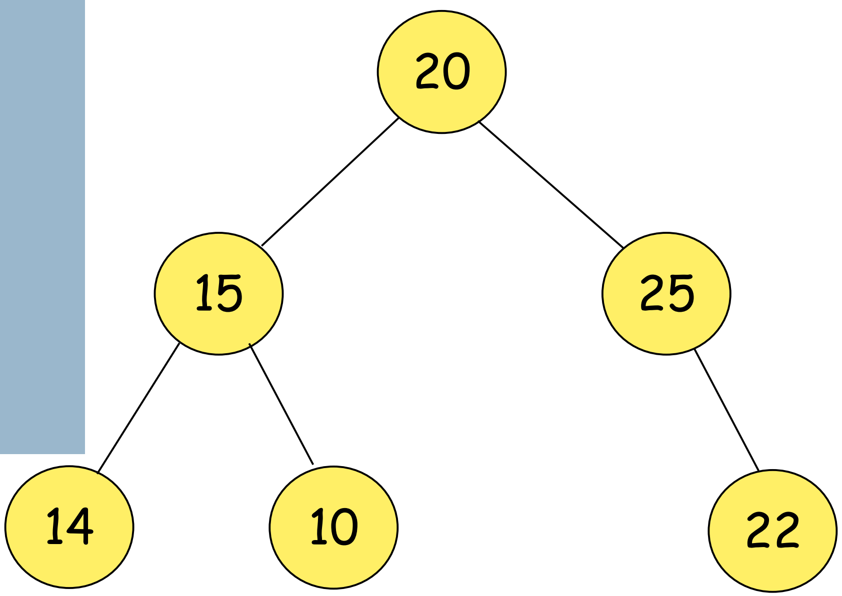
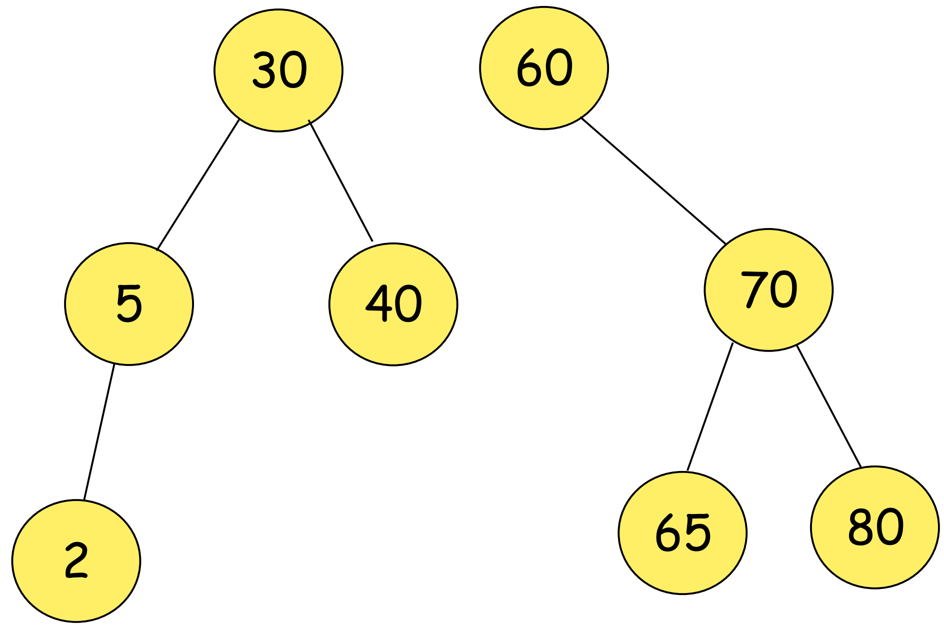
Binary Search Tree

- Definition: A binary search tree is a binary tree. It may be empty. If it is not empty then it satisfies the following properties:
  - Every node has exactly one key and no two nodes have the same key (i.e., the keys in the tree are distinct)
  - The keys (if any) in the left subtree are smaller than the key in the root.
  - The keys (if any) in the right subtree are larger than the key in the root.
  - The left and right subtrees are also binary search trees.
- Inorder traversal of BST results in ascending order of elements

Binary Trees



Not binary search tree



Binary search trees

Recursive function to create a BST

```
Nodeptr CreateBST(Nodeptr root, int item){
    if (root==NULL){
        root = getnode();
        root->data= item;
        root->lchild=root->rchild = NULL;
        return root;
    }
    else
    if (item<root->data)
        root->lchild = CreateBST(root->lchild, item);
    else
        if (item>root->data)
            root->rchild = CreateBST(root->rchild, item);
        else
            printf("Duplicates are not allowed\n");
    return root;
}
```

Searching in A Binary Search Tree

- If the root is NULL, then this is an empty tree. No search is needed.
- If the root is not NULL, compare the x with the key of root.
  - If x equals to the key of the root, then it's done.
  - If x is less than the key of the root, then no elements in the right subtree can have key value x. We only need to search the left tree.
  - If x larger than the key of the root, only the right subtree is to be searched.

Searching a BST

```
typedef struct node *Nodeptr;
struct node{
    int data;
    Nodeptr rchild;
    Nodeptr lchild;
};
Nodeptr search(Nodeptr root,int key)
{
/* return a pointer to the node that contains key.
  If there is no such node, return NULL */

  if (root==NULL) return NULL;
  if (key == root->data) return root;
  if (key < root->data)
      return search(root->lchild, key);
  return search(root->rchild,key);
}
```

Iterative Searching Algorithm

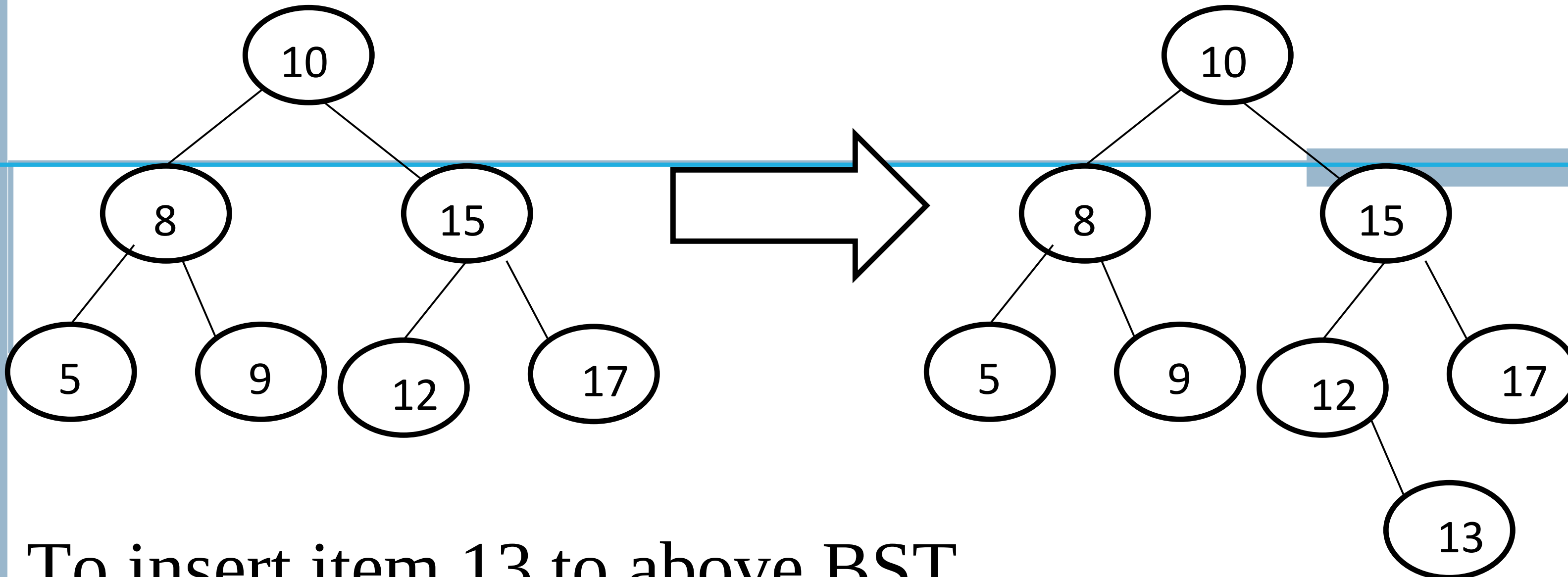
```
Nodeptr itersearch(Nodeptr root, int key)
{
  while (root) {
    if (key == root->data) return root;
    if (key < root->data)
        root = root->lchild;
    else root = root->rchild;
  }
  return NULL;
}
```

Other operations:

1. Finding the maximum element in BST: maximum element will always be the last right child in a BST. Move to the rightmost node in a BST and you will end up in the maximum element.
2. Finding the minimum element: Move to the leftmost child and you will reach the least element in BST.
3. Finding the height of a tree: height is nothing but maximum level in the tree plus one. It can be easily found using recursion.

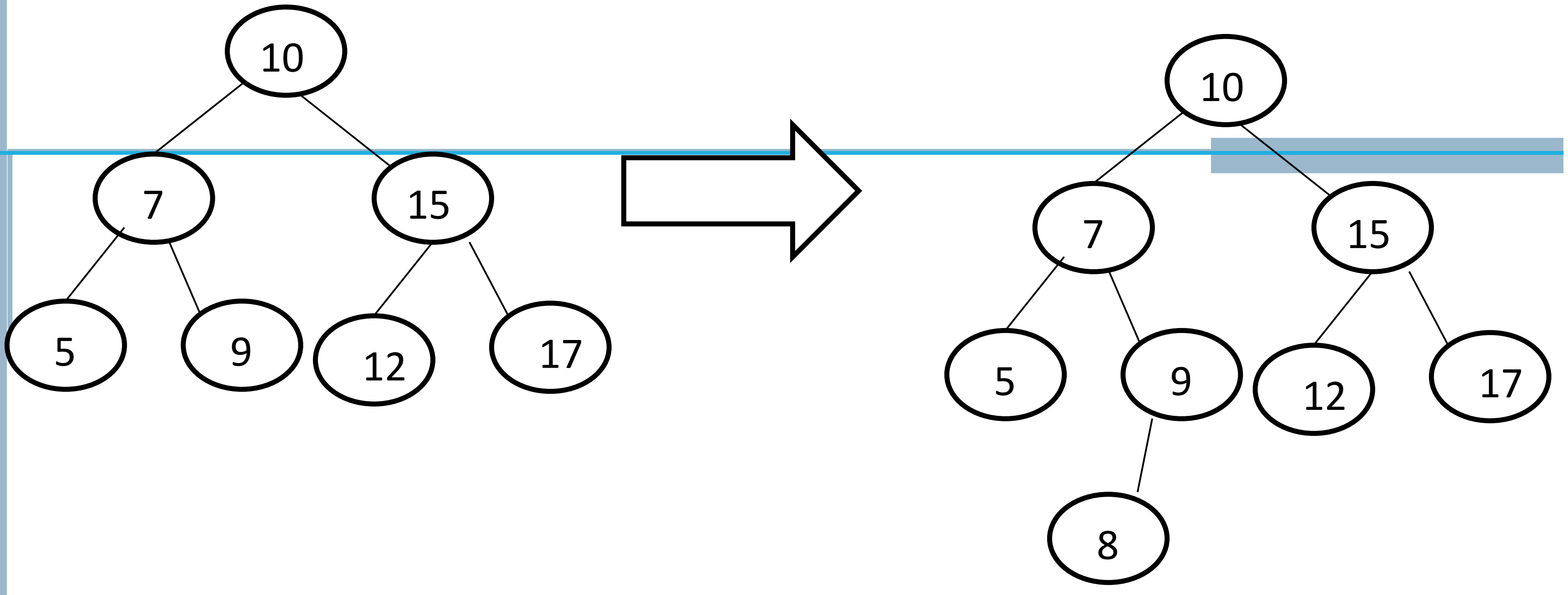
Insertion To A Binary Search Tree

- Before insertion is performed, a search must be done to make sure that the value to be inserted is not already in the tree.
- If the search fails, then we know the value is not in the tree. So it can be inserted into the tree.
- If item is lesser than the root item, move to left or else move to the right of root node.
- This process is repeated until the correct position is found.



To insert item 13 to above BST

- Compare with the root item. 13> 10, hence move to right and reach 15.
- Now 13<15, So go to left and reach 12.
- 13>12, hence move right.
- Now the correct position is found and hence insert the new node to the right of 12.



To insert 8 into the above tree

- Compare with root item. 8<10, hence move left and reach 7.
- Now 8>7. So move right and reach 9.
- 8<9. Move left and the correct position is obtained.
- Insert 8 to the left of 9.

Insertion Into A Binary Search Tree

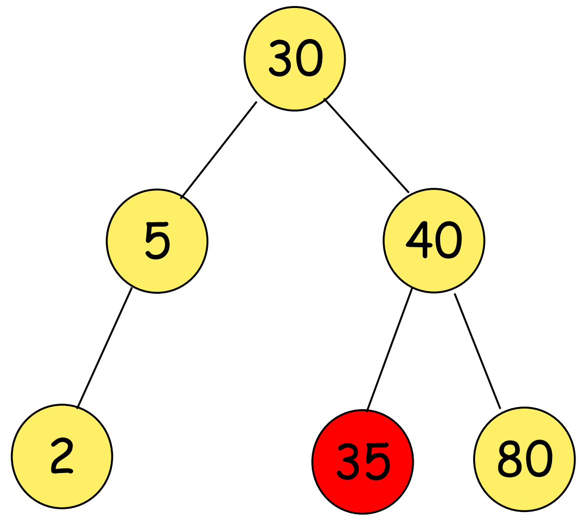
```
typedef struct node *Nodeptr;

struct node{
    int data;
    Nodeptr rchild;
    Nodeptr lchild;
};
void Insert(Nodeptr *root, int item){

    Nodeptr temp= getnode();
    temp->data = item;
    temp->lchild = NULL;
    temp->rchild = NULL;
    if (*root==NULL) {
            *root = temp; return;
     }

    Nodeptr parent, cur;
```

```
/*traverse until correct position is
found*/

 parent=NULL;
 cur=*root;
 while(cur){
        parent=cur;
        if (item==cur->data ){
            printf("Duplicates Not allowed");
            free(temp);
            return;
        }
        else if (item<cur->data)
                cur=cur->lchild;
             else
                cur=cur->rchild;
    }
    if (item<parent->data)
        parent->lchild = temp;
    else
        parent->rchild = temp;
    return;
}
```

Deletion From A Binary Search Tree

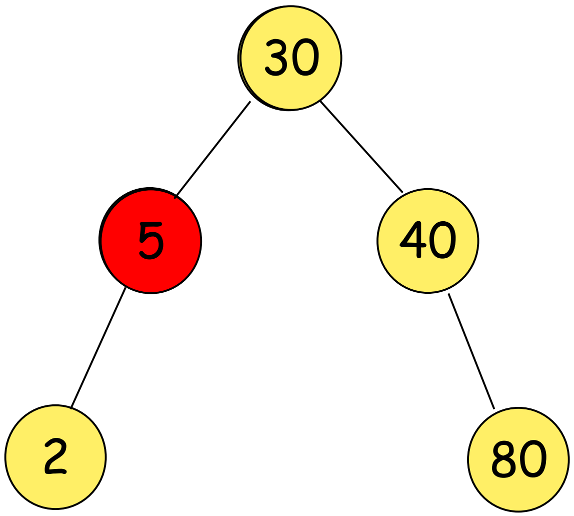
- Delete a leaf node
  - A leaf node which is a right child of its parent
  - A leaf node which is a left child of its parent
- Delete a non-leaf node
  - A node that has one child
  - A node that has two children
    - Replaced by the largest element in its left subtree, or
    - Replaced by the smallest element in its right subtree

Deleting From A Binary Search Tree



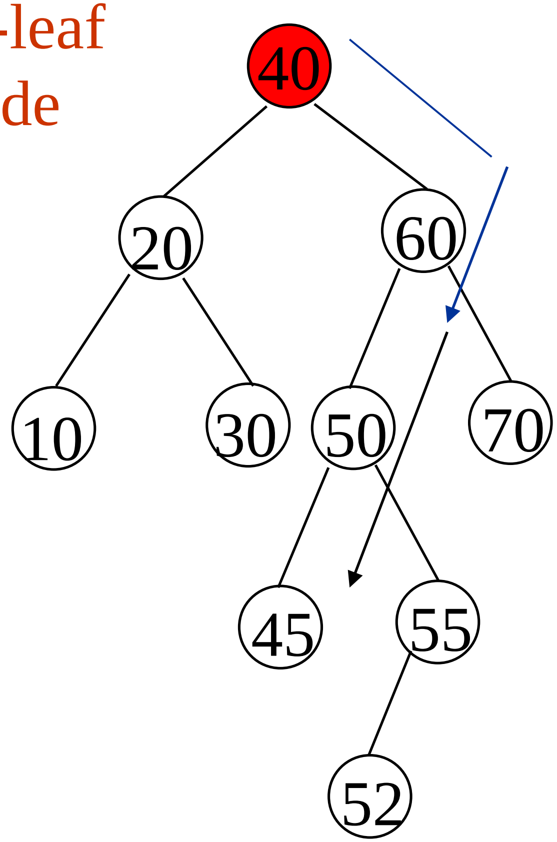
Deleting From A Binary Search Tree

non-leaf
node

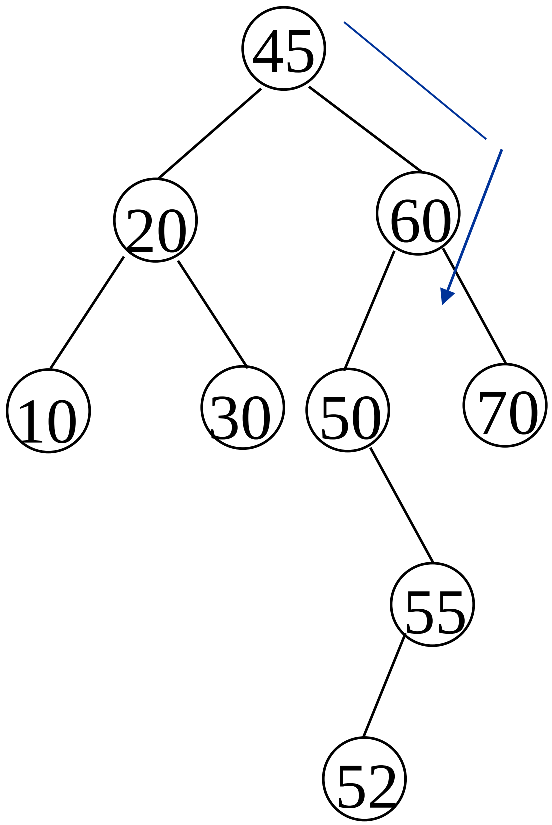


Deletion from a Binary Search Tree

non-leaf
node



Before deleting 40



After deleting 40

Function for BST Delete (Replaced by the smallest element in its right subtree)

```
void Delete(Nodeptr *root, int item){
       Nodeptr parent, cur;
        Nodeptr q, succ;

       if (*root== NULL){
            printf("Empty Tree\n");   return;
       }
       //traverse the tree until the item is found or entire tree is traversed
       parent = NULL;
       cur = *root;
       while(cur && (cur->data!= item)){
           parent = cur;
           if (item<cur->data)
               cur = cur->lchild;
           else
               cur = cur->rchild;
       }
       if (cur==NULL) {
            printf("Item Not Found\n");
            return;
       }
```

Function for BST Delete (Replaced by the smallest element in its right subtree)

```
//item found and check for case 1
if (cur->lchild == NULL) //node to be deleted has empty left subtree
   q= cur->rchild;    //get the address of right subtree
else if (cur->rchild == NULL) //node to be deleted has empty right subtree
   q = cur->lchild;        //get the address of left subtree
else //interior node
{
    //find inorder successor->smallest element in the right subtree
   parent = cur;
   succ = cur->rchild; //get address of rightchild of node to be deleted*/

   while (succ->lchild){   //move to the leftmost node of succ
      parent = succ;
      succ= succ->lchild;
   }
   cur->data = succ->data;//exchange the data of current and succ;
   cur = succ;
   q = cur->rchild;
}
```

Function for BST Delete (Replaced by the smallest element in its right subtree)

```
    if (parent == NULL){
        free(cur);
        *root = q;
        return;
    }
if (cur== parent->lchild)
        parent->lchild = q;
   else
        parent->rchild = q;
     free(cur);
     return;
}
```

Construct a binary tree from its inorder and preorder traversals

inorder [4, 2, 1, 7, 5, 8, 3, 6]

preorder [1, 2, 4, 3, 5, 7, 8, 6]

step 1: select root from preorder

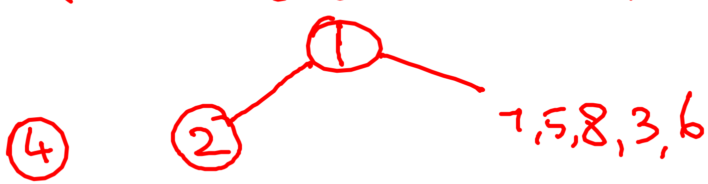
1

step 2: Look for nodes of LST and RST of root in inorder

LST (4, 2)

RST (7, 5, 8, 3, 6)

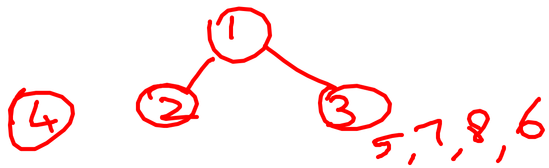
step 3: Select root of LST from preorder



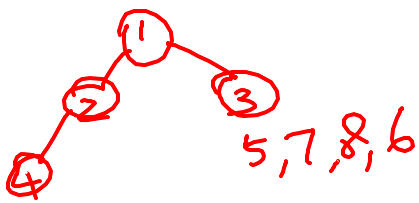
inorder [4,2,1,7,5,8,3,6]

preorder [1,2,4,3,5,7,8,6]

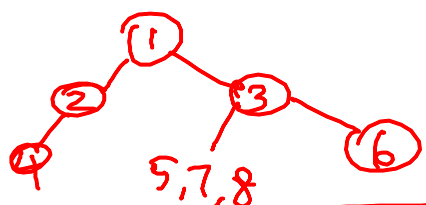
Step 4. Select root of RST from preord



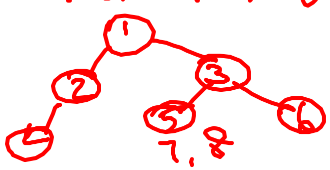
Step 5: Look for LST & RST of 2 in inorder



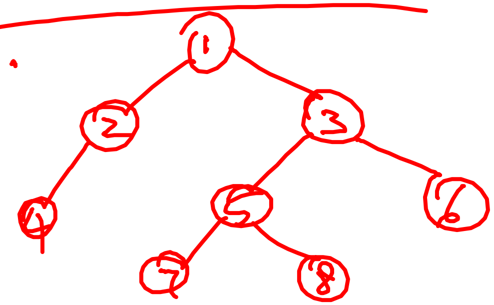
Step 6: Select LST & RST of 3 from inorder



Step 7: select root of LST of 3



Step 8.



Construct a binary tree from its inorder and postorder traversals

inorder [4, 2, 1, 7, 5, 8, 3, 6]

postorder [4, 2, 7, 8, 5, 6, 3, 1]

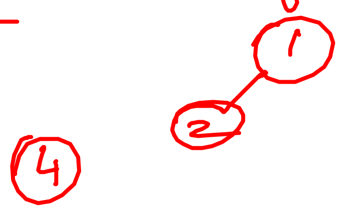
Step 1: select root from postorder

1

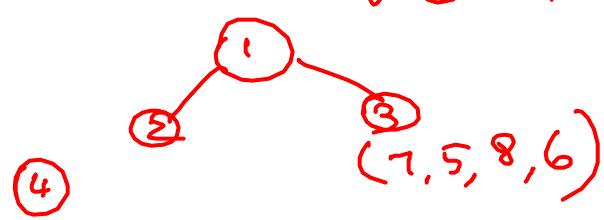
Step 2: LST & RST of 1 from inorder

(4, 2) (7, 5, 8, 3, 6)

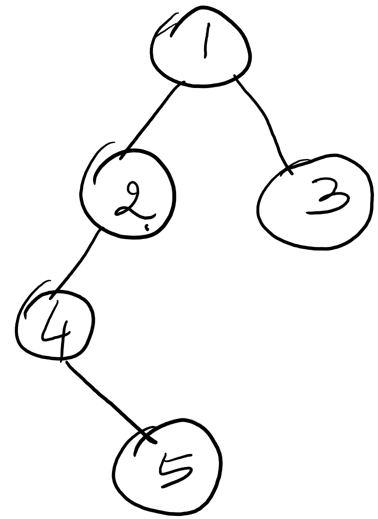
Step 3: root of LST of 1



Step 4: Select root of RST of 1 from



```
bool printAncestors(struct node *root, int target)
{

  if (root == NULL)
     return false;
   if (root->data == target)
     return true;

  if ( printAncestors(root->left, target) ||
       printAncestors(root->right, target) )
  {
    printf("%d  ", root->data );
    return true;
  }
  return false;
}
```



print Ancestors (root, 5)

o/p = 4, 2, 1

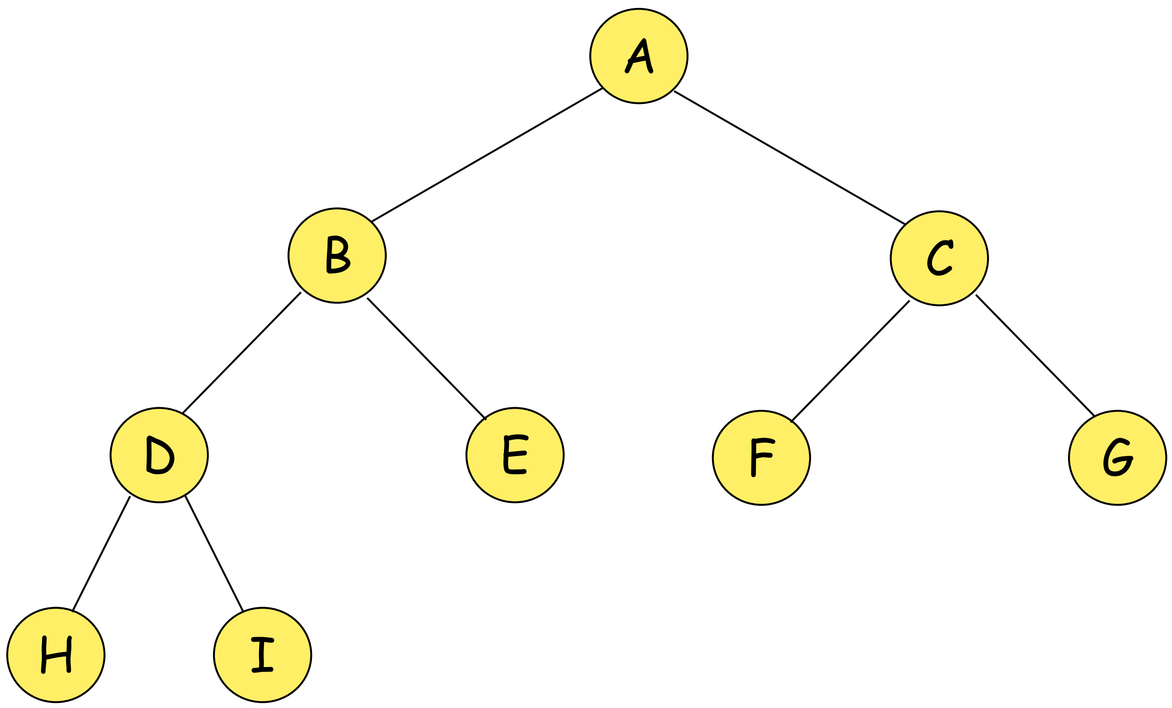
Threaded Binary Tree

- Threads
  - In a linked representation of a binary tree, there are more NULL links than actual pointers.
  - In a binary tree with n nodes containing 2n links, there are n+1 NULL links.
  - Perlis and Thornton devised a way to make use of NULL links.
  - Here the NULL links are replaced by pointers, called *threads*, to other nodes in the tree.

Threaded Binary Tree

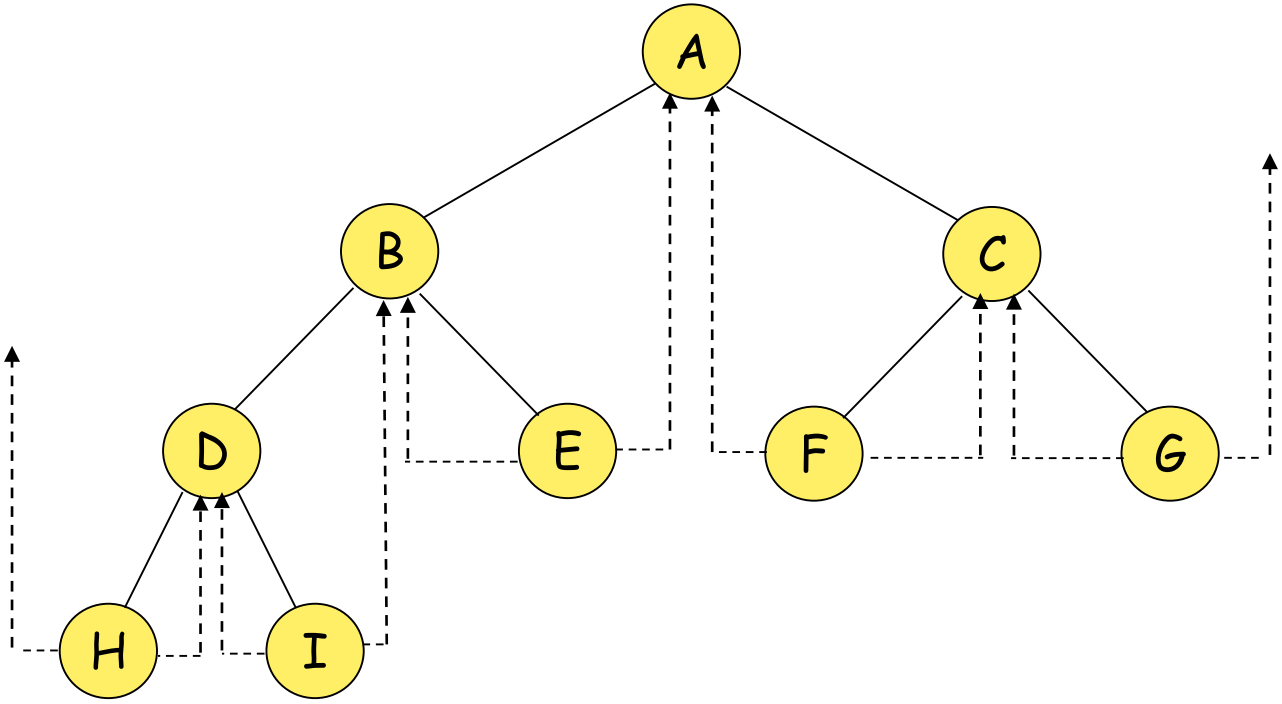
- Threading Rules
  - A NULL RightChild field at node p is replaced by a pointer to the node that would be visited after p when traversing the tree in inorder. That is, it is replaced by the inorder successor of p.
  - A NULL LeftChild link at node p is replaced by a pointer to the node that immediately precedes node p in inorder (i.e., it is replaced by the inorder predecessor of p).

A Binary Tree



Inorder sequence: H, D, I, B, E, A, F, C, G

Threaded Tree Corresponding to Given Binary Tree



Inorder sequence: H, D, I, B, E, A, F, C, G

Threads

- To distinguish between normal pointers and threads, two boolean fields, LeftThread and RightThread, are added to the record in memory representation.
  - t->leftThread= TRUE

    => t->lchild is a **thread**
  - t->leftThread= FALSE

    ⇒t->lchild is a **pointer** to the left child.
  - t->rightThread= TRUE

    => t->rchild is a **thread**
  - t->rightThread= FALSE

    => t->rchild is a **pointer** to the right child.

Threaded Binary Tree Node Structure Declaration

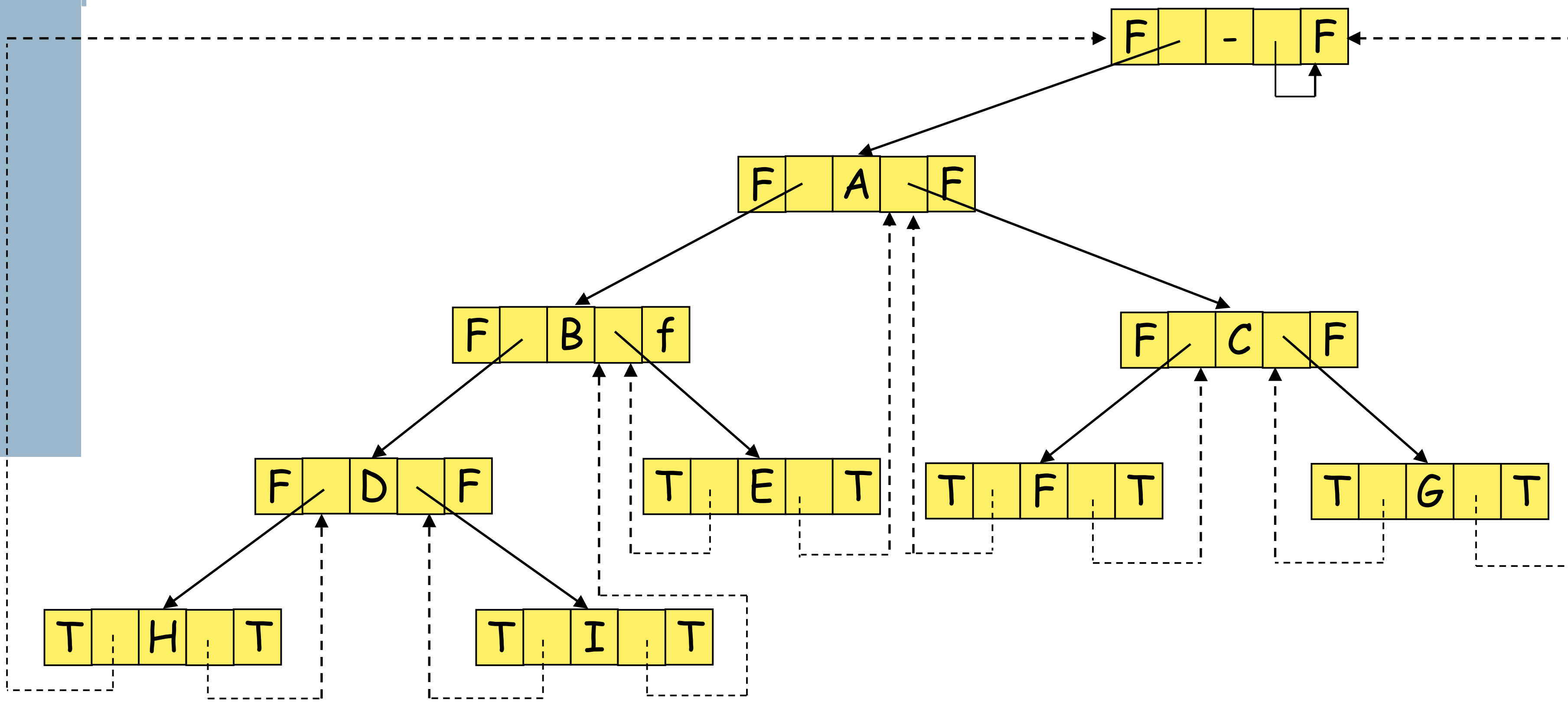
```
typedef struct threadedTree *threadedPointer;

struct threadedTree{
        short int leftThread;
        threadedPointer lchild;
        char data;
        threadedPointer rchild;
        short int rightThread;
};
```

Threads (Cont.)

- To avoid dangling threads, a head node is used in representing a binary tree.
- The original tree becomes the left subtree of the head node.
- Empty Binary Tree

leftThread	lchild	data	rchild	rightThread
TRUE				FALSE

Memory Representation of Threaded Tree



Finding the inorder successor without stack

- By using the threads, we can perform an inorder traversal without making use of a stack.

```
threadedPointer insucc(threadedPointer node)
{//Return the inorder successor of node
     threadedPointer temp = node-> rchild;
    if (node->rightThread==FALSE)
          while (temp->leftThread==FALSE)
                    temp = temp -> lchild;
    return temp;
    }
```

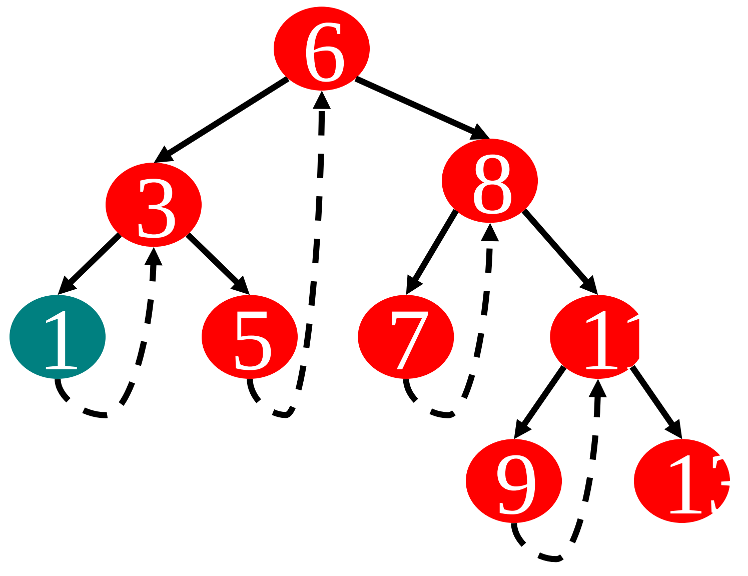
Inorder Traversal of a threaded Binary Tree

```
void tinorder(threadedPointer treehead)
{
   threadedPointer temp = treehead;
   while(1){
      temp = insucc(temp);
      if (temp == treehead) break;
      printf("%c", temp->data);
   }
}
```

Threaded Tree Traversal

- We start at the leftmost node in the tree, print it, and follow its right thread
- If we follow a thread to the right, we output the node and continue to its right
- If we follow a link to the right, we go to the leftmost node, print it, and continue

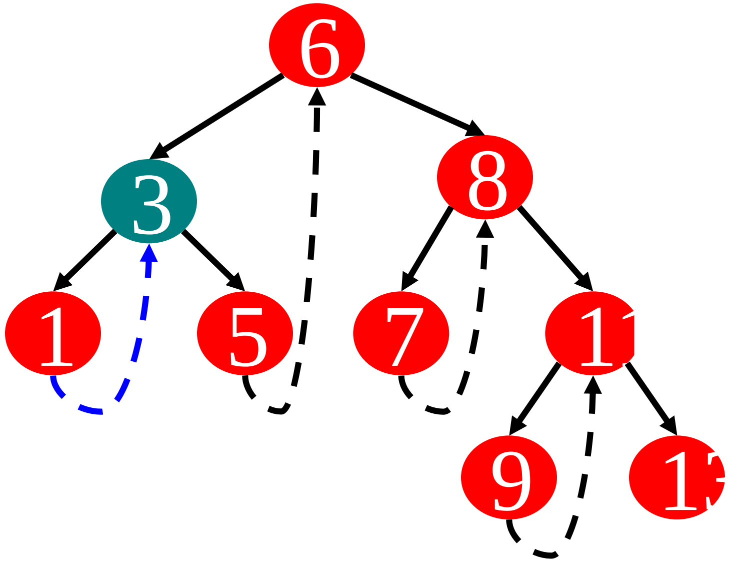
Threaded Tree Traversal



Output
1

Start at leftmost node, print it

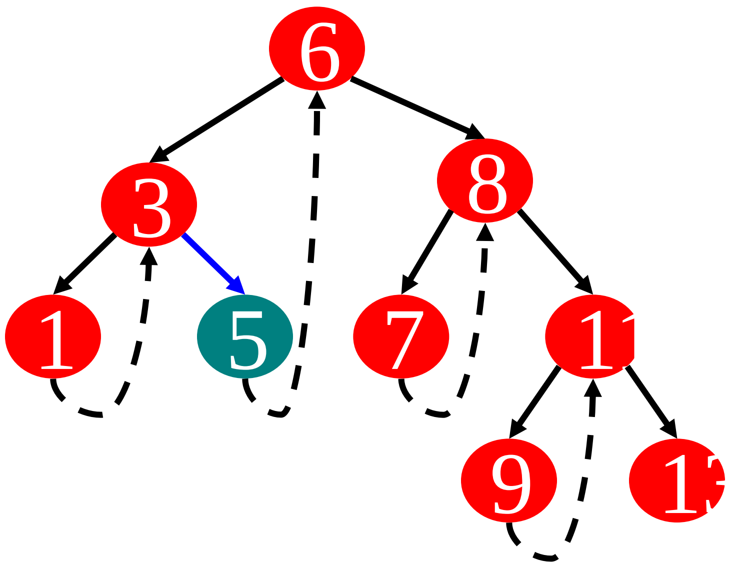
Threaded Tree Traversal



Output
1
3

Follow thread to right, print node

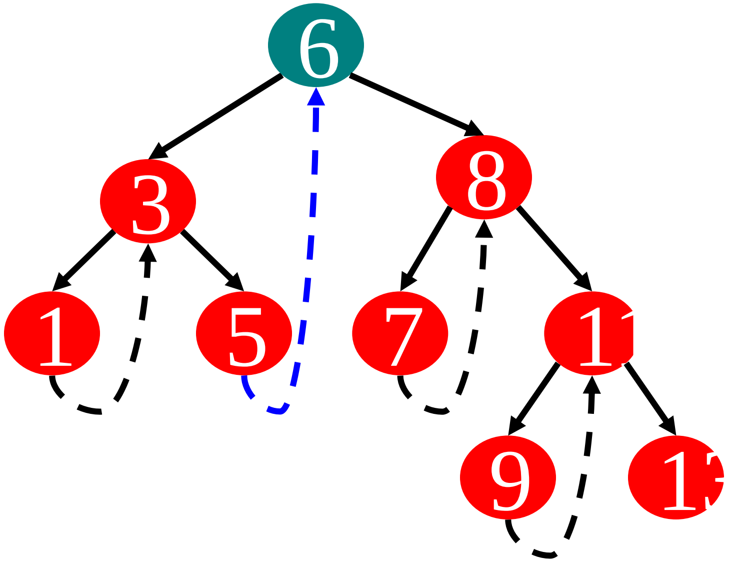
Threaded Tree Traversal



Output
1
3
5

Follow link to right, go to leftmost node and print

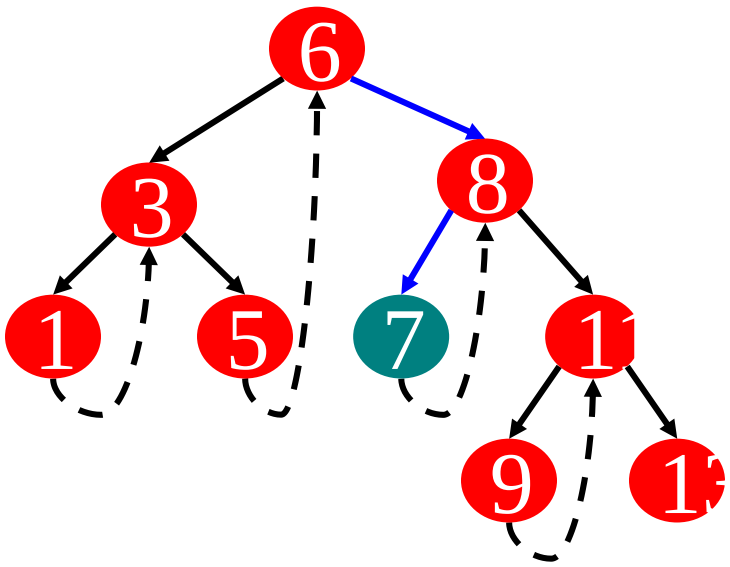
Threaded Tree Traversal



Output
1
3
5
6

Follow thread to right, print node

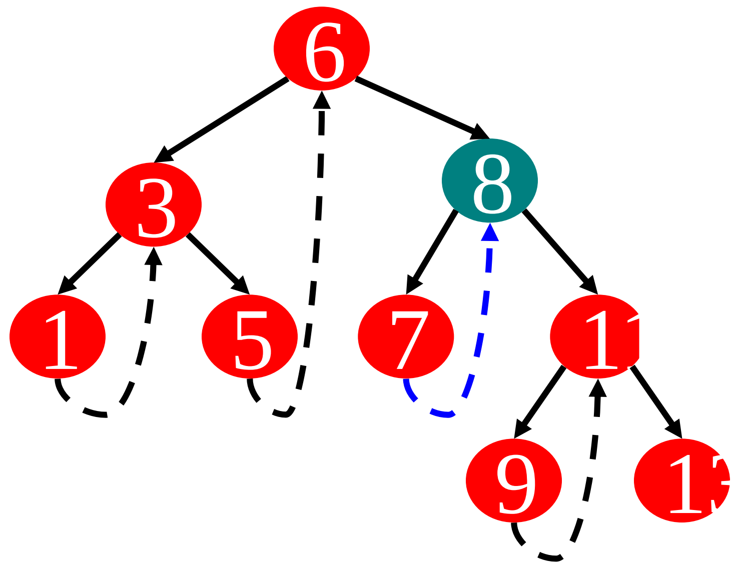
Threaded Tree Traversal



Output
1
3
5
6
7

Follow link to right, go to leftmost node and print

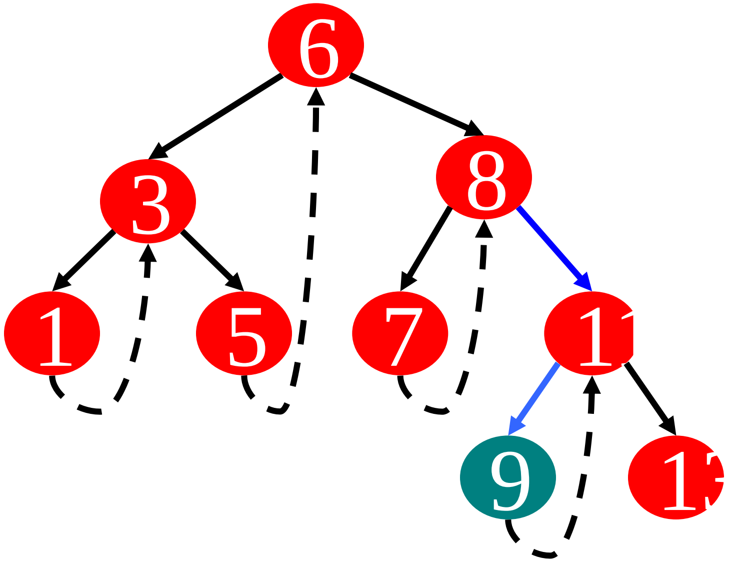
Threaded Tree Traversal



Output
1
3
5
6
7
8

Follow thread to right, print node

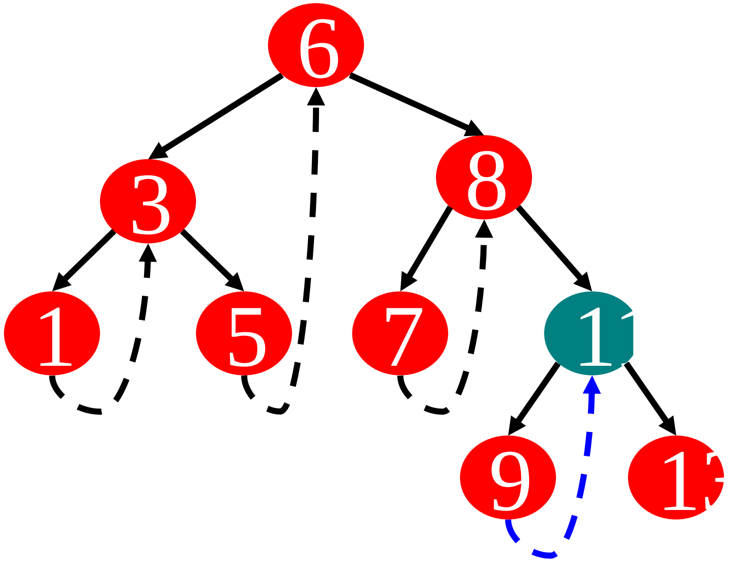
Threaded Tree Traversal



Output
1
3
5
6
7
8
9

Follow link to right, go to leftmost node and print

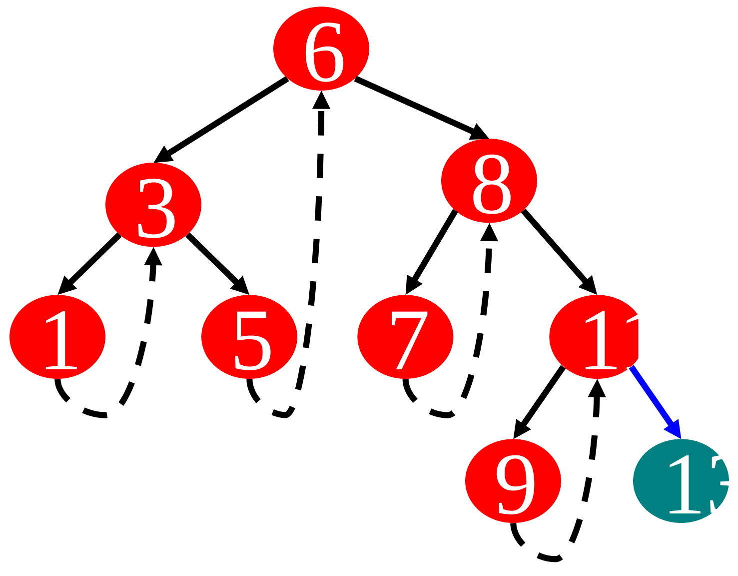
Threaded Tree Traversal



Output
1
3
5
6
7
8
9
11

Follow thread to right, print node

Threaded Tree Traversal



Output
1
3
5
6
7
8
9
11
13

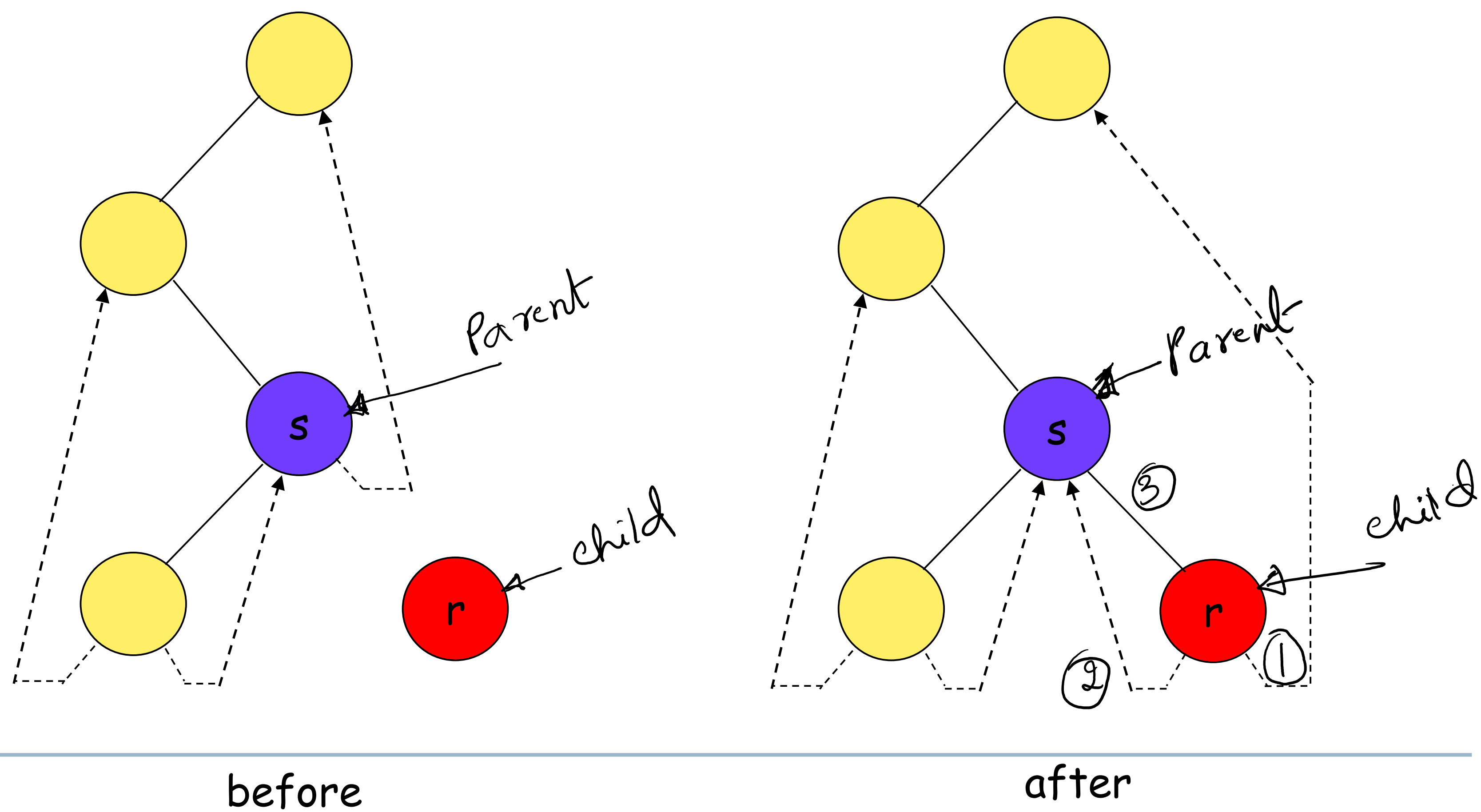
Follow link to right, go to leftmost node and print

Inserting A Node to AThreaded Binary Tree

- Inserting a node r as the right child of a node s.
  - If s has an empty right subtree, then the insertion is simple (as shown in diagram next slide)
  - If the right subtree of s is not empty, then, this right subtree is made the right subtree of r after insertion. When this is done, r becomes the inorder predecessor of a node that has a leftThread==TRUE field, and consequently there is an thread which has to be updated to point to r. The node containing this thread was previously the inorder successor of s. Figure illustrates the insertion for this case.

Insertion of r As A Right Child of s in A Threaded Binary Tree

case (a) (Empty rt subtree for s)



Insertion of r As A Right Child of s in A Threaded Binary Tree (Cont.)