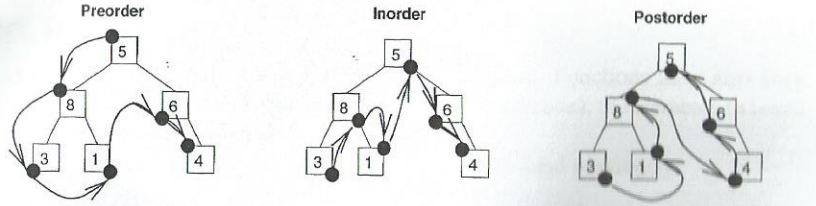

Figure 2.3 Tree traversal strategies.

These strategies are illustrated in Figure 2.3. The corresponding functions for converting a tree into a list are:

```
preorder                    :: BinTree a -> [a]
preorder Empty              = []
preorder (NodeBT a lf rt)   = [a] ++ preorder lf ++ preorder rt

inorder                     :: BinTree a -> [a]
inorder Empty               = []
inorder (NodeBT a lf rt)    = inorder lf ++ [a] ++ inorder rt

postorder                   :: BinTree a -> [a]
postorder Empty             = []
postorder (NodeBT a lf rt)  = postorder lf ++ postorder rt ++ [a]
```

2.7 Arrays

An array is used to store and retrieve a set of elements, each element having a unique *index*. In this section, we describe how arrays are created and manipulated in Haskell. In Haskell, arrays are not part of the Standard Prelude but are provided as the `Array` library module, so before using any array-related function described in this section, this library should be 'imported' using the following directive:

```
import Array
```

The implementation of arrays will be discussed later in Section 4.3.

2.7.1 Creation of arrays

Arrays are created by three predefined functions called `array`, `listArray` and `accumArray`. The first function:

`array` *bounds list_of_associations*

is the fundamental array creation mechanism. The first parameter, *bounds*, gives the lowest and highest indices in the array. For example, a zero-origin vector of five elements has bounds `(0,4)` and a one-origin 10 by 10 matrix has bounds `((1,1),(10,10))`. The values of the bounds can be arbitrary expressions. The second parameter is a list of associations where an *association*, of the form `(i,x)`, means that the value of the array element `i` is `x`. The list of associations is often defined using a list comprehension (see Section 2.4.2). Here are some examples of building arrays:

```
a'  = array (1,4) [(3 ,'c'),(2 ,'a'), (1 ,'f'), (4 ,'e')]
f n = array (0,n) [(i , i*i) | i <- [0..n]]
m   = array ((1,1),(2,3)) [((i,j) , (i*j)) | i<-[1..2],j<-[1..3]]
```

The type of an array is denoted as `Array a b` where `a` represents the type of the index and `b` represents the type of the value. Here are the (possible) type definitions of the previous expressions:

```
a' :: Array Int Char
f  :: Int -> Array Int Int
m  :: Array (Int,Int) Int
```

An array is undefined if any specified index is out of bounds; if two associations in the list have the same index, the value at that index is undefined. As a consequence, `array` is *strict* in the bounds and in the indices but *non-strict* (or *lazy*) in the values. This means that an array can contain 'undefined' elements (we will discuss lazy data structures in Section 3.1.3).

We can thus use recurrences such as:

```
fibs n = a
    where
    a = array (1,n) ([(1, 1), (2, 1)] ++
                     [(i, a!(i-1) + a!(i-2)) | i <- [3..n]])
```

as we will see later in this section, the operator `(!)` denotes indexing. The second function:

`listArray` *bounds list_of_values*

is predefined for the frequently occurring case where an array is constructed from a list of values in index order. The following defines `a''` to be equivalent to `a'` above:

```
a''= listArray (1,4) "face"
```

The last function:

`accumArray` *f init bounds list_of_associations*

removes the restriction that a given index may appear at most once in the association list but instead combines these 'conflicting indices' via an accumulating function *f*. The elements of the array are initialized with *init*. For example, given a list of values `vs`, `histogram` produces an array giving for each index value within bounds the number of occurrences of the corresponding value in `vs`.

```
histogram bounds vs = accumArray (+) 0 bounds [(i , 1) | i <- vs]
```