Data Structures and Algorithms

Binary Search and Threaded Binary Tree

PRASHANT HEMRAJANI

Assistant Professor

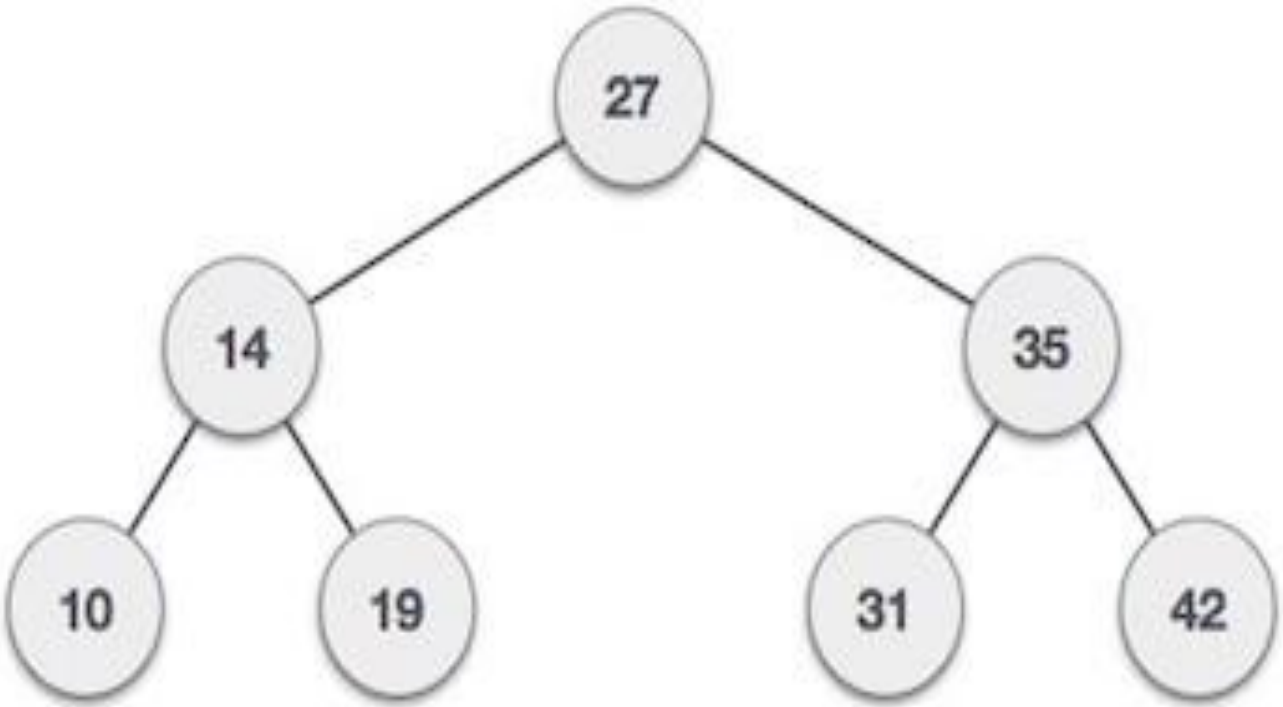
(Computer and Communication Engineering)

Introduction

- A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties −
  - The left sub-tree of a node has a key less than or equal to its parent node's key.
  - The right sub-tree of a node has a key greater than to its parent node's key.
- Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as −

  left_subtree (keys) ≤ node (key) ≤ right_subtree (keys)

Representation



Practice Work

Create a binary search tree with the input given below:

38, 2, 48, 12, 56, 32, 4, 67, 23, 87, 55, 46

- Insert 21, 39, 45, 54, and 63 into the tree
- Delete values 23, 56, 2, 45

Basic Operations

Following are the basic operations of a tree –

- **Search** – Searches an element in a tree.
- **Insert** – Inserts an element in a tree.
- **Pre-order Traversal** – Traverses a tree in a pre-order manner.
- **In-order Traversal** – Traverses a tree in an in-order manner.
- **Post-order Traversal** – Traverses a tree in a post-order manner.

Insertion

FIND (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)

- A binary search tree T is in memory and an ITEM of information is given. This procedure finds the location LOC of ITEM in T and the location PAR of the parent of ITEM. There are three special cases.

1. LOC= NULL and PAR=NULL will indicate that the tree is empty.
2. LOC ≠ NULL and PAR = NULL will indicate that Item is the root of T.
3. LOC = NULL and PAR ≠ NULL will indicate that ITEM is not in T and can be added to T as a child of the node N with location PAR

Insertion (cont…)

1. [Tree empty?]

   IF ROOT = NULL then Set LOC = NULL and PAR = NULL and Exit.

2. [ITEM at root?]

   IF ITEM= INFO [ROOT], then : Set LOC=ROOT

   and PAR = NULL and Exit.

3. [Initialize pointer PTR and SAVE.]

   IF ITEM < INFO [ROOT], then:

   Set PTR: = LEFT [ROOT] and SAVE= ROOT

   Else:

   Set PTR =RIGHT [ROOT] and SAVE = ROOT

   [End of If structure]

Insertion (cont…)

4. Repeat Steps 5 and 6 while PTR ≠ NULL
5. [ITEM found?]

   If ITEM= INFO [PTR], then: Set LOC: = PTR and PAR=SAVE, and Return.
6. IF ITEM < INFO [PTR], then:

   Set SAVE: =PTR and PTR= LEFT [PTR]

   Else

   Set SAVE= PTR and PTR=RIGHT [PTR]

   [End of If structure.]

   [End of step 4 loop.]
7. [Search unsuccessful.] Set Loc=NULL and PAR=SAVE
8. Exit.

Insertion (cont…)

INSBST (INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM, LOC)

1. **Call FIND (INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM, LOC and PAR)**
2. IF LOC ≠ NULL , then Exit.
3. [Copy ITEM into new node in AVAIL list]
   a) IF AVAIL = NULL, then: Write OVERFLOW and Exit.
   b) Set New: = AVAIL, AVAIL= LEFT [AVAIL] and INFO [NEW]=ITEM.
   c) Set LOC=NEW, LEFT[NEW]=NULL and RIGHT[NEW]=NULL
4. [Add ITEM to tree.]

   If PAR = NULL then;

   Set Root:= NEW.

   Else if ITEM< INFO[PAR], then

   Set LEFT [PAR]:= NEW.

   Else Set Right [PAR]=NEW.

   [End of If structure.]
5. Exit

Deletion

CASEA (INFO, LEFT, RIGHT, ROOT, LOC, PAR)

1. [Initializes CHILD]
   IF LEFT[LOC]= NULL and RIGHT [LOC]=NULL, then
      Set CHILD = NULL
   Else if LEFT[LOC] ≠ NULL then
      Set CHILD =LEFT[LOC]
   Else
      Set Child =RIGHT[LOC]
   [End of If structure]
2. If PAR ≠ NULL, then:
   If LOC =LEFT[PAR], then:
      Set LEFT [PAR]=CHILD
   Else:
      SET RIGHT [PAR]=CHILD
   [End of if structure.]
   Else
      Set ROOT=CHILD
   [End of If structure.]
3. Exit.

Deletion (cont...)

CASEB (INFO, LEFT, RIGHT, ROOT, LOC, PAR)

1. [Find SUC and PARSUC]
   a. Set PTR: = RIGHT [LOC] and SAVE = LOC
   b. Repeat while LEFT [PTR] ≠ NULL
      Set SAVE=PTR and PTR = LEFT [PTR]
      [End of loop]
   c. Set SUC= PTR and PARSUC = SAVE.
2. [Delete in-order successor, using Procedure CASEA.]
   CASEA (INFO, LEFT, RIGHT, ROOT, SUC, PARSUC)
3. [Replace node N by its in-order, successor.]
   a. If PAR ≠ NULL, then:
      IF LOC =LEFT [PTR], then
         Set LEFT [PAR]= SUC.
      Else:
         Set RIGHT [PAR]=SUC
      [End of if structure.]
   Else
      Set ROOT=SUC
   [End of if structure.]
   b. Set LEFT [SUC]= LEFT [LOC] and
   RIGHT [SUC]=RIGHT [LOC]
4. Exit.

Deletion (cont…)

DEL (INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM)

1. [Find the location of ITEM and its parent, using procedure FIND]
   Call FIND (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)
2. [ITEM in tree?]
   If LOC=NULL, then:= Write ITEM not in tree and Exit.
3. [Deletes node containing ITEM]
   If RIGHT [LOC] ≠ NULL and LEFT [LOC] ≠ NULL, then:
   Call CASEB (INFO, LEFT, RIGHT, ROOT, LOC, PAR)
   Else:
   Call CASEA (INFO, LEFT, RIGHT, ROOT< LOC, PAR)
   [End of If structure]
4. [Return deleted node to the AVAIL list]
5. Exit

Pre Order Traversal

PREORD (INFO, LEFT, RIGHT, ROOT)

1. [Initially push NULL onto STACK, and I initialize PTR]
   Set TOP=1, STACK [1]=NULL and PTR=ROOT
2. Repeat Steps 3 to 5 while PTR ≠ NULL:
3. Apply PROCESS to INFO [PTR].
4. [RIGHT child?]
   IF RIGHT [PTR] ≠ NULL, then : [Push on STACK]
   Set TOP: = TOP +1, and STACK [TOP]=RIGHT [PTR]
   [End of If structure.]
5. [Left child?]
   If LEFT [PTR] ≠ NULL, then
   Set PTR=LEFT [PTR].
   Else [Pop from STACK.]
   Set PTR = STACK [TOP] and TOP =TOP-1
   [End of step 2 loop]
6. Exit.

In Order Traversal

INORD (INFO, LEFT, RIGHT, ROOT)

1. [Push NULL onto STACK and initialize PTR.]
   Set TOP =1, STACK [1]=NULL and PTR=ROOT.
2. Repeat while PTR ≠ NULL: [Pushes left most path onto STACK.]
   Set TOP = TOP +1 and STACK [TOP]=PTR [SAVES Node].
   Set PTR= LEFT[PTR. [Updates PTR]
   [End of loop]
3. Set PTR=STACK [TOP] and TOP=TOP-1 [Pops node from STACK.]
4. Repeat Steps 5 to 7 While PTR ≠ NULL [Backtracking.]
5. Apply PROCESS to INFO [PTR].
6. [Right child?]
   IF RIGHT [PTR] ≠ NULL, then:
   Set PTR= RIGHT [PTR]
   Go to step 2
   Else:
   Set PTR=STACK [TOP] and TOP=TOP-1. [Pops node]
   [End of step 4 loop.]
7. Exit.

Post Order Traversal

POSTORD (INFO, LEFT, RIGHT, ROOT)

1. [Push NULL onto STACK and initialize PTR]
   Set TOP=1, STACK [1]=NULL and PTR=ROOT
2. [Push left most path onto STACK]
   Repeat steps 3 to 5 while PTR ≠ NULL:
3. Set TOP=TOP +1 and STACK [TOP]=PTR
   [Pushes PTR on STACK]
4. If RIGHT [PTR] ≠ NULL then [Push on STACK.]
   Set TOP = TOP+1 and STACK [TOP]= RIGHT[PTR]
   Else
   Set PTR=LEFT [PTR] [Updates pointer PTR]
   [End of if structure]
   [End of step 2 loop]
5. Set PTR =STACK[TOP] and TOP = TOP-1
   [Pops node from STACK.]
6. Repeat while PTR>0:
   Apply PROCESS to INFO [PTR]
   Set PTR =STACK [TOP] and [TOP]=TOP-1
   [Pops node from STACK]
   [end of loop]
7. If PTR< 0 then:
   Set PTR = -PTR
   GO to step 2
   [End of if structure.]
8. Exit.

Pre Order (Recursive)

PREORD (INFO, LEFT, RIGHT, ROOT)

- This is a recursive procedure to traverse the tree starting from root in preorder (i.e. root, left, right and APPLY a PROCESS of INFO of each node).

1. If ROOT = NULL, then Return
2. Apply a PROCESS on INFO [ROOT]
3. Call PREORD (INFO, LEFT, RIGHT, LEFT [ROOT])
4. Call PREORD (INFO, LEFT, RIGHT, RIGHT[ROOT])
5. Exit.

In Order (Recursive)

INORD (INFO, LEFT, RIGHT, ROOT)

- This is a recursive procedure to traverse the tree starting from root in in-order (i.e. left, root, right and APPLY a PROCESS of INFO of each node.)

1. If ROOT =NULL then, Return.
2. Call INORD (INFO, LEFT, RIGHT, LEFT [ROOT])
3. Apply a PROCESS on INFO[ROOT]
4. Call INORD(INFO, LEFT, RIGHT, RIGHT[ROOT])
5. Exit.

Post Order (Recursive)

POSTORD (INFO, LEFT, RIGHT, ROOT)

- This is a recursive procedure to traverse the tree starting from root in post-order (i.e. left, right, root and Apply a Process of INFO of each node.)

1. If ROOT = NULL then: Return
2. Call POSTORD (INFO, LEFT, RIGHT, LEFT[ROOT])
3. Call POSTORD (INFO, LEFT, RIGHT, LEFT[ROOT])
4. Apply a PROCESS on INFO [ROOT]
5. Exit.

Threaded Binary Tree

- A binary tree is threaded by making all right child pointers that would normally be null point to the inorder successor of the node (if it exists), and all left child pointers that would normally be null point to the inorder predecessor of the node.
- We have the pointers reference the next node in an inorder traversal; called threads
- We need to know if a pointer is an actual link or a thread, so we keep a boolean for each pointer.

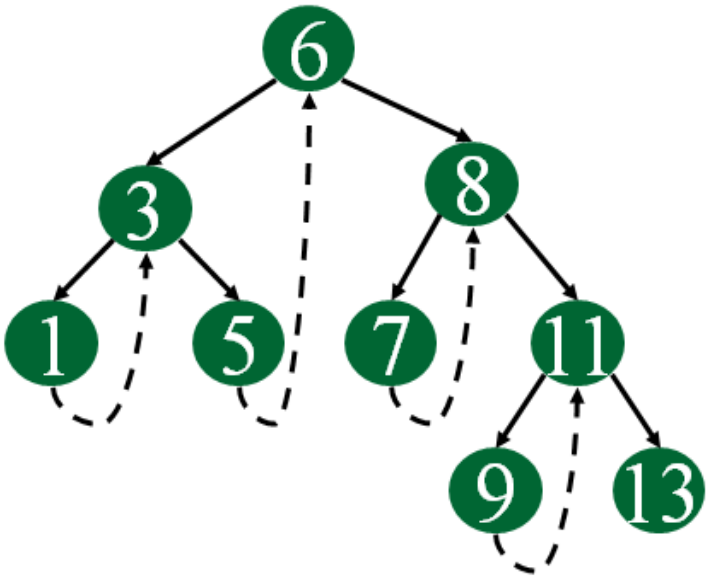
Need of Threaded Binary Tree

- Binary trees have a lot of wasted space: the leaf nodes each have 2 null pointers. We can use these pointers to help us in inorder traversals.
- Threaded binary tree makes the tree traversal faster since we do not need stack or recursion for traversal

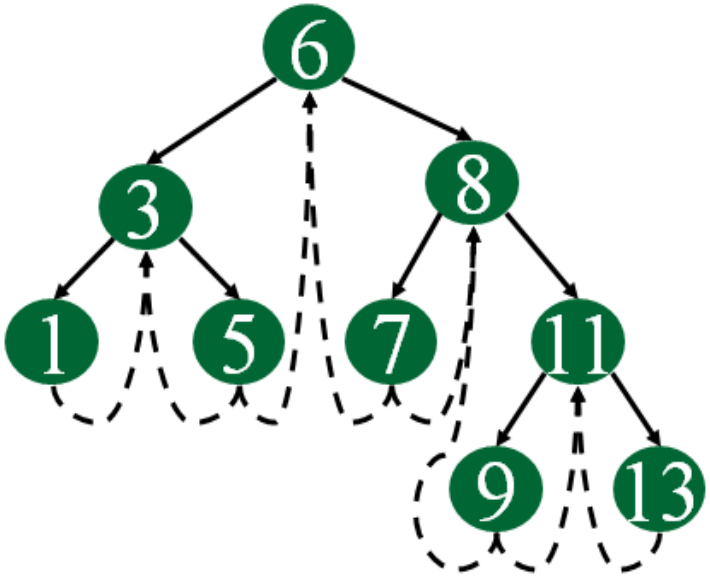
Types of Threaded Binary Tree

- **Single Threaded**: Each node is threaded towards either the in-order predecessor or successor (left **or** right) means all right null pointers will point to inorder successor **OR** all left null pointers will point to inorder predecessor.
- **Double threaded:** Each node is threaded towards both the in-order predecessor and successor (left **and** right) means all right null pointers will point to inorder successor **AND** all left null pointers will point to inorder predecessor.

Single and Double Threaded Binary Tree

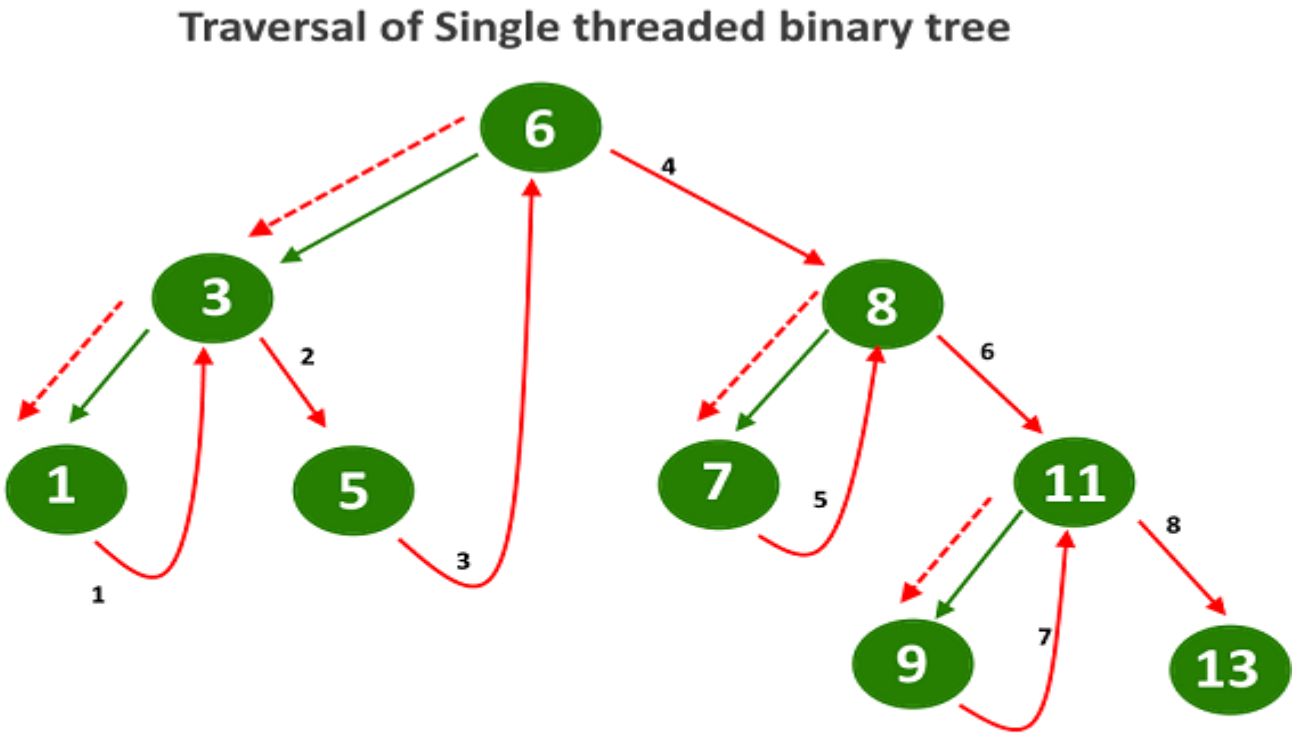


Single Threaded Binary Tree



Double Threaded Binary Tree

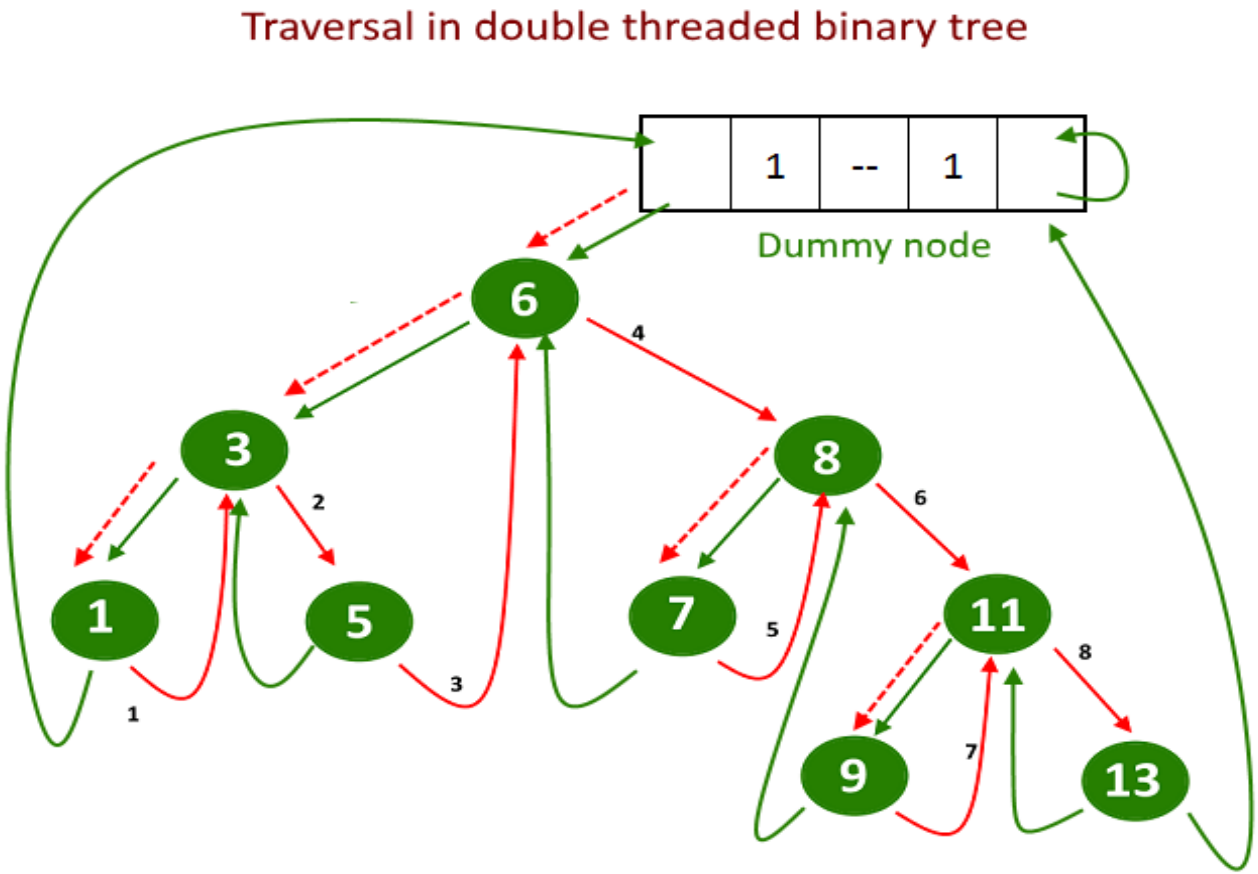
Traversal of Single Threaded Binary Tree



Output : 1 3 5 6 7 8 9 11 13

Follow the red arrow, dotted arrow when moving to left most node from the current node and solid arrow when using the right pointer to move it to it's inorder successor.

Traversal of double Threaded Binary Tree



Follow the red arrow, dotted arrow when moving to left most node from the current node and solid arrow when using the right pointer to move it to it's inorder successor.

Any Queries
????