTreeRoot
                   move $v1,$zero
                   li $v0, 11
                    move $a0, $s0
                    syscall
                    li $v0, 4
                    la $a0, printSCommand
                    syscall

                    #lw $s0, root
                    move $a2 , $s7
                    li $v0,4
                    la $a0,message
                    syscall

                    li $v0,5
```

syscall Issue a system call

```
isD:
                    li $v0, 4
                    la $a0, ChooseTreeNum
                    syscall

                    li $v0 , 5
                    syscall
                    move $a1 , $v0
                    lw $a2 , NumOfTrees
                    bgt $a1 , $a2 , EnterAgain
                    jal SelectTreeRoot
                    #lw $s7 , Heap_Start

                   move $v1,$zero
                    move $v0,$zero
                      li $v0, 11
                    move $a0, $s0
                    syscall
                    li $v0, 4
                    la $a0, prinDCommand
                    syscall

                    #lw $s0, root
                    move $a2 , $s7
                    li $v0,4
                   la $a0, DeleteMessage
                   syscall

                   li $v0,5
                   syscall
                   move $a3,$v0
                    move $t9 , $zero

                   jal Delete

                   jal after
```

```
390 isDepth:
391                     li $v0, 4
392                     la $a0, ChooseTreeNum
393                     syscall
394
395                     li $v0 , 5
396                     syscall
397                     move $a1 , $v0
398
399                     lw $a2 , NumOfTrees
400                     bgt $a1 , $a2 , EnterAgain
401                     jal SelectTreeRoot
402                    move $v0,$zero
403                     li $v0, 4
404                     la $a0, prinDepthCommand
405                     syscall
406                     #lw $s0, root
407                     lw $a0 , AboAlaaWeAbdo
408
409     addi $a0 , $a0 , 4
410     lw $a0 , 0($a0)
411     move $v0 , $zero
412     beq $a0 , $zero  , Skip
413     move $v0 , $zero
414
415     move $a0 , $s7
417                     jal TreeDepth
418                     Skip:
419                     move $a0, $v0
420                     li $v0, 1
421                     syscall
422                     jal after
423
424
```

After Function:

prints a new line in each branch

```
425   after:
426                         li $v0, 4
427                         la $a0, printNextline
428                         syscall
429
430                         jal main_loop
431
```

Output: