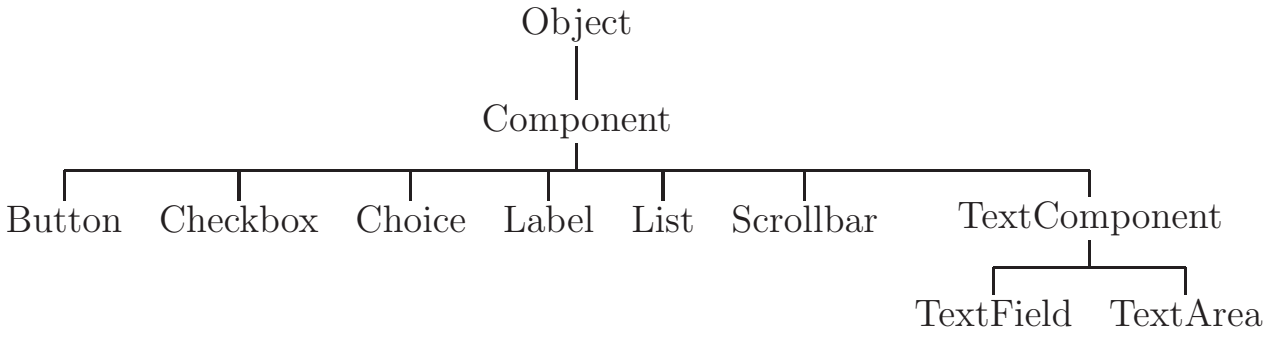
1 Remaining Topics

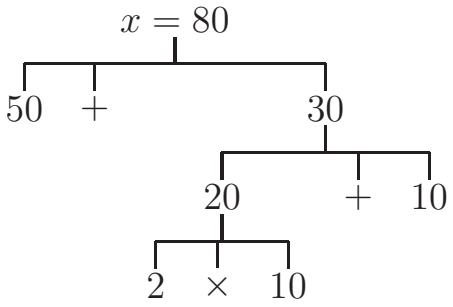
1. Trees
    - Terminology
    - Traversals
    - Binary Search Trees
    - Other Trees
2. Hash Tables
3. Sorting and Searching
4. Graphs

2 Trees

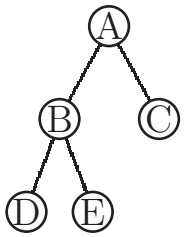


Components in Java

$$x = 50 + ((2 \times 10) + 10)$$



Evaluation of a Mathematical Expression



Simple Binary Tree

2.1 General Tree Terminology

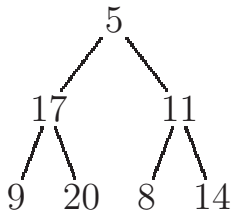
- **root** node at the top of a tree.
- **leaf** a node with no children.
- **parent** the parent of a node is the node linked above it.
- **sibling** two nodes are siblings if they have the same parent.
- **ancestor** a node's parent is its first ancestor.
- **subtree** any node in a tree that can be viewed as the root of a new, smaller tree.
- **left and right subtrees** the nodes beginning with a left or right child.
- **height** the maximum depth of any leaf.

2.2 Binary Trees

A **binary tree** is a finite set of nodes. The set might be empty (no nodes, the *empty tree*). If the set is not empty, it follows these rules:

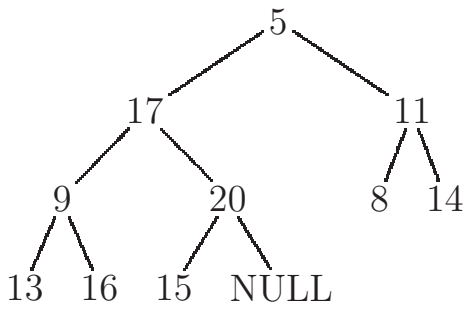
1. There is one special node—the *root*.
2. Each node may be associated with up to two other different nodes, called its *left child* and its *right child*. If a node c is the child of another node p, then p is c's parent.
3. If you start at a node and move to the node's parent (if there is one), then move again to that node's parent, and keep moving upward to each node's parent, you will eventually reach the root.

- **Full binary Tree** a binary tree of height h with no missing nodes. All leaves have the same depth and every non-leaf has two children.
- **Complete binary Tree** a binary tree of height h that is full to level $h-1$ and has level h filled in from left to right.
- **Balanced Binary Tree** a binary tree in which the left and right subtrees of any node have heights that differ by at most 1.



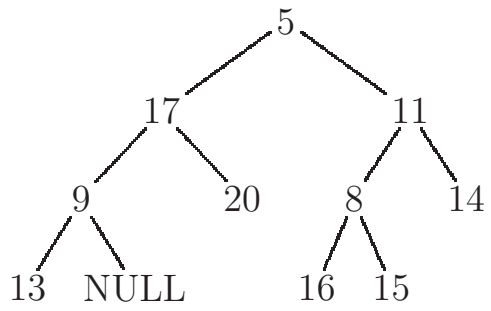
Full binary tree

Full binary Tree a binary tree of height h with no missing nodes. All leaves have the same depth and every non-leaf has two children.

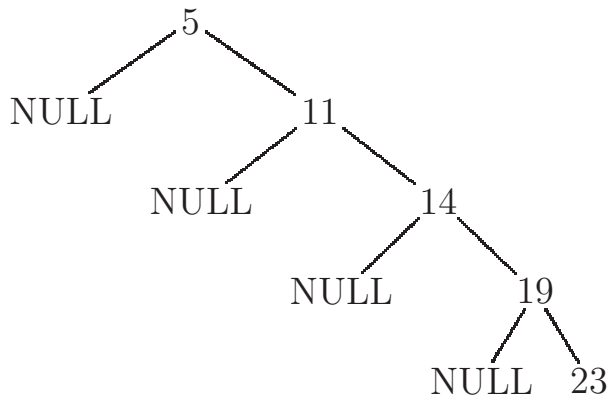


Complete but not Full binary tree

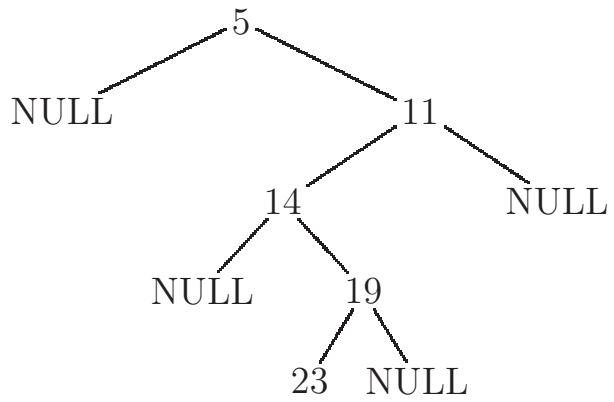
Complete binary Tree a binary tree of height h that is full to level $h-1$ and has level h filled in from left to right.



Not Complete or Full binary tree



Unbalanced binary tree



Unbalanced binary tree (*zipper* tree)

2.3 Binary Tree Definition

A binary tree is either

1. An empty tree
2. Or it has a root and the remaining nodes are divided into two disjoint sets named the **left subtree** and the **right subtree**.

Since the definition of binary trees is recursive, most of the algorithms used to manipulate binary trees are defined recursively.

2.4 Typical Binary Tree Representation

A node in a binary tree can be defined using the following:

```
1 struct BinaryTreeNode
2 {
3     int  data;
4     BinaryTreeNode *left;
5     BinaryTreeNode *right;
6 };
```

As we saw with nodes in linked lists, it is convenient to declare a new data type for pointers to the nodes.

```
1 typedef struct BinaryTreeNode *BinaryTreeNodePtr;
```

2.5 Tree Operations

- Print a tree (traversal)
- Create a node
- Delete a node
- Test if a node is a leaf
- Search a tree
- Copy a tree

2.5.1 Creating a binary tree node

```
1  BinaryTreeNode *CreateNode( int newVal )
2  {
3      BinaryTreeNode *newNode = new BinaryTreeNode;
4
5      newNode->data  = newVal;
6      newNode->left  = NULL;
7      newNode->right = NULL;
8
9      return  newNode;
10 }
```

```
1  BinaryTreeNode *CreateNode(
2          int newVal,
3          BinaryTreeNode *leftPtr  = NULL,
4          BinaryTreeNode *rightPtr = NULL )
5  {
6      BinaryTreeNode *newNode = new BinaryTreeNode;
7
8      newNode->data  = newVal;
9      newNode->left  = leftPtr;
10     newNode->right = rightPtr;
11
12     return  newNode;
13 }
```

2.5.2 Testing if a node is a leaf

```
1 bool IsLeaf( BinaryTreeNode *t )
2 {
3     return ((t->left == NULL) && (t->right == NULL));
4 }
```

2.6 Tree Size

```
1  int TreeSize( BinaryTreeNode *t )
2  {
3      if( t == NULL )
4          return 0;
5      else
6          return 1 +
7                  TreeSize( t->left  ) +
8                  TreeSize( t->right );
9  }
```