Tree Applications

1

Application of Binary tree

1. Expression representation : Known as expression tree which have a significant role to play in the principles of compiler design.

Eg:

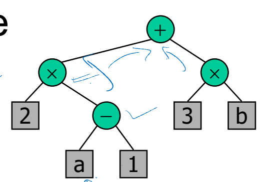
(a+b)*(c-d)^c

(┌ A ^B)V(B^E)

(T<W) V (A≤B)

2

Expression tree



- An Expression tree has
- Nonterminal node as operator
- Terminal node as operands.
- As shown in the fig.
- How precedence and associativity is reflected in expression tree?

postfix

Binary tree associated with an arithmetic expression

internal nodes: operators

external nodes: operands

Example: arithmetic expression tree for the expression (2 × (a − 1) + (3 × b))

3

Decision Tree

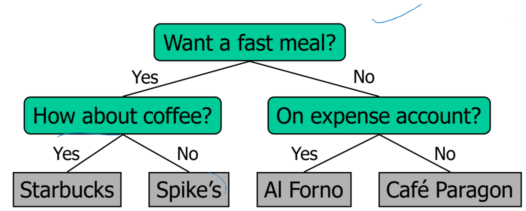
ML

Binary tree associated with a decision process

internal nodes: questions with yes/no answer

external nodes: decisions

Example: dining decision



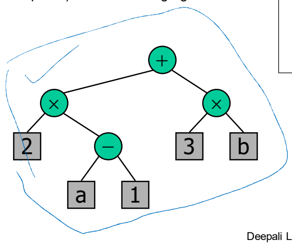
4

Print Arithmetic Expressions

Specialization of an inorder traversal

- print operand or operator when visiting node
- print "(" before traversing left subtree
- print ")" after traversing right subtree

```
Algorithm inOrder (v)
    if isInternal (v){
        print("(")
        inOrder (leftChild (v))}
    print(v.element ())
    if isInternal (v){
        inOrder (rightChild (v))
        print (")")}
```



$((2 \times (a - 1)) + (3 \times b))$

Building a Binary Expression Tree

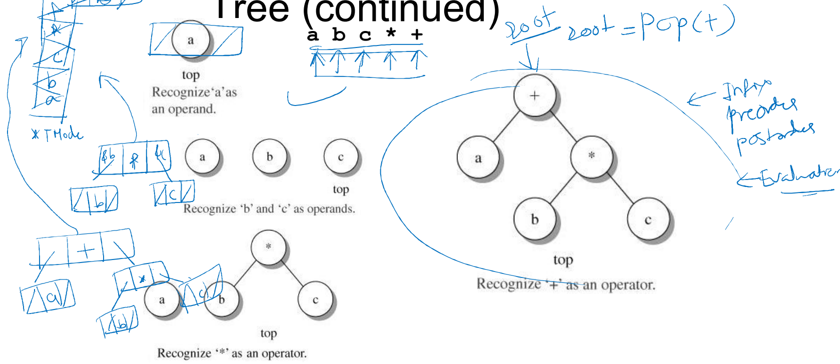
Build an expression tree from a postfix expression using an iterative algorithm. An operand is a single character such as 'a' or 'b'.

If the token is an operand, create a leaf node whose value is the operand and whose left and right subtrees are null. Push the node onto a stack of TNode references.

Building a Binary Expression Tree (continued)

If the token is an operator, create a new node with the operator as its value. Pop the two child nodes from the stack and attach them to the new node. The first child popped from the stack becomes the right subtree of the new node and the second child popped from the stack becomes the left subtree.

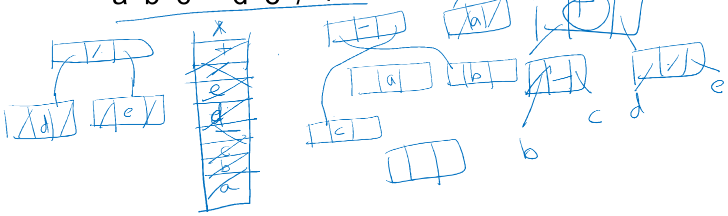
Building a Binary Expression Tree (continued)

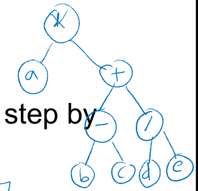


Student Question

Show the stack and the partial expression trees, step by step, for the expression

a b c - d e / + *

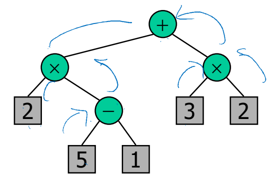




9

Evaluate Arithmetic Expressions

- recursive method returning the value of a subtree
- when visiting an internal node, combine the values of the subtrees



Algorithm *evalExpr*(*v*)
  if *isExternal* (*v*)
    return *v.element* ()
  else
    x ← *evalExpr*(*leftChild* (*v*))
    y ← *evalExpr*(*rightChild* (*v*))
    ◊ ← operator stored at *v*
    return *x* ◊ *y*

10

Traversal of ET

- Inorder yield an infix expression (Without parenthesis)
- Postorder traversal yields in postfix expression.
- Preorder traversal yields in prefix expression.

11

Creation of Expression tree

```
NODEPTR Create Exp_Tree ()
// Take infix expression →I
// Convert it into postfix form →P
// Initialize  the stack.
while (P! =NULL){
    if(P=operand)
     {   temp=getnode(P)
         push (S,Temp)
    }

else{
       if(P= operator)
        {
        temp=getnode(P)
        temp->lchild= pop(S)
        temp->Rchild=pop(s)
        push(S,temp)
        }
   } //end of else
}//end of while
Pop(s)  // which is the pointer to root of the tree
```

12

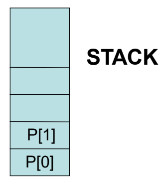
C Implementation basics of ET:

- Node Structure

```
struct ET
{
      char  op
      struct  ET * Lchild;
      struct ET  *Rchild;
}ET;
```

Lchild	OP	Rchild

```
typedef  struct Stack
   {
      ET  * p [10];
       int top;
} Stack
```




```
ET* create_ET(ET *T)
{
  char postfix[20];
  int i=0,len;
  ET *T1,*T2;
  ST S1;

  cout<<" \t Enter Postfix expression";

  gets(postfix);

  len=strlen(postfix);
  S1.top=-1;
  while(i<len)
  {
   if(isalpha(postfix[i]))
   {
    T1=  getnode(postfix[i]);
      S1=push(S1,T1);
      i++;
  }
else
    {
       if(isoperator(postfix[i]))
       {
          T1= getnode(postfix[i]);
          T1->Lchild=pop(S1);
          T1->Rchild=pop(S1);
          S1=push(S1,T1);
          i++;
       }
    }

}
 T =pop(S1);
 return(T);
}
```



```
// Check if character is OPERATOR
int isoperator(char o_p)
{
   if(o_p=='*' || o_p=='+' ||o_p=='-' || o_p=='/'
     ||o_p=='%' || o_p=='^')
     return(1);
   return(0);
}
```

```
ET  * Makenode( char o_p)
 {
    ET *P;

     p=new ET;
     p->op=o_p;
     p->Lchild=NULL;
     p-> Rchild= NULL;
     return(p);
}
```

```
// INORDER TRAV
void inorder(ET *T)
{
   if(T!=NULL)
   {
    inorder(T->Lchild);
    cut<<T->op;
    inorder(T->Rchild);
   }
}
```


STACK PUSH AND POP FUN

```
ST push( ST s,  ET *e  )
{

   if(s.top>=10)
     {
      cout<<"\n\t Stack is full";
      return(s);
     }
   s.top++;
   s.p[s.top]=e;
   return(s);
}
```

```
ET * pop(ST *s)
{
   ET *T;

   if(s->top==-1)
     {
      cout<<"\n\t Stack is empty";
      return(T);
     }
   T=s->p[s->top];
   s->top--;
   return(T);
}
```

Application of BT - BST

17

Application of BT (2)

1 .Finding all duplication numbers in a number series.
2. Find a given number is present in tree or not ?
3. Find the location to insert a given element.
4. Delete a particular data element from a tree.(needs searching and deletion)

$O(n^2)$

$O(\log n)$

Eg. : 14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5

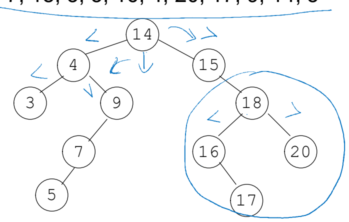
- Build a tree : ?

18

An application of binary trees

- Finding all duplication numbers in a number series.
- 14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5

BST adding duplicate No's



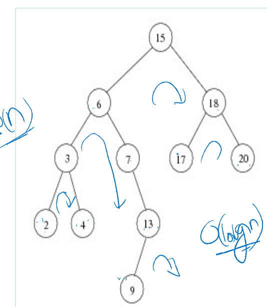
Build a binary tree: In such a way that
smaller numbers stored in the left subtree.
larger numbers stored in the right subtree.
Duplicate numbers: no duplicates are allowed.

5 -19

19

Binary Search Tree

$O(n)$



$O(\log n)$

Binary search tree
- Values in left subtree less than parent
- Values in right subtree greater than parent
- Facilitates duplicate elimination
- Fast searches - for a balanced tree, maximum of log n comparisons
- -- Inorder traversal – prints the node values in ascending order

Tree traversals: specifically
Inorder traversal – prints the node values in ascending order

Traversals of Tree
- In Order Traversal : 2 3 4 6 7 9 13 15 17 18 20
- Pre Order Traversal : 15 6 3 2 4 7 13 9 18 17 20
- Post Order Traversal : 2 4 3 9 13 7 6 17 20 18 15

20

Binary Search Tree

- Binary search tree has a better performance than any of the data structures when the functions to be performed are search, insertion, and deletion.
- 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 element is a key and no two elements are same key value (i.e., the keys 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.

21

```
algorithm insertBST (ref root <pointer>,
                     val new <pointer>)
Insert node containing new node into BST using iteration.
   Pre  root is address of first node in a BST
        new is address of node containing data to be inserted
   Post new node inserted into the tree
 1 if (root is null)
    1 root = new
 2 else
    1 pWalk = root
    2 loop (pWalk not null)
       1 parent = pWalk
       2 if (new->key < pWalk->key)
          1 pWalk = pWalk->left
       3 else
          1 pWalk = pWalk->right
       4 end if
    3 end loop
      Location for new node found
    4 if (new->key < parent->key)
       1 parent->left = new
    5 else
       1 parent->right = new
    6 end if
 3 end if
 4 return
end insertBST
```

Algorithm 8-4 Iterative binary search tree insert

(creation

new = 15

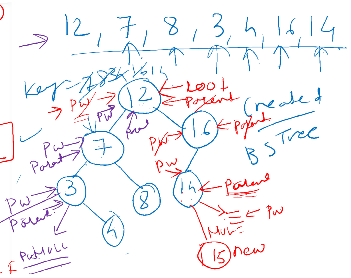
new = 1

15 < 14 F

15 < 16 = T

15 > 12 = F

15 < 14 = F



22

Binary Search Tree Creation

```
NODEPTR binsert(root, x)  {
//Insert first node
      if  (root=NULL)
          {
               root←makenode(x)
               return (root)
          }
// else find position for node
    P=Q=root
    while( Q!=NULL  &&  X!=P->data)
     {         P←Q
              if(X < P→ data)
                  Q = P→ Lchild
              else if (X>P→ data)
                  Q=P→ Rchild
     }
```

```
// Duplicate node??
  f( X = P->data)
  { write ('DUPLICATION NOT ALOOWED')
    return   }
//Insert node
     if (X < p->data)
          P->Lchild = makenode(X)
     else
          P->Rchild = makenode(X)
//finished
  return(root)
}
```

23

```
algorithm addBST  (ref root <pointer>,
                   val new <pointer>)
Insert node containing new data into BST using recursion.
   Pre  root is address of current node in a BST
        new is address of node containing data to be inserted
   Post new node inserted into the tree
 1 if (root is null)
    1 root = new
 2 else
   Locate null subtree for insertion
    1 if (new->key < root->key)
       1 addBST (root->left, new)
                                                (Contd.)
    2 else
       1 addBST (root->right, new)
    3 end if
 3 end if
 4 return
end addBST
```

Algorithm 8-5 Add node to BST recursively

Recursive Insert/creation

New = 13

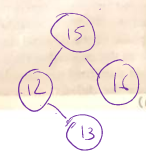
13<15

addBST(15,13)

addBST(12,13)

13 < 12 = F

addBST(NULL, 13)



24

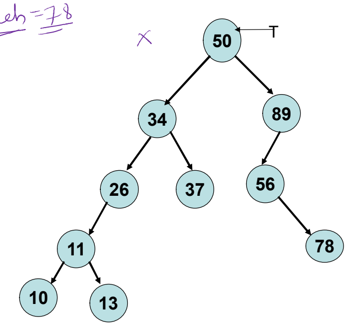
Recursive BST Creation

```
NODEPTR insert(NODEPTR p, int x)
{   if(p == NULL)
    {
        p = getnode();
        return p;
    }
    else{
        if(x < p->data)
          {
            if (p->lchild !=NULL)
                p->lchild = insert(p->Lchild, x);
            else
                p->lchild=getnode();
          }
      Else  //x>p->data
        {if(p->rchild !=NULL)
              p->Rchild = insert(p->Rchild, x);
         else p->rchild=getnode(); }
      }
      return p; }
```


Example

- Given : 50,34,89,56,26,78,37,11,13,10.

search = 78



Now

-- insert 45

-- insert 98

-- insert 36

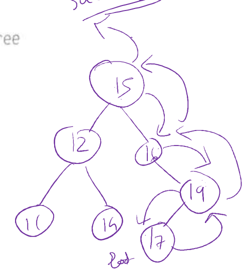
-- insert 30


```
algorithm searchBST (val root      <pointer>,
                     val argument  <key>)
Search a binary search tree for a given value.
     Pre    root is the root to a binary tree or subtree
            argument is the key value requested
     Return the node address if the value is found
            null if the node is not in the tree
  1 if (root is null)
     1 return null
  2 end if
  3 if (argument < root->key)
     1 return searchBST (root->left, argument)
  4 elseif (argument > root->key)
     1 return searchBST (root->right, argument)
  5 else
     1 return root
  6 end if
end searchBST
```

Algorithm 8-3 BST search

x = 17

Successful



Searching a node BST (recursive)

```
BSTNodeptr RBST_Search( BST root, int key)
 {
   if(root=  NULL)
         write ('Empty Tree' )
  p=root
  else
       if(key <  p->data)
            p= RBST_Search(p->lchild, key)
      else
           if(key>p->data)
           p= RBST_Search(p->Rchild, key)
        // end of else
Return (p)
```

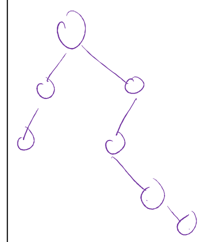
Searching a node BST (non recursive)

```
BSTNodeptr NRBST_Search( BST  root,  int key)
  {     if(root=   NULL)
           write (‘Empty Tree’ )
     p=root
     else
      while(p)
       {
            if(key= p->data)
                 Return (p)
             else if(key <  p->data)
                  p=p->Lchild
            else
                  p= p->Rchild, key    }
write (‘element not present in the tree’)
return(root)}
```

29

Height of BT (Recursive)

```
int  height ( root)
{
  if (root=null)
    return -1
  else {
    return 1 + max(height(root->left),
                   height(root->right))
  }
}
```



30

```
int  BST_NR_Height( root)
{
 // Base Case
    if (root = NULL)
        return 0;

 // Create an empty queue for level order tarversal

 // Enqueue Root and initialize height
    q.insert(root)
     height = 0
    while(1)
    {
      // nodeCount (queue size) indicates
         number of nodes
      // at current level.
        nodeCount = q.size1()
        if (nodeCount == 0) //break from the while(1)
            return height
       height++
```

```
// Dequeue all nodes of current level
and Enqueue all
      // nodes of next level
      while (nodeCount > 0)
      {

          temp=q.delete1()
         if (temp->llink != NULL)
             q.insert(temp->llink)
         if (temp->rlink != NULL)
             q.insert(temp->rlink)
         NodeCount - -
      }
   }
   return height
}
```

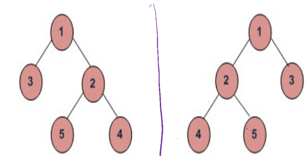
31

Practical

Mirror image of Binary tree

```
Void ptr Mirror_BST(BST root)
  {
     if(root)
     {
       temp =root->Rchild
       root->Rchild= root->Lchild
       root->Lchild=temp
       Mirror_BST(root->Lchild)
       Mirror_BST(root->Rchild)


  }
```



Mirror Trees

Inorder 3 1 5 2 4

Inorder 4 2 5 1 3

32

Iterative mirror Image

```
MirrorIterative()
{   Queue Q
   if(root = NULL)
      Retrurn root
else {  Q. insertroot);
  while(!Q.isEmpty()) {
     T  = Q.delete();
     if(T->llink = null && T->rlink= null)
          continue
     if(T->llink != null && T->rlink != null)
      {    temp = T->llink
           T->llink = T->rlink
           T->rlink=temp
           Q.insert(T->llink);
          Q.insert(T->rlink);        }
```

```
else if(T->llink == NULL)
{
        T->link =T->rlink
        T->rlink= NULL;
         Q.insert(T.llink);
      }
   else {
        T->rlink= T->llink;
        T->llink=NULL;
         Q.insert(T->rlink);
      }
   }
}
```

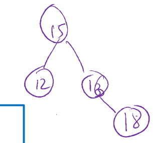

Other operations are

- Successor of Node.
- Predecessor Node.
- Inorder, preorder, postorder successor and predecessor.

```
BSTnode
BSTIN_successor(BSTnode root,
K)
{ // k is key
 successor = NULL
  current = root
  if(root==NULL)
         return NULL
while(current->data != K)
{
    if(current->data >K)
    {          successor = current;
          current= current->llink;
     }
```

```
else
      current = current->rlink
}
if(current && current->rlink)
{
   successor = BSTmin(current->rlink)
}

return successor;
}
```



12, 15, 16, 18

Overview of Binary Search Tree

Binary search tree definition:

T is a binary search tree if either of these is true

– T is empty; *or*

– Root has two subtrees:

- Each is a binary search tree
- Value in root > all values of the left subtree
- Value in root < all values in the right subtree

Deleting a node in BST

- As is common with many data structures, the hardest operation is deletion.
- Once we have found the node to be deleted, we need to consider several possibilities.
- Based on whether node to be deleted is :
  - leaf node
  - nonleaf node.

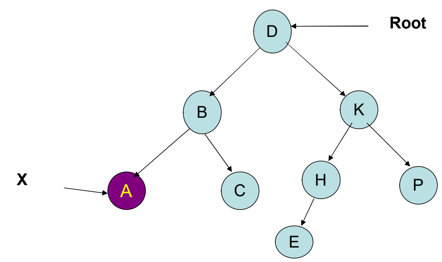
Deletion operation

- There are the following possible cases when we delete a node:
- The node to be deleted has no children. In this case, all we need to do is delete the node.
- The node to be deleted has only a right subtree. We delete the node and attach the right subtree to the deleted node's parent.
- The node to be deleted has only a left subtree. We delete the node and attach the left subtree to the deleted node's parent.
- The node to be deleted has two subtrees. It is possible to delete a node from the middle of a tree, but the result tends to create very unbalanced trees.

37

Case 1: Leaf node deletion

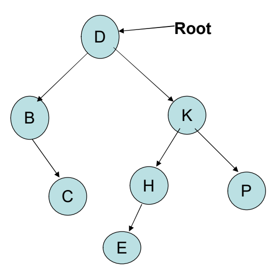
- If the node is a *leaf*, it can be deleted immediately. Eg :Delete (A)



38

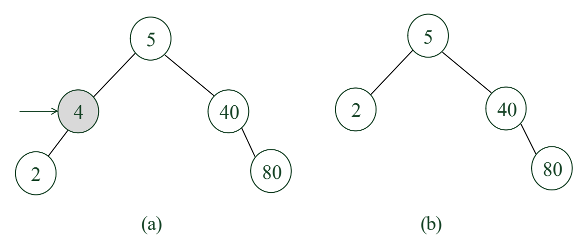
Tree after deletion

- Deleted the leaf node (A):



39

Case 2:Deletion from a BST

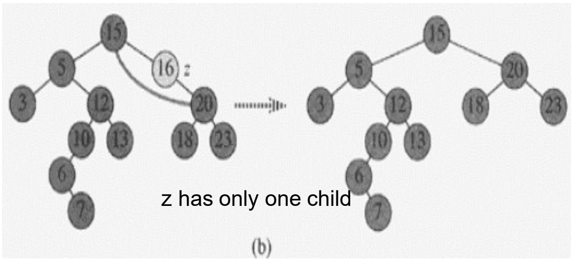


•The node has one child
•The node has two children

40

Case 2 : Node with one child

- If the node has one child, the node can be deleted after its parent adjusts a pointer to bypass the node and connect to **inorder successor**.



41

Deletion from the middle of a tree

- Rather than simply delete the node, we try to maintain the existing structure as much as possible by finding data to take the place of the deleted data. This can be done in one of two ways.

42

Deletion from the middle of a tree

- We can find the largest node in the deleted node's left subtree and move its data to replace the deleted node's data.
- We can find the smallest node on the deleted node's right subtree and move its data to replace the deleted node's data.
- Either of these moves preserves the integrity of the binary search tree.

43

- **1) *Node to be deleted is leaf:*** Simply remove from the tree.
- **2) *Node to be deleted has only one child:*** Copy the child to the node and delete the child
- **3) *Node to be deleted has two children:*** Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor.
- Note that inorder predecessor can also be used.
- The important thing to note is, inorder successor is needed only when right child is not empty.
- In this particular case, inorder successor can be obtained by finding the minimum value in right child of the node.

44

```
Algorithm deleteBST (root, dltKey)
This algorithm deletes a node from a BST.
  Pre    root is reference to node to be deleted
         dltKey is key of node to be deleted
  Post   node deleted
         if dltKey not found, root unchanged
  Return true if node deleted, false if not found
1 if (empty tree)
  1  return false
2 end if
3 if (dltKey < root)
  1  return deleteBST (left subtree, dltKey)
4 else if (dltKey > root)
  1  return deleteBST (right subtree, dltKey)
5 else
     Delete node found--test for leaf node
  1  If (no left subtree)
     1  make right subtree the root
     2  return true
  2  else if (no right subtree)
     1  make left subtree the root
     2  return true
  3  else
        Node to be deleted not a leaf. Find largest node on
        left subtree.
     1  save root in deleteNode
     2  set largest to largestBST (left subtree)
     3  move data in largest to deleteNode
     4  return deleteBST (left subtree of deleteNode,
                          key of largest
  4  end if
6 end if
end deleteBST
```


Find MIN /Smallest

```
BST  Find_Min(BST  X)
{
   if( X ->left == NULL){
        return X
  }
 else{
       return find_Min(X->left)
    }
}
Time  Complexity is O(h)
```

```
BST- NonRec_minimum(x)
{
  if x =nil
     thenreturn(\EmptyTree")
   y= x
   whileleft[y] !=null
        Y=left[y]
  return(key[y])
```


Find Max /Maximum/largest

```
BST node find_Max(  BST x)
{
while x->Rchild ≠ NULL
 {
 x ← x->Rchild
}
return x
}
```

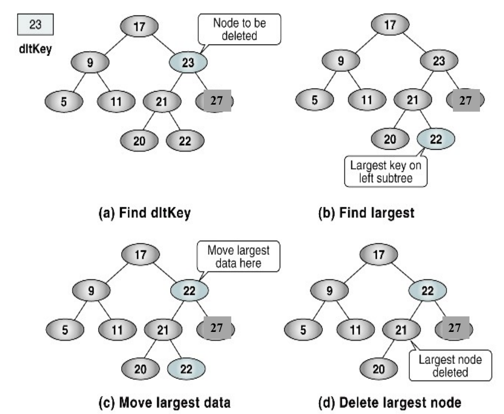



FIGURE 7-10 Delete BST Test Cases

Deletion

```
SearchTree Delete( ElementType X, SearchTree T )
{
   //  Position TmpCell
   if( T == NULL )
       Error( "Element not found" )
    else
    {   if( X < T->Element )  /* Go left */
               T->Left = Delete( X, T->Left )
     else
         if( X > T->Element )  /* Go right */
               T->Right = Delete( X, T->Right )
    }
 else  /* One or zero children */
         {     TmpCell = T
                if( T->Left == NULL )
             /* Also handles 0 children */
                       T = T->Right
             else if( T->Right == NULL )
                   T = T->Left
             free( TmpCell )
          }
         return T}
```

```
else  /* Found element to be deleted */
          if( T->Left && T->Right )  /* Two children */
          {
             /* Replace with smallest in right subtree  */
              TmpCell = FindMin( T->Right )
                 T->Element = TmpCell->Element
                 T->Right = Delete( T->Element, T->Right )
          }
```

49

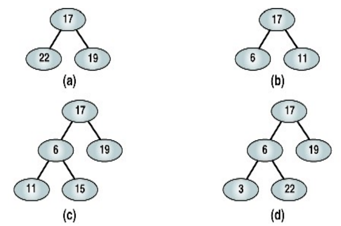


FIGURE 7-3 Invalid Binary Search Trees

50

Three BST search algorithms:

- Find the smallest node
- Find the largest node
- Find a requested node

51

ALGORITHM 7-1 Find Smallest Node in a BST

```
Algorithm findSmallestBST (root)
This algorithm finds the smallest node in a BST.
   Pre    root is a pointer to a nonempty BST or subtree
   Return address of smallest node
1 if (left subtree empty)
   1  return (root)
2 end if
3 return findSmallestBST (left subtree)
end findSmallestBST
```

52

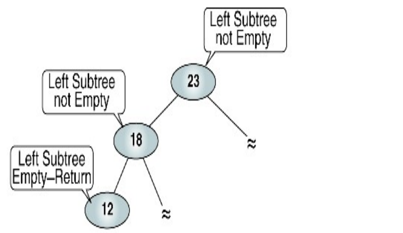


FIGURE 7-5 Find Smallest Node in a BST

53

```
void DeleteItem (treenode *&tree, int item)
  { if (tree == NULL) // empty tree or not in the tree
      return;
if (item < tree->info)
 // Go Left
  DeleteItem (tree->left, item);
  else if (item > tree->info) // Go Right
 DeleteItem (tree->right, item);
  else // This is Item
 DeleteNode (tree); }
```

54

```
void DeleteNode (treenode *&tree)
 { node *temp;
  temp = tree;
  if (tree->left == NULL)
      // no left child is easy
        { tree = tree->right; delete temp; }
     else if (tree->right == NULL) // no right is also easy
           { tree = tree->left; delete temp; }
    else // both left & right exist
   {     temp = tree->left; // find right-most node of left sub-tree
           while (temp->right != NULL)
             temp = temp->right;
              tree->info = temp->info; // move just that value to root
          DeleteItem (tree->Left, temp->info);
  // delete duplicate data }
 }
```

55

Level Order traversal

```
template <class T>
void levelOrder(binaryTreeNode<T> *t)
{// Level-order traversal of *t.
   arrayQueue<binaryTreeNode<T>*> q;
   while (t != NULL)
   {
      visit(t);  // visit t

      // put t's children on queue
      if (t->leftChild != NULL)
         q.push(t->leftChild);
      if (t->rightChild != NULL)
         q.push(t->rightChild);

      // get next node to visit
      try {t = q.front();}
      catch (queueEmpty) {return;}
      q.pop();
   }
}
```

56