
—→ **Trees: Definitions** ←—
Data Structures Fall 1998

- collection of nodes and edges
- one *root* node
- zero or more subtrees
- roots of subtrees connected to *root* node
- parent–child relationship
- degree of a node = # of children
- internal nodes, leaf nodes

→ **Trees: More Definitions** ←
Data Structures Fall 1998

- n nodes $\Rightarrow n - 1$ edges
- *path length*: # of edges in the path
- *node depth*: path length from root
- *node height*: max path length to a leaf
- *tree height*: height of root

Implementation: linked list of pointers for children

—→ **Hierarchical File System** ←—
Data Structures Fall 1998

- **tree structure**
 - **pathname: directories traversed from top**
 - **file: uniquely identified by pathname**
- ⇒ **file name need not be unique**

—→ **Tree Traversals** ←—
Data Structures Fall 1998

Preorder traversal: node first, children later

- running time = $O(n)$

Inorder traversal: consider a binary tree

- traverse left subtree
- process current node
- traverse right subtree

Postorder traversal: subtrees first, node later

—→ **Binary Trees** ←—
Data Structures Fall 1998

- at most 2 children/node
- linked list implementation: three fields
 1. value
 2. left child pointer
 3. right child pointer

Expression Trees

- operands = leaf nodes
- operators = internal nodes
- degree of internal node = # of operands required by operator

→ **Expression Tree Construction** ←
Data Structures Fall 1998

Input: postfix expression

Rules

1. read one symbol at a time

2. operand:

- create one-node tree
- push pointer onto stack

3. operator:

- pop pointers to T_1 and T_2
- form a new tree
- root = operator
- left child = T_2 , right child = T_1
- push new tree's pointer onto stack

—→ **Binary Search Trees** ←—
Data Structures Fall 1998

- each node has a key
- assume distinct keys
- keys in left subtree < current node
- current node < keys in right subtree
- average depth = $O(\log n)$
- **NO TEMPLATES! INTEGERS!**

Make_Empty

```
/* 2*/ void
/* 3*/ Binary_Search_Tree::
/* 4*/ Make_Empty( Tree_Node * & T)
/* 5*/ {
/* 6*/     if( T != NULL)
/* 7*/     {
/* 8*/         Make_Empty( T->Left);
/* 9*/         Make_Empty( T->Right);
/*10*/        delete T;
/*11*/        T = NULL;
/*12*/    }
/*13*/ }
```

Find

```
/* 2*/Tree_Node*
/* 3*/Binary_Search_Tree::
/* 4*/Find(const int &X, Tree_Node *T)
/* 5*/{
/* 6*/     if( T == NULL)
/* 7*/         return NULL;
/* 8*/     else
/* 9*/         if( X < T->Element)
/*10*/             return Find( X, T->Left);
/*11*/     else
/*12*/         if( X > T->Element)
/*13*/             return Find( X, T->Right);
/*14*/     else
/*15*/         return T;
/*16*/}
```

Find_Min (Find_Max similar)

```
/* 2*/  Tree_Node*
/* 3*/  Binary_Search_Tree::
/* 4*/  Find_Min( Tree_Node *T) const
/* 5*/  {
/* 6*/      if( T == NULL)
/* 7*/          return NULL;
/* 8*/      else
/* 9*/          if( T->Left == NULL)
/*10*/             return T;
/*11*/          else
/*12*/             return Find_Min( T->Left);
/*13*/  }
```

Insert

```
/* 2*/void Binary_Search_Tree::
/* 4*/Insert(const int &X, Tree_Node*&T) // BST has ints
/* 5*/{
/* 6*/    if( T == NULL)
/* 7*/    {
/* 8*/        T = new Tree_Node(X);
/* 9*/        if( T == NULL )
/*10*/            Error( "Out of space");
/*11*/    }
/*12*/    else
/*13*/    if( X < T->Element)
/*14*/        Insert( X, T->Left);
/*15*/    else
/*16*/    if( X > T->Element)
/*17*/        Insert( X, T->Right);
/*18*/    // Else X is in the tree already.
/*19*/}
```

—→ **Remove** ←—
Data Structures Fall 1998

- **leaf node: delete immediately**
- **node with one child:**
 - adjust parent's pointer to bypass node
 - delete node
- **node with two children:**
 - replace with smallest node in right subtree
 - delete node

—→ **Balanced Trees** ←—
Data Structures Fall 1998

- additions, deletions \Rightarrow create imbalance
- if insertion input presorted: insertion takes quadratic time!
- need for imposing *balance*

Two approaches

1. insist on balance: no node allowed to go too deep.
2. forgo balance: apply restructuring rule to make future operations efficient.

Restructuring example: a sequence of m operations take $O(m \log n)$ time

→ **Balancing Trees** ←
Data Structures Fall 1998

Simplest idea: left and right subtrees have same height

- doesn't work all the time

Alternative: for every node, left subtree height = right subtree height

- too rigid
- requires perfect balance

Need a compromise between the two approaches