Chapter 13

Lists, dictionaries and percolation

The black 99 had been a *nozoki*: it aimed to the center of the white triangle. The whites, with the 100, connected.

Yasunari Kawabata, *Il Maestro di Go* (1942).

In this chapter we discuss different types of data structures, and some applications. We introduce a new C construct, namely the `union`, and we discuss linked *lists*. These data structures are very simple, but they fundamental in many contexts. We discuss how to build a simple dictionary with lists, suggesting how to lexically analyze a text. To this purpose we introduce doubly linked lists, which we can easily scroll through in both directions.

We define the concept of *recursivity*, and we apply to the *factorial*, and to a the design of a *tree* (in order to efficiently manage data). A useful application in various scientific and engineering contexts is the *reconstruction of connected components of a cluster*. After introducing an elementary algorithm that can do this, we use the lists to improve the code. These algorithms allow us to study the problem of *percolation*.

13.1 The unions

A `union` is a type of C variable, which at different times, during execution, can contain elements of different types. This might seem incompatible with the paradigm we usually adopt, namely that a variable is of a well-defined given type, and that each environment knows which type tit has to expect and return. A strict control of the used types (even in a simple multiplication) helps to avoid programming errors. Nevertheless, as we discuss in the following, there are some situations in which this flexibility comes in handy. In these cases the `union` construct is essential.

The syntax of a `union` is very similar to the one of a `struct`, though the object defined in this way is very different. Let us define a `union` of the `chameleon` type (as a chameleon easily changes color depending on its needs of the moment). We also define two instances of this `union` type, say `theChameleonJohn` and `theChameleonMary`, thus allocating the memory required to store them:

```
union chameleon{
  double green;
  int red;
  char yellow;
} theChameleonJohn, theChameleonMary;
```

Knowing that the `double` type takes eight bytes, the `int` type four bytes and the `char` type just one byte, each `union` of the `chameleon` type allocates eight bytes. Indeed, a `union` allocates the space necessary to store its largest possible member. Contrary to the case of a `struct`, in which a memory location is allocated to each one of its components, a single memory location, large enough to contain any member of the `union`, is allocated. The variables `theChameleonJohn` and `theChameleonMary` can contain, at different times during the execution, variables of the `double`, `int` or `char` type. As for the `struct` we refer to the members of a `union` by writing something like *name_ union.name_ element*, as in `theChameleonMary.red`.

At a given time, the type of a `union` is the one of the last data it has been assigned. The program does not have any information on this type, and it is the programmer's task to keep track of this (obviously an error could have catastrophic consequences). One possible way out is to define an associated variable which is updated each time the `union` is. For example, let us define the constants

```
#define GREEN 0
#define RED 1
#define YELLOW 2
```

and a new variable for each `union` we defined

```
int colorOfTheChameleonJohn, colorOfTheChameleonMary;
```

When we change the `union` we should always remember to update the associated variable as well, as in

```
theChameleonJohn.green = 256.32;
colorOfTheChameleonJohn = GREEN;
```

or, in case of an integer,

```
theChameleonJohn.red = 25632;
colorOfTheChameleonJohn = RED;
```

From the value of `colorOfTheChameleonJohn` we always know the type of value stored in the `union`. If, after the above statement has been executed, we would assume that the `union chameleon theChameleonJohn` contains a variable of the `double` type, we would probably run into a disaster. The allowed `union` operations and the ways of accessing its components are the same as for the `struct`. Moreover, it is possible to define `struct`s or arrays of `union`s. The `union` construct is not used very often. We quote three cases in which they can be very useful.

(1) Sometimes we need to align short words (for example variables of the `char` type) delimiting long words (for example of the `long int` type). In this case an array of `union`s can be the most efficient solution.
(2) An `union`s can be useful to initialize different parts of a large word (consisting, for example, of eight bytes) with variables of shorter sizes (consisting, for example, of a single byte).
(3) Sometimes, we might need to communicate an object whose type might depend on the context. For example, depending on the result obtained in a function, the function's return value might either be a variable of the `float` type or a variable of the `int` type.

Hands on 1 - A safe factorial and its logarithm.

The result of a factorial $n!$ can be represented as an integer variable of 64 bits only if $n \leq 20$. Write a function that returns a value of the `unsigned long long int` type containing the factorial, when called with an argument $n \leq 20$, while for $n > 20$ it returns a value of the `double` type containing the factorial's logarithm obtained by using the Stirling approximation:

$$n! \simeq \sqrt{2\pi}\, n^{n+\frac{1}{2}}\, e^{-n} .$$

The function, returning a `union`, should be used by considering these two possibilities.

13.1.1 *A virtual experiment*

We now discuss a typical situation in which the `union` construct is particularly useful. We consider the *on-line* analysis of the results of an experiment. Often a computer is connected by means of an appropriate hardware interface to an experimental instrument collecting data. Here, we obviously limit ourselves to a simulation of an experiment and we analyze the random results that our code generates. Still, the procedure we show is the same when we want to check a real experiment.

For example, think of a detector measuring what happens to a particle traveling through a physical medium. We start from the hypothesis that one out of three possible situations occur in each one of our measurements. In the first case the particle travels in the medium without creating any other particle. In this case, the variable we want to measure and analyze is the particle's speed. Correspondingly, the device returns a `double` value. In the second case the particle strongly interacts with the medium, and generates a swarm of particles. In this case, the device cannot measure the particles' velocities, but rather counts their number and returns a value of the `int` type. In the third case, an error occurs. This is a rare case which nevertheless might occur. The error might have several causes: the detector's electrical components possibly responded too slowly, the particle interaction occurred too close to the border of the material (i.e, outside the "confidence zone"), an error occurred in the memory of the device. In these cases we only want to write an error message on screen to notify the researcher performing the experiment. This error message, i.e., a character string, is produced directly by the device. In this situation the hardware interface returns a variable of the `double` or `int` type or a series of characters. Therefore, a `union` is the ideal data structure to manage these different cases when calling this function.

The relevant code for this operation is easy and compact, while the overall code simulating the experiment and randomly deciding which results to return is still easy, but a bit long. Therefore, we only describe the essential features and do not discuss it in detail.

The type of `union` we need is

```
union experimentData {
  double speed;
  int num;
  char *errorMessage;
};
```

The code simulating the experiment is of the form of the one given in Listing

13.1.

```
1  int main(void) {
2    int experimentOutput = 1;
3    int experimentNumber = 0;
4
5    srand(MY_SEED);
6    while (experimentOutput < 4) {
7      union experimentData scratchData;
8      experimentNumber++;
9      experimentOutput = setExperiment(experimentNumber);
10     scratchData = experimentBody(experimentOutput);
11     analyzeExperiment(experimentOutput, scratchData);
12   }
13   myEnd(experimentNumber);
14 }
```

Listing 13.1 The `main` function of the virtual experiment.

As we said before, we need to simulate the experiment, but do not want to discuss it in detail: the parts relevant for this discussion are the analysis of the experiment11 and the final output of the results13. In the remainder of the `main` function, we initialize the random number generator (line 5), and we iterate the experiment to have several independent tests. The function `setExperiment` on line 9 asks the user to insert a number; the value 4 ends the experiment, while the input value 3 causes the type of result to be random. The function `setExperiment`, which we do not report here, decides the type of result of a single experiment, and `experimentBody`, on line 10, randomly choses the output values (the particle's velocity in case 0, the number of particles in case 1 and the type of error in case 2). The experiment's output is defined by the `union experimentData scratchData` and the value of `experimentOutput`, which is, respectively, equal to 0, 1 or 2 in the three cases.

The function `analyzeExperiment` of Listing 13.2 is very easy (thanks to the fact that it uses a `union`!). Its input is `scratchData` and `experimentOutput`.

```
1  void analyzeExperiment(unsigned long int experimentOutput,
2                         union experimentData newData) {
3    if (experimentOutput == EXP_SPEED) {
4      averageSpeed += newData.speed;
5    } else if (experimentOutput == EXP_JET) {
6      hystogramNumber[newData.num]++;
7    } else if (experimentOutput == EXP_ERROR) {
```

```
1     printf("ERROR IN EXPERIMENT: %s\n", newData.errorMessage);
2   }
3 }
```

Listing 13.2 The function `analyzeExperiment`.

The variable `experimentOutput` tells us of which type the experiment's result is, and therefore, tells us which type of value is contained in `union experimentData newData`. In case of a velocity it is a `double`, and on line 4 the new velocity is added to the `averageSpeed` to compute the average. In case of a group of particles it is an `int`, and on line 6 we update a histogram. In case an error occurred, we just print the error message, on line 1.

Finally, the function `myEnd` prints the average velocity, the histogram of the distribution of particles produced during the interactions and the number of errors made.

Hands on 2 - A virtual experiment

Describe a problem analogous to the previous one, related to the control of an industrial production line. Write a simulator which takes various cases into account, and manage them with a `union`.

13.2 Linked lists

A problem appearing in many contexts and often leading to a dramatically narrow bottleneck is the management of small or large quantities of data. The interesting problems could be very different from each other: reading a text, creating a dictionary, inserting and storing addresses, managing a student *database* with time-dependent *records*. Also think of engineering applications, such as collecting data of industrial processes, or the control or analysis of experiments in any scientific field (signals obtained when treating biologically interesting molecules, information of chemical reactions, controlling experiments of collisions between highly energetic particles generating a multitude of new particles).

13.2.1 *Lists, strings and a dictionary*

Let us discuss how to organize words in a structure which can easily be consulted. We initially use a rather inefficient, though absolutely elementary structure, which helps us understand the concept of a *linked list*. We read the words from a data file and we progressively add them to our list. A first example consists in creating a list to which we add the words while we read them. In a second example we eliminate the repeated words, thus creating a dictionary starting from a text. The core of a structure referring to a structure of the same type is shown in the following definition of the `struct` of the `word` type,

```
struct word {
  char *pointerToString;
  struct word *pointerToNextWord;
};
```

The structure consists of two pointers. A first pointer points to a memory location where we write the word we have read as a character string (this is useful because the word length may vary and in this way the structure `word` does not depend on these details). The second pointer, pointing to a structure *of the same type*, is the crucial one. It points, in this case, to the next word of the list. Note that we do not reserve any memory for the data, but only the memory required for the two pointers. The memory for the data is reserved separately. An even simpler case consists in recording some sorted telephone numbers. In this case the memory space required to contain the data (the telephone number) could have been included in the structure itself,

```
struct phoneNumber{
  unsigned long int thisNumber;
  struct phoneNumber *pointerToNextPhoneNumber;
};
```

In this case the memory space needed to store the telephone number is allocated when declaring the structure of the `phoneNumber` type as it always has the same dimension. The fundamental feature of the linked list is still present at least one member of the structure is a pointer to a structure of the same type (in this case, the next element of the list). Figure 13.1 helps to clarify how the data are organized in the case of telephone numbers, with data containers and pointers to the next structure.

This type of list structure is more flexible than an array. In an array the data are classified in necessarily consecutive memory locations. It is not possible to change the order of the memory locations, as we need to do, for example, for dynamic structures, whose order changes throughout the execution. Adding memory locations to an array (by means of a `realloc` for example) is possibly a rather burdensome task due to the fact that the data need to be contiguous. It could be that the total space required by the `realloc` is only available in a different area than the used one. In this case, in order to be extended the array must be copied in a new part of the memory. In case of a very large array this can take a lot of CPU time.

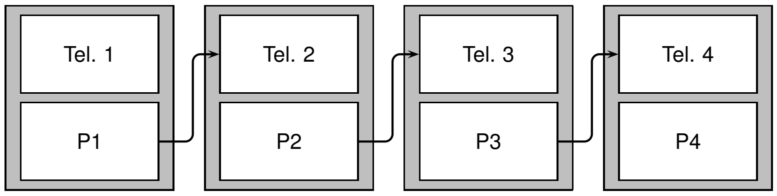

Fig. 13.1 A linked list of telephone numbers. The numbers are stored in the containers on top. The containers on the bottom store the pointer to the next structure.

Hands on 3 - A linked list

Write a code reading a text (for example a Shakespeare poem), which recognizes the words it contains and organizes them in a linked list. The detailed organization of the list can be chosen in different ways. It is instructive to try several of these.

Let us take a look at the important points of the code analyzing a text. First of all, we define a structure of the `word` type:

```
struct word {
  char *pointerToString;
  struct word *pointerToNextWord;
} *wordList = NULL;
```

Apart from defining a type of structure, `word`, we also declared a pointer to this type of structure: `wordList`. Note we did not reserve a structure of the `word` type, but only a pointer to a structure. Because we have not yet read any word, our list is still empty. Therefore, the pointer is initialized to `NULL`, a symbol defined by the compiler. The essential part of the program, typically included in the `main` function, has the following form

```
myEnd = 0;
while (myEnd != 1) {
  myEnd = readWord();
  buildInverseList();
}
printInverseList();
```

The function `readWord` reads a word, and the function `buildInverseList` inserts it in the list. This is an inverse list, because the links between two elements start from the last inserted element up to the first one. This is the easiest way to create a list. Note that for simplicity we did not include any arguments in the functions, and we rather use a certain number of global variables. When programming we always need to find a compromise between ease of writing and code robustness. In general, it is best to use as little global variables as possible as any function of the code could change their value, thus possibly causing programming errors.

The function `readWord` (Listing 13.3) reads the text from the global variable `fInput` (a pointer) to `FILE` and identifies the words. The function identifying the objects (in this case the words) in a context is called the *lexical analysis* function. It is an important component of the *parsing* process, i.e., the analysis of the grammatical structure of a character flow by decomposing the flow in more or less complex units (lexical elements) and interpreting them with respect to the grammatical rules.

The lexical analysis is a complex process, also because a word can be delimited in many ways inside a text. In most cases, one or more spaces appear at the end of a word, but also a punctuation mark (a comma, a period), parentheses or a simple "end of line" are allowed. Creating a complete lexical analyzer is a difficult task that goes beyond the scope of this textbook. For now, we only consider an elementary lexical analyzer. We leave any improvement to it as an exercise to the reader. To simplify the memory management, we assign a maximum number of characters per word, namely `MY_MAX` to the function `readWord`. If for some reason this maximum is exceeded, the program terminates with an error. The variable `myFlag` is reset when the word is complete. In this case the function transfers the control to the `main` function, while returning the value `0` unless the end of the file has been reached. In the latter case, the return value is `1`. In our elementary lexical analyzer a word can only end with a single space or with the end of the file. We leave the case of more consecutive blank spaces as an exercise. The end of line characters are ignored and we do not consider that they can possibly signal the end of a word. Therefore, also the last word of a line must be followed by a blank space. At the end of the word we add the terminator "\0" to the string. The character array `myString` is declared as a global variable, and can be seen by all functions of the program. The variable `myString` is used to temporarily store the words we have read, before they are inserted in the list.

The characters of the word are read one by one with the `fgetc` function. The function `fgetc` (included in `<stdio.h>`) takes a file pointer argument (the file must be opened and be accessible) and returns the character read from the file as a `char` converted into an `int`, or, if the end of file has been reached, the end of file signal, i.e. `EOF`, defined in the system headers. Its prototype is

```
int fgetc(FILE *stream);
```

At this point, the function given in Listing 13.3 should be clear.

```
int readWord(void) {
  char myFlag = 1;
  int j, myEnd = 0;
  j = 0;
  while ((j < MY_MAX) && (myFlag == 1)) {
    myString[j] = fgetc(fInput);
    if (myString[j] == ' ') {
      myString[j] = '\0';
      myFlag = 0;
    } else if (myString[j] == '\n') {
      j--; /*to ignore the end of a line*/
    } else if (myString[j] == EOF) {
      myEnd = 1;
      myString[j] = '\0';
      myFlag = 0;
    }
    j++;
  }
  if (j >= MY_MAX-1) {
    printf("Program interruption: "
           "the word was too long.\n");
    printf("Recompile with a new value of MY_MAX "
           "larger than %d\n", MY_MAX);
    exit(EXIT_FAILURE);
  }
  printf("Word of length %d: %s\n",
         strlen(myString), myString);
  return myEnd;
}
```

Listing 13.3 The function `readWord` to lexically analyze a text.

Note that it is much wiser to use the `fgetc` function than `fscanf`, which is more complex and sometimes deceiving, leading to reading errors.

The `buildInverseList` function builds the list (Listing 13.4). It is an easy and compact function calling `malloc` to reserve the required memory for the string that has just been read and is stored in the array `myString` of the global memory. The function `strlen` (included in `<string.h>`) takes a string pointer argument and returns the string's length:

```
size_t strlen(const char *myString);
```

(`size_t` is discussed in Section 10.2.2).

The pointer `scratchPointer` points to the new permanent memory space (reserved by the program for as long as it is being executed) in which we store the new word. To permanently (until the code has stopped executing) store the words, we copy the content of `myString` in a memory location specifically allocated and pointed to by `scratchPointer`. If the memory request is unsuccessful, the pointer `scratchPointer` is set to `NULL`, and the program is terminated with an error. Once this is done, we store the pointer to the last `word` structure that is in this moment stored in `wordScratchPointer` (without this pointer the communication between the various elements of the list would be broken). We then ask for new memory for the new `word` structure to contain the new word. This is the only moment in which we reserve a new `word` structure (before we only reserved a pointer to a structure). Again, we check whether the `malloc` was successful.

We are almost done now. We copy the word we just read from the temporary buffer `myString` into the buffer pointed by `scratchPointer`. This can be done by means of the system function `strcpy` (included in `<string.h>`) copying a string (including its terminating character `\0`) from one location to another. Its prototype has the form

```
char *strcpy(char *destination, const char *source);
```

and the pointer it returns points to the destination string (which is left unused in our code).

All we have to do now is fill the `word` structure we just created by copying the pointer to the new word in `wordList->pointerToString` and by copying the pointer to the second last structure (which we had wisely saved on line 11) in `wordList -> pointerToNextWord;`. In this way the new structure that we added contains a pointer to the word we have just found, and a pointer to the structure we had created before. The first structure we created points to the value NULL we used to initialize the first pointer we declared in the beginning of the code. A pointer to NULL in the pointer list wordList->pointerToNextWord indicates that the list is complete. At this point the function buildInverseList should be clear to us.

```
void buildInverseList(void) {
  char *scratchPointer;
  struct word *wordScratchPointer;

  scratchPointer = (char *)malloc(strlen(myString));
  if (scratchPointer == NULL) {
    printf("Program interruption: "
           "malloc failure number 1\n");
    exit(EXIT_FAILURE);
  }
  wordScratchPointer = wordList;
  wordList = (struct word *)malloc(sizeof(struct word));
  if (wordList == NULL) {
    printf("Program interruption: "
           "malloc failure number 2\n");
    exit(EXIT_FAILURE);
  }
  strcpy(scratchPointer, myString);
  wordList->pointerToString = scratchPointer;
  wordList->pointerToNextWord = wordScratchPointer;
}
```

Listing 13.4 The function buildInverseList to build a linked list.

A better understanding of how the linking of this list works can be obtained by looking at the function used to print it, given in Listing 13.5.

```
void printInverseList(void) {
  struct word *wordScratchPointer = wordList;
  while (wordScratchPointer != NULL) {
    printf("%s\n", wordScratchPointer->pointerToString);
    wordScratchPointer =
      wordScratchPointer->pointerToNextWord;
  }
}
```

Listing 13.5 The function printInverseList printing a linked list.

The function's mechanism is clear. The pointer wordList (a global variable) points to the structure containing the last word that has been read. We print it and we continue with the previous word, by means of `wordScratchPointer->pointerToNextWord`, until we reach the pointer to `NULL`, indicating that the word list is finished and that the reconstruction has been completed.

Hands on 4 - More lists and dictionaries

The lexical analyzer we discussed is extremely simple. Improve it by taking into account possible punctuation marks and uppercase and lowercase characters. Also consider that the "end of line" character terminates a word (unless a selected symbol, for example '`-`', is present indicating the word has been divided in two parts). Include letters with accents, apostrophes and the fact that the accents are sometimes inserted by a symbol following the word, into account. More generally, this is the right time to take another look at the table of ASCII characters, discussed in Section 1.7.2 and in Appendix20.4.2, in order to build a lexical analyzer which takes the most interesting cases into account.

Write a code that implements a direct list, starting from the first read word up to the last one read. The code is very similar to the one described for the inverse list. Finally, change the code to eliminate repeated words.

To eliminate repeated words it suffices to add four simple lines of code at the beginning of `buildInverseList` (Listing 13.4). A local pointer scrolls through the words we found so far and compares them to the last one we read. If it coincides with one of the previous ones, it is not recorded. The two strings are compared by means of the system function `strcmp`, included in `<string.h>`, which has as prototype

```
int strcmp(const char *string1 , const char *string2);
```

The function compares the two strings pointed by `string1` and `string2`. With the ASCII encoding (Section 1.7.2 and Appendix 20.4.2), it basically associates a numeric value to each string such that a string preceding it in alphabetic order is assigned a smaller numeric value. Thus, the function returns an integer smaller than, equal to or larger than zero if `string1` is, respectively, smaller, equal to or larger than `string2`. Note that, in the following code, in this case the function is left by means of a `return` statement placed inside a block. Not everybody considers this to be a structured construct because it entails we have a block of statements that we can access only in one way, but that we can quit in two different ways, namely at the end or by means of the `return`. We should not abuse this form, though sometimes it is useful. With this so-called *safeguard clause*, we basically reserve the right to immediately quit the function if a certain condition is verified (in our case if the word is already included in the list).

```
struct word *wordScratchPointer, *pointerCheck;
for (pointerCheck = wordList; pointerCheck != NULL;) {
  if (strcmp(pointerCheck->pointerToString, myString) == 0) {
    return;
  }
  pointerCheck = pointerCheck->pointerToNextWord;
}
```

A single link list is the easiest example among many possible cases. In Figure 13.2 we give an example of a double link list. Thanks to the double pointer structure it can be scrolled in both directions, from head to tail and the other way around. In this case, each structure stores, apart from the relevant data, also a pointer to the preceding structure and a pointer to the next one.

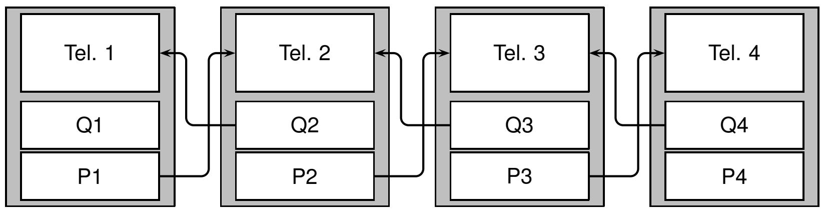

Fig. 13.2 A double link list. The structure is richer than a single link list. We can easily scroll through it in both directions, form head to tail and the other way around. The pointers P_i point to the next structure, the pointers Q_i to the preceding one.

13.2.2 *Recursive functions: computing the factorial*

To continue our analysis about how to organize data and how to build and manage lists, it is useful to briefly introduce the concept of *recursion*, which is treated in detail in Section 15.3.1.

A function is said to be recursive if it can call itself. Not all programming languages allow the use of this type of functions: C does, and this is one of its particularly interesting features. As an example we consider how to compute a factorial, $n! \equiv 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$, defined for integer, non-negative values n. The Section 4.3.1 contains a simple non-recursive code computing the factorial, while the recursive function is given in Listing 13.6.

```
unsigned long long int factorial(unsigned int n) {
  if (n == 0) return 1;
  return (n * factorial(n - 1));
}
```

Listing 13.6 Computing the factorial with a recursive function.

The key is the mathematical identity $n! = n \cdot (n-1)!$. When called with a positive integer, the function calls itself until it is called with the value zero. In the latter case, it returns one. So, when the function is called with argument 2 it returns in turn $2 = 2 \cdot 1$, when called with argument 3 it returns $6 = 3 \cdot 2$ and so on. A specific situation defines the condition ending the recursive chain, which in this case is the call to the function with argument equal to zero. Without this termination condition (or if the condition is wrong or never encountered) the function will call itself until it fills the computer's memory and blocks it. A function like ours, based on variables of the `unsigned long long int` type, can compute factorials up to $n = 20$, i.e., until the result's representation takes more than 64 bits. This is why the argument can be of the `unsigned int` type, which is sufficiently large.

A recursive function calling itself is completely analogous to calling a new function. The whole scope (local variables, parameters) must be stored, such that it can be recovered when returning to the calling function. An instructive way to think of a recursive function which is called many times is as if we wrote the function the same number of times, reproducing the same number of lines of code for each call to the function. In practice this is obviously not possible, given that a function could call itself recursively hundreds of times (and that we do not know this number a priori). Still, it helps us to figure out how complex this mechanism is. One of the main contexts in which a recursion is applied are *tree-like* data structures, which we discuss in Section 13.2.3.

13.2.3 *Binary trees and dictionaries*

The way we built the list in Section 13.2.1, storing one by one the (non repeated) words found in the text, may be very slow to sort. As the number of words increases, organizing the list, to sort it in alphabetic order for example, takes a large amount of computer resources. In the third part of this textbook, we analyze *scaling laws* describing how the execution time grows in function of the size of the input, which in this case is the number of words composing the list.

Building and managing a *tree* with recursive functions helps to create efficient dynamic structures. Trees are structures whose elements point to other elements of the same structure, as in lists. The elements of the trees, though, are organized in a more complex way. A list is a linear structure, while a tree is a branched structure with a ramification in each *node*. The initial node has k branches who all end in other nodes. Each of these k nodes can have again k branches, and so on. For example, think of a *binary tree* , i.e., a tree with $k = 2$. The first part of a binary tree is shown in Figure 13.3. The first word that has been read (word 1) occupies the first node; the (non repeated) words read next are placed along the branches of the tree. If, according to the alphabet, a new word precedes the word that has found its place in the first node, it follows the left branch, while a word following it in alphabetic order will follow in the branch on the right. This is done on every occupied node, and eventually the new word is placed in the first empty node along this path. This is a simple recipe to build a binary tree. We always consider and discuss trees which are upside down (as in Figure 13.3), i.e., with the root on top and the branches developing downward.

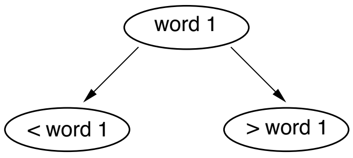

Fig. 13.3 The root of the tree. word 1 is the first read word. A word alphabetically preceding word 1 moves along the left branch, while a word alphabetically following word 1 moves along the branch on the right. The word stops in the first empty node along this path. The root of the tree is shown on top, while its leaves are shown at the bottom.

Hands on 5 - The binary tree

Write a code that reads a text and organizes it in a binary tree as the one we just discussed, eliminating the repeated words. Use a recursive algorithm. The task of the recursive function is to place the new word in the tree. It should call itself until it reaches the tree branch on which the word should be stored. Write a function printing the word list in alphabetic order.

The use of recursive functions makes the code reading a word list and organizing it in a binary tree much more compact. We want to write a function addToTree (Listing 13.7) calling itself up to when it has reached the leaf of the tree where it can deposit the new word.

Organizing words in a tree is extremely efficient. If the N words are read in a random order, the time it takes to sort them in alphabetic order only grows like $N \log(N)$. In this case, the tree is said to be *balanced*. This is for example also the case if the words were not picked in a random way, but rather taken from a novel. Typically even in this case we will obtain a balanced tree, since the author usually does not choose to only use words starting with the letter "a" in the first chapter, with the letter "b" in the second chapter, and so on. Instead, if the read words have a well-organized order, the tree is unbalanced, and searching the tree is a lot slower. For example, reading a dictionary which is already sorted, the words are all placed on a single branch, producing a completely unbalanced tree, whose depth grows linearly with the number of read words. In this case the tree structure is as inefficient as a list.

The node's structure has two pointers to nodes of the same type,

```
struct word {
  char *pointerToString;
  struct word *nextWordLeft;
  struct word *nextWordRight;
} *treeRoot = NULL;
```

As usual, the first pointer points to the word stored in the corresponding character array, while the second and the third pointers allow to find the following word on the tree, respectively, along the left or right bifurcation.

Upon execution, we start by creating a pointer to a structure of the word type pointing to NULL. The basic structure of the program is very easy:

```
fW1.myEnd = 0;
while (fW1.myEnd != 1) {
  fW1 = readWord();
  treeRoot = addToTree(treeRoot, fW1.scratchPointer);
}
```

The structure fW1 contains the pointer to the word we have read and a control variable signaling when to terminate the program execution (because the last word of the text file has been read). It has the form

```
struct fromWord {
  int myEnd;
  char *scratchPointer;
} fW1;
```

The function declared as

```
struct fromWord readWord(void);
```

reads a word, whose pointer is returned, together with a control variable, or *flag*, which is equal to one if the end of file is read. Also in the latter case, the read word is a legitimate word and should be analyzed and possibly stored. This function's structure is the same as the one for the management of the simple list discussed in Section 13.2.1. The function `addToTree` adds the read word to the tree starting from its root and moving along its branches until an empty node (a leaf) is found.

Note that in this simple example we are not treating lowercase or uppercase letters any differently, and we simply assume we only have to deal with lowercase letters. We leave the correct treatment of uppercase letters as an exercise.

```
struct word* addToTree(struct word* inputWord,
                       char *localPointer)
{
  int stringDifference;
  if(inputWord == NULL){
    inputWord =
      (struct word *) malloc(sizeof(struct word));
    if(inputWord == NULL){
      printf("Program interrupted: malloc failure 1\n");
      exit(MY_RUIN);
    }
    inputWord->pointerToString = localPointer;
    inputWord->nextWordLeft = NULL;
    inputWord->nextWordRight = NULL;
  } else if((stringDifference =
             strcmp(inputWord->pointerToString,
                    localPointer)) != 0){
    if(stringDifference > 0) {
      inputWord->nextWordLeft =
        addToTree(inputWord->nextWordLeft,localPointer);
    } else {
      inputWord->nextWordRight =
        addToTree(inputWord->nextWordRight, localPointer);
    }
  }
  return inputWord;
27 }
```

Listing 13.7 The function `addToTree` adding a word to the tree.

Let us have a look at how this function, given in Listing 13.7, works. If the function is called with an argument which is an empty location, the word can be added to the tree. In this case `thisWord == NULL`. We add a leaf to the tree, where we store the word and add a bifurcation leading to two empty positions (lines 12-14). A `malloc` (line 7) reserves the memory location required to store the word we just read, and we check whether the memory has been allocated correctly (if not, the program is terminated with an error). We assign to the string pointer the address of the word we have read and the two pointers to the left and the right are initialized to `NULL`. The latter information is then returned with a single `return` (line 26). The word has been inserted and all open calls in the recursive chain are closed, in a cascade. Instead, if the considered location is occupied, the function `strcmp` on line 16 (`strcmp` is discussed in Section 13.2.1) plays a fundamental role. If the new word is equal to the one in the considered node, nothing happens: the search is finished, without leading to any update of our list. In this case the function returns the address of the word already occupying the considered memory, and the recursive chain is closed.

Instead, if the considered tree node is not a leaf and does not contain a word equal to the one we just read, we check whether the new word alphabetically precedes or follows the one already present in the node. In this case, the function is called recursively to check, respectively, the next node on the left and the one on the right. For example, let us assume, without loss of generality, that the new word alphabetically precedes the one already contained in the node. In this case the recursive call has the following form

```
thisWord->nextWordLeft =
  addToTree(thisWord->nextWordLeft,localPointer);
```

Thus, we want to insert the new word in the next branch on the left. The recursive procedure runs along the tree and stops either when it finds an empty space (where it inserts the new word) or when a word equal to the read one has already been inserted. In Chapter 14.5 we discuss the recursive procedures in much better detail and we clarify how crucial the definition of the return condition is.

The power of a recursive procedure is probably most clear in the function that prints in alphabetic order the word list: this function is also recursive, and it is shown in Listing 13.8.

```c
void printTreeInAlphabeticOrder(struct word* thisWord) {
 if (thisWord != NULL) {
   printTreeInAlphabeticOrder(thisWord->nextWordLeft);
   printf("%s\n", thisWord->pointerToString);
   printTreeInAlphabeticOrder(thisWord->nextWordRight);
 }
}
```

Listing 13.8 The function `printTreeInAlphabeticOrder`.

The function consists of only four short lines, accomplishing an absolutely nontrivial task! Take your time to digest what is happening: this is the only way to fully comprehend what the recursive approach entails. The function first moves down along the left side of the tree. The word in the leftmost leaf is certainly the first one in alphabetic order. We then travel along all branches of the tree from left to right. Note that when printing a tree, its left half is printed before its right half (the first read word occupies the root of the tree and divides it in two). Moreover, this is true for each subdivision of the tree.

Also note how powerful and compact this code is with respect to the one discussed in Section 5.6.3, and to the one based on pointers discussed in Section 6.5.1.

13.3 Connected clusters

We now consider a lattice problem. In particular we try to understand what happens if we place objects on the sites of a lattice: when do they start to form large connected clusters which become infinite if the lattice is infinite? This leads us to the important subject of *percolation*. An algorithm based on lists allows for an effective treatment of this problem.

In this Section we consider a two-dimensional square lattice ($D = 2$): analogous considerations can be made for other lattice shapes, such as a triangular lattice, also in a different number of physical dimensions (for example in the $D = 3$ case). We place binary (or Boolean) variables on the lattice sites. By convention, we assume that the two possible values of these variables are $(-1, +1)$, or $(0, 1)$, or red and green. The problem can be interpreted in various ways and have different applications: however, the values we choose for representing our variables do not change the problem's structure. For example, we could decide that the lattice represents our city's telephone network with, in each site, a telephone exchange which could be