Chapter 11

Data Structures

Abstract A data structure is an organization of collections of data, such that operations on them are efficient. In earlier chapters, we have already looked at some examples, e.g.,

- strings which are variable length sequences of characters and which were discussed in Chapter 3,
- tuples which are fixed length sequences of values of variable types and which were discussed in Section 5.1,
- lists which are variable length sequences of values of identical type and which were discussed in Chapter 7, and
- stacks which are specialized lists, where values can only be added and removed from its head, and which was discussed in Section 9.4.

In this chapter, we will further consider the following data structures:

- *queues* which are specialized lists, where elements can be added to the end of the list and removed from its head,
- *trees* which are hierarchical orderings of data of a variable number of values of identical types,
- *sets* which are an unordered collection of unique values of the identical type, and
- *hash maps* which are mappings between sets of keys into sets of values.

These data structures have a long history and are often discussed from an abstract point of view in terms of their conceptual interface and from a computational complexity point of view, where details of their implementation are stressed. The above-mentioned data structures occur frequently alone or in combination with many programming solutions and form a solid basis for solving problems by programming. Some of these data structures have their predefined modules in F# but not all. In this chapter, we will give a brief introduction to each.

11.1 Queues

A queue is a sequence values that can be added to its end and removed from its front as illustrated in Figure 11.1. Queues appear often in real life: Standing in line at a

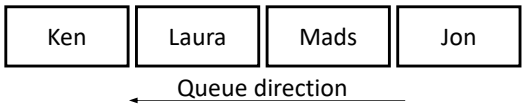

Fig. 11.1 A queue with Ken in the front and Jon in the back of the queue.

shop counter, orders await in a queue for their turn to be shipped in an online shop, and students waiting to be examined at an oral examination. Many operations on queues can be defined, but the following are always present in some form:

create: Create an empty queue.

enqueue: Add an element to the back of the queue.

dequeue: Remove the element at the front of the queue.

head: Get the value of the front element of the queue.

isEmpty: Check if the queue is empty.

As of the writing of this book, the standard collection of Fsharp libraries does not include a queues module, but they can easily be implemented using lists. For example, a queue of integers is implemented in Listing 11.1. This is a functional queue because the enqueue and dequeue operations return a new queue they are called, without destroying the old queue. Mutable queues are more common, where en- and dequeuing update the value of a queue as a side-effect. See Section 13.1 for more on mutable values. A simple application using this queue is shown in Listing 11.2 Note that this implementation, the computational complexity of all but `enqueue` is $O(1)$, while `enqueue` is $O(n)$, where n is the length of the list, since it relies on list concatenation. Faster implementations exist but are beyond the scope of this book.

11.2 Trees

A tree is a hierarchical organization of data. For example the expression

Listing 11.1 queue.fs:
Implementing a functional queue using lists.

```
module Queue

type element = int
type queue = element list

/// the empty queue
let create () : queue = []
/// add an element at the end of a queue
let enqueue (e: element) (q: queue) : queue =
  q @ [e]
/// remove the element at the front of the queue
let dequeue (q: queue) : (element option) * queue =
  match q with
    [] -> (None, [])
    | e::rst -> (Some e, rst)
/// the value at the front of the queue
let head (q: queue) : element option =
  q |> dequeue |> fst
/// check if the queue is empty
let isEmpty (q:queue): bool =
  q.IsEmpty
```

Listing 11.2 queueApp.fsx:
An application of the Queue module.

```
open Queue

let q = create () |> enqueue 3 |> enqueue 1
printfn "q = %A" q
printfn "Is q empty? %A" (isEmpty q)
let (v,newQ) = dequeue q
printfn "dequeue q = (%A, %A)" v newQ
```

```
$ dotnet fsi queue.fs queueApp.fsx
q = [3; 1]
Is q empty? false
dequeue q = (Some 3, [1])
```

$$\frac{1}{3+5} \tag{11.1}$$

can be represented as shown in Figure 11.2 Further examples of trees are the file structure on your hard disk, where a directory contains files and other directories, and the list of contents of this book, which has chapters consisting of sections which in turn consist of subsections.

Trees consists of *nodes* and *relations*. In Figure 11.2, 1, 3, 5, “Div”, and “Plus” are nodes and their relation are shown with arrows. Relations are often described as

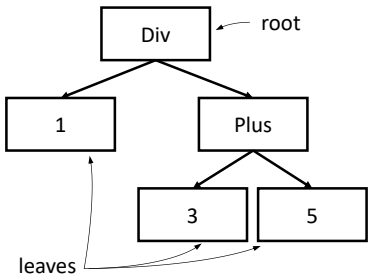

Fig. 11.2 A tree representation of $1/(3+5)$.

family relations, such that, e.g., 1 and "Plus" are *siblings* and are the *children* of the *parent* "Div". Nodes which do not have *descendants* are called *leaves*, and there must be on node, which does not have *ancestors* and that is called the *root*. Trees are often classified as being *k-ary*, if each node has at most k children. The example in Figure 11.2 is an example of a 2-ary or *binary tree*. Trees are often displayed with the children below their parent, in which case the arrowheads are neglected.

As of the writing of this book, the standard collection of Fsharp libraries does not include a tree module. Trees are somewhat complicated to program in the functional paradigm since they are non-linear structures. One way to represent them is with the use of discriminated union, as demonstrated in Listing 11.3. Such values are

Listing 11.3 bTree.fsx:
Representing a computational expression as a binary tree

```
type element = Value of float | Mul | Plus | Minus | Div
// A binary tree
type bTree = Leaf of element | Node of element * bTree * bTree
/// The tree representation of 1/(3+5)
let expr: bTree = Node (Div, Leaf (Value 1.0), Node (Plus,
   Leaf (Value 3.0), Leaf (Value 5.0)))
```

not easy to read, since it relies on nested types and tuples. Luckily, it is not difficult to make a function, which converts the tree into a string for later displaying on the screen. Such functions are commonly called `toString` and will be discussed later in Chapter 15. Here, the strategy will be to increase indentation proportional to the depth of the nodes printed, and one version of `toString` is shown in Listing 11.4 The result is to be interpreted as `Div` is the division operation of a `Value 1.0)` `and the result of performing the \lstinline{Plus` of two other values.

The `toString` function is an example of tree traversal. For binary trees 3 different traversal orders are common: *Prefix*, *infix* and *postfix* order according to the order in which the value of a node and its two children are handled. Consider a node with a value `elm` and its two children `left` and `right`, the ordering is as follows:

Prefix order: `elm`, `left`, and `right`

Listing 11.4 bTreeToString.fsx:
A `toString` method aids the interpretation of tree values.

```
type element = Value of float | Mul | Plus | Minus | Div
// A binary tree
type bTree = Leaf of element | Node of element * bTree * bTree
let rec toString (tab: string) (t:bTree) : string =
  match t with
    Leaf v -> tab+(string v)+"\n"
    | Node (op, a, b) ->
      tab + (string op) + "\n"
      + (toString (tab + "  ") a)
      + (toString (tab + "  ") b)

/// The tree representation of 1/(3+5)
let expr: bTree = Node (Div, Leaf (Value 1.0), Node (Plus,
   Leaf (Value 3.0), Leaf (Value 5.0)))
printf "%s" (toString "" expr)
```

```
$ dotnet fsi bTreeToString.fsx
Div
  Value 1.0
  Plus
    Value 3.0
    Value 5.0
```

Infix order: `left`, `elm`, and `right`

Postfix order: `left`, `right`, and `elm`

Thus, `toString` in Listing 11.4 is a prefix traversal. In Listing 11.5 the 3 traversal schemes are used to convert a binary tree to a list of `element` values.

Traversal methods linearize the tree structure and in general, 2 different traversals are required to reconstruct the original tree. However, as we saw in Section 9.4, a stack can be used to properly evaluate postfix representations of mathematical expressions. The code in Listing 9.17 can be modified for this purpose. Firstly, no computations need to be done, but every operator must result in a node and every value in a leaf, and instead of a stack of values, we must stack the sub-trees. The result is shown in Listing 11.6. Note that the resulting element on the stack is the original tree as expected.

Listing 11.5 bTreeTraversal.fsx:
Pre-, in-, and postfix traversal of a binary tree.

```
type element = Value of float | Mul | Plus | Minus | Div
type bTree = Leaf of element | Node of element * bTree * bTree

/// Prefix traversal of a binary tree
let rec prefix (t: bTree) : (element list) =
  match t with
    Leaf e -> [e]
    | Node (e, left, right) ->
      e :: (prefix left) @ (prefix right)
/// Infix traversal of a binary tree
let rec infix (t: bTree) : (element list) =
  match t with
    Leaf e -> [e]
    | Node (e, left, right) ->
      (infix left) @ [e] @ (infix right)
/// Postfix traversal of a binary tree
let rec postfix (t: bTree) : (element list) =
  match t with
    Leaf e -> [e]
    | Node (e, left, right) ->
      (postfix left) @ (postfix right) @ [e]

let expr: bTree = Node (Div, Leaf (Value 1.0), Node (Plus,
   Leaf (Value 3.0), Leaf (Value 5.0)))
printfn "Prefix traversal:\n%A" (prefix expr)
printfn "Infix traversal:\n%A" (infix expr)
printfn "Postfix traversal:\n%A" (postfix expr)
```

```
$ dotnet fsi bTreeTraversal.fsx
Prefix traversal:
[Div; Value 1.0; Plus; Value 3.0; Value 5.0]
Infix traversal:
[Value 1.0; Div; Value 3.0; Plus; Value 5.0]
Postfix traversal:
[Value 1.0; Value 3.0; Value 5.0; Plus; Div]
```

11.3 Programming intermezzo: Sorting Integers with a Binary Tree

Consider the problem of sorting a list of integers

Problem 11.1

Given a list of integers, such as `[5; 5; 7; 8; 3; 1; 8; 9; 0; 8]`, sort them and print the result on the screen.

Listing 11.6 bTreeFromPostfix.fsx:
Using a stack to reconstruct a tree from its postfix traversal.

```
open Stack

type element = Value of float | Mul | Plus | Minus | Div
type bTree = Leaf of element | Node of element * bTree * bTree

/// Postfix traversal of a binary tree
let rec postfix (t: bTree) : (element list) =
  match t with
    Leaf e -> [e]
    | Node (e, left, right) ->
      (postfix left) @ (postfix right) @ [e]
/// Convert postfix traversal back into a tree
let rec fromPostfix (tkns: element list) (stck:
   stack<bTree>): stack<bTree> =
  match tkns with
    [] -> stck
    | elm::rst ->
      match elm with
        Value v ->
          push (Leaf (Value v)) stck |> fromPostfix rst
        | _ ->
          let (a, stck1) = pop stck
          let (b, stck2) = pop stck1
          push (Node (elm, b, a)) stck2 |> fromPostfix rst

let src: bTree = Node (Div, Leaf (Value 1.0), Node (Plus,
   Leaf (Value 3.0), Leaf (Value 5.0)))
let psfx = postfix src
let tg = fromPostfix psfx (create ())
printfn "Original tree:\n%A" src
printfn "Postfix traversal:\n%A" psfx
printfn "From Postfix tranversal:\n%A" tg
```

```
$ dotnet fsi postfixLibraryGeneric.fs bTreeFromPostfix.fsx
Original tree:
Node (Div, Leaf (Value 1.0), Node (Plus, Leaf (Value 3.0),
   Leaf (Value 5.0)))
Postfix traversal:
[Value 1.0; Value 3.0; Value 5.0; Plus; Div]
From Postfix tranversal:
[Node (Div, Leaf (Value 1.0), Node (Plus, Leaf (Value 3.0),
   Leaf (Value 5.0)))]
```

This is such a common task that the list module contains a function for just this, `List.sort`, but here we will use the problem to study programming with binary trees. Our strategy will be to build a generic library for binary trees and an application with a function for sorting these values.

A strategy for sorting values is to enter them into a binary tree such that a node's value is always larger than its left child and less than or equal to its right child. The list could thus result in the tree shown in Figure 11.3. A consequence of the sorting

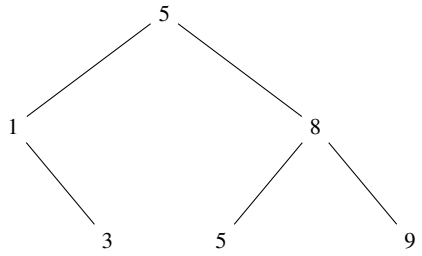

Fig. 11.3 A binary sorting tree for the list `[5; 1; 8; 3; 9; 5]`.

rule is that given a node, then its value is larger than all the values in the left sub-tree and no bigger than any values in the right sub-tree. Note also that the infix traversal of the tree gives a sorted list of values. In Figure 11.3 this is `[1; 3; 5; 5; 8; 9]`. In the spirit of our 8-step guide, let us make a sketch of the sorting algorithm to get a feeling of which types and functions could be useful. Firstly, we will need a generator of random numbers, to make sure we test many different combinations. This can be achieved with `System.Random()` and `List.map` as shown in Listing 11.7 Then we

Listing 11.7 bTreeSort.fsx:
Generating a list of random integers in the interval 0 to 9.

```
16 let rnd = System.Random()
17 let unsrt = List.map (fun _ -> rnd.Next 10) [1..10]
```

will need a type for a tree, and here we will use a generic type `bTree<'a>`, such that we reuse it for other values that can be compared with the $\leq$ operator. Our idea is to take the elements in the unsorted list to be inserted into a tree sequentially, and then finally write the result using infix traversal, e.g., as shown in Listing 11.8 Working

Listing 11.8 bTreeSort.fsx:
Using `List.fold` to sequentially insert integers into a binary sorting tree.

```
18 let srt = List.fold sortInsert (create unsrt.Head) unsrt.Tail
19 printfn "Unsorted list:\n%A\nSorted:\n%A" unsrt (infix srt)
```

with this, we realize that we will need a method for creating tree nodes and inserting them in a sorted fashion into a tree. Hence, we suggest the functions

```
create: v: 'a -> bTree<'a>
sortInsert: acc: bTree<'a> -> elm: 'a -> bTree<'a>
```

Given a value, the function `sortInsert` must start at the root of the tree and traverse down the tree until it finds a node with space for a leaf that obeys the sorting rule.

Thinking about this, we realize, that the tree type could well make use of the option type, e.g.,

```
type bTree<'a> = Node of 'a * bTree<'a> option * bTree<'a> option
```

and thus, an available position can be noted by the branch having the `None` value. Further, since we are in the functional programming paradigm, traversal must be recursive and insertion means the creation of a new tree. Finally, we arrived at the code in Listing 11.9. When programming this, it seemed useful to have

Listing 11.9 bTreeSort.fsx:
Insert a new node into an existing tree in a sorted manner.

```
3  let rec sortInsert (acc: bTree<int>) (elm: int): bTree<int> =
4    let v = retrieveValue acc
5    let l = tryRetrieveLeft acc
6    let r = tryRetrieveRight acc
7    if elm < v then
8      match l with
9        None -> replaceLeft (create elm) acc
10       | Some t -> replaceLeft (sortInsert t elm) acc
11   else
12     match r with
13       None -> replaceRight (create elm) acc
14       | Some t -> replaceRight (sortInsert t elm) acc
```

```
replaceLeft: c: bTree<'a> -> t: bTree<'a> -> bTree<'a>
replaceRight: c: bTree<'a> -> t: bTree<'a> -> bTree<'a>
retrieveValue: t: bTree<'a> -> 'a
tryRetrieveLeft: t: bTree<'a> -> bTree<'a> option
tryRetrieveRight: t: bTree<'a> -> bTree<'a> option
```

where `replaceLeft` and `replaceRight` replaces the left and right child `c` respectively in node `t`, `retrieveValue` retrieves the value stored in node `t`, and `tryRetrieveLeft` `tryRetrieveRight` follows the tradition of the other F# modules and returns a `Some bTree<'a>` or `None` depending on the existence of the child node.

At this point, we make a signature file for the library, and by the above arguments, we arrived at the code in Listing 11.10. This signature file specifically targets the problem of binary tree sorting, and a general-purpose library would have other functions as well. However, here we restrict the development to the problem at hand. The implementation of these functions turns out to be simple. In Listing 11.11, we have used pattern recognition in the definition of the arguments in several of the functions as, e.g., in `retrieveValue`. This makes the implementation particularly short, however, it may be less readable. Finally, putting it all together, a demonstration

Listing 11.10 bTreeGeneric.fsi:
The signature for a binary tree with insertion sort.

```
module bTree

type bTree<'a>

/// Create a tree with one value and no children
val create: 'a -> bTree<'a>
/// replace t's left child with c
val replaceLeft: c: bTree<'a> -> t: bTree<'a> -> bTree<'a>
/// replace t's right child with c
val replaceRight: c: bTree<'a> -> t: bTree<'a> -> bTree<'a>
/// retrieve the value of the root of m
val retrieveValue: t: bTree<'a> -> 'a
/// retrieve the left child
val tryRetrieveLeft: t: bTree<'a> -> bTree<'a> option
/// retrieve the right child
val tryRetrieveRight: t: bTree<'a> -> bTree<'a> option
/// Traverse the tree in infix order.
val infix: bTree<'a> -> 'a list
```

of the library and application can be seen in Listing 11.12. Due to the functional programming style, this implementation is robust and versatile, but not optimal. Depending on the internal workings of F#, each insertion potentially copies the pre-

Listing 11.11 bTreeGeneric.fs:
Insert a new node into an existing tree in a sorted manner.

```
module bTree

type bTree<'a> = Node of 'a * bTree<'a> option * bTree<'a>
   option

let create (v: 'a) =  Node (v, None, None)

let replaceLeft (c: bTree<'a>) (Node (v, l, r)) : bTree<'a> =
  Node (v, Some c, r)

let replaceRight (c: bTree<'a>) (Node (v, l, r)) : bTree<'a> =
  Node (v, l, Some c)

let retrieveValue (Node (v, l, r)) : 'a = v

let tryRetrieveLeft (Node (v, l, r)) : bTree<'a> option = l

let tryRetrieveRight (Node (v, l, r)) : bTree<'a> option = r

let rec infix (Node (v, l, r)) : ('a list) =
  (l |> Option.map infix |> Option.defaultValue [])
  @ [v]
  @ (r |> Option.map infix |> Option.defaultValue [])
```

Listing 11.12: The result of applying insertSorted on a random list.

```
$ dotnet fsi bTreeGeneric.fsi bTreeGeneric.fs bTreeSort.fsx
Unsorted list:
[9; 2; 1; 6; 6; 0; 7; 0; 6; 5]
Sorted:
[0; 0; 1; 2; 5; 6; 6; 6; 7; 9]
```

insertion tree and its sub-trees many times, and it is thus not to be expected to be particularly fast or memory-conserving.

11.4 Sets

A *set* is an unordered collection of data. Sets form the bases for much mathematics and much of computer science. For instance, `int` is the set of integers from $[2^16 \ldots 2^16.1]$ and `bool` is the set {false, true}. In fact, all types can be considered sets. Sets can be *empty*, a set with just 1 element is called a *singleton set*, and at least conceptually, sets can contain an infinite number of elements. In F# it is possible to represent *infinite sets*, but a discussion of this is beyond the scope of this book. The mathematical notation for sets is well-developed:

Empty set: The empty set is denoted $\emptyset$.

Set roster: A set can be written as a list of values with curly brackets and ellipses, e.g., $\{1, 2, \ldots, 10\}$. Remember, though, that the order of the elements is meaningless for sets.

Membership: An element x is a member in a set X is written as $x \in X$ and equivalently $x \notin X$ denotes non-membership.

Subsets: Subsets are denoted by $\subset$ and $\subseteq$, e.g., $\{'c', 'a',\} \subset \{'a', 'b', \ldots, 'z'\}$ and $\{'c', 'a',\} \subseteq \{'c', 'a',\}$. The negations of these are similarly defined $\{'a', 'b', \ldots, 'z',\} \not\subset \{'a', 'c'\}$.

Cardinality: The cardinality $|X|$ also known as the size of the set is the number of elements in the set X.

With sets comes a small number of basic operators

Complement: The complement of a subset $x \subset X$ is what is missing, i.e., $x^c = \{y : y \in X \text{ and } y \notin x\}$

Union: The union of two sets X and Y is $X \cup Y = \{z : z \in X \text{ or } z \in Y\}$

Intersection: The intersection of two sets X and Y is $X \cap Y = \{z : z \in X \text{ and } z \in Y\}$

Difference: The set difference between two sets X and Y is $X \setminus Y = \{z : z \in X \text{ and } z \notin Y\}$

F# has both a set class. Classes and objects will be discussed in further detail in Chapter 15. Presently, it is sufficient to think of a set class as an immutable type. F# furhter has a set module with more functions for sets, see `https://fsharp.github.io/fsharp-core-docs/reference/fsharp-collections-setmodule.html` for more details.

With the set module, an empty set can be created with `Set.empty`, and set can be created from a list `lst` as `Set lst`. Some of the important functions in the set module are:

`Set.add`: `x: 'T -> X: Set<'T> -> Set<'T>`
returns a new set $\{x\} \cup X$.

`Set.contains`: `x: 'T -> X: Set<'T> -> bool`
checks $x \in X$.

`Set.count`: `X: Set<'T> -> int`
returns $|X|$.

`Set.difference`: `X: Set<'T> -> Y: Set<'T> -> Set<'T>`
returns a new set $X \setminus Y$.

`Set.intersect`: `X: Set<'T> -> Y: Set<'T> -> Set<'T>`
returns a new set $X \cap Y$.

`Set.isEmpty`: `X: Set<'T> -> bool`
checks whether $X = \emptyset$.

`Set.isSubset`: `X: Set<'T> -> Y: Set<'T> -> bool`
checks whether $X \subset Y$.

`Set.remove`: `x: 'T -> X: Set<'T> -> Set<'T>`
returns a new set where $x \notin X$.

`Set.union`: `X: Set<'a> -> Y: Set<'a> -> Set<'a>`
returns a new set $X \cup Y$.

Sets can only be defined for elements, which can be compared, i.e., for which the $\geq$ and $\leq$ family of operators are defined.

Sets can be created from other collection types, such as lists, or by adding elements individually to an empty set, as demonstrated in Listing 11.13 A quick demonstration

Listing 11.13 setCreate.fsx:
Creating sets from lists or by adding elements one at a time.

```
let s1 = Set [3..5]
let s2 = Set.empty |> Set.add 3 |> Set.add 4 |> Set.add 5
printfn "s1 = %A\ns2 = %A" s1 s2
```

```
$ dotnet fsi setCreate.fsx
s1 = set [3; 4; 5]
s2 = set [3; 4; 5]
```

of union, intersection, and set difference is given in Listing 11.14. The module also

Listing 11.14 setUnionIntersectionDifference.fsx:
Illustration of set union, intersection, and difference.

```
let s1 = Set [3..5]
let s2 = Set [4;6;1]
printfn "s1 = %A\ns2 = %A" s1 s2
printfn "s1 union s2 = %A" (Set.union s1 s2)
printfn "s1 intersect s2 = %A" (Set.intersect s1 s2)
printfn "s1 difference s2 = %A" (Set.difference s1 s2)
```

```
$ dotnet fsi setUnionIntersectionDifference.fsx
s1 = set [3; 4; 5]
s2 = set [1; 4; 6]
s1 union s2 = set [1; 3; 4; 5; 6]
s1 intersect s2 = set [4]
s1 difference s2 = set [3; 5]
```

contains `Set.fold`, `Set.foldBack`, `Set.map`, and other functions similar to the `List` module, however, we leave it to the reader to consult the official documentation of the module for further detail.

11.5 Maps

A *map* is a discrete collection of relations between a domain and a codomain. As such, maps are discrete functions, but often they are termed databases or libraries since the elements from the domain are often called keys, and the corresponding values in the codomain are called values. The set of keys must be unique, while this is not the case for the set of values. Thus, the mapping between the set of keys and the set of values in a given map is *surjective* but not necessar-

ily *injective*. in F#, maps are sets of immutable *key-value pairs*. Similarly to sets, maps are classes and supported by a map module. A map can be created by `Map [("copenhagen", 1153615); ("berlin",3426354)]` or by `Map.empty |> Map.add "copenhagen" 1153615 |> Map.add "berlin" 3426354`. A brief list of important functions from the map module is:

`Map.add`: `k: 'K -> v: 'V -> m: Map<'K,'V> -> Map<'K,'V>`
: return a new map, which includes the `(k,v)` pair to the map `m`. If the key exists, then the existing key-value pair is replaced.

`Map.count`: `m: Map<'K,'V> -> int`
: count the number of `(k,v)` pairs there are in `m`.

`Map.isEmpty`: `m: Map<'K,'V> -> bool`
: checks whether the map `m` is empty.

`Map.keys`: `m: Map<'K,'V> -> System.Collections.Generic.ICollection<'K>`
: returns the sequence of keys in the map `m`. This can be turned into a set by `Map.keys m |> Set`.

`Map.remove`: `k: 'K -> m: Map<'K,'V> -> Map<'K,'V>`
: return a new map, which does not contain a `(k,v)` pair.

`Map.tryFind`: `k: 'K -> m: Map<'K,'V> -> 'V option`
: return the value `v` of the `(k,v)` pair if it exists.

`Map.values`: `Map<'K,'V> -> System.Collections.Generic.ICollection<'V>`
: returns the sequence of keys in the map `m`. This can be turned into a list by `Map.keys m |> List.ofSeq` or a set of unique values by `Map.keys m |> Set`.

Like sets, maps can only be defined for keys, which can be compared, i.e., for which the $\geq$ and $\leq$ family of operators are defined.

As an example of a map, consider the problem of producing the histogram of characters in a text. In Listing 11.15. The module also contains `Map.fold`, `Map.foldBack`, `Map.map`, and other functions similar to the `List` and the `Set` modules, however, we leave it to the reader to consult the official documentation of the module for further detail.

Listing 11.15 mapHistogram.fsx:
Calculating the histogram of characters using a map.

```
let rec hist (lst: char list) (m: Map<char,int>) :
    Map<char,int> =
  match lst with
    [] -> m
    | c::rst ->
      match (Map.tryFind c m) with
        None -> hist rst (Map.add c 1 m)
        | Some v -> hist rst (Map.add c (v+1) m)

let txt = "many years ago, there was an emperor, who was so
    excessively fond of new clothes, that he spent all his
    money in dress."
let lst = Seq.toList txt // Converts to a list of characters
let h = hist lst Map.empty

printfn "The text %A has the histogram\n%A" txt h
```

```
$ dotnet fsi mapHistogram.fsx
The text "many years ago, there was an emperor, who was so
    excessively fond of new clothes, that he spent all his
    money in dress." has the histogram
map
  [(' ', 22); (',', 3); ('.', 1); ('a', 8); ('c', 2); ('d',
    2); ('e', 14);
    ('f', 2); ('g', 1); ...]
```

11.6 Key concepts and terms in this chapter

In this chapter, we have looked at more data structures commonly used in programs. Key concepts have been:

- **Data structures** are often used as **models** of the real world and are defined **abstractly**. They may have many different **implementation** which vary in **computational complexity**.
- **Queues** are lists, where elements are added to the back and removed from the front. It does not have a built-in module in F# but is easy to implement.
- A **tree** is a non-linear structure, which organize data in a **hierarchical** structure. Trees are also not found as a standard data structure in F#, and unfortunately, not all facets of trees are easily implemented in the functional paradigm.
- A tree is a recursive data structure, consisting of **nodes** and their **children**, which are also trees.

- A **binary tree** is a node with at most two children.
- Binary trees are typically **traversed** in **prefix**, **infix**, or postfix order.
- A **set** is an immutable collection of values and is well-supported F#. Key set operators are **intersection**, **union**, and **set difference**.
- A **map** is a discrete, **surjective** but not necessarily **injective** function between a set of **keys** and **values**.

Part III

Imperative Programming Paradigms

In this part, we will primarily consider the *imperative* and *object-oriented programming paradigms*. Unfortunately, the imperative paradigm is used in the literature both to mean the overarching term of imperative paradigms and, as we do here, imperative programming without using object-oriented programming or other features. Thus it is ok, to say that object-oriented programming follows the imperative paradigm, but imperative programming does not necessarily follow the object-oriented paradigm.

Imperative programming is a paradigm for programming *states*. In imperative programming, the focus is on how a problem is to be solved, as a list of *statements* that affect states. In F#, states are *mutable values*, and they are affected by functions. In imperative programming, functions are sometimes called *procedures*, to emphasize that they may have *side-effects*. A side-effect is the result of the change of a state on the computer not related to the list of return parameters from the procedure. An imperative program is typically identified as using:

Mutable values
: Mutable values are holders of states, they may change over time, and thus have dynamic scope.

Procedures
: Procedures are functions that return "`()`", as opposed to functions that transform data. They are the embodiment of side-effects.

Side-effects
: Side-effects are changes of state that are not reflected in the arguments and return values of a function. The `printf` is an example of a procedure that writes text on the terminal but returns "`()`".

Loops
: The `for`- and `while`-loops typically use an iteration value to update some state, e.g., `for`-loops are often used to iterate through a list and summarize its contents.

Functional programming, can be seen as a subset of imperative programming and is discussed in Part II. *Object-oriented programming* is an extension of imperative programming, where statements and states are grouped into classes. For a discussion on object-oriented programming, see Chapters 15 to 17. Almost all computer hardware is designed for *machine code*, which is a common term used for many low-level computer programming languages, and almost all machine languages follow the imperative programming paradigm.

A prototypical example of an imperative program is a baking recipe, e.g., to make a loaf of bread, do the following:

1. Mix yeast with water.

2. Stir in salt, oil, and flour.
3. Knead until the dough has a smooth surface.
4. Let the dough rise until it has doubled its size.
5. Shape dough into a loaf.
6. Let the loaf rise until it is approximately double in size.
7. Bake in the oven until the bread is golden brown.

Each line in this example consists of one or more statements that are to be executed, and while executing them, states such as the size of the dough and the color of the bread change. Some execution will halt execution until certain conditions of these states are fulfilled, e.g., the bread will not be put into the oven for baking before it has risen sufficiently.

Object-oriented programming is a paradigm for encapsulating data and methods into cohesive units. E.g., a car can be modeled as an object, where data about the car including the amount of fuel, position, velocity, passengers, etc., can be stored together with functions for manipulating this data, e.g., move the car, add or extract passengers, etc. Key features of object-oriented programming are:

Encapsulation
: Data and methods are collected into a cohesive unit, and an application program need only focus on how to use the object, not on its implementation details.

Inheritance
: Objects are organized in a hierarchy of gradually increased specialties. This promotes a design of code that is of general use and code reuse.

Polymorphism
: By overriding methods from a base class, derived classes define new data types while their methods still produce results compatible with the base class definitions.

Object-oriented programming has a well-developed methodology for analysis and design. The analysis serves as input to the design phase, where the analysis reveals *what* a program is supposed to do, and the design *how* it is supposed to be doing it. The analysis should be expressed in general terms irrespective of the technological constraints, while the design should include technological constraints such as defined by the targeted language and hardware.