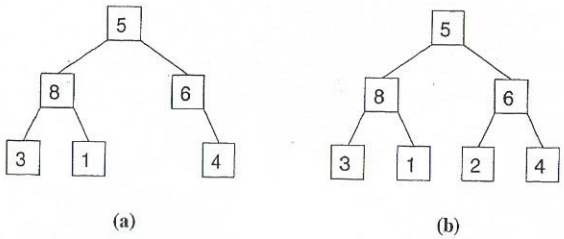

Figure 2.2 Examples of binary trees.

Defining trees

Assuming that values are stored in the nodes and that they are all of the same type, a general tree can be declared as:

```
data Tree a = Node a [Tree a]  deriving Show
```

Using this representation, the tree illustrated in Figure 2.2(a) will be expressed as follows provided we do not make any distinction between the successors:

```
Node 5 [Node 8 [Node 3 [], Node 1 []],
        Node 6 [Node 4 []]]
```

A node in a *binary tree* has at most two successors so it can be declared as:

```
data BinTree a  = Empty  | NodeBT a (BinTree a) (BinTree a)
    deriving Show
```

Using this representation, the tree illustrated in Figure 2.2(a) is expressed as:

```
NodeBT 5 (NodeBT 8 (NodeBT 3 Empty Empty) (NodeBT 1 Empty Empty))
         (NodeBT 6 Empty (NodeBT 4 Empty Empty))
```

and the tree in Figure 2.2(b) is represented as:

```
NodeBT 5 (NodeBT 8 (NodeBT 3 Empty Empty) (NodeBT 1 Empty Empty))
         (NodeBT 6 (NodeBT 2 Empty Empty) (NodeBT 4 Empty Empty))
```

If every node of a binary tree is limited to either zero or exactly two successors, some space can be saved by using the following declaration:

```
data BinTree' a = Leaf a | NodeBT' a (BinTree' a) (BinTree' a)
    deriving Show
```

In this case, the tree in Figure 2.2(a) can no longer be represented but the tree in Figure 2.2(b) can be written in a shorter form:

```
NodeBT' 5 (NodeBT' 8 (Leaf 3) (Leaf 1))
          (NodeBT' 6 (Leaf 2) (Leaf 4))
```


Tree operations

Tree operations can be grouped into *single* operations such as insertions or deletions or *global* operations that manipulate the tree as a whole. The most common single operations are adding and removing an item from the tree. An item can be inserted into the tree at different positions, depending on the purpose of the tree and the algorithm. Similarly, deleting an item contained within an interior node can be accomplished in several ways depending on the algorithm used. Examples will be described in Section 5.7.

We now consider global operations that involve traversing the tree as a whole. For example, the depth of a general tree can be computed as follows:

```
depth                :: Tree a -> Int
depth (Node _ [])    = 1
depth (Node _ succs) = 1 + maximum (map depth succs)
```

because the depth of a node is one more than the maximum depth of its successors. The depth of a binary tree can be determined in the following manner:

```
depth'                   :: BinTree a -> Int
depth' Empty             = 0
depth' (NodeBT _ lf rt) = 1 + max (depth' lf) (depth' rt)
```

Another example is counting the number of empty leaves in a tree:

```
countEmpty                  :: BinTree a -> Int
countEmpty Empty            = 1
countEmpty (NodeBT _ lf rt) = countEmpty lf + countEmpty rt
```

Supposing that nodes contain integers, summing the values in a tree can be achieved as follows depending on the representation we use for the binary trees:

```
tsum                   :: BinTree  Int -> Int
tsum Empty             = 0
tsum (NodeBT a lf rt)  = a + (tsum lf) + (tsum rt)
```

and

```
tsum'                  :: BinTree'  Int -> Int
tsum' (Leaf v)         = v
tsum' (NodeBT' v lf rt) = v + (tsum' lf) + (tsum' rt)
```

Consider now the problem of converting a tree into a list. This can be done in three ways, depending on when the node is visited:

- **Preorder** the node is visited before its left and right subtrees are visited.
- **Inorder** the node is visited after the left subtree has been visited and before the right subtree is visited.
- **Postorder** the node is visited after its left and right subtrees have been visited.

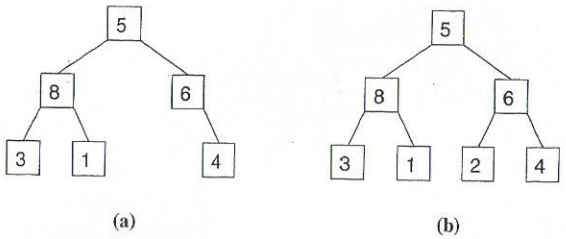

Figure 2.2 Examples of binary trees.

Defining trees

Assuming that values are stored in the nodes and that they are all of the same type, a general tree can be declared as:

```
data Tree a = Node a [Tree a]  deriving Show
```

Using this representation, the tree illustrated in Figure 2.2(a) will be expressed as follows provided we do not make any distinction between the successors:

```
Node 5 [Node 8 [Node 3 [], Node 1 []],
        Node 6 [Node 4 []]]
```

A node in a *binary tree* has at most two successors so it can be declared as:

```
data BinTree a  = Empty  | NodeBT a (BinTree a) (BinTree a)
    deriving Show
```

Empty :: BinTree a

Using this representation, the tree illustrated in Figure 2.2(a) is expressed as:

```
NodeBT 5 (NodeBT 8 (NodeBT 3 Empty Empty) (NodeBT 1 Empty Empty))
         (NodeBT 6 Empty (NodeBT 4 Empty Empty))
```

and the tree in Figure 2.2(b) is represented as:

```
NodeBT 5 (NodeBT 8 (NodeBT 3 Empty Empty) (NodeBT 1 Empty Empty))
         (NodeBT 6 (NodeBT 2 Empty Empty) (NodeBT 4 Empty Empty))
```

If every node of a binary tree is limited to either zero or exactly two successors, some space can be saved by using the following declaration:

```
data BinTree' a = Leaf a | NodeBT' a (BinTree' a) (BinTree' a)
    deriving Show
```

In this case, the tree in Figure 2.2(a) can no longer be represented but the tree in Figure 2.2(b) can be written in a shorter form:

```
NodeBT' 5 (NodeBT' 8 (Leaf 3) (Leaf 1))
          (NodeBT' 6 (Leaf 2) (Leaf 4))
```

NodeBT :: a -> BinTree a -> BinTree a -> BinTree a

Tree operations

Tree operations can be grouped into *single* operations such as insertions or deletions or *global* operations that manipulate the tree as a whole. The most common single operations are adding and removing an item from the tree. An item can be inserted into the tree at different positions, depending on the purpose of the tree and the algorithm. Similarly, deleting an item contained within an interior node can be accomplished in several ways depending on the algorithm used. Examples will be described in Section 5.7.

We now consider global operations that involve traversing the tree as a whole. For example, the depth of a general tree can be computed as follows:

```
depth                :: Tree a -> Int
depth (Node _ [])    = 1
depth (Node _ succs) = 1 + maximum (map depth succs)
```

because the depth of a node is one more than the maximum depth of its successors. The depth of a binary tree can be determined in the following manner:

```
depth'                    :: BinTree a -> Int
depth' Empty              = 0
depth' (NodeBT _ lf rt) = 1 + max (depth' lf) (depth' rt)
```

Another example is counting the number of empty leaves in a tree:

```
countEmpty                  :: BinTree a -> Int
countEmpty Empty            = 1
countEmpty (NodeBT _ lf rt) = countEmpty lf + countEmpty rt
```

Supposing that nodes contain integers, summing the values in a tree can be achieved as follows depending on the representation we use for the binary trees:

```
tsum                    :: BinTree  Int -> Int
tsum Empty              = 0
tsum (NodeBT a lf rt)   = a + (tsum lf) + (tsum rt)
```

and

```
tsum'                   :: BinTree'  Int -> Int
tsum' (Leaf v)          = v
tsum' (NodeBT' v lf rt) = v + (tsum' lf) + (tsum' rt)
```

Consider now the problem of converting a tree into a list. This can be done in three ways, depending on when the node is visited:

- **Preorder** the node is visited before its left and right subtrees are visited.
- **Inorder** the node is visited after the left subtree has been visited and before the right subtree is visited.
- **Postorder** the node is visited after its left and right subtrees have been visited.