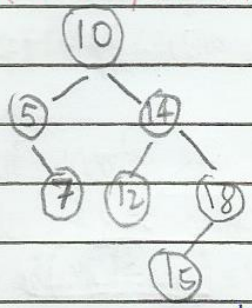
BST: Binary Search Trees

date _____

*Example of a Binary Search Tree: SET A = {10, 5, 14, 12, 7, 18, 15}



→ Observations? - The left child is less than the parent
- The right child is greater than the parent

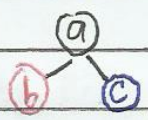
~o~ BST CHARACTERISTICS ~o~

- A Binary Search Tree (BST) is a binary tree in which the nodes are labeled with elements of a set [unique elements].
- A basic data structure used to represent sets whose elements are ordered by some linear order & can be compared using <, >, & = (or their equivalent in some language).
- // NOTE: To recall for structures comparison, choose a field in the structure then compare. Cannot use an entire structure for comparison

→ BST PROPERTY ←

* All elements stored in the left subtree of any node X are all less than the element stored at X, and all elements stored in the right subtree of X are greater than the element stored at X.

ex:

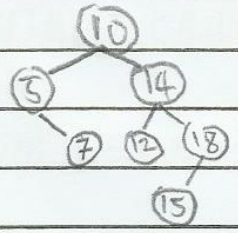


b < a
c > a

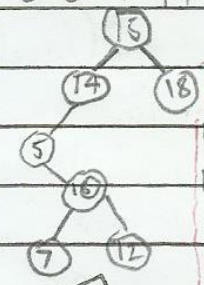
// NOTE: no equals, strictly < and > only.

→ BST - Sets A & B

SET A = {10, 5, 14, 12, 7, 18, 15}



SET B = {15, 14, 5, 10, 16, 7, 12}



• Observations: equal sets, different BST due to order of insertion

// NOTE: if inserted in ascending order: BST skewed to the Right

// if inserted in descending order: BST skewed to the left

// Both have O(N) running time

BST: Operations

date

- Operations supported by BST have a running time of $O(\log_2 N)$
  1.) insert() - every time we insert a node, we insert a leaf.
  2.) delete() -
  3.) Member() - return 1 or 0 if a node is part of a tree or not.
  4.) Min() - returns the left most node of a given tree.
  5.) Max() - returns the right most node of a given tree.
- APPLICATION: BST can be use to represent the set of identifiers of a programming language like C.

~~What is $\log_2 N$? ~~

$O(\log_2 N) = X \rightarrow 2^X = N$

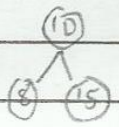
// if $O(\log_2 N) = X$, then u can represent it using $2^X = N$.

// N is the # of elements and X is the running time

X	N
1	2
2	4
3	8

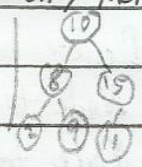
- Explanation: Since when you double the # of elements in your tree, the # of comparisons is only incremented by 1.

Example:



No. of nodes: 3

comparisons to find 8: 2



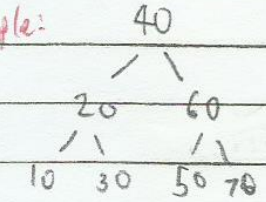
No. of nodes: 6

comparisons to find 9: 3

★ Other terms:

- Height Balanced → considered if the max difference between the left subtree & right subtree is 1.
- Completely Balanced → Considered if all nodes are filled up before moving to the next level.
- Skewed tree

Example:



// is height balanced & completely balanced

Strathmore

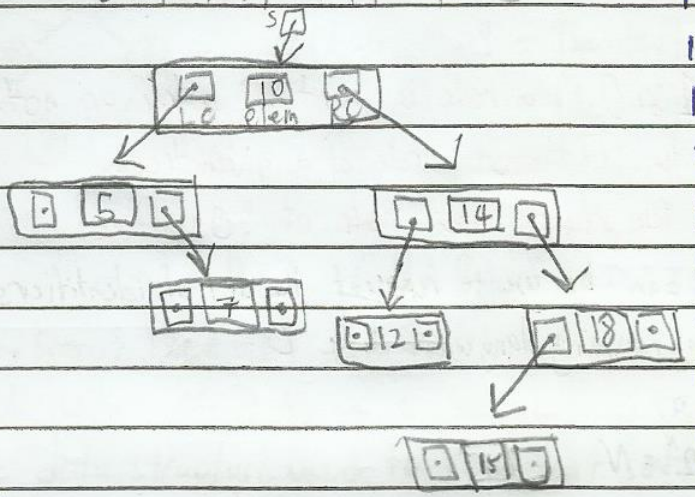
Note: $O(N)$ still needs to be avoided.

★ Pointer Implementation of BST ★

date _____

- each node will have a pointer to its LEFT & RIGHT child.

Example: SET S = {10, 5, 14, 12, 7, 18, 15}



• Data Struct Def:

```
typedef struct cell {
  int elem;
  struct cell *LC;
  struct cell *RC;
} ctype, *BST;

BST S;
```

★ Operations ★

1.) INSERT(n, S) - given a new node and the tree, the node is inserted at the leaf (as child) to maintain BST Property.

2.) DELETE(n, S) - this has 3 possibilities:
   - > Delete node at leaf.
   - > Node to delete has 1 child
   - > Node to delete has 2 children

3.) MEMBER(n, S) - given a node & a tree, look for the node if it exists in the tree or not.

4.) MIN(S) - return the minimum node value found in the tree.
   - entire tree does not need to be searched.

5.) MAX(S) - return the maximum node value found in the tree

date _____

☆ Traversal Functions ☆

1.) INORDER(s) - traverse left subtree
- visit root
- traverse right subtree
- results in ascending order

2.) PREORDER(s) - visit root
- traverse left subtree
- traverse right subtree

3.) POSTORDER(s) - traverse left subtree
- traverse right subtree
- visit root

o ~ o BST OPERATIONS EXPLAINED o ~ o

1.) MEMBER() - PN Traversal

1. Start at ROOT
2. Move to Left/Right node depending on the element to be searched for:
   Left: n < node
   Right: n > node
3. Keep traversing until element is found OR you reach a leaf node

//can be either: • Non-recursive
• Recursive

date _____

2) INSERT() - PPN Traversal

1. Start at ROOT and look for LEAF Node.
2. Compose new element n to current node:
   L: n < node
   R: n > node
3. Once leaf is found, insert node to its left or right depending on comparisons < or >.

// is either: Non-recursive
Recursive

3. DELETE() - PPN Traversal - has 3 cases

Case 1: Node to be deleted is a LEAF.
> simply remove from the tree.

Case 2: Node to be deleted has only one child.
> copy the child to the node and delete the node.

Case 3: Node to be deleted has two children.
> Find inorder successor of node.
> copy contents of the inorder successor to the node & delete the inorder successor.

Steps: 1.) start at ROOT.
2.) Compare elem to current root:
   L: n < node
   R: n > node
3.) check for the possible cases:
   Case 1: delete leaf node
   Case 2: delete L/R node
   Case 3: let root = inorder successor.

// can be Non-Recursive or Recursive

Recursive Functions

date _____

- Recursive Functions - a routine / function that calls itself directly or indirectly.

- Creating Recursive Functions - Make sure the function has at least 1 exit point or base case to ensure that the function terminates.
  - Start the func. with the exit point.

- A substitute for loops (?)
  → Recursive functions creates MULTIPLE ACTIVATION RECORDS for each function call. The process of returning a value & deallocating the space used by the A.R. This makes the process longer & slower.

  → MOST USEFUL: Recursion creates a pointer for Every level of the tree. This allows for traversing downwards AND back upwards.

  → APPLICATIONS: - INORDER, PREORDER, POSTORDER Traversals
  - sudoku

// Note: - Loop for one-directional traversing, shorter
- Recursive for traversing back upwards.

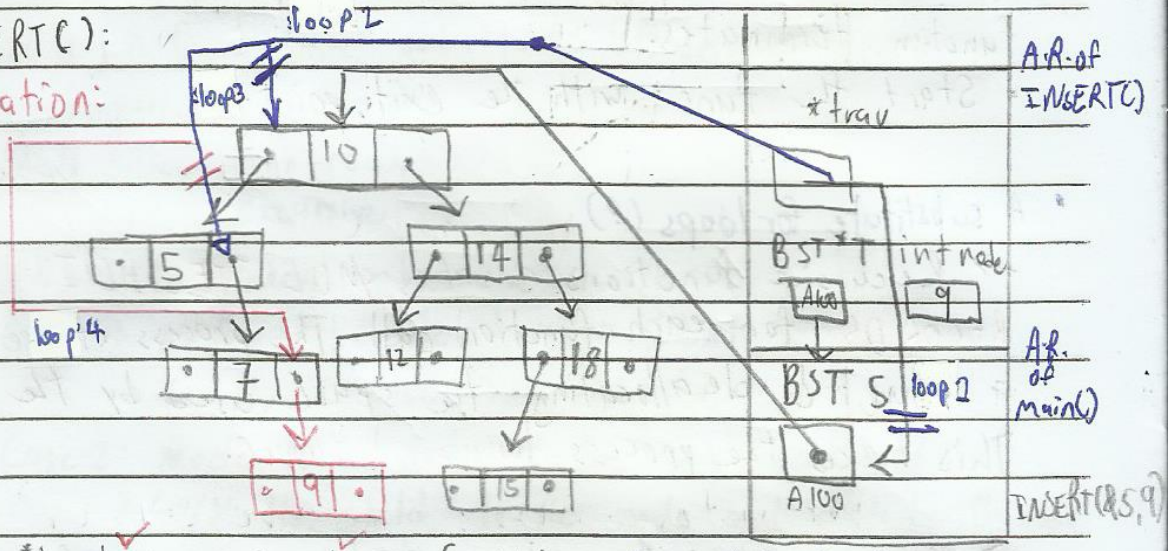
Non-recursive Ver. BST Operations

date _____

1) INITIALIZE()

```
void INITIALIZE(BST *T){
    *T = NULL;
}
```

2) INSERT():



- Simulation:

Loop 1: *trav != NULL && 10 != 9 ; (9<10) so trav = &(*trav)->LC

Loop 2: *trav != NULL && 5 != 9 ; (5<9) so trav = &(*trav)->RC

Loop 3: *trav != NULL && 7 != 9 ; (7<9) so trav = &(*trav)->RC

Loop 4: *trav != NULL - STOP LOOP

- Code:

```
void INSERT(BST *T, int node){
    BST *trav;
    for(trav=T; *trav!=NULL && (*trav)->elem!=node;){
    trav=((*trav)->elem>node)? &(*trav)->LC : &(*trav)->RC;
    }
    if(*trav==NULL){
      *trav=(BST)calloc(1, sizeof(struct node));
       if(*trav!=NULL){
          (*trav)->elem=node; simulate more
       }
    }
```

Malloc ver.:

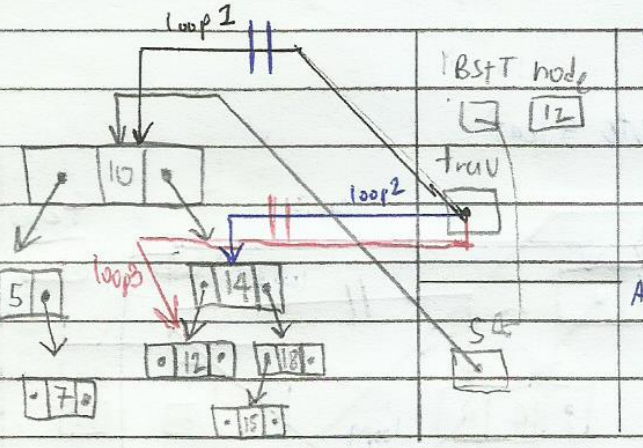
```
if(*trav==NULL){
BST temp=(BST)malloc(sizeof(struct node));
if(temp!=NULL){
   temp->elem=node;
   temp->LC=NULL;
   temp->RC=NULL;
   *trav=temp;
```

Non-Recursive Ver BST operations

date

3.) MEMBER()

• Simulation:

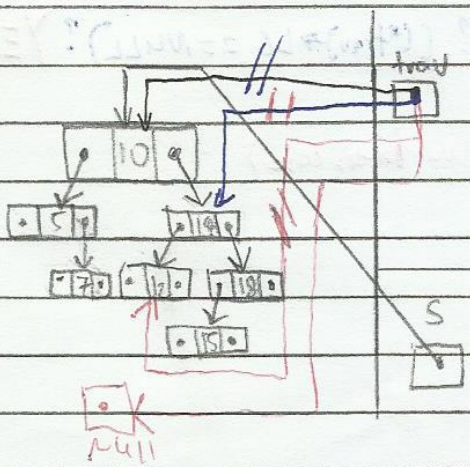


// Testcase:
MEMBER(T, 12)
// 12 is a member

// same logic with isMember() for a singly linked list. The only difference is the 3rd parameter of the loop.

```
int MEMBER(BST T, int node){
    BST trav;
    for(trav=T; trav!=NULL && trav->elem!=node;){
        trav=(trav->elem>node)?trav->LC:trav->RC;
    }
    return (trav==NULL)?0:1;
}
```

// Testcase:
MEMBER(T, 11)



// trav==NULL, 11 is not a member

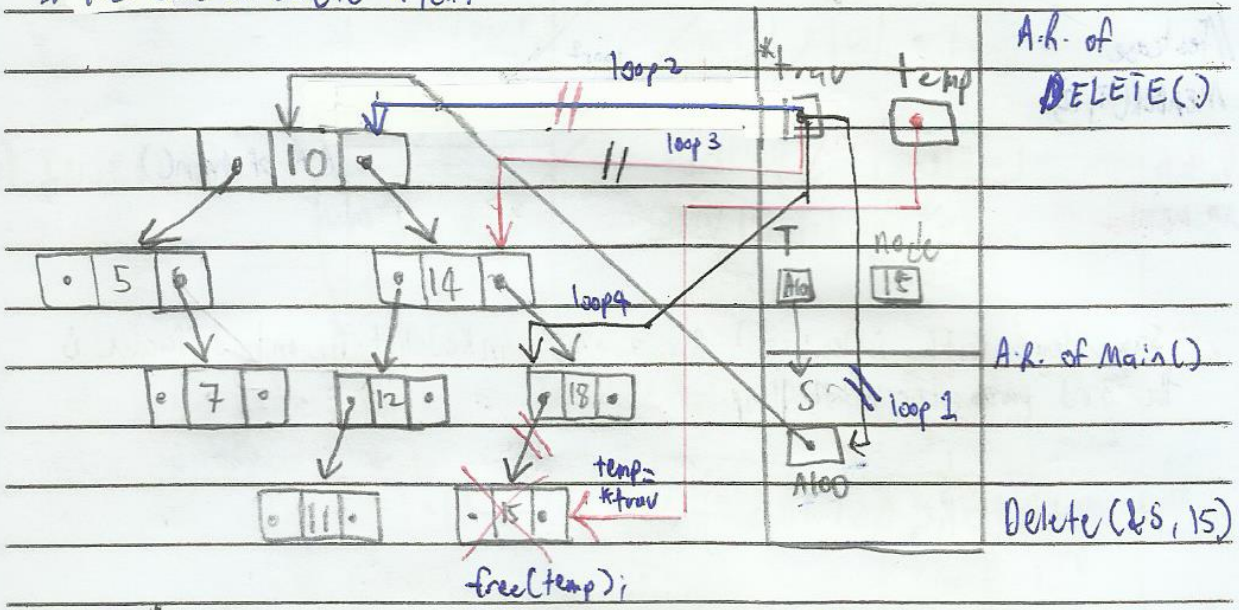
Non-Recursive BST Operations

date _____

1.) DELETE()

- Simulation (5):

// Test case 1: Delete a leaf



Loop 1: *trav != NULL && 10 != 15 ; (10 < 15) so &(*trav) → RC

Loop 2: *trav != NULL && 14 != 15 ; (14 < 15) so &(*trav) → RC

Loop 3: *trav != NULL && 18 != 15 ; (18 > 15) so &(*trav) → LC

Loop 4: *trav != NULL && 15 == 15 → STOP LOOP

- Check cases

Case 1: Does 15 have NO LC? (*trav) → LC == NULL)? YES

temp = 15

*trav = RC (null in this case)

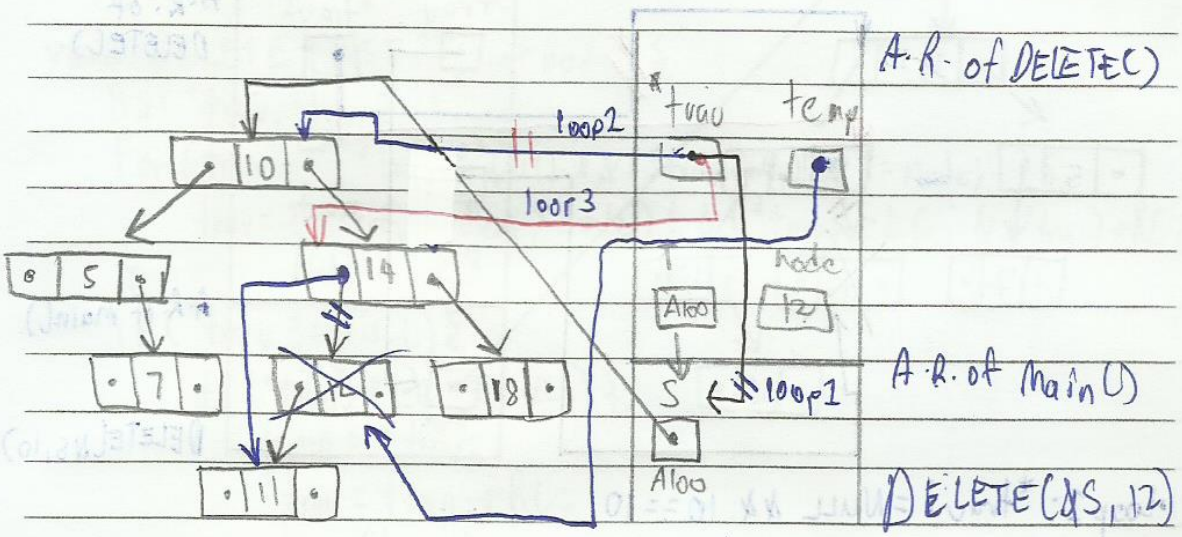
free(temp)

★ More Delete test cases ★

NO.

DATE

//Test case 2: if no Right child (left child only)



loop 1: trav != NULL && 10 != 12; (10 < 12) so & (*trav) → RC

loop 2: trav != NULL && 14 != 12; (14 > 12) so & (*trav) → LC

loop 3: trav != NULL && 12 == 12 → STOP LOOP

- Check cases:
  - case 1: 12 has no left child? NO
  - Case 2: 12 has no right child? YES

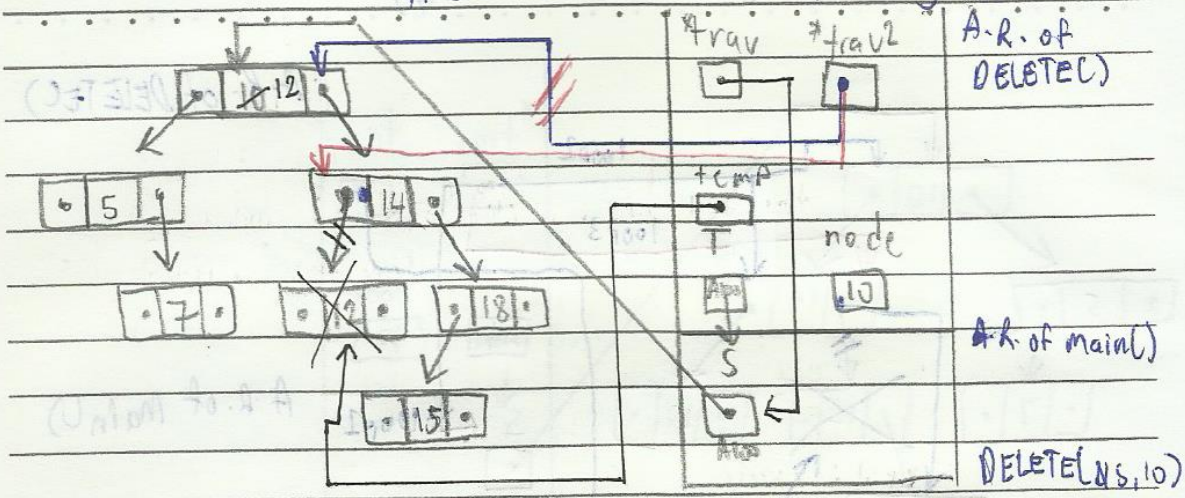
```
temp = 12
*trav = temp → LC (11)   // basically setting the new LC for 14
free (temp)
```

NO.

DATE

// Test case 3: if w/ right and left children

// NOTE: We are not actually deleting the node, just replace



- Loop 1: *trav != NULL && 10 == 10
  - Check cases:
    - Case 1: has no left child? NO
    - Case 2: has no right child? NO
    - Case 3: has two children? YES

    // loop for replacement node (1) - set trav2 to LC of 10
    - loop 1: (*trav2) → LC != NULL? YES → move

      // loop 1 checks the left child of 14.
      - loop 2: (*trav2) → LC != NULL? NO → stop loop

        // loop 2 checks the left child of 12.
      - temp = 12

        *trav2 = temp → RC (NULL in this case)

        (*trav) → elem = temp → elem (10 is replaced by 12)

        free(temp)

- Writers note: I've developed carpal tunnel

☆ THE CODE ☆
of DELETE

NO.

DATE

```
void DELETE (BST *T, int node) {
    BST *trav, *trav2, temp;
    for (trav = T; *trav != NULL && (*trav)->elem != node; ) {
        trav = ((*trav)->elem > node) ? &(*trav)->LC : &(*trav)->RC;
    }
    if (*trav != NULL) {
        if ((*trav)->LC == NULL) {
            temp = *trav;
            *trav = temp->RC;
            free(temp);
        }
        else if ((*trav)->RC == NULL) {
            temp = *trav;
            *trav = temp->LC;
            free(temp);
        } else {

            for (trav2 = &(*trav)->RC; (*trav2)->LC != NULL;
                                    trav2 = &(*trav2)->LC) {}
            temp = *trav2;
            *trav2 = temp->RC;
            (*trav)->elem = temp->elem;
            free(temp);
        }
    }
}
```

NO.

DATE

5.) MIN()

```
int MIN(BST T){
    BST trav;
    for(trav=T; trav!=NULL; trav->LC){}
    return trav->elem;
```

6.) MAX()

```
int MAX(BST T){
    BST trav;
    for(trav=T; trav!=NULL; trav->RC){}
    return trav->elem;
}
```

//leaving space incase i add more funcs.

★ Recursive Functions ★

//credits to Hannah 4 helpful codes'

NO.

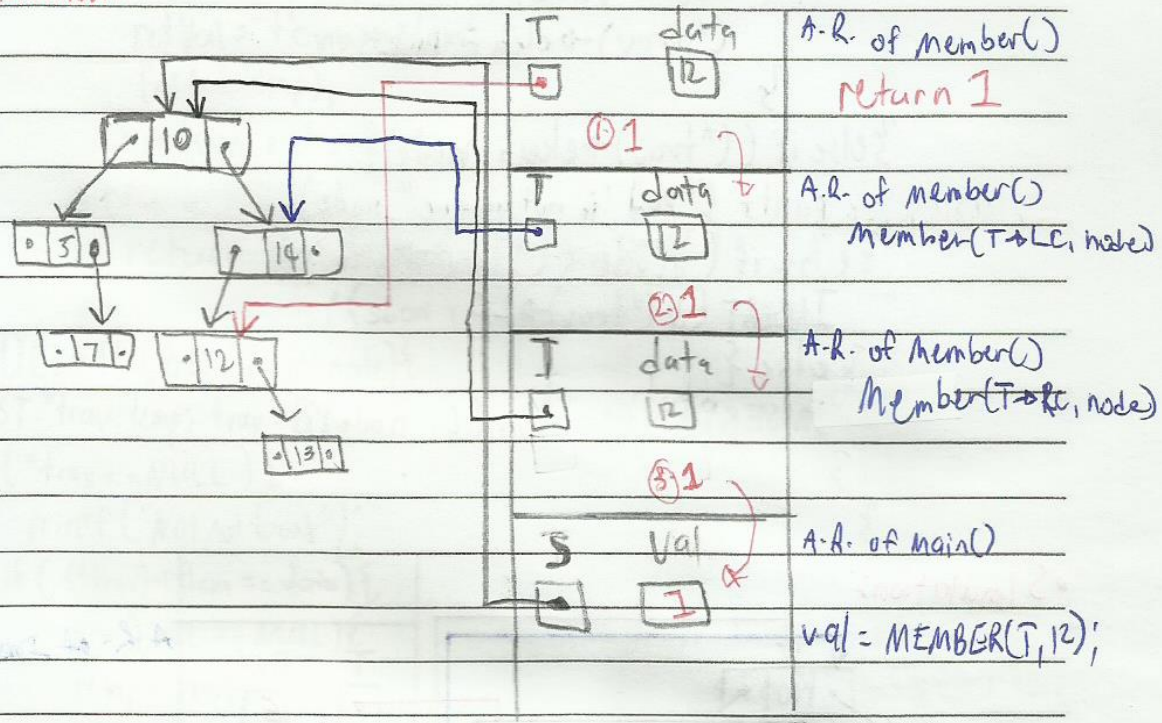
DATE

1.) MEMBER()

- Code:

```
int MEMBER (BST T, int node){
    if (T!=NULL && T->elem!=node){
     (node<T->elem)? MEMBER(T->LC, node): MEMBER(T->RC, node);
    }
    return (T!=NULL)? 1 : 0;
```

- Simulation:



Note: Simulation starts from the bottom

NO.

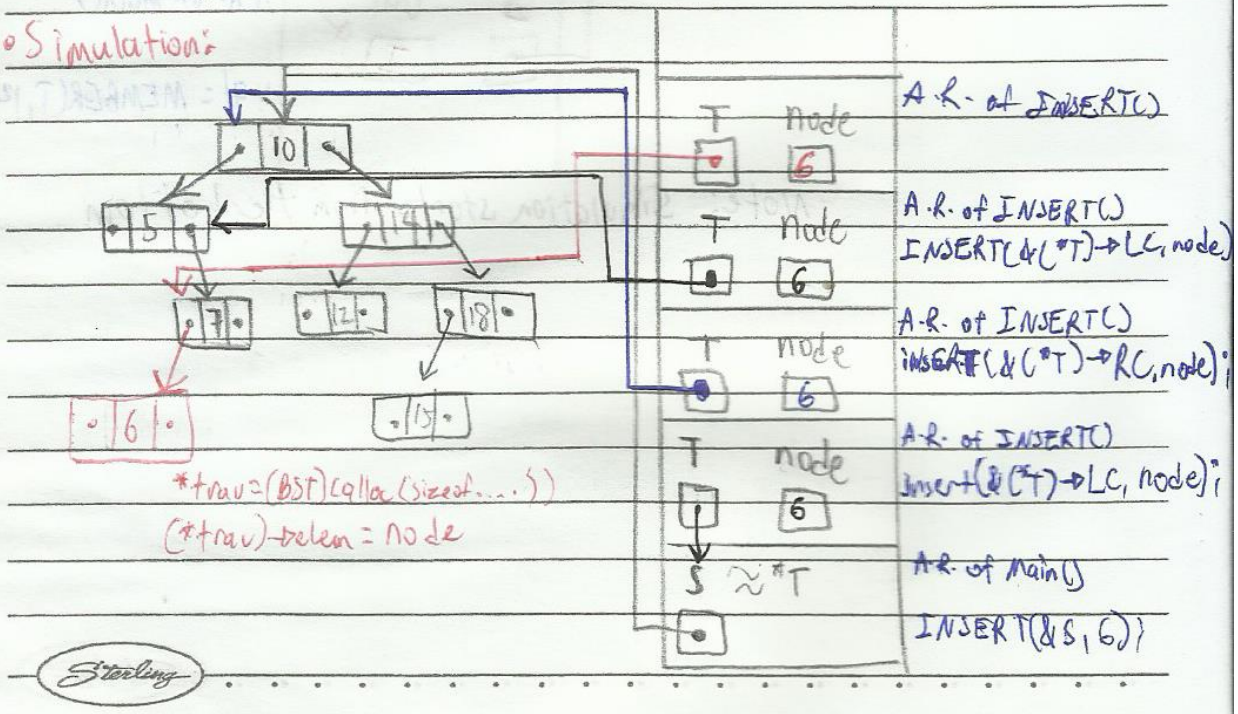
DATE

2.) INSERT()

• Code:

```
void INSERT(BST *T, int node){
    BST *trav;
    trav = T;
    if (*trav == NULL){
        *trav = (BST) calloc(1, sizeof(struct node));
        if (*trav != NULL){
            (*trav)->elem = node;
        }
    }else if ((*trav)->elem == node){
        printf("%d is not unique", node);
    }else if (node < (*trav)->elem){
        INSERT(&(*trav)->LC, node);
    }else{
        INSERT(&(*trav)->RC, node);
    }
}
```

• Simulation:



Sterling

NO.

DATE

3.) DeleteMin() & DELETEMIN()

```
int DeleteMin (BST *T){
    BST *trav, temp;
    int retval;
    trav = T;
    if ((*trav)->LC == NULL){
        temp = *trav;
        *trav = temp->RC;
        retVal = tcmp->elem;
        free (temp);
        return retval;
    }else{
        return DeleteMin(&(*trav)->LC);
```

```
void DELETE(BST *T, int node){
  BST *trav, temp; trav = B;
  if (*trav == NULL){
      printf("Not not found");
  }else if ( (*trav)->elem == data){
    if ((*trav)->RC == NULL){          // I am not simulating this it is too big
        temp = *trav;
        *trav = temp->LC;
        free(temp);
    }else if((*trav)->LC == NULL){
        temp = *trav; *trav = temp->RC; free(temp);
    }else{
    (*trav)->elem = DeleteMin(&(*trav)->RC);
    }
  }else if(data < (*trav)->elem){
    DELETE (&(*trav)->LC, node);
  }else{
    DELETE (&(*trav)->RC, node);
  }
```

NO.

DATE

~ RECURSIVE Traversals ~

1.) INORDER()

```
void INORDER(BST T){
  if(T!=NULL){
    INORDER(T->LC);
    Printf("%d ",T->elem);
    INORDER(T->RC);
  }
}
```

2.) PREORDER()

```
void PREORDER(BST T){
  if(T!=NULL){
    printf("%d ",T->elem);
    PREORDER(T->LC);
    PREORDER(T->RC);
  }
}
```

3.) POSTORDER()

```
void POSTORDER(BST T){
  if(T!=NULL){
    POSTORDER(T->LC);
    POSTORDER(T->RC);
    printf("%d ",T->elem);
  }
}
```

Sterling