ANDRE BARCZAK

DATA STRUCTURES AND ALGORITHMS

PUBLISHED BY

Massey University

Edition: February 2022

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

Preface

The study guide was created specifically for the 159.201 course (Data Structures and Algorithms). It is not supposed to replace books, at least not completely. The guide is only meant to provide an organised source of reading to students, who are then going to be able to follow the slides and the source code more easily. The chapters are in the same order in which the topics are presented in the slides.

The study guide was created using LaTeXwith Tufte's format. Students can use the extra wide margins to make notes when studying.

Thanks to Associate Professor Dr. Chris Scogings for his advice on the topics that are part of this study guide and for his notes, assignments and examples that inspired the writing of this guide.

Thanks to Dr. Teo Susnjak for helping with the revision and for the fun and motivational chats we often have about computer topics.

Thanks to Dr. Napoleon Reyes for the modification suggestions and for the debugging of many of the sample codes.

1 Introduction

This is a basic course that covers Data Structures and Algorithms. The languages used in all examples are C and C++. We start with simple C functions and data structures, linking with what students have learnt in 159101 and 159102. Later we introduce a new syntax and new programming paradigms that are available only in the C++ language. This will give students the opportunity to learn basic concepts that are relatively language independent, while learning the specific implementation details using C and C++. Once students can grasp these concepts, they will also be able to implement the same algorithms and data structures in a number of other popular languages (Java, Python, Basic etc).

This course assumes that students already know the basics of the C language and can independently write, compile, debug and run simple C programs involving pointers. This chapter starts with a revision on C language, including arrays, pointers and memory allocation.

For additional reading we recommend Drozdek's books (e.g. [1]). There are also many good sites with examples in C and C++, for example `http://www.cplusplus.com/` [2].

[1] Adam Drozdek. *Data Structures and Algorithms in C++*. Cengage Learning, Boston, MA, 2013

[2] 2019. URL `http://www.cplusplus.com/`

1.1 - Arrays and Structs

It is worth revisiting the topic of arrays in the C/C++ languages. The declaration of an array is static (its the size cannot change after the program was compiled). The use of array's elements is very simple: `myarray[index]` allows one to access and modify a certain element. We will use a simple conventional way of representing arrays schematically, as shown in figure 1.1.

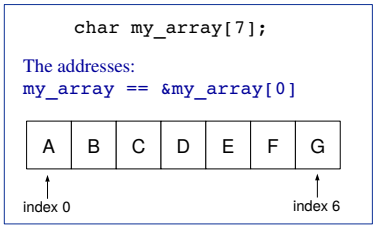


Figure 1.1: This shows the basic declaration of an array in C or C++. It also shows how the elements are contiguous in memory. Remember that the address for the entire array is also the address for the first element of the array.

The name of the array is a variable that holds a pointer to the first element, hence the address of the first element `&my_array[0]` is the same as the address `my_array` (see figure 1.1).

The main advantage of using arrays is that the declaration and usage are very simple, giving the programmer the ability to access any element by index (random access). There are a few important disadvantages. There is room for programming mistakes if the index goes beyond the limit of the array (imposed by how many elements it was declared). Although some OS are more forgiving than others, such mistakes may eventually cause the application to crash (or in computer science jargon, it gets a *segmentation fault*). Another

disadvantage is that elements cannot be inserted in between elements without some extra effort to shuffle elements to the left or to the right. Deleting elements may leave a space in memory that is not being used. Another disadvantage is the fact that the array cannot grow dynamically. If the programmer does not know how much space is needed, he/she might make two opposing decisions: either make the array too small (risking getting a segmentation fault) or make the array too big (wasting memory). The only solution for such a problem is to study how to acquire memory as it is needed during runtime (*dynamic allocation*), a topic that will be constantly discussed in this course.

In C, there are only a handful of data types available. For example: `char, int, float, double` are examples of simple data types that you have used before in the 159101 and 159102 papers. You have also learned how to group these basic data types together using a `struct`. Listing 1.1 shows a simple example of how to use `struct` with a `char` and an `int`.

Listing 1.1: struct in C

```
1  struct Address {
2    char street[100];
3    int number;
4  };
5  ...
6  main(){
7  ...
8    Address myaddress;
9    strcpy(myaddress.street,"Imaginary Road");
10   myaddress.number=12;
11 }
```

1.2 - Pointers in C/C++

Let us recall some of the reserved symbols that have special meaning in C. A very important resource in C and C++ is their ability to use virtual addresses directly, and to manipulate these addresses. Even arithmetic expressions involving memory addresses can be achieved easily. We start this section with an example of the syntax involved in dealing with pointers.

1.2.1 - *

This is used to declare a pointer (a variable that points to an address). E.g., if you want a pointer to an integer, declare it like `int * a;`. The same symbol is used to de-reference a pointer, getting its content instead. This is a source of confusion, as the same symbol is used for different purposes depending on context.

Listing 1.2: Pointers in C

```c
#include <stdio.h>
#include <stdlib.h>
main(){
  int a=5;
  int *b;
  b=(int*)malloc(sizeof(int));//allocate an integer to b
  printf("the address that b points to: %p\n",b);
  *b=7;
  printf("the content of the integer pointed to by b: %d \n", *b);
  printf("the address of integer a: %p \n", &a);
  //copy the address of a to b
  b=&a;
  printf("the content of the integer pointed to by b: %d \n", *b);
}
```

1.2.2 - &

This symbol is used to get the address of a variable. E.g., if you want to pass the address of a variable, use `&` in front of it.

1.2.3 - malloc() and new

These are called to allocate new memory to a previously created pointer. Just declaring a pointer to an integer does not make the space for an integer available. One needs to allocate the space explicitly otherwise the program may crash.

1.2.4 - ->

The arrow replaces the '.' to access a member of a pointer. It can be though as an equivalent of dereferencing the pointer and accessing the member, e.g., if `A` is a pointer to some `struct` that has `a` as a member:

`A->a` is equivalent to `(*A).a`

In listing 1.2 there is an example of how to access the address of a variable `a`. Also, the code shows how to use a variable `b` to hold a pointer to an integer. There are statements using and printing different addresses connected to `a` and `b`. Study this code very carefully to remember how to use pointers in C.

Remember that functions can accept arguments by value or by reference. The value of the original variable will only change if it is passed by reference.

In listing 1.3 there are two functions. The first one, `function1`, accepts an integer. This is called *passing by value*. The value is copied temporarily when `function1` is used, and the original value in `main` does not change. The second function, `function2`, receives a pointer

Listing 1.3: Passing by value or by reference

```
#include <stdio.h>

int b=8;

void function1(int A) {//passing by value
  printf("Function1: A contains %d. A is located at %p\n",A,&A);
  A=b;
}

void function2(int *A) {//passing by reference
  printf("Function2: A contains %d. A is located at %p \n",*A,A);
  *A=b;
}

main(){
  int a=10;
  printf("Variable b is located at %p \n",&b);
  printf("Variable a is located at %p \n",&a);
  //pass by value
  function1(a);
  printf("After function1, a=%d \n",a);
  //pass by reference
  function2(&a);
  printf("After function2, a=%d \n",a);
}
```

to an integer. This is called *passing by reference*, and the original value on `main` can be changed.

Study this code very carefully in order to understand the printouts. These two forms of passing parameters to functions are very important in the C language, and need to be understood very clearly to make it to the next step. Make sure that you copy, compile and run this code until you understand what the addresses' relationship in the printout after listing 1.3 mean.

Output:

```
Variable b is located at 0x601040
Variable a is located at 0x7fff2431429c
Function1:  A contains 10.  A is located at 0x7fff2431427c
After function1, a=10
Function2:  A contains 10.  A is located at 0x7fff2431429c
After function2, a=8
```

In listing 1.4 there is another example, this time with three different functions. The first function, `function1`, accepts a pointer to an integer, similar to what was in the previous code. Functions `function2`

and `function3` have a different syntax, but essentially do the same thing: they accept a *pointer to a pointer* (a double pointer). This is not simply coding by obfuscation, there is a very important role played by functions like that (this will become more clear in the linked-list chapter).

A function with a double pointer can change the address of the variable (a pointer) that was passed to the function. Instead of trying to modify an integer variable as we did in listing 1.3, `function2` and `function3` modify the address to which the original pointer was pointing to. Clearly `function1` could not do that (in line 31 of listing 1.4 the variable a was not printed as 20, but as 10).

The best way to understand listing 1.4 is to type, compile and run the code. Once you obtain a printout, it is important to draw the variables schematically to understand what can and what cannot be modified. In figure 1.2 a simple example is shown. Be aware that the addresses of the pointers in figure 1.2 are arbitrary, so they can be completely different every time the program runs.

Listing 1.4: Passing a pointer by value or by reference

```
1  #include <stdio.h>
2  int b;
3  int c;
4  void function1(int *A) {
5    printf("Function1: A contains %p. A is located at %p\n",A,&A);
6    A=&b;
7  }
8
9  void function2(int *&A) {
10   printf("Function2: A contains %p. A is located at %p \n",A,&A);
11   A=&b;
12 }
13
14 void function3(int **A) {
15   printf("Function3: A contains %p. A is located at %p \n",*A,A);
16   *A=&b;
17 }
18
19 main(){
20   int *a;
21   int x=10;
22   a=&x;
23   printf("Main: a contains %p. a is located at %p\n",a,&a);
24   printf("Main: b contains %d. b is located at %p\n",b,&b);
25   printf("Main: x contains %d. x is located at %p\n",x,&x);
26   printf("a=%d \n\n",*a);
27
28   b=20;
29   function1(a);
```

```
30   printf("After Function1: a contains %p\n",a);
31   printf("a=%d \n\n",*a);
32
33   b=30;
34   function2(a);
35   printf("After Function2: a contains %p\n",a);
36   printf("a=%d \n\n",*a);
37
38   b=40;
39   a=&x;
40   function3(&a);
41   printf("After Function3: a contains %p\n",a);
42   printf("a=%d \n\n",*a);
43 }
```

Output:

```
Main:  a contains 0x7fff6c491a94.  a is located at 0x7fff6c491a98
Main:  b contains 0.  b is located at 0x601044
Main:  x contains 10.  x is located at 0x7fff6c491a94
a=10

Function1:  A contains 0x7fff6c491a94.  A is located at 0x7fff6c491a78
After Function1:  a contains 0x7fff6c491a94
a=10

Function2:  A contains 0x7fff6c491a94.  A is located at 0x7fff6c491a98
After Function2:  a contains 0x601044
a=30

Function3:  A contains 0x7fff6c491a94.  A is located at 0x7fff6c491a98
After Function3:  a contains 0x601044
a=40
```

1.3 - Allocating and freeing memory in C/C++

In order to avoid the memory usage pitfalls explained earlier, one can use dynamic memory allocation to create arrays. Let us explore simple examples using the functions `malloc()`, `free()`, `new` and `delete`.

1.3.1 - malloc() and free()

In the old C style, malloc() and free() are used to control the allocation of memory. For example, allocating memory to an array can be seen in the next listing (1.5).

Listing 1.5: malloc() and free()

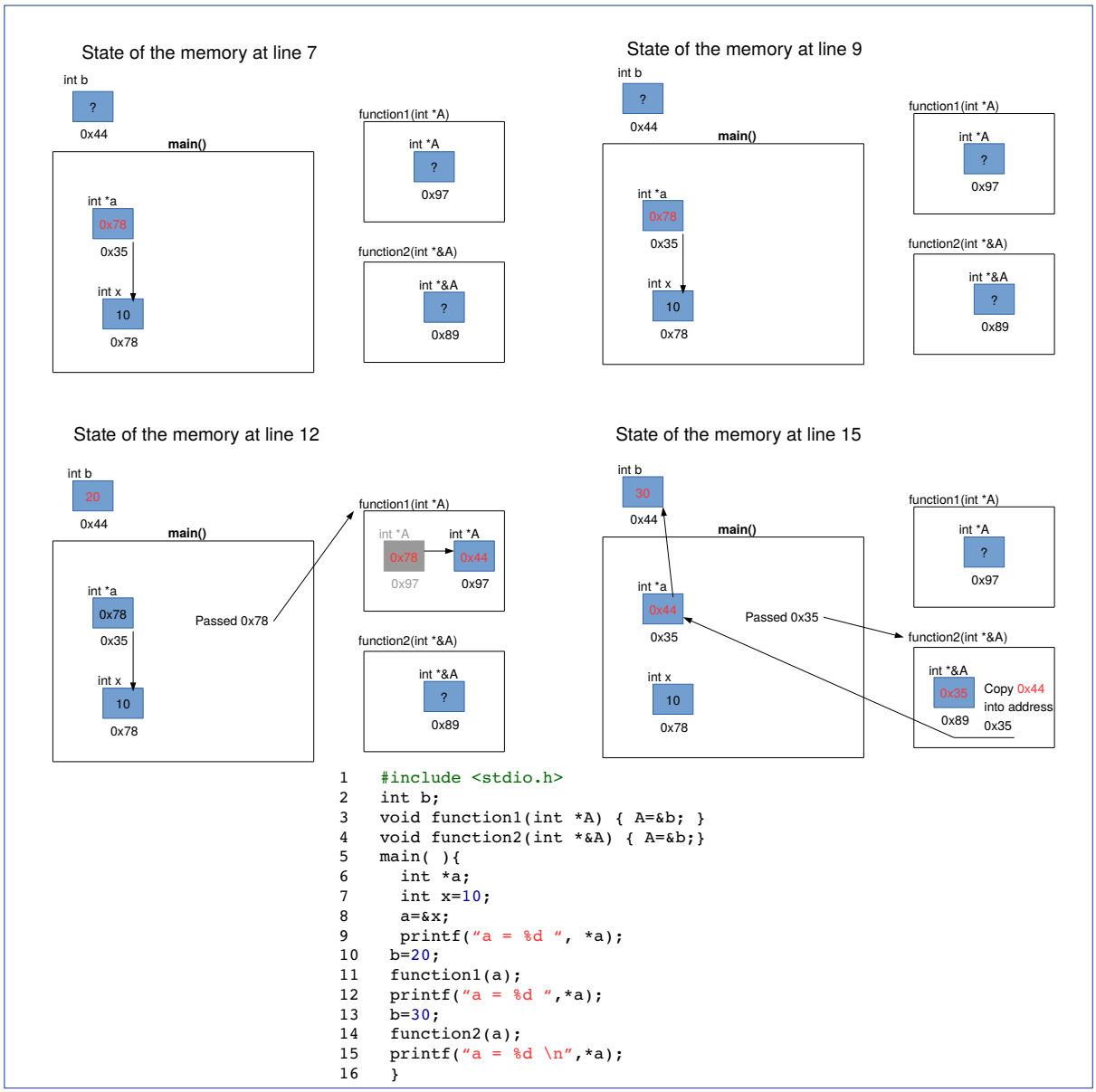
```
1 #include <stdio.h>
2 #include <stdlib.h>
3 main(){
4   int a[10];//static array, 10 places for integers
5   int *b;//pointer only, no allocation yet
```

```
6   b=(int*)malloc(10*sizeof(int));//now allocating 10 places for integers in b
7   a[5]=10;
8   b[5]=10;
9   printf("a[5]=%d and b[5]=%d\n",a[5],b[5]);
10  free(b);//10 places for integers are no longer available in b
11 }
```

If `b[]` is used before the `malloc()` or after `free()`, the result may be a segmentation fault (the executable 'crashes').

1.3.2 - new and delete

In C++ one can also use new and delete with the same purpose as seen in the previous section.




```
1   #include <stdio.h>
2   int b;
3   void function1(int *A) { A=&b; }
4   void function2(int *&A) { A=&b;}
5   main( ){
6     int *a;
7     int x=10;
8     a=&x;
9     printf("a = %d ", *a);
10   b=20;
11   function1(a);
12   printf("a = %d ",*a);
13   b=30;
14   function2(a);
15   printf("a = %d \n",*a);
16   }
```

Figure 1.2: Passing double pointers.

Listing 1.6: new and delete

```
#include <stdio.h>

main(){
  int a[10];//static array, 10 places for integers
  int *b;//pointer only, no allocation yet
  b = new int[10];//allocate 10 integers
  a[5] = 10;
  b[5] = 10;
  printf("a[5]=%d and b[5]=%d\n",a[5],b[5]);
  delete[] b;//10 places for integers are no longer available in b
}
```

If `b[]` is used before the `new` or after `delete`, the result may be a segmentation fault (the executable 'crashes'). Listing 1.6 can be only compiled with `g++` (not with the `gcc` command).

One could see the allocation and deallocation functions as a complete solution for the the dynamic allocation. However, note that once allocated and populated with data it would be difficult to add extra elements into the existing structure. One could allocate a new array with one more element and copy all the old elements to it, but this is very inefficient. There are better ways to cope with single element allocation and deallocation, and we are soon going to cover some of these data structures in the next chapters.

1.4 - Command line arguments

For this course it is very important to recall how to use arguments from command line. Command line arguments can help to test the program with different arguments without having to resort to re-compiling it. It saves time for the testing and it also opens the opportunity to use files as input for the program (the input file name can then be changed without having to re-compile the program).

Listing 1.7 shows a standard main function using arguments. A commonly accepted convention establishes that the first argument for the `main()` function is called `argc` and the second `argv`. Note that the first one is an integer, while the second one is a double pointer of a char. Listing 1.7 will simply print the number of arguments and print each argument separately.

The first argument, `argc`, counts the number of arguments entered in the command line. This includes the name of the executable file itself. So if your program is called `myprog.exe` and you call it with the command `myprog.exe 1 2 3`, then the `argc` counts 4 arguments (the name of the executable plus three numbers). The arguments are always separated by a space, so if for example you call `myprog.exe 1.2 3`, this would be counted as 3 arguments (the name of the executable plus 2 numbers). How many arguments would be counted if the command `myprog.exe A B C D` is called?

The second argument, argv, is trickier but easy to use. The double pointer stands for a pointer to a pointer, and the argv pointer can be regarded as an array of strings. So each argument is read as a string, not as a number. To be able to read each argument, one can use argv[index] to access one particular argument, as it can be seen in listing 1.7.

Listing 1.7: Command line in C/C++

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(int argc, char **argv) {
4   int counter=0;
5   printf("The number of arguments entered is %d \n",argc);
6   for(counter=0; counter<argc; counter++){
7     printf("Argument %d is %s \n", counter, argv[counter]);
8   }
9 }
```

Output (after calling program 1.7 like ./a.out A BB CCC:

```
The number of arguments entered is 4
Argument 1 is A
Argument 2 is BB
Argument 3 is CCC
```

If the arguments are numeric, an appropriate conversion function (to convert the strings to a number) needs to be used. For example, an integer can be converted from a string by using the function `atoi(argv[index])`, while a float can use `atof(argv[index])`.

Figure 1.3 shows an example of arguments from command line. Note that `argv[0]` is always a string with the name of the executable file (the symbol # is used as the OS prompt).

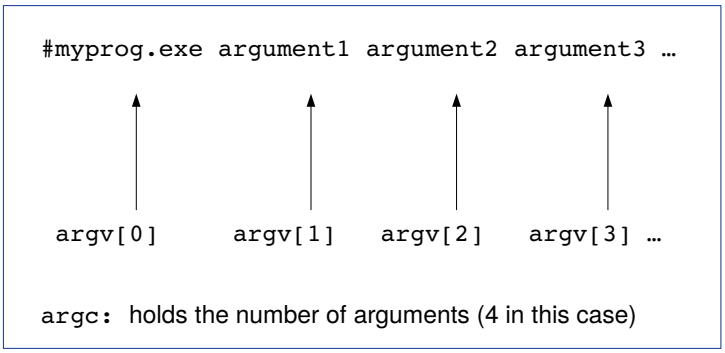


Figure 1.3: Arguments from command line. It is worth becoming acquainted with the use of terminals. For Unix or Linux user it is usually second nature, while students using Mac OS or Windows might have to teach themselves the advantages of command line. The alternative to using command line is to install an IDE that provides an input field, so the arguments can be passed in the same way as command line. Code::Blocks and Eclipse provide that.

When using command line arguments, the program has to be called from a terminal (DOS session in Windows, or an xterm in Unix/Linux). An alternative to that is offered by many of the program editors available. For example, in CodeBlocks the editor has an option to insert command line argument, so when the user pushes the button to execute it will automatically include the arguments.

1.5 - Abstract Data Types

The concept of an *Abstract Data Type* (**ADT** for short) is: *a model for a certain data structure, with a certain behaviour and certain properties*. These properties can be explored by specific algorithms in order to write and run code efficiently.

There are many advantages in using ADTs. An algorithm can make use of a certain ADT (e.g., a Stack), so all the details of how to insert data or delete data from a Stack can be hidden (assumed to be well known), reducing the details. It allows the readers of pseudo-codes to have an overview (main picture) of the algorithms, as long as they know what a Stack is and how it should behave.

The other big advantage of the concept is that different implementations can be used for the same ADT. We will see examples of ADTs that can be implemented using an array or a linked-list. A programmer using such implementations might be unaware of the details, using the well known properties and operations without knowing the full implementation of that ADT. Often ADTs can be implemented using other ADTs underneath.

Once the operations for a certain ADT are implemented (respecting the particular properties of that ADT), it is relatively easy to change or upgrade the implementation. We will see in later chapters that one convenient way of implementing ADTs is to use a C++ class. Yet another advantage of using well known ADTs is that their fundamental operations and properties have a proven theoretical limit in terms of performance and memory usage.

1.6 - Comparing Algorithms

Comparing algorithms is difficult. Firstly, what criteria should be adopted, and how should it be measured? For example, one can use runtime to compare algorithms that achieve the same output given an input. However, does that suffice? Measuring runtime is not accurate, as the operating system can be busy with other tasks while running our algorithm. The runtime depends also on the hardware platform, and on the implementation. It is better to have some neutral way of measuring the efficiency of algorithms that is independent of hardware, operating system, programming language and implementation. The answer to the dilemma discussed above is to establish the *complexity* of the algorithm.

Complexity of an algorithm is a function that describes the upper bound (asymptotic growth rate) of a certain characteristic. The most common characteristic for an algorithm is its runtime. The runtime complexity can be thought of as or its growth trend in relation to the data size. Finding the complexity of an algorithm can be achieved by counting the number of operations, or counting the number of cycles that an algorithm has to run to get its correct output. The number of operations is obviously related to the code, which would pose the question that we would be comparing different things. For example,

two algorithms that visit the entire data structure once would have to traverse it N times. If algorithm A has 5 lines of code in the loop and algorithm B has 10, it is clear that algorithm B would have more line counts. However, we can claim that both algorithms have a *linear complexity*, as the upper bound of the trend is a function of N ($f(N)$). In other words, these two algorithms would have the same complexity. We cannot make any assumptions about runtime directly, but we can state that if any of the algorithms take time t for a problem size of 100 numbers (N=100), they will take 10 times more if the problem size is 1000 (N=1000). For the complexity function the direct runtime comparison between algorithm A and algorithm B is irrelevant, what matters is the trend of the growth of the runtime.

Let us compare two well known algorithms for searching an array. If we try to traverse the entire array from the beginning to the end, the worst case scenario is that we have to traverse all the N elements of the array. Of course we could be lucky and find the key in the first position, which would then cost only 1 operation. We could also define an average case, where we would try to search for every key while measuring the effort. This would give us an average effort proportional to N/2. Often the worst case scenario is what matters for performance, as it cannot get worst than that. In the case of searching, the worst case scenario certainly happens when the key being searched for is *not* in the array. So if we want to find out about the performance of the algorithm, we can search for a key that does not exist in the array, as this will be the worst case scenario.

In mathematics we can represent the limiting trend of a function by a notation called *big O* (it is written $O(f(N))$. One can replace the function inside the big O notation to indicate the upper bound of the function's trend given a problem size N.

For example, $O(1)$ would indicate constant time for an algorithm, meaning that no matter what the size of the data structure, the algorithms runtime is fixed. We can imagine that there are other complexities such as linear $O(N)$. In this case, the worst case scenario for a linear algorithm is that the runtime grows proportionally to the growth of the size of the data structure. There are other common complexities that will be discussed later on.

Back to the comparison between searching algorithms, we can use linear searching and binary searching:

- **Linear search**: we start from the first element and look at each item until we find the key we are searching for.
- **Binary search**: we look in the middle of the array first, compare with the key and decide if we want to search the left half or the right half, depending on the key value. Note that for the binary search to work properly, the data needs to be *sorted* before starting. We repeat the process with one of the halves. This implies in repeating the search on a problem size half of what we started with. For every repetition, the data is divided by 2.

  Assuming that we have a N sized array, we can count the number

of comparisons between the given key and the element of the array. In the linear searching it is clear that the worst case scenario takes N comparisons. Therefore we can declare that the big O notation for the complexity of any implementation of a linear search is $O(N)$.

For the binary search the analysis needs to be more rigorous. We can say that for every split trying to find the key, we get to compare between the given key and the middle of a smaller and smaller set, until we can find the element. In the worst case scenario, we get to divide N by two until we have a single element. This last element is either equals the key or not, in which case we proved that the key is not contained in the array without having to look at every element. What would be the $O()$ complexity in this case? See table 1.1.

number of comparisons	0	1	2	3	...	k
length of the sub-array	N	$N/2$	$N/4$	$N/8$	...	$N/2^k$

Table 1.1: Binary search.

In the worst case scenario, we get to a single element (the length of the sub-array) in the sub-array after splitting it k times. Therefore:

$$
\begin{aligned}
N/2^k &= 1 \\
N &= 2^k \\
\log N &= \log 2^k \\
\log N &= k \log 2 \\
k &= \log(N)/\log(2)
\end{aligned}
$$

Therefore we can state that the number of comparisons to find a key in the array using binary search is proportional to $\log N$, and we represent that: $O(\log N)$

Why we only use the $f(N)$ without any constants? Because we are interested in the term of the equation that *dominates* the runtime. Suppose that a certain algorithm has a number of steps to run. After analysing the code, you conclude that the runtime function is $t = N^2 + 3N + 62$. As N gets bigger, only the first factor dominates the runtime:

$$
\begin{aligned}
N = 10,\ t = 192\ (N^2 accounts\ for\ 8\%) \\
N = 100,\ t = 10362\ (N^2 accounts\ for\ 96\%) \\
N = 1000,\ t = 1003062\ (N^2 accounts\ for\ 99.7\%)
\end{aligned}
$$

So after the analysis, one can write the big O as the dominant factor of the equation, or $O(N^2)$. [3]

Common complexity found in various algorithms are (see figure 1.4):

- $O(1)$ constant time

[3] find complexities for classical algorithms using data structures at http://bigocheatsheet.com/

- $O(\log(N))$ logarithmic time
- $O(N)$ linear time
- $O(N \log(N))$ linearithmic time (loglinear time, or "en log en")
- $O(N^2)$ quadratic time
- $O(N^3)$ cubic time
- $O(N^k)$ polynomial time
- $O(K^N)$ exponential time
- $O(N!)$ factorial time

So what does the complexity of an algorithm tell us? It gives us an idea of how the runtime can grow when the data grows. Typically with current hardware one can run algorithms with $O(N^3)$ with relatively large amounts of data, but one might have to wait for a bit. If the application needs to run in real-time, any complexities above quadratic are bad news. Computer scientists work hard to both find algorithms with better (lower) complexities as well as to prove mathematically that there is no algorithm better than a certain complexity (a very hard task indeed).

There are still many open questions: for example, can we get a better algorithm to factorise numbers? With the current algorithms it is *infeasible* to factorise large integer numbers (infeasible means, in this context, theoretically possible, but taking a huge amount of time, even longer than our lifetimes or the known universe's existence). In this particular case it works to our advantage, as the fact that it is difficult to factorise large integers is used for cryptographic applications (the RSA algorithm is an example).

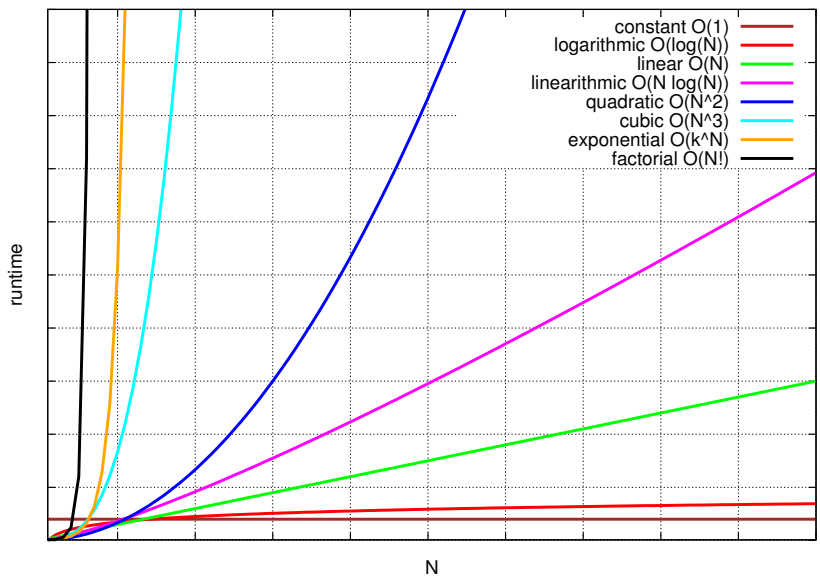


Figure 1.4: Complexity $O()$ trends. A higher complexity does not imply that the program will run slower than another algorithm with lower complexity. Often with small N algorithms can run fast as well, it is with increasing N that the runtime can be very large.

1.6.1 - Other complexity measurements

Complexity can also express the best case scenario or even the average case scenario. Notations for other bounds include:

Θ, Ω, ω, o etc. [4]

Other characteristics other than runtime can also be expressed by the same complexity notations. One important property for an algorithm is the *memory complexity*, which tells us what is the upper bound trend (asymptotic growth rate) for memory occupation to run certain algorithms with size N. This seems unnecessary for an array being searched, but it can be the case that certain algorithms need to copy data in a special format before they can carry out their operations. Typically, algorithms have an *input*, an *output* and a *working memory*. It is the latter that is measured in a memory complexity analysis. In the case of sorting, many algorithms have constant $O(1)$ because they just need some variables to hold temporary values. So even if the input is larger the working memory remains the same. In chapter 11 we will study an algorithm called *merge sorting*, which usually needs a copy of the data as working memory. In this case the sorting algorithm's memory complexity is $O(N)$, as the entire input has to be copied for the algorithm to work.

[4] These notations (including the Big O notation) are called Bachmann-Landau notation, after the two mathematicians that invented them. Θ is called Big Theta and it represents the average time rather than the worst-case scenario. Ω is called Big Omega, and it usually represents the best case scenario, although there are two contradictory definitions for it in the literature. The other small letter complexities (o, or small O, ω, or small omega etc) are usually related to representing a function that dominates over the complexity, i.e., are more restrictive than their counterparts.

1.7 - Exercises

1. Re-type, compile and run the program in listing 1.2.
2. Re-type, compile and run the program in listing 1.3. Draw the virtual addresses that are printed in each function and understand what is being passed as a parameter in each function.
3. Re-type, compile and run the program in listing 1.4. Draw the virtual addresses that are printed in each function and understand what is being passed as a parameter in each function. Compare the output with the output after listing 1.4 (and also figure 1.2).
4. Re-type, compile and run the program in listing 1.5. Try to use some index value outside the boundary of the array. Observe that sometimes this causes a segmentation fault.
5. Re-type, compile and run the program in listing 1.6. Try to use some index value outside the boundary of the array. Observe that sometimes this causes a segmentation fault.
6. Re-type, compile and run the program in listing 1.7. Use the program in a terminal or a DOS session so you can pass parameters to main(). Alternatively use the editor (e.g., CodeBlocks, Visual Studio etc) to pass the parameters, so you can still press the run button.

2 *Linked-lists*

The first ADT (Abstract Data Type) we are going to study is the *linked-list*.

Linked-lists are sequences of connected nodes. Linked-lists are initially empty after they are declared (no nodes). Nodes are added (inserted) dynamically, and they can also be deleted dynamically.

Each node of a linked-list contains at least one pointer that points to another node within the linked-list. Similar to an array, the pointer to the linked-list points to the first node (element) of the linked-list.

Linked-lists are linear data structures, in the sense that one node points to a single node, in such a way that there is a chain of nodes. The only way to reach a node is by traversing the linked-list from the first node (head) to the node of interest.

Linked-lists can be used as a dynamic alternative to arrays, as long as there is no need for random access (with linked-lists we can only do sequential access). Table 2.1 compares arrays against linked-lists.

Linked-lists	Arrays
Grows during runtime (dynamically allocated)	Fixed size (compilation time)
Easy to insert in the middle	Inserting may require shuffling
Easy to delete from the middle	Deleting may leave empty spaces
Sequential access only	Random access (via index)
Slow to search (complexity O(N))	Fast to search (complexity O(1))
Needs specific functions to insert and delete	Simple access to every element via index

Table 2.1: Linked-lists compared to Arrays.

The schematic representation of a linked-list and its nodes can be seen in figure 2.1. Note that each node links to the next and that the last nodes points to nothing, represented by the `NULL` pointer.

The linked-list's address can be thought as a pointer that always points to the head (the first element). The end of the linked-list is indicated by a `NULL` pointer, i.e., the last element's pointer is `NULL`. Schematically the `NULL` pointer can be represented by a slash.

The declaration of a Node of the linked-list can use a `struct`. Inside the struct, one needs to include at least some data (a char, an integer etc) and a pointer of the same type of the node. The compiler is able to understand that and do the recursion for the Node type (listing 2.1).

For pure C syntax the Node should also be declared as a type:

```
typedef struct Node Node;
```

However, in C++ this line is not necessary.

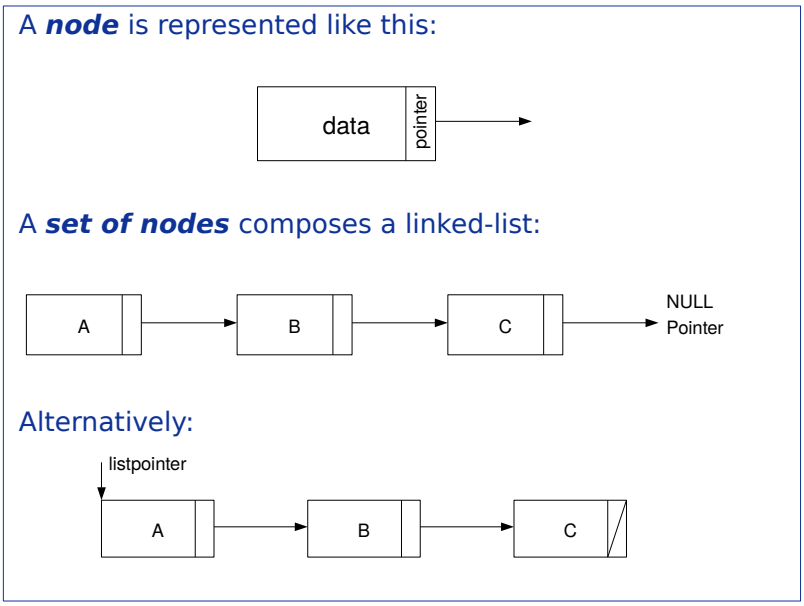


Figure 2.1: Nodes of a linked-list. Inside each node, there is at least a key (or a key with a whole record with other data) and a pointer. The key does not need to be fixed when declaring the node, but when implementing simple searching functions the programmer has to choose a key from the data.

Listing 2.1: Linked-list Node declaration

```
1 struct Node {  //declaration of a Node
2   int accnumber; //some data...
3   float balance; //more data...
4   struct Node *next; //the pointer to the next node
5 }; //Note the recursive declaration
```

Remember that the single integer pointer does not get a space in memory until it is allocated some space via `malloc()` or `new`. The same happens with the linked-list nodes, they need to be allocated memory space. The nodes (or elements) should be inserted to the linked-list by allocation and then linking.

We have to start a linked-list by declaring its head, also of type Node. It is convenient to declare a pointer to the head that is initially `NULL`, and then use a function to insert new nodes as required. We need to choose where to link the newly inserted nodes. It will become clear in the next section that the easiest way to insert nodes to an existing linked-list is to insert nodes in the front of the linked-list (or in the head of the linked-list). This is the case because the only access point to the linked-list is the pointer to its head.

2.1 - Inserting Nodes into a Linked-list

The insertion of new nodes into an existing linked-list should use the functions `new` or `malloc`. After creating the new node and loading it with some data, the new node has to be linked with the rest of the linked-list. The simpler way to do it is to link the new node to the head of the linked-list, so the head pointer (listpointer) points to the new node. The node's data usually includes a *key*.

We introduce the first algorithm in the form of pseudo-code (algorithm 1). This explains the logic of the algorithm without the formal syntax of a proper programming language. The pseudo-code can later be implemented as real code in some language.

Algorithm 1 Add Node

```
1: function AddNode(key)                                   ▷ New item
2:     declare a new Node temp
3:     allocate memory to temp
4:     copy key to the data of temp
5:     copy the address of listpointer to next pointer of temp
6:     copy the address of temp to listpointer
7: end function
```

Listing 2.2: Linked-list AddNode function

```
1  #include <stdio.h>
2  struct Node {
3    int accnumber;
4    float balance;
5    struct Node *next;
6  };
7
8  // function to add a new node to the FRONT of the list
9  void AddNode(Node *& listpointer, int a, float b) {
10   Node *temp;
11   temp = new Node;
12   temp->accnumber = a;
13   temp->balance = b;
14   temp->next = listpointer;
15   listpointer = temp;
16 }
17
18 Node* A = NULL; //declaration of linked-list A
19
20 main(){
21   AddNode(A, 1, 1.11);
22   AddNode(A, 2, 2.22);
23   AddNode(A, 3, 3.33);
24   //...insert other elements
25 }
```

Algorithm 1 can be implemented in C as per listing 2.2 `AddNode()` function.

Note that because the insertion is happening at the head (insert at front), the order of the keys are reversed. So the order of the keys of the linked-list in listing 2.2 would be 3,2 and 1 (see figure 2.2).

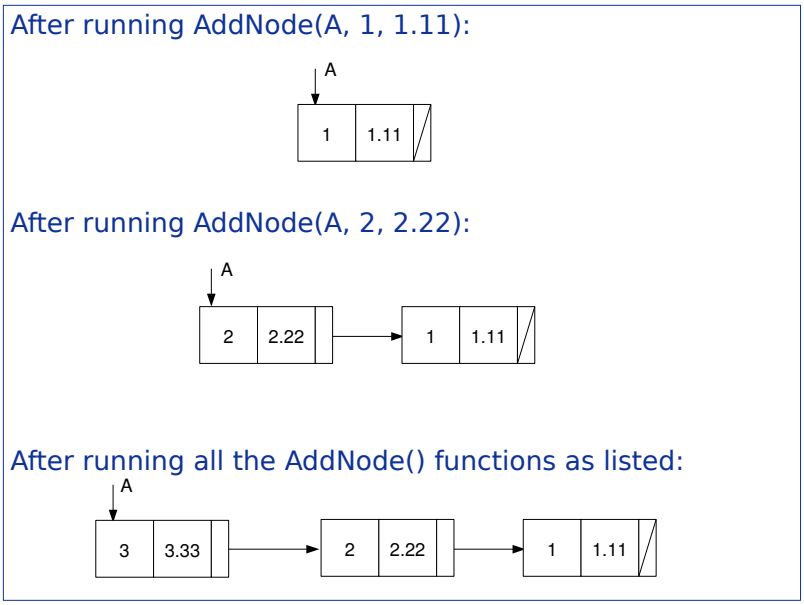


Figure 2.2: Linked-list after running program 2.2. Note that the elements are in reverse order because the new node is always inserted in the front.

Also a very important issue for the function `AddNode` is the use of `Node*& listpointer` as an argument. If we passed only `Node* listpointer` (without the &) the original listpointer could not be modified by the function, causing the insertion to fail. Although the insertion fails, a new node would have been created but no longer accessible.

2.2 - Printing all the Nodes of a Linked-list

Traversing a data structure is the act of finding and doing something with all its elements. Printing is one example of traversing a data structure to simply print the data inside each element. Traversing could also be used to actually carry out some computation of the data inside the element. In this course we are often going to use traversing as synonym for printing.

In order to traverse a linked-list, we need to start from the head, as no other elements (nodes) have external pointers that give us access to them. We standardised the name of the head of a generic linked-list to `listpointer`, but any variable that points to a `Node` type could be used. The head has a pointer to the next node. The second node has a pointer to the third node and so on and so forth. Therefore, the traversing process can use a temporary pointer (we will call it `current`) that changes to the next node in line, printing them all. When `current` finally becomes the `NULL` pointer, the end of the linked-list has been reached and the loop can be interrupted. Algorithm 2

shows the pseudo-code for a printing function.

Algorithm 2 Print linked-list

```
1:  function PrintLL(key)
2:      declare a new Node current            ▷  pointer of type Node*
3:      current = listpointer                 ▷  current points to the head
4:      while true do
5:          if current is NULL then
6:              break from while loop
7:          end if
8:          print data at current
9:          copy next at current to current   ▷  or: current=current->next
10:     end while
11: end function
```

Listing 2.3 shows an example of C implementation for the printing function. Note that here the argument for the linked-list is `Node *listpointer` (*without* the &). This is the case because we only need access to where `listpointer` points to, and do not need to modify it while printing.

The crucial point to understand is the statement `current=current->next`. During the runtime of the algorithm 2 the pointer `current` is moving along the linked-list. We can at any moment access the values inside the `Node` that `current` is pointing to, including the `next` pointer (which is a member of the `Node` struct). Therefore copying the `next` of `current` to itself makes `current` move one node forward (see figure 2.3).

Listing 2.3: Printing function

```
1  void PrintLL(Node *listpointer) {
2    Node *current;
3    current = listpointer;
4    while (true) {
5      if (current = = NULL) { break; }
6      printf("Account %i ", current->accnumber);
7      printf("balance is %1.2f\n", current->balance);
8      current = current->next;//this is important!!
9    }
10   printf("End of the list.\n");
11 }
```

Listing 2.4: Main function using AddNode() and PrintLL()

```
1  int main() {
2    AddNode(A, 1, 9.99);
3    AddNode(A, 2, 8.88);
4    AddNode(A, 3, 7.77);
5    PrintLL(A);
6  }
```

After running listing 2.4, the output is:

```
Account 3 balance is 7.77
Account 2 balance is 8.88
Account 1 balance is 9.99
End of the list.
```

Figure 2.3 shows the `PrintLL()` function working schematically.

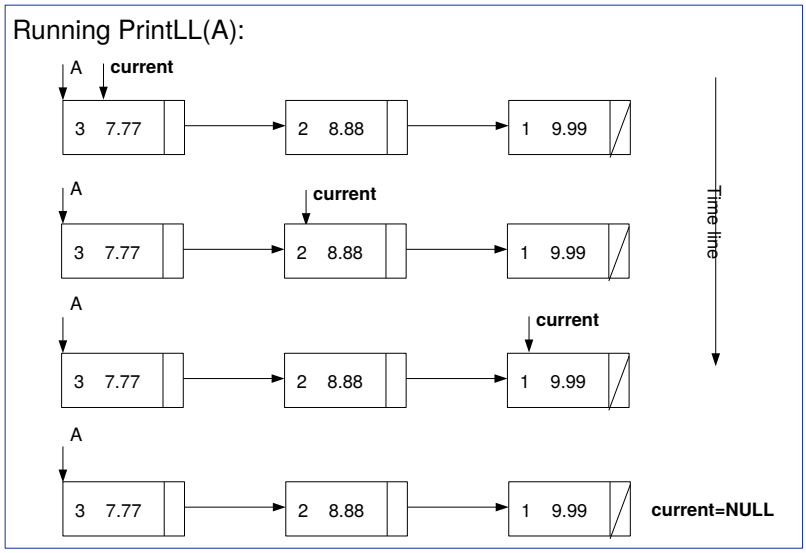


Figure 2.3: Printing the entire Linked-list. Note how `current` traverses the linked-list.

2.3 - *Inserting Nodes at the end of a Linked-list*

It was easy to write a function that inserts new elements at the head of the linked-list. But now in order to insert the new node *after* the tail of the linked-list we have to find the last element first. We learnt how to traverse a linked-list to print it, so we can stop at the last element *before* the pointer `current` becomes `NULL`. How can we change the while loop to achieve that? If we ask the question: is `current->next` not `NULL`? This solves the problem. We can present the C code straight away (listing 2.5), as this is just a variation on what we have done so far.

Listing 2.5: AddNode to the **end** of the linked-list

```
1  void AddNode2(Node*& listpointer, int a, float b) {
2    Node *current;
3    current=listpointer;
4    if(current!=NULL){
5      while (current->next!=NULL){
6        current=current->next;
7      }
8    }// now current points to the last element
9    Node *temp;
10   temp = new Node;
```

```
11    temp->accnumber = a;
12    temp->balance = b;
13    temp->next = NULL;
14    if(current!=NULL) current->next = temp;
15    else listpointer=temp;
16  }
```

One important aspect to observe is that inserting nodes in this way is very inefficient. Every time we insert a new node, the entire linked-list has to be traversed from the head to the last element. If the linked-list is large enough, the `AddNode2()` function will run very slowly. An alternative is to keep an extra pointer pointing to the end of the linked-list.

Another important observation is the fact that now the order in which the elements are inserted is the *same* order as they were entered. For example, if one runs the code in listing 2.6:

Listing 2.6: main() using AddNode to the **end**

```
1  int main() {
2    AddNode2(A, 1, 9.99);
3    AddNode2(A, 2, 8.88);
4    AddNode2(A, 3, 7.77);
5    PrintLL(A);
6  }
```

The output is:

```
Account 1 balance is 9.99
Account 2 balance is 8.88
Account 3 balance is 7.77
End of the list.
```

Figure 2.4 shows the state of the linked-list after every insertion.

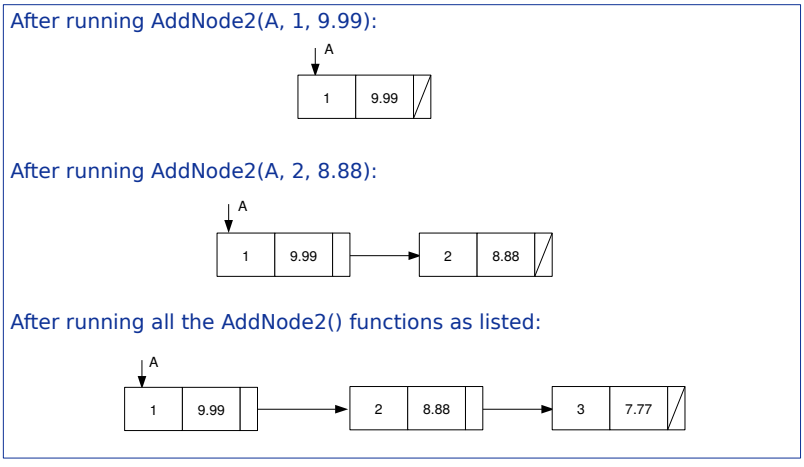


Figure 2.4: Adding nodes to the end of the linked-list. Note that the elements are in the same order in which they were inserted.

2.4 - Searching Nodes in a Linked-list

Now that we know how to traverse the entire linked-list and to find the last element, we can modify the existing functions and use them to search for a certain key or a certain node (say, by its position) in the linked-list.

Using a `current` pointer to a node, we can traverse the linked-list until the `key` of the node pointed by current is equal to the argument of the search. If during the traversal we never find the `key` and `current` becomes `NULL`, then we know the there is no node with the same key. A possible solution for a simple search is illustrated in algorithm 3 and listing 2.7.

Algorithm 3 Search a key in a linked-list

```
1:  function SearchLL(key)
2:      declare a new Node current          ▷  pointer of type Node*
3:      current = listpointer
4:      while true do
5:          if current is NULL then
6:              break from while loop
7:          else
8:              if key is equal to data of current then
9:                  print data of current
10:                 return
11:             end if
12:             copy next of current to current   ▷  current = current->next
13:         end if
14:     end while
15:     print error message   ▷  this indicates that there is no Node with this key
16: end function
```

Listing 2.7: Searching function

```
1  void Search(Node *listpointer, int x) {
2    Node *current;
3    current = listpointer;
4    while (true) {
5      if (current == NULL) { break; }
6      if (current->accnumber == x) {
7        printf("Balance of %i is %1.2f\n",
8          x, current->balance);
9        return;
10     }
11     current = current->next;
12   }
13   printf("Account %i is not in the list.\n", x);
14 }
```

Listing 2.8: main() function using the Search() function

```
1 int main() {
2   AddNode(A, 1, 9.99);
3   AddNode(A, 2, 8.88);
4   AddNode(A, 3, 7.77);
5   Search(A,123);
6   Search(A,1);
7   Search(A,2);
8   Search(A,3);
9 }
```

As an example, listing 2.8 using AddNode() and the SearchLL() functions would produce the following output:

Output:

```
Account 123 is not in the list.
Balance of 1 is 9.99
Balance of 2 is 8.88
Balance of 3 is 7.77
```

2.5 - Deleting Nodes from a Linked-list

Removing a node implies in search for it first. After finding the node with the required key, then one can delete the node. However, deleting a node affects the pointer `next` of the previous node. The pointer `next` of the previous node has to be updated in order to keep the linked-list in a consistent state. So one needs to keep track of *two* nodes to delete one node. We can create a second temporary pointer called `previous` to use in the function. The following algorithm (algorithm 4) summarises the discussion above.

A possible implementation in C is in listing 2.9.

This implementation works fine for most linked-lists. However, note that if the node to be deleted is the first one, then the function will fail (the program crashes). The reason for that is the fact that in this state the `previous` pointer is `NULL`, and therefore the statement with `previous->next` will cause a segmentation fault.

The other situation in which this function fails is when the key is not found in the linked-list. Note that in this case at the end of the while loop the node `current` is also `NULL`, so the program might crash. Fixing these problems is left as an exercise (see exercises 4 and 5 at the end of the chapter).

As an example, after adding some nodes to a linked-list we remove the node with accnumber=2. The listing 2.10 and figure 2.5 show the results.

Algorithm 4 Delete a key in a linked-list

```
1:  function RemoveNode(key)
2:      declare two new Node called current and previous
3:      current = listpointer
4:      previous = NULL
5:      while (current is not NULL) do
6:          if (key is equal to accnumber of current) then
7:              break
8:          end if
9:          previous = current
10:         current = current− >next
11:     end while
12:     previous− >next = current− >next
13:     delete node pointed by current
14: end function
```

Listing 2.9: The Remove node function

```
1  void Remove(Node * & listpointer, int x) {//not all cases
       considered
2    Node *current, *prev; //why do we need 2 pointers?
3    current = listpointer;
4    prev = NULL;
5    while (current != NULL) {
6      if (current->accnumber == x) { break; }
7      prev = current;
8      current = current->next;
9    }
10   prev->next = current->next;
11   }
12   delete current;
13 }
```

Listing 2.10: main() function using Remove()

```
1  int main() {
2   Node A = NULL;
3   AddNode(A, 1, 9.99);
4   AddNode(A, 2, 8.88);
5   AddNode(A, 3, 7.77);
6   AddNode(A, 4, 6.66);
7   AddNode(A, 5, 5.55);
8   Search(A,2);
9   Remove(A,2);
10  Search(A,2);
11 }
```

And the output is:

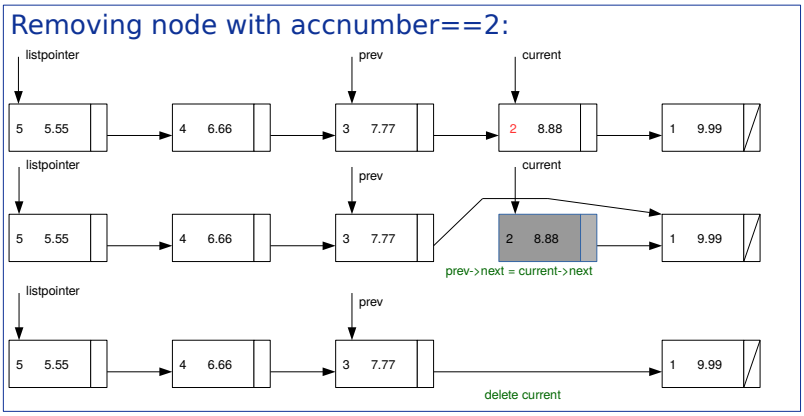


Figure 2.5: Removing nodes by key

```
Balance of 2 is 8.88
Account 2 is not in the list.
```

2.6 - Other functions for Linked-lists

2.6.1 - Concatenate two different linked-lists

Sometimes we want to join or concatenate two existing linked-lists. This is easily achieved by modifying the `NULL` pointer of the last element of the first list to point to the head of the second linked-list. In order to do that, one needs to find the last element of the first linked-list by traversing the entire list in a similar way to what we have done when inserting node to the end of the list.

Listing 2.11 shows a simple implementation and figure 2.6 shows an example.

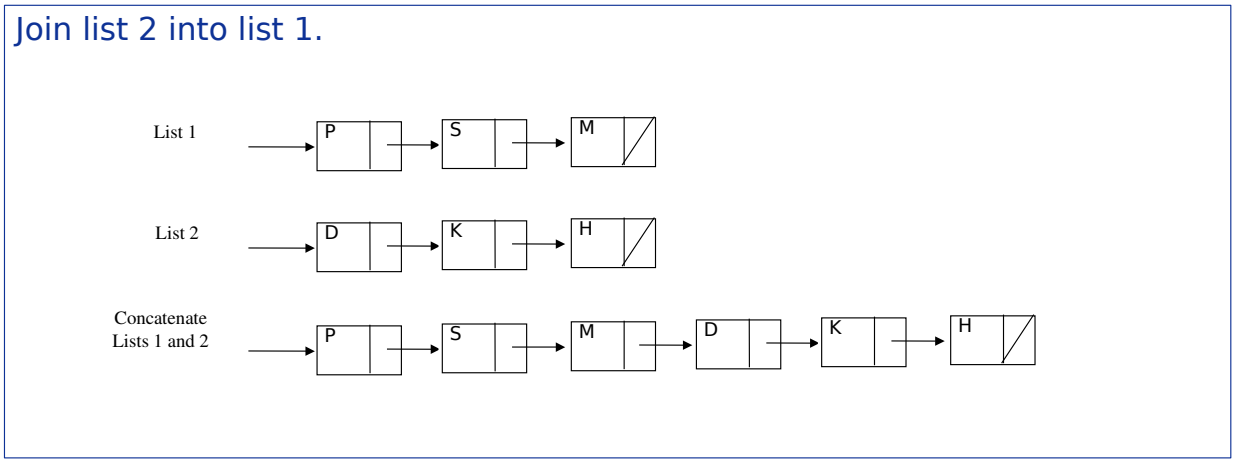


Figure 2.6: Joining two linked-lists

An example of a main() using the concatenate function is shown in listing 2.12, and the output is shown after that.

Listing 2.11: Concatenate two linked-lists

```
void Concatenate(Node * &listpointer1, Node * listpointer2) {
        Node *current, *prev;
        current = listpointer1;
        prev = NULL;
        while (current != NULL) {
                prev = current;
                current = current->next;
        }
        if (prev == NULL) {    //in this case listpointer1 is empty
                printf("list1 was empty, join anyway\n");
                listpointer1 = listpointer2;
        } else {   //join lists
                printf("join lists\n");
                prev->next=listpointer2;
        }
}
```

Listing 2.12: main() example of joining two linked-lists

```
Node *A=NULL;
Node *B=NULL;
int main() {
  AddNode(A, 1, 9.99);
  AddNode(A, 2, 8.88);
  AddNode(A, 3, 7.77);
  AddNode(B, 4, 6.66);
  AddNode(B, 5, 5.55);
  AddNode(B, 6, 4.44);
  PrintLinkedList(A);
  PrintLinkedList(B);
  Concatenate(A,B);
  PrintLinkedList(A);
}
```

Output:

```
Account 3 balance is 7.77
Account 2 balance is 8.88
Account 1 balance is 9.99
End of the list.
Account 6 balance is 4.44
Account 5 balance is 5.55
Account 4 balance is 6.66
End of the list.
join lists
Account 3 balance is 7.77
Account 2 balance is 8.88
Account 1 balance is 9.99
Account 6 balance is 4.44
Account 5 balance is 5.55
Account 4 balance is 6.66
End of the list.
```

2.6.2 - Reversing the order of the keys

Often the user wants to reverse the order of the keys in a linked-list. We will present three different methods to show that despite achieving the same results, some methods are better than others:

1. **Create** a new linked-list and copy all the elements using insertion to the front and then delete the old linked-list.

2. Scan the linked-list once to count the number of nodes. **Swap** the contents of the first and the last nodes, then swap the contents of the second with the previous to the last node and so on until it reaches the middle node.

3. **Change** pointers only in such a way every node points to their predecessors, changing the head to the last node and the old head points to `NULL`.

2.6.3 - Method 1

Method 1 is simple to understand and it is a good exercise of deleting an entire linked-list. Listing 2.13 shows an implementation of method 1 reversing function and listing 2.14 shows an example of `main()` using the function.

If the programmer forgets to delete the old linked-lists, then we have a situation called *leaking memory* problem. Running the function `ReverseLL1()` several times will cause more and more nodes to be dangling in memory without any pointers to access them. Eventually the operating system may no longer has enough memory to keep the program running. In order to delete an entire linked list properly requires to delete each node individually. Deleting only the `listpointer` will not have any effect on the remaining nodes. The other issue with method 1 is that every node has to be allocated again

and a deep copy of the contents have to be made, so the runtime can be very slow for large linked-lists.

Listing 2.13: Reverse linked-lists method 1

```
void ReverseLL1(Node * &listpointer) {
  Node* R=NULL;//reversed list
  Node* current=listpointer;
  while(current!=NULL){
    AddNode(R,current->accnumber, current->balance);
    current=current->next;
  }
  delete-linked-list(listpointer); //avoids memory leaking
  listpointer=R; //copy reversed list back to the listpointer
}
```

Listing 2.14: main() example of method 1

```
Node *A=NULL;
int main() {
  AddNode(A, 1, 9.99);
  AddNode(A, 2, 8.88);
  AddNode(A, 3, 7.77);
  AddNode(A, 4, 6.66);
  AddNode(A, 5, 5.55);
  PrintLinkedList(A);
  ReverseLL1(A);
  PrintLinkedList(A);
}
```

The output of listing 2.14 is:

```
Account 5 balance is 5.55
Account 4 balance is 6.66
Account 3 balance is 7.77
Account 2 balance is 8.88
Account 1 balance is 9.99
End of the list.
Account 1 balance is 9.99
Account 2 balance is 8.88
Account 3 balance is 7.77
Account 4 balance is 6.66
Account 5 balance is 5.55
End of the list.
```

2.6.4 - Method 2

Method 2 is a little bit more cumbersome. It is also a good exercise to keep track of different positions within the linked-list, but this method is very inefficient unless the linked-list is a doubly linked-list and can

be traversed backwards (see section 2.7.2). The way it is presented here requires to traverse large portions of the linked-list repeatedly, making it slow for large linked-lists. It also requires a hard copy of the contents from one node to the other. Figure 2.7 shows the main idea behind the algorithm that needs to be implemented.

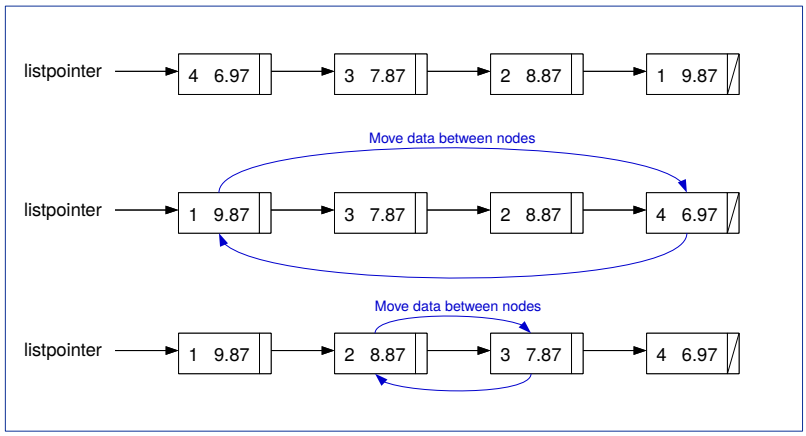


Figure 2.7: Reverse a linked-list, method 2. This method is less wasteful than method 1, but it still requires to move data around the nodes.

Listing 2.15 shows an implementation of method 2 reversing function.

Listing 2.15: Reverse linked-lists method 2

```
1  void ReverseLL2(Node * listpointer) {
2    Node *current, *temp, *temp2;
3    current = listpointer;    int numbelements=0;
4    while (true) { //scan once to count
5      if (current == NULL) { break; }
6      current = current->next;
7      Numbelements++;//count number of nodes
8    }
9    if(numbelements!=0){
10     for(int count=0;count<numbelements/2;count++){
11         temp=SearchByPosition(listpointer, count);
12         temp2=SearchByPosition(listpointer, numbelements-1-count);
13         //swap values inside the nodes:
14         int accnumber_temp=temp->accnumber;
15         float balance_temp=temp->balance;
16         temp->accnumber = temp2->accnumber;
17         temp->balance = temp2->balance;
18         temp2->accnumber = accnumber_temp;
19         temp2->balance = balance_temp;
20     }
21     printf("the list is reversed\n");
22     return;
23   }
24 }
```

Note that the function `SearchByPosition()` needs to be implemented separately. This function returns a pointer of type Node, i.e., the location of the node that is in a certain position within the linked-list. This function can have a negative impact on the performance of the reversal. Every time it is called, it requests for a traversal from the head. The listing of the function is shown in listing 2.16. Two arguments have to be passed, the `listpointer` and the `position` of the node within the linked-list.

Note also that we did not use the `&` in the argument of function `ReverseLL2()`. The reason we did not need to use it is that the `listpointer` address remains unchanged during the reversal (this was not the case in method 1).

Listing 2.16: Search by position of the node for reverse method 2

```
1  Node * SearchByPosition(Node *listpointer, int x) {
2    Node *current;
3    current = listpointer;
4    int pos=0;
5    while (current!=NULL) {
6      if (pos == x) {
7        return current;
8      }
9      current = current->next;
10     pos++;
11   }
12 }
```

2.6.5 - Method 3

Finally method 3 is the best one in terms of performance. It only requires *pointers* to be changed. There is no need for allocation of new nodes or deletion of old ones, only changes in the way the nodes point to each other. To understand method 3 it is better to start with algorithm 5.

Algorithm 5 uses three pointers. One pointer (**`temp2`**) holds the position of the original head. Other two pointers (**`temp1`** and **`temp3`**) are used to traverse the list and to change the **`next`** pointer of each element of the linked-list. After traversing the linked-list, both pointers **`temp1`** and **`temp3`** are NULL. All the **`next`** pointers of all elements are reversed, except the one that belongs to the old head, where **`temp2`** points to. The **`next`** pointer of **`temp2`** needs to change to NULL, as it became the last element of the linked-list. This last change is achieved after the **`while`** loop.

A simple implementation of Method 3 can be seen in listing 2.17.

An example of Algorithm 5 running with a small linked-list is shown in figure 2.8.

Algorithm 5 Reverse a linked-list (Method 3)

```
1:  function ReverseLL3
2:      declare three pointers (Node) temp1, temp2 and temp3
3:      temp2 = listpointer                  ▷ holds the old head of the linked-list
4:      temp3 = listpointer− >next           ▷ points to the second node
5:      while (temp3 is not NULL) do
6:          temp1 = temp3− >next
7:          temp3− >next = listpointer       ▷ start reversing the pointers
8:          listpointer = temp3
9:          temp3 = temp1                    ▷ move temp3 forward
10:     end while
11:     temp2− >next = NULL      ▷ the old head turns into the tail of the linked-list
12: end function
```

Listing 2.17: Reverse linked-lists method 3

```
1  void ReverseLL3(Node * &listpointer) {
2    Node *temp1, *temp2, *temp3;
3    temp2=listpointer;
4    temp3=listpointer->next;
5    while(temp3!=NULL){
6      temp1=temp3->next;
7      temp3->next=listpointer;
8      listpointer=temp3;
9      temp3=temp1;
10   }
11   temp2->next=NULL;
12   printf("the list is reversed\n");
13   return;
14 }
```

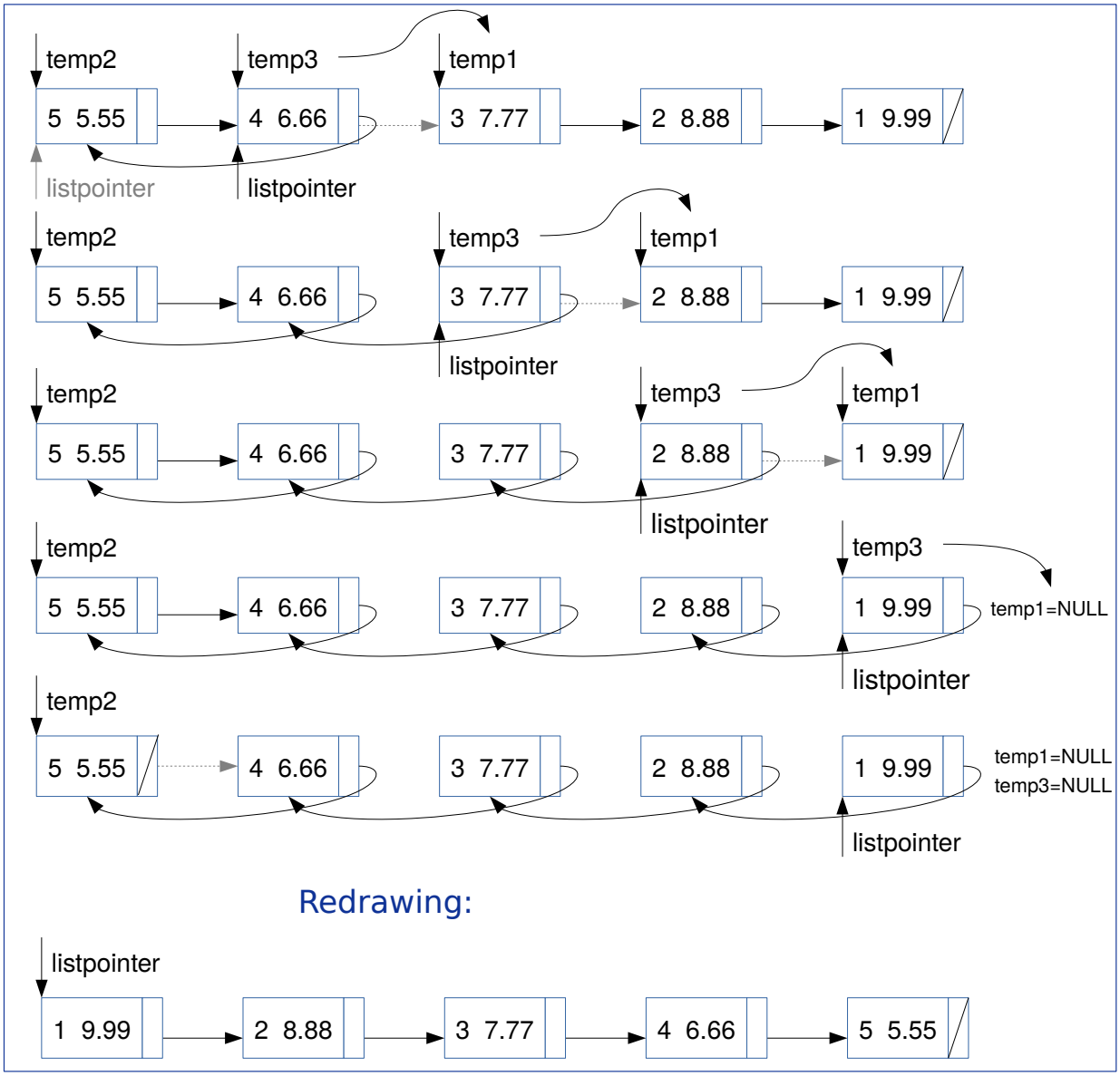


Figure 2.8: Reverse a linked-list, method 3. This is the fastest method, as it does not require moving data around nor allocating and deallocating nodes.

2.6.6 - Split a linked-list into two different linked-lists

To split a linked-list, firstly we need to locate the node where the split has to occur. Then we use a new node pointer to point to the `next` node, so this becomes the head of the new linked-list. Finally, the `next` pointer of the splitting node becomes `NULL`. Listing 2.18 presents the C implementation, and figure 2.9 shows an example.

Listing 2.18: Split a linked-list into two

```
1  bool Split(Node *& listpointer, Node *& listpointer2, int n){
2   if(listpointer2!=NULL){
3     printf("Error: the list should be empty\n");
4     return false;
5   }
6   Node * temp=SearchByPosition(listpointer,n);
7   listpointer2=temp->next;
8   temp->next=NULL;
9   return true;
10 }
```

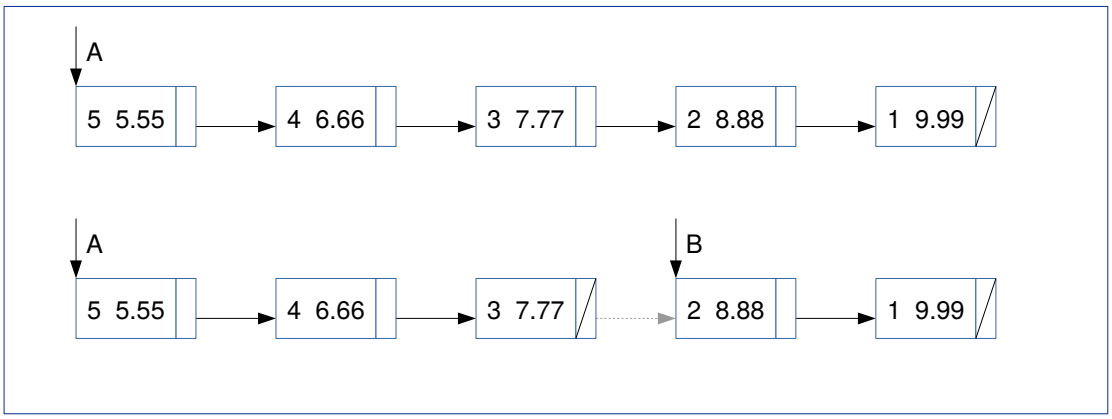


Figure 2.9: Splitting a linked-list.

2.7 - Other types of linked-lists

This section describes two important variations on linked-lists: circular linked-lists and doubly linked-lists.

2.7.1 - Circular Linked-list

Sometimes it is convenient to set the end (tail) of a linked-list to something different than `NULL`. This is the case of circular linked-lists, where the tail (the last node) points to the head instead. It is easy to modify the insertion functions to cater for that. Figure 2.10 shows an example.

When trying to print using listing 2.19, there is the issue of an infinite loop, as the stopper is no longer the `NULL` pointer. One can easily fix that by using a different condition for the loop or the break.

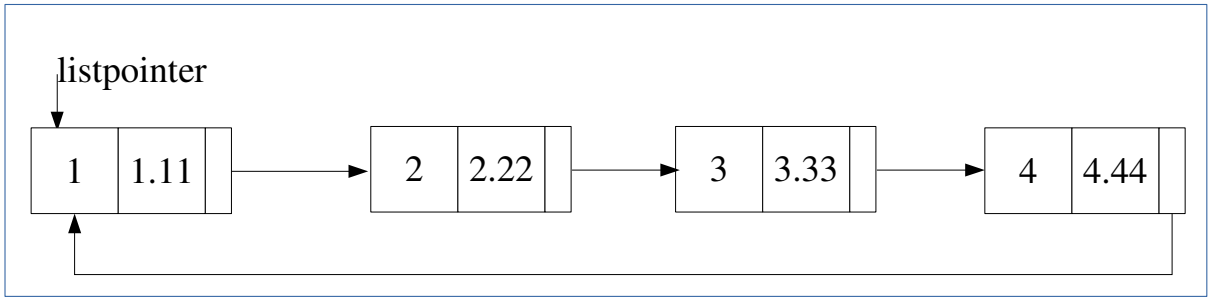


Figure 2.10: Circular linked-list.

Listing 2.19: Printing a circular linked-list

```
1  void CLL_PrintLinkedList(Node *listpointer) {
2    Node *current;
3    current = listpointer;
4    int element=1;
5    while (current->next != listpointer) {
6      printf("Element %d: Balance of acc %i is %1.2f\n",
7      element, current->accnumber, current->balance);
8      current = current->next;
9      element++;
10   }
11   //print last element
12   printf("Element %d: Balance of acc %i is %1.2f\n",
13   element, current->accnumber, current->balance);
14   printf("End of the list.\n");
15 }
```

The condition of a while loop that traverses the entire circular linked-list was changed from `(current != NULL)` to `(current->next != listpointer)`. This avoids the infinite loop.

We did not implement a function to insert or delete nodes from a circular linked-list. This is left as an exercise (see exercises 8 and 9 at the end of this chapter).

2.7.2 - Doubly linked-lists

Another important type of linked-list is one that has pointers pointing to the previous node as well as the next node. Schematically the double linked-list looks like the linked-list in figure 2.11.

A huge advantage of doubly linked-lists is the ability of traversing the list in both directions, even though it usually requires an extra pointer to the tail to be effective. However, this also costs extra space within the nodes themselves, as another pointer has to be declared in the `struct`.

The node declaration is found in listing 2.20. We also had to declare two pointers, one to the *head* and another to the *tail* of the linked-list. The tail pointer is important to give access to the tail without having

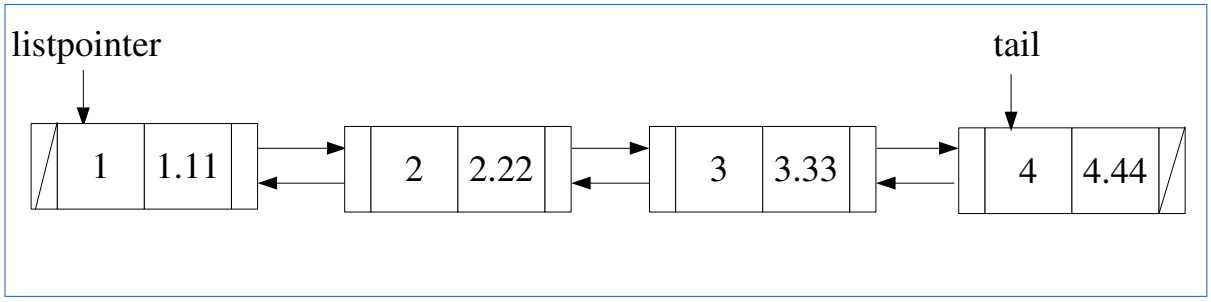


Figure 2.11: Doubly linked-list.

to traverse the whole linked-list first.[1]

[1] There is a better way of doing that when using C++ classes. We will implement that when we study Queues.

Listing 2.20: Doubly linked-list node

```
1  struct Node {  //declaration of a doubly linked-list
2    char key;
3    Node *next;
4    Node *previous;
5  };
6  ...
7  Node *LL;
8  Node *Tail_LL;
```

Any functions related to inserting and deleting nodes need to be modified to cater to the doubly linked-list. Listing 2.21 shows an implementation to the insertion at the front of the linked-list. Listing 2.22 shows the insertion after the tail for the same linked-list. Note that both the front (`listpointer`) and the tail (`tailpointer`) have to be passed as arguments because when the list is empty or has one element the front and the tail have to be modified. When the list is initially empty or has one node, inserting nodes will affect both `listpointer` and `tailpointer`, and therefore both arguments in the functions use &.

Listing 2.21: Doubly linked-list insert to front

```
1  void AddNode(Node *& listpointer, Node *& tailpointer, char a) {
2    Node *temp;
3    temp = new Node;
4    temp->key = a;
5    temp->next = listpointer;
6    //we need to take care of the previous pointers
7    temp->previous=NULL;
8    //was the list previously empty?
9    if(listpointer != NULL) listpointer->previous=temp;
10   listpointer = temp;
11   //if the list was empty, listpointer=tailpointer=temp
12   if(tailpointer = = NULL) tailpointer = temp;
13 }
```

Listing 2.22: Doubly linked-list insert after the tail

```
1  void AddNode_Tail(Node *& listpointer,Node *& tailpointer,char a) {
2    Node *temp;
3    temp = new Node;
4    temp->key = a;
5    temp->previous = tailpointer;
6    temp->next = NULL;
7    //was the list empty?
8    if(tailpointer != NULL ) tailpointer->next = temp;
9    tailpointer = temp; // now tail points to the new element
10   //if the list was empty, listpointer=tailpointer=temp
11   if (listpointer = = NULL) listpointer = temp;
12 }
```

Using AddNode() and AddNode_Tail()

```
int main() {
 Node *LL = NULL;
 Node *Tail_LL = NULL;
 AddNode(LL, Tail_LL, 'A');
 AddNode(LL, Tail_LL, 'B');
 AddNode(LL, Tail_LL, 'C');
 AddNode_Tail(LL, Tail_LL, 'D');
 AddNode_Tail(LL, Tail_LL, 'E');
}
```

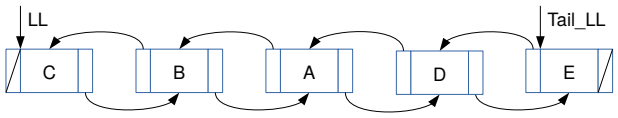


Figure 2.12: Doubly linked-list with nodes inserted at the head and the tail.

Now there is the issue of traversing the doubly linked-list to print all the elements in a certain order. The normal order, from head to tail, will be exactly like a normal linked-list. But we can also print the list in reverse by starting from the tail and following the `previous` pointers. This is shown in listing 2.23.

Listing 2.23: printing the doubly linked-list in reverse order

```
1  void PrintLinkedList_Backwards(Node *tailpointer) {
2    printf("Printing list in reverse order... \n");
3    Node *current;
4    current = tailpointer;
5    while (true) {
6      if (current == NULL) { break; }
7      printf("Key %c \n", current->key);
8      current = current->previous;
9    }
10   printf("End of the printout.\n");
11 }
```

The output of printing the doubly linked-list of figure 2.12 would be;

```
Printing list in reverse order...
Key E
Key D
Key A
Key B
Key C
End of the printout.
```

2.8 - Exercises

1. Copy the programs for inserting nodes at the head and at the tail. Also copy the `PrintLL()` into the same file, making one coherent program. In the `main()`, use the following sequence:

Listing 2.24: Exercise 1

```
1  AddNode2(A,1,1.11);
2  AddNode(A,2,2.22);
3  AddNode2(A,3,3.33);
4  AddNode(A,4,4.44);
5  AddNode2(A,5,5.55);
6  AddNode(A,6,6.66);
7  AddNode2(A,7,7.77);
8  AddNode(A,8,8.88);
9  AddNode2(A,9,9.99);
10 PrintLL(A);
```

2. Draw the linked-list manually and check in which sequence the nodes are arranged. Does it comply with what was printed?

3. Implement a function that returns the integer key of the N^{th} element of a linked-list. If the element does not exist, return -1;

4. Fix the `Remove()` (listing 2.9) function to work with the deletion of the first node. Fix it again for the case in which the `key` does not exist in the linked-list.

5. Modify the `Remove()` function to delete the N^{th} element. Remember to include the special cases for the removal, i.e., first node and the fact that the N^{th} node may not exist.

6. Write a function to delete an entire linked-list (to be used with method 1 of the `reverse()` functions.). The prototype of the function should be `delete-linked-list(Node *& listpointer)`. At the end of the function, `listpointer` should be `NULL` and every node should have been deallocated (use `free()` or `delete`).

7. Write a program to insert a node in a linked-list *after* a certain key (i.e., search for the key and insert the new node after it).

8. Write a program to insert nodes to the head of a circular linked-list. Draw the schematic first to figure out what to change in the AddNode function.

9. Write a program to insert nodes to the middle (after a certain key) of a circular linked-list. Draw the schematic first to figure out what to change in the AddNode function.

10. Using a doubly linked-list, write a printing function that traverses the list from the head to tail and then again from the tail to the head (this should be done within the same function).

3 Stacks

Our next abstract data type (ADT) is called *Stack*. A stack is a pile of things, where only the element on the top of the pile is (usually) accessible by the programmer. If the programmer wants to access some data below the top element, then elements need to be deleted before. Figure 3.1 shows different representations of a Stack. The first two represent a schematic view of a Stack. The two stacks on the right side of the figure 3.1 represent a Stack implemented as an array and as a linked-list respectively.

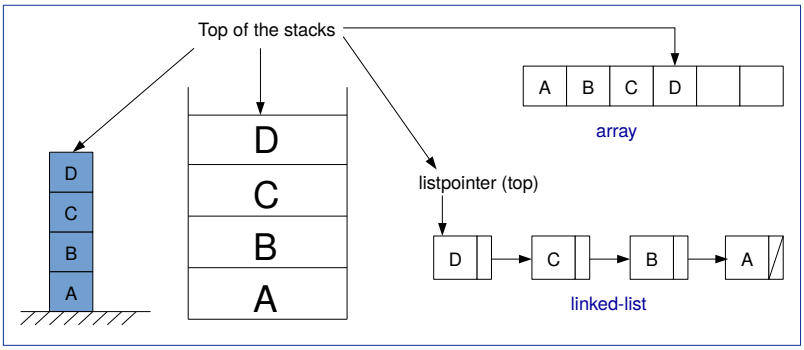


Figure 3.1: Alternative representations of a Stack.

Stacks can have many operations, but there are 4 main operations that are essential (see figure 3.2). These are:

1. Push - place (insert) a new item on the top of the stack

2. Pop - remove (delete) the item on the top of the stack

3. Top - retrieve (look at) the top item of the stack

4. IsEmpty - returns true or false depending whether the stack is empty or not

A Stack is constructed in such a way that the last item in is the first item out (LIFO, last in first out).

From this point of the course we are going to start using simple C++ classes to implement ADTs. For Stacks, two different implementations will be used, one with `arrays` (static) and the other with *linked-lists* (dynamic allocation). Before we can discuss the implementation details we need to explain some basic concepts of C++ classes.

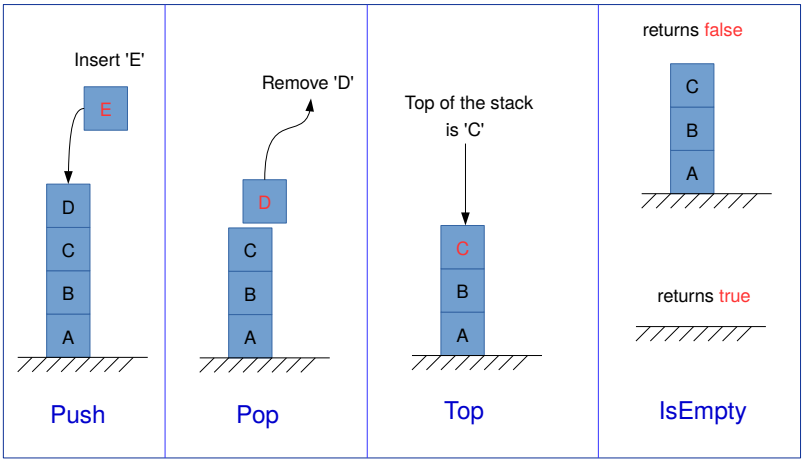


Figure 3.2: The four standard operations for a Stack.

3.0.1 - C++ classes

In this section we briefly introduce the concept of classes and discuss basic topics about the *object oriented* approach (OO). Classes are template definitions that include variables and methods (functions) that can access and modify the variables. An instance of a class is called an *object*.

Classes can be considered templates for *objects*. Objects usually contain *members*. Members are (in a simplistic way) either *data* (properties that can be represented by variables) or *methods* ("functions" that modify or inquire about the object's properties). In the same way that one can declare a `struct` and then declare several instances of that `struct`, classes can also have many instances. Often the instance of a class is referred to as an object.

In pure C *functions* play the role of modifying or accessing data, either from a global scope or via parameters passed by value or by reference. In C++ object-oriented approach, the "functions" became *methods*. Due to the fact that the data is hidden (in the `private` scope), the only way to access them is via the class's methods. One could still write a function for the implementation of a certain algorithm, but the function would have to rely on the methods to access the object's data. More examples later on will make this idea more clear.

In a C++ class, there are different scopes for variables and methods. They can be `private` or `public`. Anything in the `private` scope can only be modified or accessed via a `public` method. Although classes in C++ can have other scopes and can have a complicated structure regarding the variables and methods, we are going to use `private` data (for variables and containers) and `public` methods ("functions") to implement all the ADTs in this study guide. [1]

An important novelty with C++ classes is the way an object can be created. Remember how allocations can be done with an integer? The same can be done with a class. Declaring a class immediately reserves spaces for the data members of an object. Declaring a pointer of that class type only reserves space to the address, and the programmer

[1] A complete object oriented approach programming course is available as paper 159.234.

has later to call `new` to allocate space to it. Therefore an object can be represented by a variable of type *class* (in this section, a Stack) or by a pointer of type *class*. Due to the more complex nature of the classes, variables cannot be initialised in the declaration as we would normally do with variables. Instead, when the object is instantiated the compiler relies on one special method to do that: a **constructor**.

3.0.2 - Constructors and Destructors

A *constructor* is a special method that is called upon automatically when a new object is created. This allows the programmer to code all initialisation needed for the object. E.g., if a variable `int a` needs to be initialised to zero, this will be done inside the constructor method. The constructor method has the same name as the class itself, and it is not usually called explicitly in the main function (there are some circumstances in which it is).

The *destructor* is the other special method that needs to be considered when coding with classes. The destructor is needed when objects are deleted from memory, so their data members need to be properly deallocated. You may remember the example of a linked-list that was copied in reverse and had to be deleted. Just deleting the head of the linked-list did not suffice, as the nodes are dangling independently in memory. A destructor would have to be coded to delete all elements, one by one. If an object (of a certain `class`) is to be deallocated, the destructor method is called automatically (the destructor is *never* called explicitly by the programmer). The destructor has the same name of the class with the symbol ~ in front.

Listing 3.1: Constructor and Destructor example

```
1  void Function1(){
2    Stack A; //constructor is used here for A, static but inside a function
3    A.Push(4.8);
4  }//destructor for A is used when function ends
5
6  Stack S;//constructor is used here for S, static memory allocation
7  Stack *P;
8  int main() {
9    S.Push(6.2);
10   P = new Stack; //constructor is used here for P, dynamic memory allocation
11   P->Push(6.2);
12   Function1();
13   delete P; //destructor is used here for P
14 }
```

If constructors and destructors are not used explicitly, when are they used at all? It depends on how the object (the instance of the class) is being created. The example in listing 3.1 shows three situations. In statement 6 a static instance `S` was declared, so the constructor is used when the instance is declared, and the destructor

is never used. In statement 7 a pointer of type Stack is used (`P`), so one needs to allocate space first by using `new`. In this case, the constructor is used when `new` is called (statement 10), and the destructor is used when `delete` is called (statement 13). Finally, if a temporary instance of the class is used inside a function or method, the constructor is used when the instance appears (declaration inside the function in statement 2 called `A`) and the destructor is used when the function ends and returns control to `main`.

There are a few more specific syntax details that are better explained with an example. We start by implementing a Stack class using *arrays*.

3.1 - Stacks implemented with Arrays

For the first implementation of a Stack, it should be clear by now that the four standard operations (Push, Pop, Top and IsEmpty) will become methods of the class itself. As every class also has to declare a constructor and a destructor, our class will have six public methods. More methods can be added later, but these six methods are part of the minimum implementation.

The other members of the class are data variables. One of them is the array that will store the data for the Stack. We need one extra variable to indicate what is the top element of the Stack at a certain moment, so we can both insert and delete other elements and mark where the top element should be. For this purpose we are going to use a simple integer variable called `index`. The class can be implemented as per listing 3.2.

Note that the class declaration does not implement the methods themselves yet, just their prototypes. Note also that the constructor and the destructor are void methods, i.e., no parameters and no returning values. However, the other methods may have parameters or returning values.

Listing 3.2: The `Stack` class

```
1  class Stack {
2    private: //this holds the data
3      float data[10];
4      int index;
5    public:  //this holds the declaration of the methods
6      Stack(); //the constructor
7      ~Stack();  //the destructor
8      void Push(float newthing);
9      void Pop();
10     float Top();
11     bool isEmpty();
12 };
```

In C++, the implementation of methods of a class should be preceded by the name of the class and the symbols `::`. The four methods

are shown in listing 3.3. This listing should be copied to the same file after the declaration of the class in listing 3.2.

The constructor has a single statement, the initialisation `index=-1;`. This indicates an empty stack at the start (no valid number in the array) For now, we are going to use an empty destructor just to please the compiler, and because the two data members of the class are static. Later we will show a case where the destructor is useful.

Listing 3.3: The `Stack` methods

```
1  Stack::Stack() { // constructor
2    index = -1; //the only initialisation needed
3  }
4
5  Stack::~Stack() {
6    // an empty destructor
7  }
8
9  void Stack::Push(float newthing) { //new thing on top of the stack
10   index++;  //Warning: watch for overflow
11   data[index] = newthing;
12 }
13
14 void Stack::Pop() { //removes the top item from the stack
15   if (index > -1) { index--; } //Takes care of underflow
16 }
17
18 float Stack::Top() { //return value of the top item in the stack
19   return data[index]; //Warning: what if stack is empty?
20 }
21
22 bool Stack::isEmpty() { // return true if the stack is empty
23   if (index < 0) { return true; }
24     return false;
25 }
```

An important point to pay attention to: when methods are called, the name of the Stack (the *instance* of the class that are declared) is not passed as an argument as we do with C functions and linked-lists. Instead, we use a similar syntax when accessing members of a `struct` in C. So when trying to push 4.4 into the Stack `A`, we should code:

`A.Push(4.4);` (see listing 3.4)

If the object (the instance of the class Stack) is a pointer to that object, e.g., after declaring `Stack *B = new Stack;`, then we should use;

`B->Push(4.4);`

Figure 3.3 represents the `main` function in listing 3.4. Here we represent the array inside the Stack horizontally, but one could also represent it vertically when drawing the Stack schematically. The important thing is to know where the top of the stack is, and in which

order the elements are. The variable `index` has to point to the *top* element.

Listing 3.4: main() example using the `Stack` methods

```
1  int main() {
2    Stack A;  //This is how to declare a new stacks
3    A.Push(1.1);
4    A.Push(2.2);
5    A.Push(3.3);
6    A.Pop();
7    if (A.isEmpty()) {
8      printf("Stack A is empty");
9    }
10   else {
11     float x = A.Top();
12     printf("Top item in A is %1.1f", x);
13   }
14 }
```

The output of listing 3.4 is: `Top item in A is 2.2`

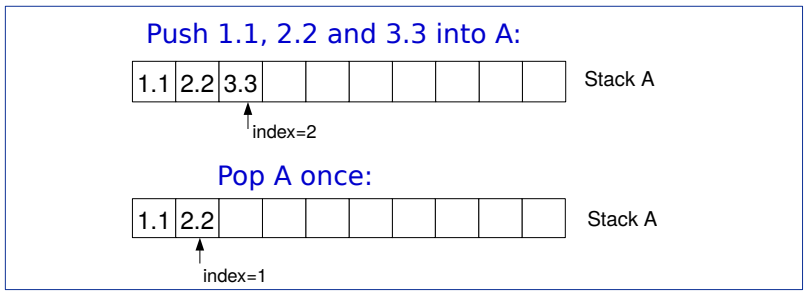


Figure 3.3: A Stack implemented as an array

Is it important to initialise the array inside the Stack? As the elements are always inserted in increasing order of the variable `index`), any array value for which the index is bigger than the variable `index` is invalid. The only valid values in the array are the ones smaller or equal the current `index`. In principle there is no need to initialise the array, as long as the array in the Stack is used within that context.

This implementation does not take care of possible overflow of the `index`. A simple `if` statement in the `Push()` method would take care of it.

Finally, note that the method `Pop()` does not underflow. Therefore, we could call it repeatedly without causing any harm to the Stack. On the other hand, what happens if an empty Stack is asked about its top element (call `Top()` when the Stack is empty)? That would also incur in a error, as the `index = -1`. This is the reason why every call of `Top()` should be wrapped by a call to the `isEmpty()` method, as shown in listing 3.4.

Another option would be to protect the `Top()` method from errors. This would imply in returning some integer value (this is what the

Listing 3.5: The `Stack` class with linked-list

```
struct Node{
  float data;
  Node* next;
}

class Stack {
  private: //this holds the data
    Node* listpointer;
  public:  //this holds the declaration of the methods
    Stack(); //the constructor
    ~Stack();  //the destructor
    void Push(float newthing);
    void Pop();
    float Top();
    bool isEmpty();
};
```

method has to return). What is an invalid integer in this case? It depends on the application. Suppose that negative integers should not appear in this Stack. In this case, one could modify `Top()` to cater for that, and there would be no risk of segmentation fault. However, the disadvantage to do that is that the meaning of the negative numbers could change with another application under a different context. On the other hand, using the method exactly how it is listed in listing 3.4 implies in making more calls to the `isEmpty()` method in the `main` function.

3.2 - Stacks implemented with Linked-lists

The next C++ implementation of Stacks uses a linked-list to store the stack's data (rather than an array). In order to represent a linked-list, a single pointer of type `Node` would suffice, as this is the head of the linked-list. In other words, the class only needs a single pointer to the head of the linked-list (`listpointer`). As long as only public methods can access `listpointer`, the linked list is protected from external functions. This is part of the data hiding strategy that we used before. The declaration of the class is in listing 3.5.

Note that the elements inside the Stack (inside the linked-list that is part of the class) depend completely on the definition of the `Node struct`. If instead of floats one wants integers, the Node has to change, as well as any methods that either receive a parameter or return a value. In the case of the Stack, methods `Push()` and `Top()` would have to be changed to reflect an `int` instead of a `float`.

The implementation of the methods now need to consider that the container is a linked-list. Should the elements in the linked-list be in

order of input or in reverse order of input? The reader should look again at figure 3.1. Note that the array and the linked-list examples have their elements in different order.

A convenient way of implementing this class is to consider that the **top** of the Stack is the **head** of the linked-list. This is so because it is easier to insert and remove the first node of a linked-list. This does not require any traversal before removing the top node.

The first method, `Push()`, should push a new value into the Stack (on top, or "before" the previous top of the Stack). This is equivalent to the *add node to front* that we have seen before. Note that there is no overflow, as the linked-list can grow dynamically.

The method `Pop()` should delete the top element, which can be done without traversing the linked-list. Note that there is no underflow, as the linked-list can shrink dynamically.

The method `Top()` should return the value inside the first element, also done without any traversal.

Finally, the `isEmpty()` method should make use of the linked-list properties to guess whether the Stack is empty or not. If `listpointer` is `NULL`, clearly there are no nodes. If `listpointer` is different than `NULL`, then there is at least one node (we do not need to know how many elements are in the Stack, just that it is not empty).

The constructor simply initialises the pointer `listpointer` to `NULL` to indicate an empty stack at the start. The destructor is again empty.

Listing 3.6 shows the implementation of the four methods.

Listing 3.6: The `Stack` methods using a linked-list

```
1  Stack::Stack() {// constructor
2    listpointer = NULL; //empty linked-list at the start
3  }
4
5
6  Stack::~Stack() { }// destructor
7
8
9  void Stack::Push(float newthing) {//newthing on top of the stack
10   Node *temp;
11   temp = new Node;
12   temp->data = newthing;
13   temp->next = listpointer;
14   listpointer = temp;
15 }
16
17 void Stack::Pop() {// remove the top item from the stack
18   Node *p;
19   p = listpointer;
20   if (listpointer != NULL) {
21     listpointer = listpointer->next;
22     delete p; //always delete a TEMPORARY variable
```

```
23    }
24  }
25
26  float Stack::Top() {//return value of the top item in the stack
27    return listpointer->data; //what if listpointer is NULL?
28  }
29
30  bool Stack::isEmpty() {// return true if the stack is empty
31    if (listpointer ==NULL ) {
32      return true;
33    }
34    return false;
35  }
```

There is one scenario that was not considered when implementing `Top()`. What happens if `listpointer` is `NULL` (i.e., the Stack is empty?). The method attempts to access the data from the `listpointer` (the head of the linked-list), but there is none. The executable will possibly get a segmentation fault. One could prevent this by asking if `listpointer` is `NULL` inside `Top()`, but then we would have the same dilemma we had before with the application context. If the Stack is empty, is a negative float a good response? If we keep the method as is, then `Top()` has to be wrapped up by the `isEmpty()` method every time it is called (see listing 3.4).

The same `main()` function used in listing 3.4 can be used with the new implementation of Stack without any modification. The result of running listing 3.4 is shown in figure 3.4.

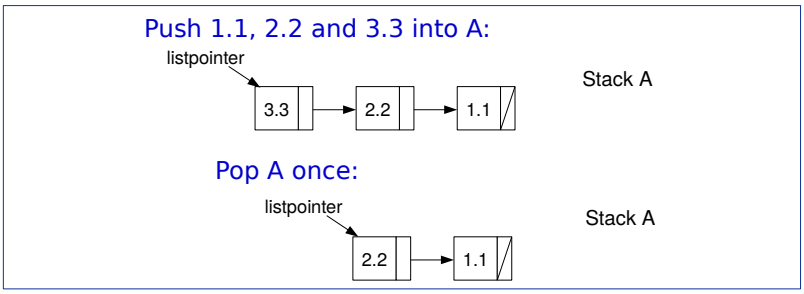


Figure 3.4: A Stack implemented as a linked-list. The first element is always the top of the stack. Therefore, we never need to resort to searching an element of the linked-list if we are using the 4 basic methods for a stack.

3.3 - Comparing Stacks: `A = B;` and `A == B;`

Remember the cases in C language when you had two instances of a certain struct? You should not compare nor copy them directly using pointers. Instead, one should copy or compare the *elements* inside the struct.

With classes the same thing happens. One cannot copy or compare the classes using = nor == directly. Without proper overloading, the meaning of = and == are to copy or compare addresses of the Stacks objects rather than their content. [2]

[2] Actually C++ allows programmers to do just that. This is called *operator overloading* (covered in 159234).

One solution would be to write methods that have loops that compare or copy all the elements inside the Stack (array or linked-list).

Listing 3.7: Methods Compare and Copy

```
1 Stack A, B;
2 ...
3 main(){
4   A = B;
5   if(A.Compare(B)) printf("they are identical");
6   else printf("they are different");
7   A.Copy(B);
8   if(A.Compare(B)) printf("now they are identical for
        sure");
9 }
```

We would have to have two methods, Copy() and Compare(). The programmer would have to decide which order the parameters go, either A is copied onto B or B is copied onto A. These implementations are left as exercises (see exercises 2 and 3).

3.4 - Using the Standard Template Library (STL)

The *Standard Template Library* (STL) is a software library for the C++ programming language available for free (it is part of the GNU GCC for example). It provides implementations of many of the ADTs that we are going to study in this course. The STL has four major components: containers, functional, algorithms and iterators. We are not going to cover all, but mostly show the equivalent containers that we can use. This makes the link between our own classes and the classes represented by one of the STL's containers.

The stack is implemented in the STL as a container and can be used by adding the line `#include <stack>` in the header.

The method names are slightly different than our implementations before. The four methods in STL are called `push`, `pop`, `top` and `empty` (note the lower case). For example, listing 3.4 could be rewritten using the stack from the STL as seen in listing 3.8.

Note that the codes are almost identical. One important difference is the first use of a template. The stack is declared with a specific type `float`, and the syntax to indicate that is `stack<float> A;`. This is somewhat different to the classes we showed before using our own implementations. In our previous classes the data type had to be changed in all the methods for it to work. With templates it suffices to indicate the type without actually changing anything in the methods themselves. The method implementation is hidden from the programmer, who can use any data type. We are going to see other examples of templates later, and see how to create our own template class.

The other important aspect to note is that when using the STL stack we have no idea how the implementation of the stack was carried out, i.e., using arrays or linked-lists or something else internally. However the programmer can freely use the STL implementation in the hope that it was implemented in the best way possible.

There are extra methods available for the STL stack that are also common with other containers. An important one is `size`, exemplified by statement 10 in listing 3.8.

Listing 3.8: main() example using the `Stack` from STL

```
1  #include <stdio.h>
2  #include <stack>
3  using namespace std;
4  int main() {
5    stack<float> A;
6    A.push(1.1);
7    A.push(2.2);
8    A.push(3.3);
9    A.pop();
10   printf("the current size of the stack is %ld \n",A.size());
11   if (A.empty()) {
12     printf("Stack A is empty");
13   }
14   else {
15     float x = A.top();
16     printf("Top item in A is %1.1f", x);
17   }
18 }
```

3.5 - Exercises

1. Modify the methods of the class Stack using an array to make sure that overflow and underflow errors do not happen.

2. Write an extra method for the class Stack called `Copy(Stack B)` that copies a Stack onto another. You need to traverse the entire linked-list and copy it. Be careful with the order of the elements.

3. Write an extra method `bool Compare(Stack B)` that traverses all the elements of both Stacks and compare them. If there is a difference, return false, otherwise return true.

4. Draw schematically the array that is part of the `Stack` class. The stack contains integers that are inserted/deleted using for the following sequence of statements:

Listing 3.9: main() function using a Stack

```
... //copy the Stack class here
Stack S;
main(){
  S.Push(1);
  S.Push(2);
  S.Pop();
  S.Push(3);
  if(!S.isEmpty()) printf("%d ",S.Top());
  S.Push(4);
  S.Push(5);
  S.Push(6);
  S.Push(7);
  S.Push(8);
  if(!S.isEmpty()) printf("%d ",S.Top());
  S.Pop();
  if(!S.isEmpty()) printf("%d ",S.Top());
  S.Push(9);
  S.Push(10);
  S.Push(11);
  S.Pop();
  S.Pop();
  S.Pop();
  if(!S.isEmpty()) printf("%d ",S.Top());
}
```

5. For the same sequence of statements above, draw the linked-list inside the Stack and indicate what is the top item.

6. For the same sequence of statements above, write the output of the printf calls.

7. Now suppose that the Stack shown in listing 3.9 is implemented as an array. For the same sequence of statements above, draw the array and indicate what is the top item.

8. Write a new method or function to the list of functions inside the Stack class. This is a Boolean function called `TwoorMore()` which returns true if there are two or more items in the stack, and returns false otherwise. Do not make use of any other methods in your function. Write two versions of the method, one for the case of an implementation using an array and the other for a linked-list.

9. Add a new method to the list of functions inside the Stack class. This is a void function called `PopTwo()`. which pops the top two items off the stack. This may be a useful function to use when evaluating Reverse Polish Expressions. How would the two items be returned from the function? Do not make use of any other methods in your function. Write two versions of the method, one for the case of an implementation using an array and the other for a linked-list.

4 Queues

Our next abstract data type (ADT) is called *Queue*. A queue works exactly like a real life queue. When you join a queue you join it at the end (unless you jump the queue, in which case you are going to be told off by someone who is already waiting in the queue). The first people to be served are the ones in front of the queue, so you need to wait your turn. This manner of treating the elements of an ADT is called First in, First Out (FIFO). This is opposite to the idea of the Stacks, which were LIFO.

Figure 4.1 gives an idea of the concept.

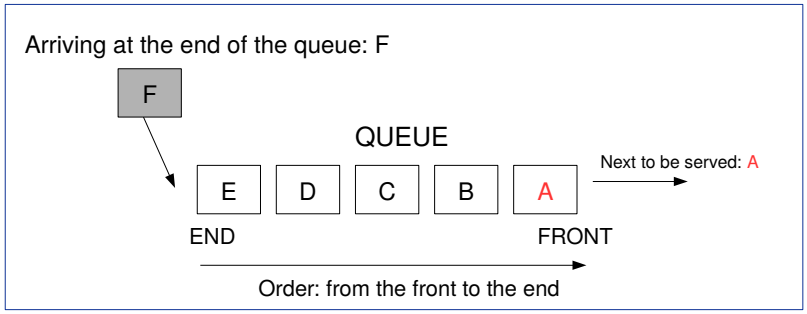


Figure 4.1: Representation of a queue. The order of the elements is not important, as both the tail and the front are marked.

Queues can have many operations, but there are 4 main operations that are essential. [1] These are:

1. Join - place (insert) a new item after the end of a queue
2. Leave - remove (delete) the item from the Front of a queue
3. Front - retrieve (look at) the Front item of the queue
4. IsEmpty - returns true or false depending whether the queue is empty or not

[1] The names of the operations are somewhat arbitrary. For example, in C++ STL (the Standard Template Library) the operation's name *Join* is usually replaced by `push()`.

The queue's elements have to be used and inserted in such a way that the first item to be inserted has to be used first (i.e., first in first out, or FIFO). Figure 4.2 shows the four operations schematically with an arbitrary queue. Note the positions of the front and rear. This is going to play an important role when implementing queues.

4.1 - Queues implemented with Arrays

For the first implementation of a queue we are going to implement the four standard operations using an array to hold the elements. As

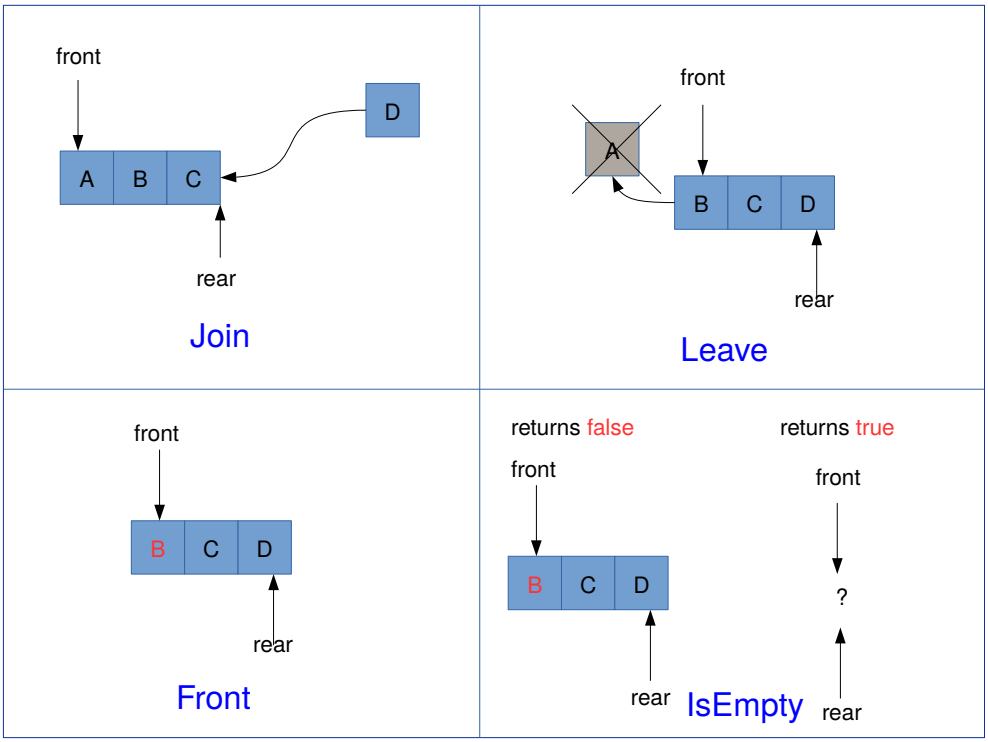


Figure 4.2: Basic operations of a queue.

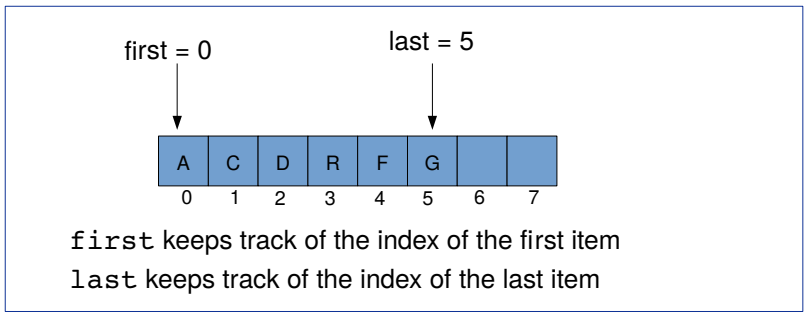


Figure 4.3: Queue implemented with an array.

we did before with Stacks, each operation will become a method of the class `Queue`.

We need to decide which end is going to be the front and which one the rear of the queue. It is easier to consider that the front is initially at the index 0, at the beginning of the array. Any elements added after that will have larger indices. We tend to draw arrays from left to right, so that is the way represented in the examples (see figure 4.3). Every time the method `Join()` is used, a new element enters the queue after the `last` index, and 1 is added to the variable `last`. Every time the method `Leave()` is used, then 1 is added to the variable `first`, essentially deleting the element in the front (note that one does not need to explicitly overwrite it with a zero).

As a queue grows and shrinks, it needs to change the front and the rear positions of the queue. In a queue implemented with an array, this can be achieved by adding two variables to the class `Queue` to mark where the ends are. In the class implementation example, these

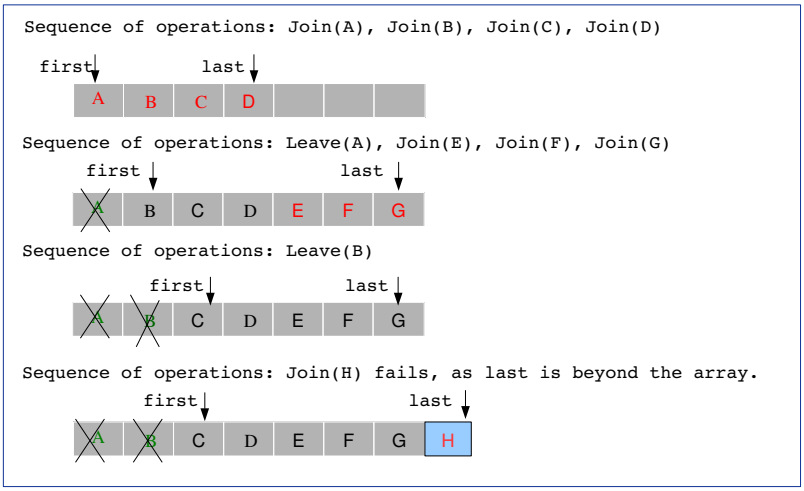


Figure 4.4: Queue implemented with an array: the *creeping problem*.

variables are `first` and `last` (see listing 4.1).

Listing 4.1: The `Queue` class using an array

```
1  const int Qmax = 10; //note the use of constants
2
3  class Queue {
4  private:
5    float data[Qmax]; //the array containing Stack elements
6    int first, last, count;
7  public:
8    Queue();
9    ~Queue();
10   void Join(float newthing);
11   void Leave();
12   float Front();
13   bool isEmpty();
14 };
```

Now an interesting problem remains to be solved. When elements are inserted and deleted from the queue, there could be a scenario where the array still has empty places to be filled up, but the variable `first` already moved away from the index 0. New elements cannot be inserted into the queue. This is known as the *creeping problem*. Figure 4.4 illustrates the problem.

How do we resolve this issue? We would like to use the array fully, until no spaces in the array are vacant. A simple solution is to use the array in a *circular* fashion. When the variable `last` reaches the end of the array, then the next attempt to insert an element using `Join()` inserts the new item into index 0, *as long as that position is vacant*. If it is not vacant, it means that the limit of the array (`Qmax`) was reached, and we can no longer insert elements in this queue. Figure 4.5 illustrates how to resolve the creeping problem in queues implemented with arrays.

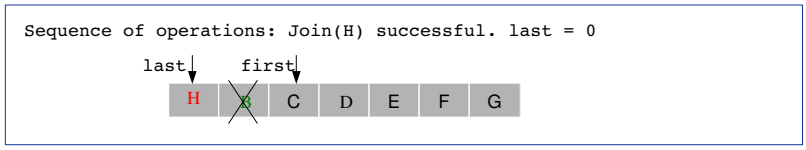


Figure 4.5: Queue as an array: the *creeping problem* solved using a circular array.

The methods for the queue using a circular array can be found in listing 4.2. Note that the comments in the code call attention to certain issues. The methods are not error proof. If one keeps inserting elements using `Join()`, old elements will be overwritten by new items. The modification of the methods to cater for this error is left as an exercise.

Listing 4.2: The `Queue` methods using an array

```
1  Queue::Queue() {// constructor
2    first = 0;  last = -1;  count = 0;
3  }
4
5  Queue::~Queue() {// empty destructor
6  }
7
8  void Queue::Join(float newthing) {// place the new thing at the rear of the
       queue
9    last++;
10   if (last >= Qmax) { last = 0; }
11   data[last] = newthing;
12   count++; //watch out for overflow!!
13 }
14
15 void Queue::Leave() {// front item is removed from the queue
16   first++;
17   if (first >= Qmax) { first = 0; }
18   count--; //watch out for underflow!!
19 }
20
21 float Queue::Front() {// return the value of the first item
22   return data[first];
23 } //what if the queue is empty?
24
25 bool Queue::isEmpty() {// returns true if the queue is empty
26   if (count == 0) { return true; }
27   return false;
28 }
```

To have an idea on how to use a queue in the main, refer to listing 4.3. Note that before using `Front()`, the program asks if the queue is empty or not using `IsEmpty()`. This avoids the problem of getting the wrong answer when querying the queue.

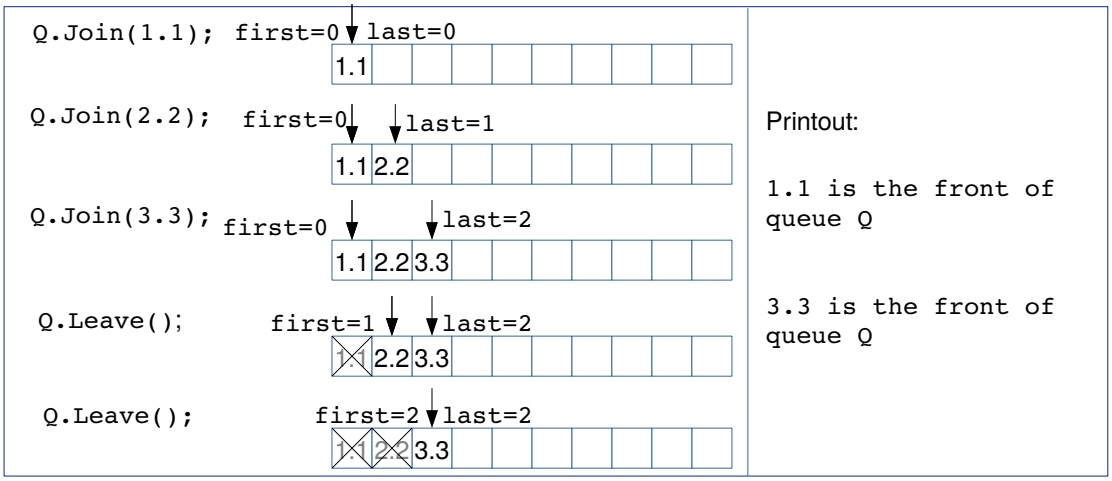


Figure 4.6: Queue implemented with an array running listing 4.3.

To illustrate the method usage, figure 4.6 shows a queue schematically when running listing 4.3

Listing 4.3: The `Queue` methods used in the `main`.

```
1  Queue Q;
2  int main() {
3    Q.Join(1.1);
4    Q.Join(2.2);
5    Q.Join(3.3);
6    if (Q.isEmpty()) printf("Queue Q is empty\n");
7    else printf("%f is the front of queue Q\n",Q.Front());
8    Q.Leave();  //Note: delete without knowing content
9    Q.Leave();
10   if(Q.isEmpty()==false) printf("%f is the front of queue Q
         \n", Q.Front());
11 }
```

Small queues that do not need to grow over a certain limit and need good runtime performance can be implemented with arrays. For larger dynamic queues it is better to use linked-lists, an approach described in the next section.

4.2 - Queues implemented with Linked-lists

Linked-lists can also be used as containers of elements of the queue. When we used linked-lists to implement stacks, we didn't have to look at any other element but the first one (the head of the linked-list). This is because the stacks are LIFO. With queues however, we need the ability to insert at the end of the queue and to delete elements from the front of the queue. The question is which is the best arrangement: should the front of the queue be head of the linked-list or the tail of the linked-list?

The answer may look obvious, but it is not. If the front of the queue coincides with the head of the linked-list, then it is easy to

delete elements when the method `Leave()` is called. The first element is deleted, and the new front of the queue is again the head of the linked-list. On the other hand if the front was on the tail of the linked-list, then deleting the last element would imply modifying the previous element (modifying the `next` pointer of the previous element) for which the only way to access would be to traverse the entire linked-list. As we saw in the linked-list chapter, traversing the linked-list can be onerous to the performance.

In this implementation we also need to know the front and the tail of the queue. We can add two variables that we are going to call **`front`** and **`rear`**. Instead of being `int` variables meaning indices, these variables are simply linked-list *pointers* (of type `Node`). The class implementation can be seen in listing 4.4. The public part of the class `Queue` is identical to the case when we used an array.

Listing 4.4: The `Queue` class using a linked-list

```
1  struct Node {
2    float data;
3    Node *next;
4  };
5
6  class Queue {
7  private:
8    Node *front, *rear;
9  public:
10   Queue();
11   ~Queue();
12   void Join(float newthing);
13   void Leave();
14   float Front();
15   bool isEmpty();
16 };
```

The prototypes of the methods are identical to the queues with arrays, but the implementation is very different.

We need to analyse listing 4.5 very carefully. Do all methods consider all possible cases?

The first method is `Join()`. This method allocates and inserts a new item after the tail of the linked-list, and then modifies the pointer `rear` to point to the new element. In a queue with elements already added to it, that should be all. However, if the queue is initially empty, then we also need to modify the `front` pointer. Figure 4.7 a) shows a general case of insertion when there are already some elements in the queue. Figure 4.7 b) shows the scenario when the queue is empty, and therefore will have to use the last statement to modify `front`.

It is also worth noting that when the queue is empty, both `front` and `rear` pointer are `NULL`. When the queue has a single element, both pointers point to the single element of the linked-list.

Listing 4.5: The Queue methods using a linked-list

```
Queue::Queue() {// constructor
  front = NULL;  rear = NULL;
}

Queue::~Queue() {// destructor
}

void Queue::Join(float newthing) {
// place the new thing at the rear of the queue
  Node *temp;
  temp = new Node;
  temp->data = newthing;
  temp->next = NULL;
  if (rear != NULL) { rear->next = temp; }
  rear = temp;
  if (front == NULL) { front = temp; }
}

void Queue::Leave() {// remove the front item from the queue
Node * temp;
  if (front == NULL) { return; }
  temp = front;
  front = front->next;
  if (front == NULL) { rear = NULL; }
  delete temp;
}

float Queue::Front() {// return the value of the front item
  return front->data;
}

bool Queue::isEmpty() {// return true if the queue is empty
  if (front == NULL) { return true; }
  return false;
}
```

After running the same `main()` function used before on the queue implemented with an array (listing 4.3), figure 4.8 shows the result in the queue implemented with a linked-list. Note that in the linked-list the front is always the front of the linked-list and that the rear is always the tail of the linked-list. Therefore elements inserted by `Join()` will be inserted after the tail of the linked-list and elements deleted by `Leave()` are deleted from the front of the linked-list.

It is also important that when the class Queue is instantiated, the pointers to the front and rear are NULL. The method `IsEmpty()` just needs to query the front pointer to correctly return the status of the queue. When there is only one element in the queue, both `front` and

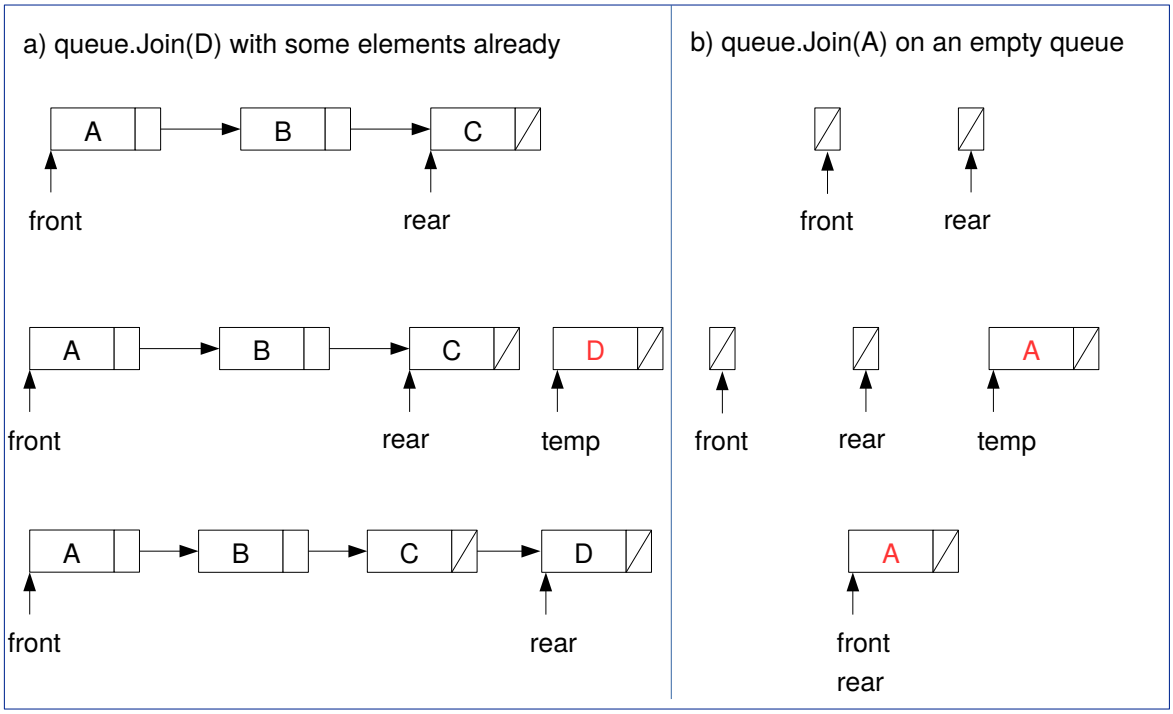


Figure 4.7: Queue implemented with a linked-list.

`rear` pointers point to the same element of the linked-list.

The method Front can only be used if the queue is not empty. Therefore it has to be wrapped by the `IsEmpty()` method, otherwise the program risks a segmentation fault. The alternative to that would be to add a statement to the method `Front()` to make sure that the pointer `front` is different than NULL before attempting to access any data. However that would demand some return value to be passed back. The value could be tied to a certain context given to the application, e.g., if the application does not deal with negative numbers, the failure of finding an element in the queue could make `Front` return -1, indicating that the queue is empty. However that would make the class dependent on the application.

Finally, which implementation is better? Queues implemented with an array serve a limited purpose but tend to be faster. The creeping problem suffered by the array can be solved by using a circular array.

The queues implemented with linked-lists are more generic in the sense that they can use dynamic allocation of the elements, and they do not suffer from the creeping problem. However the linked-lists make the methods slower than their counterparts using arrays.

4.3 - Using the Standard Template Library (STL)

As we did with stacks, we are going to show how to create simple queues using the STL. The include line is `#include <queue>` in the header.

The declaration of a queue is equivalent to the one used for a STL stack, i.e.:

```
queue<float> A;
```

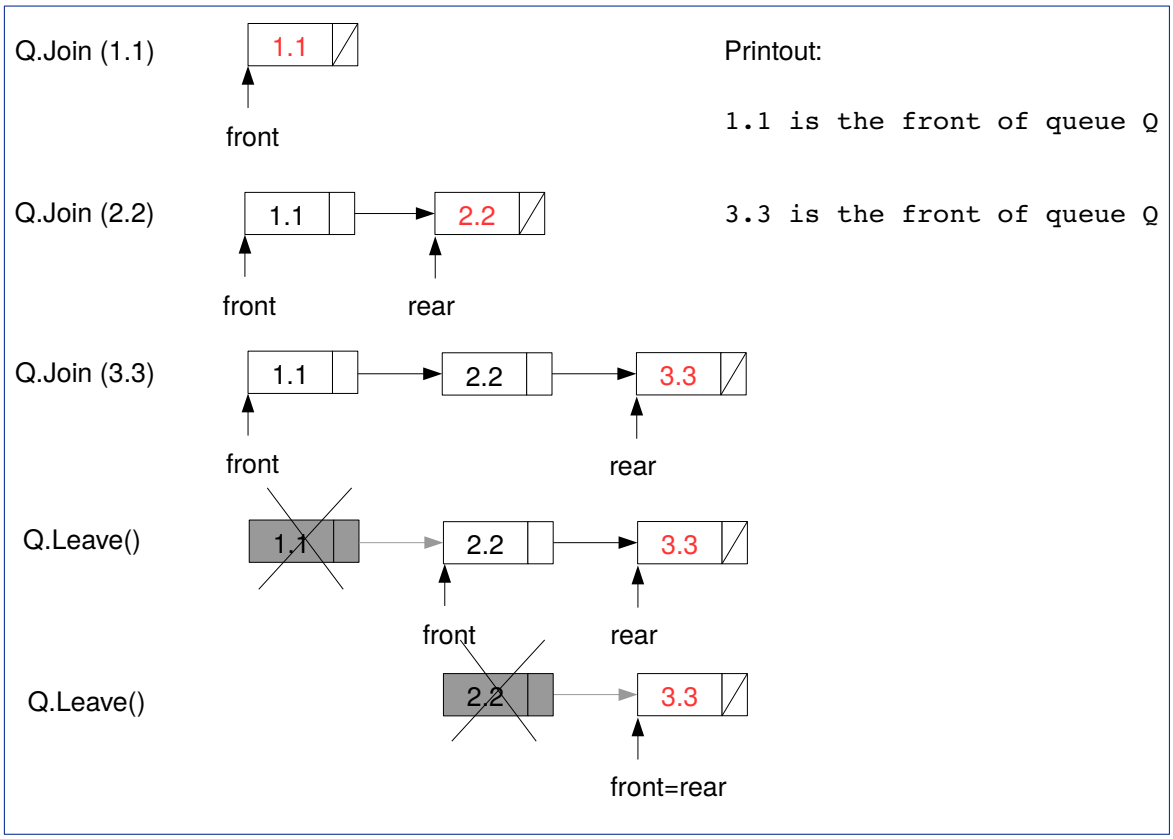


Figure 4.8: Queue implemented with a linked-list running listing 4.3.

The equivalent code for listing 4.3 can be seen in listing 4.6

As the reader can see, there are some important differences between these listings. The names of the methods are different than our own class implementations of the Queue. The `Join()` method is called `push` in STL. The method `Leave()` is called `pop`. The method `isEmpty()` is called `empty`. Finally, the method `Front()` is called `front`.

Listing 4.6: The `Queue` methods used in the `main`.

```
#include <stdio.h>
#include <queue>
using namespace std;
queue<float> Q;
int main() {
  Q.push(1.1);
  Q.push(2.2);
  Q.push(3.3);
  if (Q.empty()) printf("Queue Q is empty\n");
  else printf("%f is the front of queue Q\n",Q.front());
  Q.pop();  //Note: delete without knowing content
  Q.pop();
  if(Q.empty()==false) printf("%f is the front of queue Q
      \n", Q.front());
}
```

The queue in STL is also a template, meaning that any data type can be used to instantiate a queue in STL. The method `size` can also be used with STL queues.

The extra statement 3 in listing 4.6 was not explained. As include files need to be used, there are definitions of names of classes that are not understood by the compiler without some extra references. Normally to use the queue template we would have to refer to it including the standard namespace like this:

```
std::queue<float> A;
```

To avoid that extra reference, we just add statement 3.

4.4 - Exercises

1. Modify the methods of the class `Queue` using an array to make sure that the size Qmax is respected when using `Join()`. Also, add an error message in case the method Front is used on an empty queue.

2. Modify the methods of the class `Queue` using a linked-list to print a message in case the queue is empty and return -1.

3. The listing 4.7 shows a series of operations on a Queue of integers. Considering that this Queue was implemented as an array, draw it schematically. Indicate where are the front and rear values in the array.

Listing 4.7: main() function using a Stack

```
1  ... //copy the Stack class here
2  Queue Q;
3  main(){
4    Q.Join(1);
5    Q.Join(2);
6    Q.Leave();
7    Q.Join(3);
8    if(!Q.isEmpty()) printf("%d ",Q.Front());
9    Q.Join(4);
10   Q.Join(5);
11   Q.Join(6);
12   Q.Join(7);
13   Q.Join(8);
14   if(!Q.isEmpty()) printf("%d ",Q.Front());
15   Q.Leave();
16   if(!Q.isEmpty()) printf("%d ",Q.Front());
17   Q.Join(9);
18   Q.Join(10);
19   Q.Join(11);
20   Q.Leave();
21   Q.Leave();
22   Q.Leave();
23   if(!Q.isEmpty()) printf("%d ",Q.Front());
24 }
```

4. For the same sequence of statements above, and considering that the Queue is implemented as a linked-list, draw it schematically. Indicate the front and rear nodes.

5. In a city there is an intersection where two (single lane) one-way streets cross each other. Write a C or C++ program that simulates the situation at this intersection and measures how long each queue gets during a 2 hour period. Adjust the phasing of the traffic lights to minimise the two queues that form at the intersection. Download the sample program on Stream. The program is able to receive two command line arguments, phase for Q1 and phase for Q2 (the equivalent of adjusting the phasing of the traffic lights for each intersection). You must use command line arguments to get Q1 and Q2. The intersections will be represented by two queues (implemented as linked-lists, so it can cater for an unknown queue size).

5 Vectors and Lists

In this chapter we are going to learn about *vectors* and `lists`. In order to use and implement these two ADTs, we are also going to learn more about templates. Initially we are just using templates without creating the class. Later with lists we are going to learn how to create your own templates.

5.1 - Vectors

Vectors can be considered arrays that can extend dynamically. Vectors should be accessible by index, therefore they have random access to every element. However due to implementation issues the random access might not be as efficient as a static array because it might involve some pointer arithmetic.

Vectors are usually implemented in two forms. One uses blocks of a given size. When the vector is instantiated, a single block is allocated. When the block is full, then a larger block is allocated, the elements are copied to the new block and the old block is deleted from memory. This form of array is very fast, as the memory is always contiguous and random access does not need pointer arithmetic. However, every time the vector needs resizing it may slow down until all the elements are copied to the new block and the old block is deallocated.

The second form for vectors is to implement it as a linked-list of blocks. Each block holds a certain number of elements. The vector starts with a single element (one block). When the block is full, a new block is allocated and linked with the previous block. New blocks are created and linked just like a linked-list. Resizing is faster than the previous form of vectors, however random access suffers from having to traverse the linked-list. Suppose that the vector has three blocks and the index points to an element in the last block. The last block is not accessible directly, so it would take two steps to find the pointer of the third block, and only then could the element be found using the index.

Instead of implementing our own vector class, we are going to use one already available through the **STL**, short for *Standard Template Library*. We have seen two simple examples using STL's stacks and queues. Templates in C++ are implementations of data structures where the data type is specified at the instantiation step in the program. That means that the compiler will only know what the type of the data is when it reaches a declaration of an instance of that class.

This is a much better proposition than the simple classes we used for Stacks and Queues. Remember that we have to modify the class itself for different data types and also modify the methods that use the data. Imagine how much effort it takes for maintaining classes, and also the chances of making mistakes.

When using templates the programmer might not know any detail about the implementation, only about how the ADT should respond to standard methods.

5.2 - Using STL's Vector

We start to explore vectors in STL using a simple approach. The first listing creates a vector without any specific size. We then use the method `resize` to start using the vector as we would use an array. The advantage is that the program can change the size of the vector to whatever the user wishes. The code is in listing 5.1.

Firstly you need to include `<vector>` in the code. Secondly, you need to declare a vector of a certain data type and a name for the instance. In the listing we choose `int` and the name is `v`.

Listing 5.1: Resizing and using a vector.

```
1  #include <stdio.h>
2  #include <vector>
3  using namespace std;
4  vector<int> v;
5  int size=0;
6  int num;
7  int main() {
8    printf("What size vector do you want? ");
9    scanf("%d", &size);
10   v.resize(size);             // you MUST set a size
         before using
11   printf("Max size of this vector is %d\n", (int)v.size()
         );
12   for (int i = 0; i < v.size(); i++) {
13     printf("Enter a number ");
14     scanf("%d", &num);
15     v[i] = num;
16   }
17   for (int i = 0; i < v.size(); i++) {
18     printf("%i\n", v[i]);
19   }
20 }
```

Note that the `for` loop requires the limit to coincide with that of the vector, so it is safer to limit the loop to `v.size()`. Using an index above the limit of the vector might cause a segmentation fault (just like with static arrays).

Under different circumstances the vector may have to be stretched (insert more space for data). For this reason vectors have a method that allows one to push a new element to the end of the vector. The end of a vector is its last index (or `v.size()-1`). The method is called `push_back` in STL. The syntax is similar to what we used before for stacks.

This method has to be used carefully, as it requires some form of memory allocation. Sometimes it is better to `resize()` and allocate a big block instead of resizing one element at a time (pushing one at a time).

In listing 5.2 there is an example where the numbers are *pushed into* the vector. Note that after the numbers are copied, the vector elements can be accessed as if it were an array.

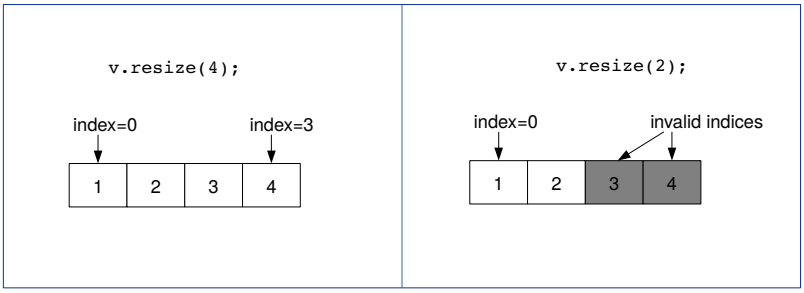


Figure 5.1: Vector: care should be taken about the index limit.

Listing 5.2: Numbers being pushed onto the vector

```
#include <stdio.h>
#include <vector>
using namespace std;
vector<int> v;

#define MAXSIZE 10

int main() {
  int num; // here we DO NOT set a size
  for (int i = 0; i < MAXSIZE; i++) {
    printf("Enter a number ");
    scanf("%d", &num);
    v.push_back(num);
  }
  // the printed size will be 10
  printf("Max size of this vector is %d\n", (int)v.size()
      );
  // now print the numbers as above
  for (int i = 0; i < v.size(); i++) {
    printf("%i\n", v[i]);
  }
}
```

To learn more about STL, you can use the online reference http://www.cplusplus.com/reference/stl/. There are different implementations of STL. It is important to know that, as code for a certain STL might not compile on a different one.

For vectors, the STL in GNU GCC offers the following methods: `begin, end, rbegin, rend size, max_size, resize, capacity, empty, reserve operator[], at , front, back assign, push_back, pop_back, insert, erase, swap, clear, get_allocator`. We have only used a few in the examples above, but just enough to use vectors later on to implement certain types of trees.

Figure 5.1 calls the attention about the limit of the index of a vector, the same as an array. Remember that you can always inquire about the current size of a vector using the method `size`.

5.3 - Lists

A `List` can be considered a special form of linked-list. A `List` should include certain operations and auxiliary variables to help with these operations. The elements in a List are not accessible randomly (no index). Therefore, a List has to be traversed to find elements or to find the position where new elements are to be inserted or where elements have to be deleted. For that reason, it is typical that lists have at least two pointers: `front` and `current`.

Inside every element there should be at least a pointer to the next (`next`), and sometimes a pointer to the `previous` node. When the elements have a `previous` pointer the `List` is using a *doubly linked-list*, and it is possible to traverse the `List` backwards (this is left as an exercise, see exercises 5, 6 and 8).

Lets implement a `List` using a linked-list with only three basic methods: `FirstItem`, `NextItem` and `AddtoFront`. This example is a minimum implementation for a List, and one can imagine many other operations that could be included in the class. Moreover, we are going to code a template for the `List` class. As seen in the vectors section, this will allow us to define a new instance of a List using any data type without changing anything in the class itself (listing 5.3). Note that the pointer `next` belongs to the `struct Node`, while `front` and `current` are pointers that are private variables.

So where is the linked-list in this class? The linked-list is addressed by the pointer `front`. The address of the first element of the linked-list is written into `front`. The other pointer `current` contains an ephemeral address, it can point to any other element at a certain time (it is going to be driven by the methods `FirstItem` and `NextItem`).

The other important observation is the syntax of the template class. Note that the words `template <class T>` indicate that this class has an undefined data type, and that the data (contained inside the Node) is of type T. The data type will be resolved during the compilation, when specific instances of the class are instantiated by the program. For example, in `main()` one could use a List of integers by declaring: `List<int> mylist;`

The prototypes of the three methods also reflect that fact that this is a template. One needs to pass `newthing` to the method `AddtoFront`, and a pointer to an item (type `T`) for the methods `FirstItem` and `NextItem`.

Listing 5.3: `List` class.

```
1  template <class T>
2  class List {
3  private:
4    struct Node {
5      T data;
6      Node *next;
7    };
8    Node *front, *current;
9
10 public:
11   List();
12   ~List();
13   void AddtoFront(T newthing);
14   bool FirstItem(T & item);
15   bool NextItem(T & item);
16 };
17
18
19 template <class T>
20 List<T>::List() { front = NULL;  current = NULL; }
21
22 template <class T>
23 List<T>::~List() { }
24
25
26 template <class T>
27 void List<T>::AddtoFront(T newthing) {
28   Node *temp;
29   temp = new Node;
30   temp->data = newthing;
31   temp->next = front;
32   front = temp;
33 }
34
35 template <class T>
36 bool List<T>::FirstItem(T & item) {
37   if (front == NULL) { return false; }
38   item = front->data;
39   current = front;
40   return true;
41 }
42
```

```
43 template <class T>
44 bool List<T>::NextItem(T & item) {
45   current = current->next;
46   if (current == NULL) { return false; }
47   item = current->data;
48   return true;
49 }
```

About the methods, the first method AddtoFront is essentially the same as we used in linked-lists before. The only difference is the generalisation of the data type (template). Note that the process of creating and linking a new `Node` is the same.

The other two methods have one peculiarity. The return type is a `bool`, and the parameter passed to the methods is a pointer to `T`. What does that mean? In order to avoid problems with the state of the List (e.g. an empty list) we are now resorting to only get the data if the return of the method is true. This is going to be more clear in the main() function example. The pointer to `T` is being used to get the data and copying it onto an external temporary variable. If the list is empty or the end of the list is reached, random data might be copied to the temporary variable, but no harm done, as the condition of the bool variable indicates that this is invalid data. Look at listing 5.4 for details (pay particular attention to the `while` loop).

Listing 5.4: `List` class main() example.

```
1  List<int> L; //create a list of integers
2
3  int main() {
4    L.AddtoFront(28);
5    L.AddtoFront(54);
6    L.AddtoFront(53);
7    L.AddtoFront(52);
8    bool ok;  int x;
9    ok = L.FirstItem(x);
10   while (ok) {
11     printf("%i \n", x);
12     ok = L.NextItem(x);
13   }
14 }
```

The way in which the List is printed is very different than what we were doing in linked-lists. Note that to traverse all elements, the `FirstItem` method is called once. If the first element exists, it will copy the data onto `x`. If successful (indicated by `bool ok`) then the `while` loop condition is true. Only then the data of the element is printed, and an attempt to reach the next element of the list calls NextItem (also using `ok` and `x`). If the second element exists, the while loop condition is true again, and the second element's data is printed. A new attempt to reach the next element is carried out. When the

data of the last element is printed, the `NextItem` call will return false, leaving the `while` loop and terminating the program.

There is an important observation about the pointers `front` and `current`. Pointer `front` is used to *keep track of the head of the linked-list* that implicitly is part of the List. Pointer `current` is used to *keep track of where the traversal stopped* the last time `NextItem` was called. When `FirstItem` is called at any point, current always points back to the first element (or if the List is empty, becomes a NULL pointer). Figures 5.2, 5.3 and 5.4 show listing 5.4 at work and illustrates these observations.

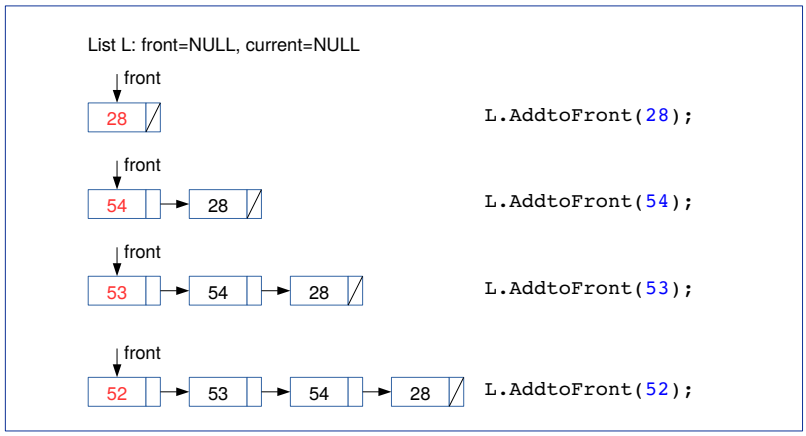


Figure 5.2: Using List's `AddtoFront(item)`.

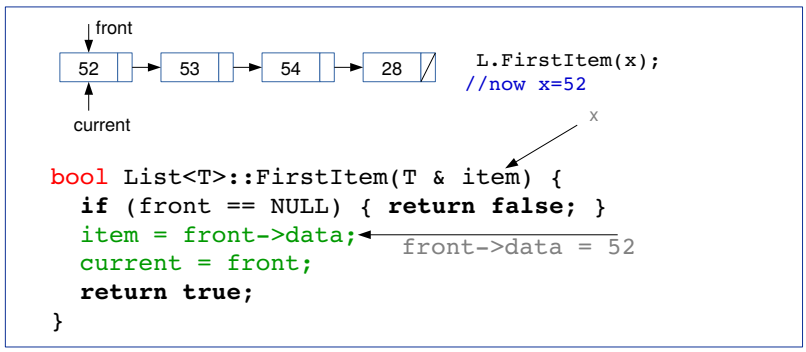


Figure 5.3: Using List's `FirstItem()`.

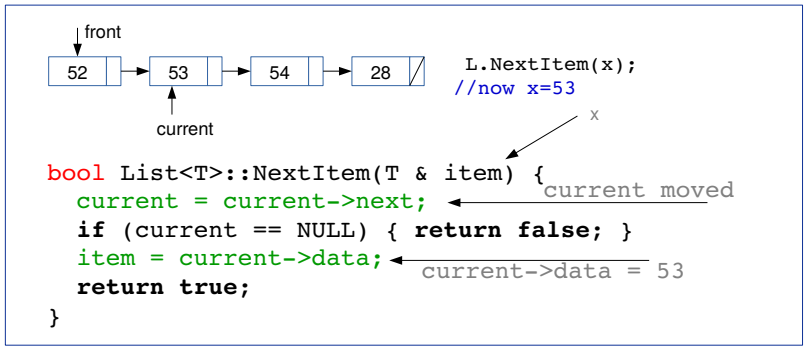


Figure 5.4: Using List's `NextItem()`.

5.3.1 - Using another class as the item for a List

Now we can test the power of the templates. Instead of using a traditional data type, we can use another class inside the class `List`. This is not going to require any changes to the class `List` itself. However, when it comes to printing the data inside each Node of the `List`, then a method has to be devised for the other class being used as items. We can say that we have a class instance contained in the Node that belongs to another class.

The complete listing will include (copy/paste) the original class `List` (call that section A), then a new class has to be created for Book items (section B) and finally a display function has to be coded (section C).

To clarify the idea, lets say that a List of Books need to be implemented. The Book class can be coded separately, and it is going to be as per listing 5.5.

Listing 5.5: The `Book` class (section B).

```
1  class Book {
2  private:
3    char title[80];
4    float callnumber;
5  public:
6    Book() {callnumber = 0.0;}// inline function
7    ~Book() { }
8    void GetData();
9    void Display();
10 };
11
12 void Book::GetData() {
13   cout << "Enter the title " << endl;
14   cin >> title;
15   cout << "Enter the call number " << endl;
16   cin >> callnumber;
17 }
18
19 void Book::Display() {
20 cout << callnumber << " " << title << endl;
21 }
```

Now one can instantiate books, but we want to do that using a `List`, so the data input is stored in the `List`. The `main()` function can be seen in listing 5.6. Note that the first statement is the declaration of an instance of `List` called `booklist`. The data type is now `Book`, not `int` nor `float` (no problems compiling that!). The rest of the code is more or less straightforward and similar to listing 5.4. The difference is that the Book data cannot be accessed directly (the `title` and `callnumber` are private), so the only way to change these variables is by using the method `GetData`. That is the reason to have a temporary `Book` instance

(called `temp` in the listing). The data can be copied onto `temp` first, and `temp` is then used as a parameter for the List method `AddtoFront`. The data for the book is then copied onto the new element of the List `booklist`. A single element of the List `booklist` looks like the one depicted in figure 5.5.

Listing 5.6: The `Book` main().

```
1  List<Book> booklist;  //global var, a List of Books
2
3  int main() {
4    Book temp;  char ch;
5    cout << "Enter several books into the list" << endl;
6    while (true) {
7      temp.GetData();
8      booklist.AddtoFront(temp);// will this have copy
           problems?
9      cout << "Enter another book? (Y/N) " << endl;
10     cin >> ch;
11     if (ch == 'N') { break; }
12   }
13   cout << "Here is the list" << endl;
14   DisplayAllBooks();
15 }
```

Finally, the last statement in listing 5.6 is `DisplayAllBooks();` Listing 5.7 shows this function. Note that in order to get the data from the Nodes, one has to use a temporary Book variable to receive what the List's methods `FirstItem` and `NextItem` are getting. To actually print, one has to use the Display method for the class `Book`. Apart from that, the `DisplayAllBooks` looks like our previous traversal of a List depicted in listing 5.4. By combining these two classes, we successfully created and printed a list of books.

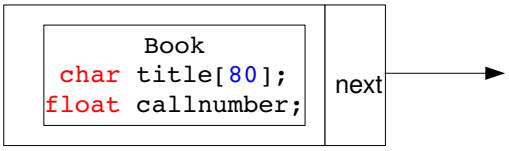


Figure 5.5: The Node in a List of Books.

Listing 5.7: The DisplayAllBooks function (section C).

```
1  void DisplayAllBooks(){
2    bool ok;
3    Book temp;
4    ok = booklist.FirstItem(temp);
5    while (ok == true) {
6      temp.Display();
7      ok = booklist.NextItem(temp);
8    }
9  }
```

5.4 - Exercises

1. Write a C++ program for a Stack class using vectors instead of arrays.

2. Write a C++ program for a Queue class using vectors instead of arrays. Is the *creeping problem* still an issue?

3. Write a C++ program to create a vector of a relatively large size (say 1000). Create two different functions to add 1 to every element of the vector. The first function receives the vector by 'value'. The second function receives a pointer to the vector. Print the vector at the end of each function to make sure it worked in both cases.

4. Write a recursive function where you need to pass vectors as parameters. Which one is faster, passing:

   `vector<> v`

   or: `vector<> &v`?

   Why?

5. Modify the class `List` to include a `previous` pointer inside the Node. Also, include a `rear` pointer (besides `front` and `current`). Now modify the AddtoFront method to make sure that the rear pointer is always at the tail of the linked-list. You just created a List that can be traversed backwards (also called a bi-directional List).

6. Using the modified class List from the previous exercise, create two new methods called `LastItem` and `PreviousItem`. The first method should get the data of the Node at the tail of the List (`current` should be the same address as `rear` after the call). The second method should get the data of the previous Node as well as move `current` one position backward. Using the two new methods, show that you can print the List in reverse.

7. Based on the function that reverses the elements of a linked-list, write a *method* (not a function) that can reverse the elements of a List (based on the reversal function seen in the linked-list chapter).

8. Using the bi-directional list you have created before, write a method to insert a new item into a List at the end of the list (after `rear`).

6 Binary Trees

6.1 - Introduction

Trees are hierarchical data structures. They are very useful ADTs, as there are many applications where the best performance is reached by using trees.

Binary trees are trees in which every node has only two children, so they are very easy to implement. For this reason they are more popular than general trees. There are also special forms of trees that have their own application scope, such as quad-trees (each node has four children), or oct-trees (each node has eight children) etc.

In this chapter we introduce simple binary trees, teaching how to implement a simple node class. We also introduce the traversals of binary trees, using both recursive and iterative functions.

6.2 - Binary Trees

A *node* is the single unit of a tree. The node contains data and two pointers, one to the left child and another to the right child.

A *root* of a binary tree is a node that has no parents, i.e., it is the first node accessible directly via a pointer with the name of the tree.

A *leaf* is a node with no children. A leaf sits at the bottom a sub-tree.

Parent and *child* are relative status of a node. Each child has only one parent, but a parent can have up to two children.

A *full binary tree* is a binary tree in which every node has either zero or two children. Note that a full binary tree can still be asymmetric.

A *complete (or perfect) binary tree* is a full binary tree where all the leaves are at the same level. Note that in this case only symmetric binary trees can be complete binary trees. See figure 6.1 for examples.

A binary tree is either:

a) the NULL tree (a tree with no nodes)

b) a root with two sub-trees, where each sub-tree is also a binary tree

In other words, each node of a binary tree is also a tree in itself. We refer to parts of the binary tree as *sub-trees*. That means that the sub-tree can be used as an independent tree depending on application. Note that the very definition of a tree implies in recursion.

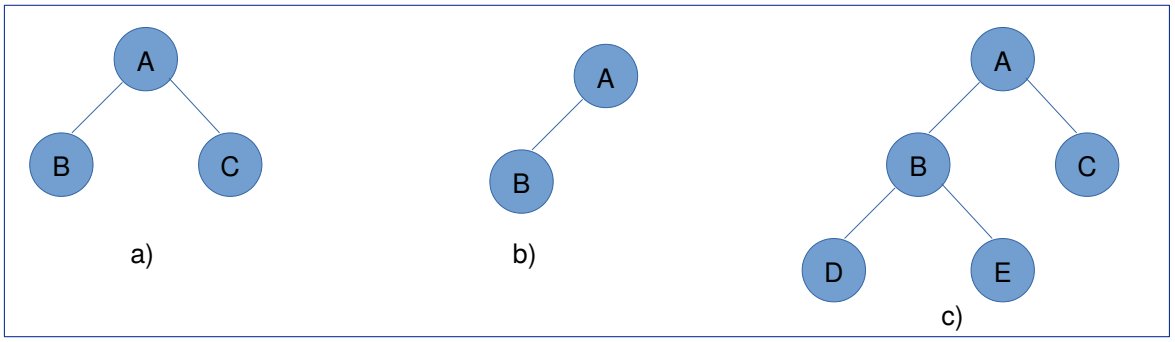


Figure 6.1: a) a full and complete binary tree b) not full nor complete c) full but not complete.

6.3 - *Implementing Binary Trees With a Node of Two Pointers*

In a similar way that we created linked-lists, we can define a *Tree Node* with two pointers of the same type, making one pointer to the left child and the other to the right child. If one or two of the children to not exist, then the pointers should be NULL. In that sense, a binary tree can be viewed as a 2 dimensional data structure, while the linked-list is a 1 dimensional data-structure (linear).

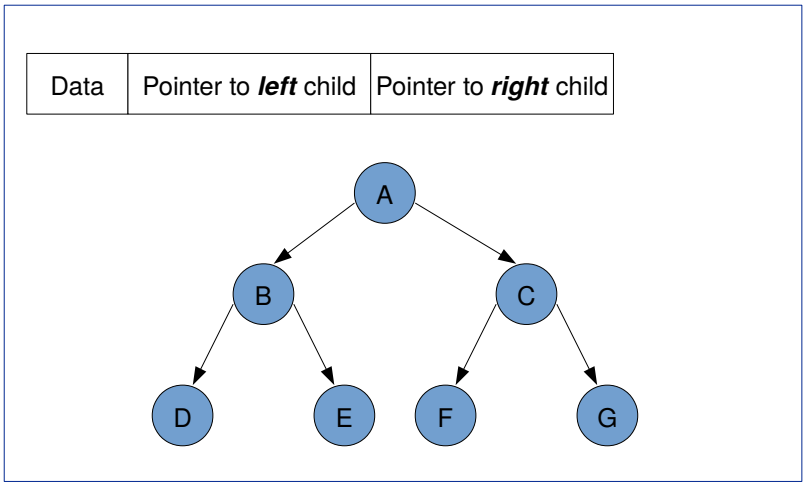


Figure 6.2: Implementing Trees using a class node with two pointers.

Figure 6.2 shows that the Node should have some data (including the *key* to find the nodes) and two pointers, the `left` and the `right`.

There is a difference from what we implemented in the linked-list and this implementation of binary trees: instead of using a `struct` for the Node, we are going to use a **class**. In the same way that the compiler could resolve the recursion involved in defining a linked-list node, it can also resolve the class recursion.

Listing 6.1 shows the class with the declaration of the data, the pointers and the methods. Let us start with a simple class, and we can elaborate later. This will be suitable for a generic binary tree that has no specific rules on how the keys should be ordered. It is completely random or dependent on the order in which the nodes are connected to each other.

Analysing the definition of the Tree class we can already see one

Listing 6.1: The `Tree` class

```
class Tree {
  private:
    char data;
    Tree *leftptr, *rightptr;

  public:
    Tree(char newthing, Tree* L, Tree* R);  // constructor
        with parameters
    ~Tree() { }
    char RootData() { return data; }  // inline methods
    Tree* Left() { return leftptr; }
    Tree* Right() { return rightptr; }
};
```

Listing 6.2: The constructor of the `Tree` class

```
Tree::Tree(char newthing, Tree* L, Tree* R) {
  data = newthing;
  leftptr = L;
  rightptr = R;
}
```

important difference. The *constructor* has parameters passed to it. What does that mean? It means that when we instantiate a Tree node, we have to pass the parameters to it. The three parameters in the code are clearly all the variables contained in the private scope (`data`, `leftptr` and `rightptr`). In other words, the way in which we are going to allocate a node for a tree is very different than what we did in the linked-list: the Tree nodes can be created independently of an existing Tree, without using a specific insertion function, with the possibility of linking to a Tree later.

The constructor implementation is in listing 6.2. The only three statements are simply copying the data from the parameters passed to the constructor to the private variables of the Tree class. However, how can we know what the pointers (the addresses of the right and left children) are? For now just assume that a single node that is a leaf (i.e., a tree with a single node) can be created. In this case, both the pointers should be NULL, and one could create a single node Tree with content A using the following statement:

```
Tree * myTree = new Tree('A', NULL, NULL);
```

The other three methods are simply returning the tree private variables. Therefore, these three methods are going to be used to gather information about the data contained in the node as well as where the node points to.

Listing 6.3: Creating a Tree bottom-up.

```
1  Tree *T1, *T2, *T3, *T4, *T5, *T6, *myTree;
2
3  int main() {
4          T1 = new Tree('D', NULL, NULL);
5          T2 = new Tree('E', NULL, NULL);
6          T3 = new Tree('B', T1, T2);
7          T4 = new Tree('F', NULL, NULL);
8          T5 = new Tree('G', NULL, NULL);
9          T6 = new Tree('C', T4, T5);
10         myTree = new Tree('A', T3, T6);
11  }
```

In order to fully understand the power of such a simple class, we are going to create a full binary tree *manually*. Each of the nodes are going to be created by the use of the constructor, making sure that they can be linked to the tree as they are created. It is intuitive that one needs to start from the leaves of the tree because the leaves have no children and therefore the `right` and `left` pointers are known to be NULL. Any other node has to wait until its children are created, defining the address to where `right` and `left` needs to point to. Therefore this is a bottom-up operation (children first, parents later).

Using only the constructor, we want to create the binary tree seen in figure 6.2. Listing 6.3 does exactly that. Note that the only statements are constructors, the various tree pointers receive the addresses of the new tree nodes as a result of the allocation. As the nodes are being allocated, they also receive the data, in this case a single `char` (also the key of the nodes) and the `left` and `right` pointers. At this stage there are no rules in relation to the key's order nor any restriction on repeated keys. The only reason we use different keys for every node is the convenience of understanding the code.

The insertion of nodes on the tree follows the manual process of creating the two children first, then creating the parent. Figure 6.3 shows the creation of a 3 node tree.

After creating a 3 nodes tree, it becomes a sub-tree of a larger tree. The last statement links all the sub-trees together. At this point, all the pointers `Tree* T1, *T2... *T6` could have been discarded. We will see that with a single pointer to the root of the tree (`Tree* myTree`) we can traverse the entire tree.

6.4 - Traversing a Binary Tree

There are different forms of traversals covered in this course. The most common traversals are called *depth-first* and *breadth-first* (or depth-first search and breadth-first search).

Depth-first traversals try to visit nodes going all the way to the

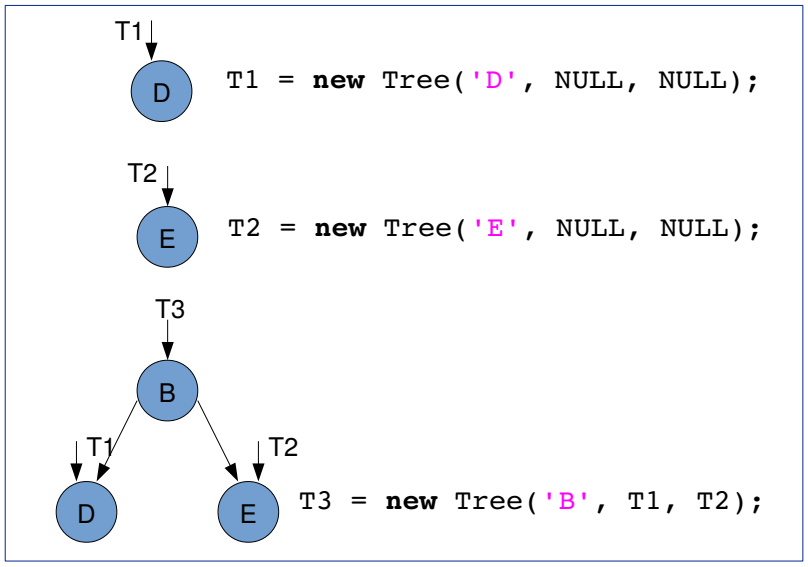


Figure 6.3: A binary Tree constructed bottom-up.

bottom of the tree via the left nodes until it reaches a leaf, and then going back to the origin of the traversal and carrying on visiting the right sub-tree. However, there are three ways in which this can be achieved, namely *pre-order*, *in-order* and *post-order*.

In the pre-order traversal, the root of the tree is visited first. Then the sub-tree on the left is visited as a pre-order traversal. Finally the sub-tree on the right is visited as a pre-order traversal. Note the recursion involved in the definition of the traversal. The other two traversals are almost identical, with the difference in which the root is visited in relation to the left and right sub-trees.

Pre-order traversal:

1. Visit the **root**
2. Pre-order traversal of the **left** subtree
3. Pre-order traversal of the **right** subtree

In-order traversal:

1. In-order traversal of the **left** subtree
2. Visit the **root**
3. In-order traversal of the **right** subtree

Post-order traversal:

1. Post-order traversal of the **left** subtree
2. Post-order traversal of the **right** subtree
3. Visit the **root**

A simple example of traversal follows. For the binary tree in figure 6.2, the results of the traversals are:

- Pre-order: A B D E C F G
- In-order: D B E A F C G
- Post-order: D E B F G C A

One can see some patterns emerging. In the pre-order, the root of the tree ('A') is the first one to be printed. In the in-order traversal, the root is in the middle, as the tree is complete. Finally in the post-order traversal, the root is the last one to be printed.

To understand how the traversal works, it is easier to start with a simple recursive algorithm (algorithm 6) that is be able to print all the nodes of any binary tree.

Algorithm 6 In-order traversal of a Binary Tree.

```
1: function InOrder(Tree address T)
2:     if (T = = NULL) then
3:         return;
4:     end if
5:     InOrder(left of T)
6:     Print Root Data
7:     InOrder(right of T)
8: end function
```

The function relies on recursion (notice the `inOrder()` recursive calls on statements 5 and 7). We can think of recursion as operations that are pushed onto a Stack, and then popped from the Stack again to carry on the next statement (from where it stopped when it was pushed onto the stack). Each recursion carries out some statements until it calls itself with different parameters. The "instance" of that function stops, gives the control to a new instance, so the new call carries out the first statement. After that instance finishes, it gives the control back to the caller, so the caller continues from where it stopped when it was pushed onto the Stack.

Imagine that you have a Stack that controls the recursion. The rules are: every time a new instance of the algorithm 6 is called, it will run from statements 2 to 8. However, every time it runs statements 5 or 7 the instance of the algorithms stops and is pushed onto the Stack. The information pushed onto the stack is the node's key (this identifies which instance of the algorithm is) and the last statement it runs before being pushed. It is easy to see that the only statements for any instance that are stopped and represented in the Stack are statements 5 and 7.

There is only one active instance of the algorithm at any time, and when it finishes it will give the control back to the instance that is the **top** of the Stack. Figure 6.4 shows part of the in-order traversal of the tree in figure 6.2. The completion of the traversal is left as an exercise (the answer is DBEAFCG).

The implementation of algorithm 6 in C++ is shown in listing 6.4. Note that the function needs a single parameter, and this is the address of a Tree node (a `Tree*`). When calling the function

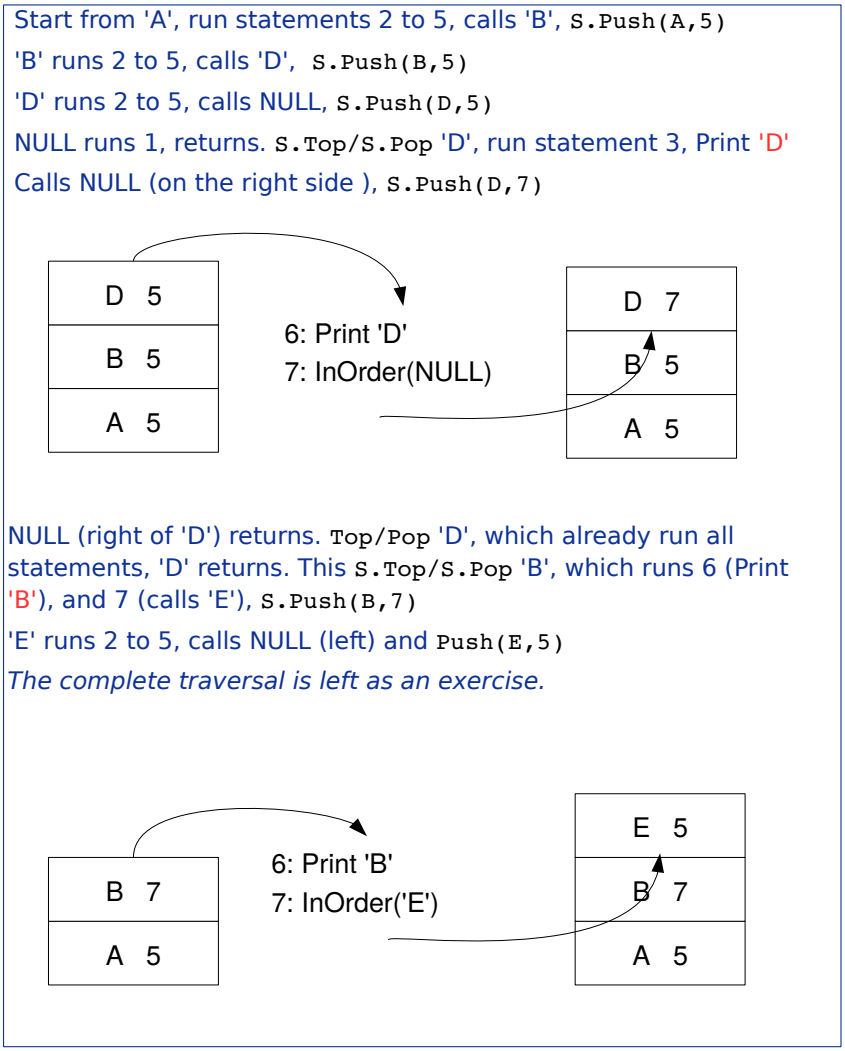


Figure 6.4: Using an imaginary Stack to explain the recursion in algorithm 6.

recursively, one needs to pass the same parameter type. Looking at the method `Left` and `Right` in the `Tree` class (listing 6.1), indeed both return a `Tree*`, fulfilling the requirement of the single argument passed to the `inOrder` function.

Listing 6.4: In-order traversal (recursive).

```
1 void inOrder(Tree *T) {
2   if (T = = NULL) { return; }
3   inOrder(T->Left());
4   cout << T->RootData() << " ";
5   inOrder(T->Right());
6 }
```

If we examine the definition of the other two traversal orders, it is very easy to modify listing 6.4 to cater for the other two (see listings 6.5 and 6.6). The only changes were the position of the printing statement and the names of the functions.

Listing 6.5: Pre-order traversal (recursive).

```
1  void preOrder(Tree *T) {
2    if (T = = NULL) { return; }
3    cout << T->RootData() << " ";
4    preOrder(T->Left());
5    preOrder(T->Right());
6  }
```

Listing 6.6: Post-order traversal (recursive).

```
1  void PostOrder(Tree *T) {
2    if (T = = NULL) { return; }
3    PostOrder(T->Left());
4    PostOrder(T->Right());
5    cout << T->RootData() << " ";
6  }
```

6.4.1 - Non-recursive Traversals

Recursive functions are very useful and easy to implement. However, sometimes it is convenient to use an iterative function instead. An iterative function can use a Stack as an auxiliary ADT (explicitly declared for this purpose). The content of the stack is different to the example seen previously to explain recursions. For the purpose of iterative functions, the stack needs to contain a tree node address (a `Tree*`). Intuitively, we think that we can easily code the three traversals, but that is not the case. The pre-order traversal is the easiest traversal to write iteratively.

Listing 6.7: Pre-order traversal using stacks.

```
1  void preOrder_with_Stacks(Tree *T) {
2    Stack S;//modify the Stack class to use Tree*
3    Tree * tempT = T;
4    if(tempT!=NULL) S.Push(tempT);
5    while(S.isEmpty()==false){
6      tempT = S.Top();
7      S.Pop();
8      cout << tempT->RootData() << " ";
9      if(tempT->Right()!=NULL){
10       S.Push(tempT->Right());
11     }
12     if(tempT->Left()!=NULL){
13       S.Push(tempT->Left());
14     }
15   }
16   cout << endl;
17 }
```

The pre-order should use the stack as a way to make the node wait before printing its key. When calling the traversal from the root, the first step is to push the address of the root onto the stack. After that, a loop can `top` (and `pop`) the node to print it (it makes sense, as the root is the first one to the printed in this traversal). Inside the loop the right and left children (if they exist) are pushed onto the stack. The loop keeps calling `top` and `pop` and the next node is printed, and so on and so forth. Listing 6.7 is a complete function, but it assumes that the stack data is of `Tree*` type.

The in-order and post-order iterative traversals implemented in C++ are listed in listings 6.8 and 6.9.

Listing 6.8: In-order traversal using stacks.

```
void inOrder_with_Stacks(Tree *T) {
  Stack S;  Tree* temp=T;
  while(temp!=NULL){
    while(temp!=NULL){
      if(temp->Right()!=NULL) S.Push(temp->Right());
      S.Push(temp);
      temp=temp->Left();
    }
    temp=S.Top();
    S.Pop();
    while(S.isEmpty()==false && temp->Right()==NULL){
      cout << temp->RootData() << " ";
      temp=S.Top();
      S.Pop();
    }
    cout << temp->RootData() << " ";
    if(S.isEmpty()==false){
      temp = S.Top();
      S.Pop();
    }
    else temp=NULL;
  }
}
```

Listing 6.9: Post-order traversal using stacks.

```
void postOrder_with_Stacks(Tree *T) {
   Stack S;
   Tree* temp1, *temp2;
   temp1=T;   temp2=T;
   while(temp1!=NULL){
     while(temp1->Left()!=NULL){
       S.Push(temp1);
       temp1=temp1->Left();
     }
```

```
10      while(temp1->Right()==NULL || temp1->Right()==temp2)
            {
11        cout << temp1->RootData() << " ";
12        temp2=temp1;
13        if(S.isEmpty()==true) return;
14        temp1=S.Top(); S.Pop();
15      }
16      S.Push(temp1);
17      temp1=temp1->Right();
18    }
19    cout << endl;
20  }
```

6.5 - *Breadth-first traversal*

In the previous section we learnt about traversing a binary tree using depth-first search. Sometimes we are interested in printing the data by level. This is also a way to print a tree in human readable format.

The easiest way to implement a breadth-first traversal is to use a queue of `Tree*`. Algorithm 7 explains how it works. The queue QT accumulates the addresses of the tree nodes in the same order that you would if traversing the tree by level (from top to the bottom, left to right). For example, visiting node B in figure 6.2 will insert nodes D and E in the queue, but those are positioned to be served after node C, already in the queue by the time we print B. Therefore, the nodes are visited in level order. For a complete example please refer to the slides.

Algorithm 7 Breadth-first traversal of a Binary Tree.

```
1:  function BREADTH-FIRST(Tree's root address T)
2:      declare a Tree node temp and a Queue QT ▷ QT contains Tree*
3:      temp = T
4:      if (temp is not NULL) then
5:          temp joins QT
6:          while (QT is not empty) do
7:              temp = front of QT
8:              QT leave                          ▷ delete front of QT
9:              visit temp                        ▷ print nodes's data
10:             if (left of temp is not NULL) then
11:                 left joins QT
12:             end if
13:             if (right of temp is not NULL) then
14:                 right joins QT
15:             end if
16:         end while
17:     end if
18: end function
```

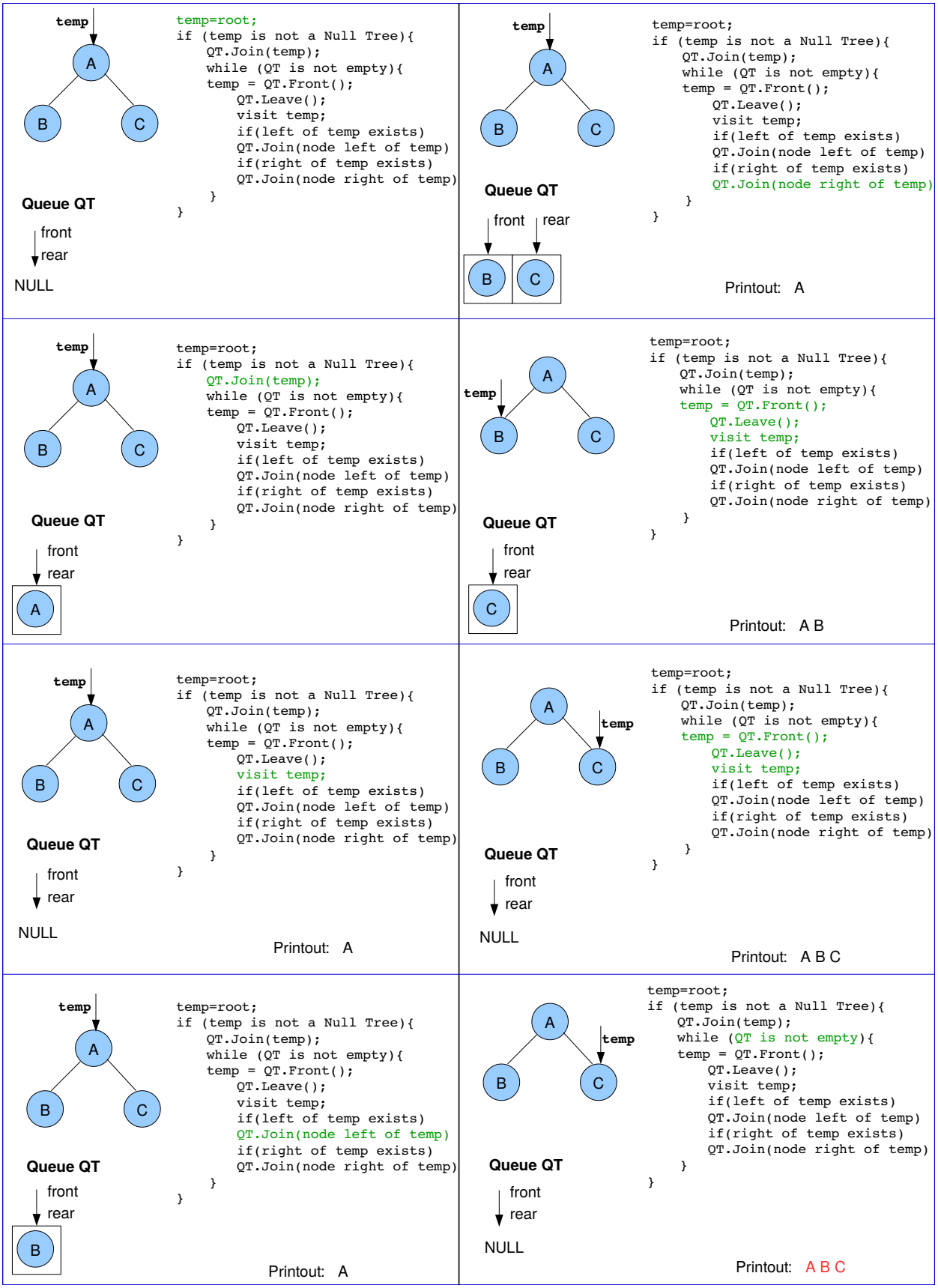


Figure 6.5: Breadth First example, following algorithm 7.

6.6 - Height of a Binary Tree

The height of a binary tree is the highest possible number of values within the tree.

Listing 6.10: Height of a Binary Tree

```
1 int Height(Tree * T){
2   if (T==NULL) return 0;
3   else{
4     int LDepth = Height(T->Left());
5     int RDepth = Height(T->Right());
6     if (LDepth>=RDepth) return (LDepth+1);
7     else return (RDepth+1);
8   }
9 }
```

A simple way to get the height (or depth) of a binary tree is to use a simple recursive function. The listing 6.10 shows a simple example of a C++ function for that purpose.

The code works in a simple manner. If the root of the sub-tree is NULL, then the height should be zero and the function returns zero. Using recursion, one can inquire about the height of the left and the right sides of the root. The code does that by recursively calling the left and the right children of the root. If both children are NULL, then both recursions return zero, and the top recursion returns 1+0, making the height=1. If one or both children exist, and they are leaves, then their recursion will return 1, making the top one return 1+1, so height=2.

The reader should be convinced, by trying a few examples schematically, that because of statement 6 and 7 in listing 6.10, the highest part of the binary tree will be measured, and the height of an asymmetric tree will be measured correctly.

6.7 - Printing a Tree in Readable Format

With a modified version of the Breadth-first traversal and the height of the tree, we can code a simple function to print a tree in readable format. The height is needed to estimate how much space we need to spread all the leaves evenly on the terminal. The modified code is presented in listing 6.11 (the NULL trees indicated by underscore symbol '_').

The `while` loop in statement 6 in algorithm 7 was replaced by the two `for` loops (in listing 6.11). The first `for` loop (line 8) deals with each level of the tree. The second `for` loop (line 10) deals with every node of that level, and this is the reason why the limit of the loop is the number of nodes in that level (given by the simple equation $numbernodes = 2^{currlevel}$).

We can improve the listing 6.11 by including the right number of spaces and also use the symbols '\', '/', and '|' to show the links

between the nodes. This code is available on Stream and it can be used to debug other binary tree programs.

Listing 6.11: Printing a Binary Tree by level.

```
void Print_by_level(Tree* root){
  Queue Q;//a queue of Tree *
  Tree *current=root;
  int height=Height(root);
  if(current!=NULL){
    Q.Join(current);
    //for every level
    for(int currlevel=0; currlevel<Height(root);
        currlevel++){
      //for all possible nodes (including NULL)
      for(int c=0; c<pow(2,currlevel) && Q.isEmpty()==
          false; c++){
        current=Q.Front();
        Q.Leave();
        if(current!=NULL) {
          cout << current->RootData() << " ";
          Q.Join(current->Left());
          Q.Join(current->Right());
        }
        else {
          cout << "_" << " ";//"_" indicates a NULL node
          Q.Join(NULL);
          Q.Join(NULL);
        }
      }
      cout << endl;
    }
  }
}
```

The listing 6.12 will print a tree in a more readable format. Follow an example of the printout in figure 6.5.

Listing 6.12: Printing a Binary Tree by level.

```
void printspace(int n){
  for(int a=0;a<n;a++) cout << " ";
}

void printunderscore(int n){
  for(int a=0;a<n;a++) cout << "_";
}

void PrintBinaryTree_WithSpacesandLinks(Tree* root){
  Queue Q;//a queue of Tree *
```

```
  Tree *current=root;
  int height=Height(root);
  printf("height is %d \n",height);
  int initialspacepernode=7;//5 or 7 only please
  vector<int> spacebetweennodes;
  spacebetweennodes.resize(height);
  spacebetweennodes[height-1]=initialspacepernode;
  for(int a=height-2;a>=0;a--){
      spacebetweennodes[a]=spacebetweennodes[a+1]*2+1;
  }
  if(current!=NULL){
    Q.Join(current);//for every level
    for(int currlevel=0; currlevel<Height(root); currlevel++){
      int printthislevel=currlevel+1;
      int previousnnodes=pow(2,printthislevel-2);
      //print '_' and '|'
      if(printthislevel!=1){//don't print that before the root
        printspace(spacebetweennodes[currlevel+1]-(initialspacepernode/2)+1);
        if((spacebetweennodes[currlevel+1]-(initialspacepernode/2)+1)<1) printspace(1);
        for(int a=0;a<previousnnodes;a++){
          printunderscore((spacebetweennodes[currlevel]-(initialspacepernode/2))/2+1);
          printf("|");
          printunderscore((spacebetweennodes[currlevel]-(initialspacepernode/2))/2+1);
          printspace((spacebetweennodes[currlevel-1]-(initialspacepernode/2))/2+3);
        }
      }
      printf("\n");
      //print '/' and '\'
      if(printthislevel!=1){
        printspace(spacebetweennodes[currlevel+1]-(initialspacepernode/2));
        for(int a=0;a<previousnnodes;a++){
          printf("/");
          printspace(spacebetweennodes[currlevel]);
          printf("\\");
          printspace(spacebetweennodes[currlevel]);
        }
      }
      cout << endl;
      bool printtab=true;
      //for all possible nodes (including NULL)
      for(int c=0; c<pow(2,currlevel) && Q.isEmpty()==false; c++){
        current=Q.Front();
        Q.Leave();
        if(printtab) printspace(spacebetweennodes[currlevel+1]-(initialspacepernode/2));
        printtab=false;
        if(current!=NULL) {
          cout << current->RootData();
          Q.Join(current->Left());
          Q.Join(current->Right());
```

```
60          }
61          else {
62            cout << " ";//space indicates a NULL node
63            Q.Join(NULL);
64            Q.Join(NULL);
65          }
66          printspace(spacebetweennodes[currlevel]);
67        }
68        cout << endl;
69      }
70    }
71  }
```

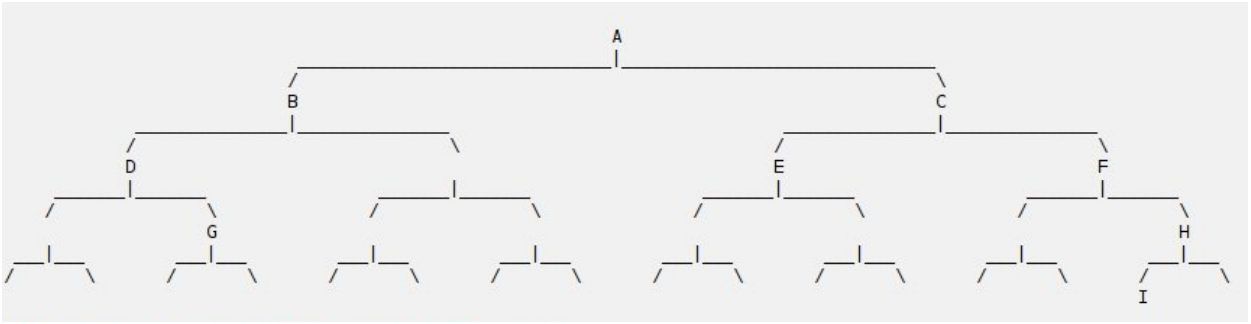


Figure 6.6: Printing a binary tree using the listing 6.12.

This chapter covered general binary trees. These trees have no other special rules and they can be simply built manually by the programmer. However, such generic trees cannot be used in applications where it depends on insertions and deletions (no insertion or deletion methods were explored in this chapter). In order to do that some extra rules or constraints must apply to the binary tree.

The same can be said for searching elements in the tree. Given the tree class seen in this chapter, the only way to search for an element is by traversing the tree, which is not more efficient than a linear data structure such as a linked-list. With extra rules for building the tree, one can find much more efficient ways to search the tree, which means that the search can be made faster than using a linear data structure.

In the next chapters we are going to study other forms of trees: arithmetic binary trees, binary search trees (also known as BSTs) and heaps.

6.8 - Exercises

1. Re-write the binary tree class, including the functions for the traversals. Test your code by building different trees (manually) and traversing them (printing the keys). Use complete and non-complete binary trees.

2. Write the pre-order non-recursive function and test it to see whether it yields the same result of the recursive one. Remember to use a Stack of `Tree*` (see listing 6.7).

3. Repeat the previous exercise for the other two iterative traversals, in-order and post-order.

4. Implement algorithm 7 in C++ using the Tree class defined before. Note that you need to change the queue type to a `Tree*` (use either the STL queue or modify our Queue class).

5. Copy (or download) the listing 6.12 code available on Stream. Print the existing tree. Modify the tree and print it again until you are comfortable with the format of printing. You can use this printing function to help to debug other binary tree functions/methods.

7 Special Binary Trees I: BST, AVL and Arithmetic Trees

In this chapter we are going to study some of the special cases for binary trees. The first special binary tree is one that represents an arithmetic expression. We can demonstrate how the standard traversals are related to the different notations (pre-fix, in-fix and post-fix). The second special binary tree in this chapter is the *binary search tree* (or *BST* for short). The BST can be used for efficient search. The very definition of a BST allow us to easily implement methods for insertion and deletion of nodes into an existing tree.

7.1 - Arithmetic Binary Trees

Binary trees can be used to hold arithmetic expression. Operands and operators are put in the same node types, but have a different place in the tree. An operator node will always have two children. The children may be operands (numbers) or other sub-trees that contain other expressions. The operands (numbers) are always leaves on the arithmetic binary tree.

The arithmetic tree can be interpreted as holding an expression where the children are going to be processed by their parents, which happen to be operators. The order is important: the left child contains the first operand and the right child contains the second operand. Figure 7.1 shows the binary tree equivalent to the arithmetic expression $(2 * (3 + 4)) / (7 - 5)$.

Using the standard traversals seen in the previous chapter will print the expression in three different forms. Traversing in-order prints the expression in in-fix format. Traversing the tree using the pre-order traversal prints the expression in pre-fix format. Finally, traversing the three using the post-order traversal prints the expression in post-fix format (in Reverse Polish Notation). The reader should carry out the three traversals for the tree in figure 7.1 and find that indeed the printouts should be:

- Pre-order traversal: $/ * 2 + 3\ 4 - 7\ 5$
- In-order traversal: $2 * 3 + 4 / 7 - 5$
- Post-order traversal: $2\ 3\ 4 + * 7\ 5 - /$

As you know the RPN expressions do not need parenthesis, but

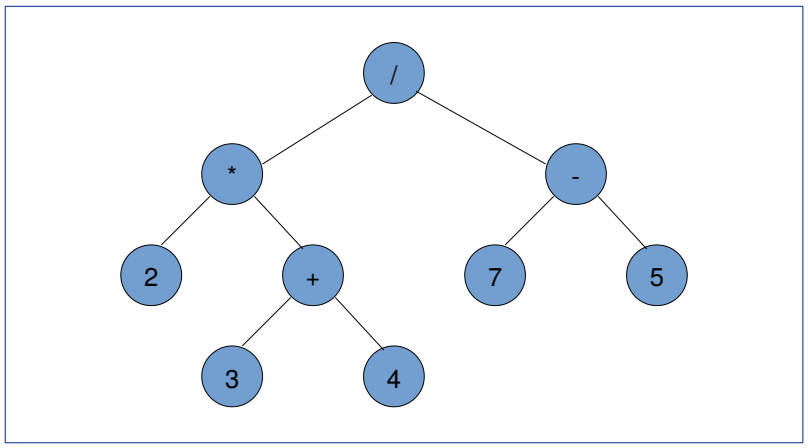


Figure 7.1: An arithmetic binary tree.

the other two do. The in-fix expression is ambiguous without the parenthesis. Fully parenthesised expressions can be created during the traversals. But for now it suffices to know that the operators and operands are printed in the correct order if we traverse the same binary tree using different traversals.

With that in mind, it is also possible to read an RPN expression and create the equivalent binary tree on the fly. Although we still do not have specific methods for insertion and deletion, we no longer need to rely on building the binary tree manually as we did in the previous chapter. Algorithm 8 shows how to build an arithmetic binary tree from an RPN expression.

Algorithm 8 Build an arithmetic tree.

```
1:  function CreateArithmeticTree(RPN expression E)        ▷ RPN from a file
2:      declare Tree *T1, *T2, *T; declare char x;          ▷ tree pointers and a char
3:      declare a Stack S;                                  ▷ S contains tree pointers, not chars
4:      while (E has data) do
5:          x = read next item from E;
6:          if (x is a number) then
7:              create a new Tree with (x,NULL,NULL);
8:              Push the Tree node onto S;
9:          end if
10:         if (x is an operator) then
11:             T1 gets Top of S;
12:             Pop S;
13:             T2 gets Top of S;
14:             Pop S;
15:             create a new Tree with (x,T2,T1);       ▷ careful with the order of T1 and T2
16:             Push the Tree node onto S;
17:         end if
18:     end while
19:     make the root of the arithmetic tree the Top of S;
20: end function
```

The algorithm considers a Tree node composed of one character (this is the key of the node) and two pointers, `left` and `right`. A new node can be created (memory allocation with arguments) by using `new`, which calls the node constructor. In the previous chapter we have seen that the constructor has three parameters, and they are the three members of the Tree node. Recall that the simple class `Tree` has no method to modify its pointers, and therefore the general binary tree has to be built from the bottom to the root. The algorithm below does just that, creating the leaves first (as we know both the `left` and `right` pointer should be NULL in this case) and creating the parents only once the pointers to the children are known. The difference is that algorithm 8 does that in an ordered way, linking existing sub-trees.

To understand algorithm 8 refer to the slides for a walk through example. Also, make sure you can build your own arithmetic tree (see exercises 1 and 2 at the end of this chapter).

7.2 - Binary Search Trees (BST)

Another important special binary tree is the Binary Search Tree (BST). These trees are important because they make it possible to search for keys very efficiently. However, the keys have restrictions. In a BST:

- the key for each node is unique (no duplicated keys in the BST)
- for every node of the BST all children's keys to the **left** are **smaller**, and all keys to the **right** are **bigger**.

These simple rules allow for the creating of simple insertion, deletion and searching algorithms. Searching for a key from the root of the BST involves a simple decision: go to the left or to the right sub-trees depending on the value of the search key compared to the key of the current node. The key should be found before reaching a NULL pointer, otherwise it means that the searched key is not present in the BST. Figure 7.2 shows a simple BST. The reader should try to find a certain key starting from the root and navigating the tree depending on the value of the searched key.

7.2.1 - BST Search

One can devise algorithms for BSTs that are recursive or iterative. We are going to use a simple iteration based on a temporary pointer called `current`. This is similar to the search we have done with linked-lists, except that instead of always making `current=current->next` (linear), this time we have an option to make either `current=current->leftptr` or
`current=current->rightptr`. Algorithm 9 shows this simple search.

Note that the simple BST search could be implemented as a *function* or as a *method*. The reason for that is the fact that the Tree class used in the previous chapter represents a node, so each node of a tree can be accessed by the three methods `RootData()`, `Left()` and `Right()` (see listing 6.1). The implementation is left as an exercise (see exercises 3

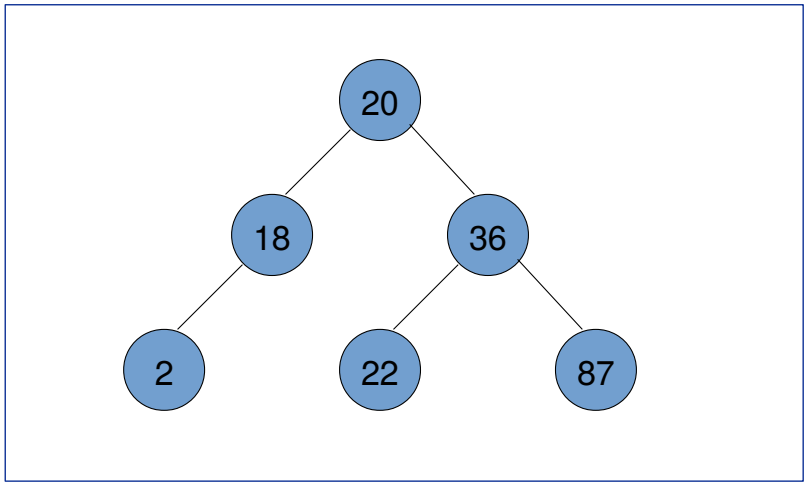


Figure 7.2: A Binary Search Tree.

to 5). A recursive alternative can also be found in the sample code folders on Stream.

7.2.2 - BST Insertion

Another important algorithm for the BST is the insertion. In this chapter we are going to implement the insertion in a simplistic way: always inserting the new node as a *leaf*. This is a simplistic approach because the BST is only efficient in being used for search if it is *balanced*. Randomly inserting nodes as leaves makes the tree unbalanced. We are going to study ways of balancing trees later in this chapter.

In order to insert a node as a leaf, we have to resort to the same process that we have seen in searching, i.e., the new item needs to be searched and follow the tree until it reaches a NULL pointer. If during this search we find that the key already exists, then the algorithm should not carry out the insertion (the BST has unique keys). Otherwise, reaching the NULL pointer shows where to place the new node.

As seen before with the search algorithm, the insertion could also be implemented either as a function or a method. Listing 7.1 shows a C++ *method* for inserting a new node in the BST. The implementation is recursive, rather than using a while loop as we did in algorithm 9 (iterative). The method only works if the BST already has at least one node, so the first node of the BST has to be created manually at `main()` (one can use the constructor directly, where the two pointers `left` and `right` are initially `NULL`). The method can also be modified to cater for a NULL tree.

When the search starts, the first node is the root of the BST. Therefore, the item can be compared directly to `data` (without any reference to `tree->` as the `data` belongs to the root in this case).

If the `item` is bigger than `data`, then it follows the right side. The `rightptr` can be either NULL (we reached a leaf) or not NULL. If it is NULL, we create a new tree, linking it with the parent (`rightptr`

Algorithm 9 Search a BST.

```
1:  function SEARCH(given the root of a BST Tree* R and a Key K)
2:      declare a Tree* current;
3:      if (K is the same as the key in R) then
4:          Found item in the root of the BST;
5:          return;
6:      else
7:          if (K is smaller than the key at R ) then
8:              current = left of R;
9:          end if
10:         if (K is bigger than the key at R) then
11:             current = right of R;
12:         end if
13:         while (current is not NULL) do
14:             if (K == the key in current) then
15:                 Found item at current;
16:                 return;
17:             else
18:                 if (K > key in current) then
19:                     current = right of current;
20:                 else
21:                     if (K < key in current) then
22:                         current = left of current;
23:                     end if
24:                 end if
25:             end if
26:         end while
27:         Item K is not in this BST;
28:     end if
29: end function
```

is changed to point to the newly created node). Otherwise, call a recursion on the right node with the same `item` as a parameter. Similarly, if the `item` is smaller than the `data`, it follows the left side of the tree. At any point in the recursion it might find that `item == data`, in which case the insertion terminates with an error message.

Listing 7.1: Insertion for a BST in the form of a recursive method.

```
1  void Tree::Insert(int item) {
2    // BST must already contain at least one item
3    if (item > data) {  // move to the right
4      if (rightptr == NULL) {
5        rightptr = new Tree(item, NULL, NULL);
6      }
7      else {rightptr->Insert(item);}
8    }
9    else {
10     if (item < data) {  // move to the left
```

```
11        if (leftptr == NULL) {
12          leftptr = new Tree(item, NULL, NULL);
13        }
14        else { leftptr->Insert(item);}
15      }
16      else {
17        if (item == data) {  // should be unique
18          printf("Error: key is not unique");
19          return;
20        }
21      }
22    }
23  }
```

7.2.3 - BST deletion

The deletion is the most difficult of the three algorithms. This is so because when deleting a node that already has children, the connection between the BST and the sub-trees below the deleted node are lost. The operation has to be carefully coded to avoid changing the BST properties, i.e., smaller on the left and bigger on the right side for each node. Analysing the scenarios that could happen during the deletion we can come up with three cases:

1. The node to be deleted is a leaf.
2. The node to be deleted has only one child.
3. The node to be deleted has two children.

7.2.4 - Case 1: leaf

This is the easiest case. The parent of the node to be deleted needs to point to NULL, then the node can be deleted. No adjustments need to be done. Figure 7.3 shows the case.

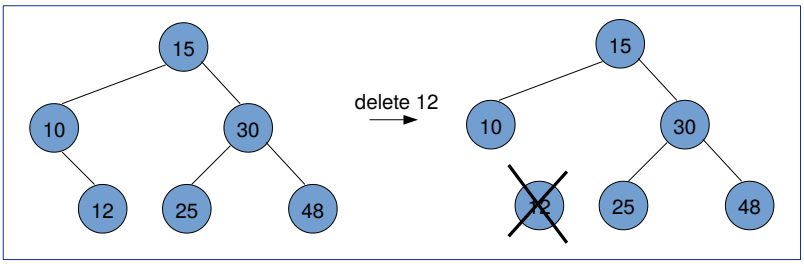


Figure 7.3: Case 1 of the deletion for BSTs.

7.2.5 - Case 2: one child

In this case, the link of the parent (of the node to be deleted) needs to be changed to the child of the node to be deleted. Then the node can

be deleted safely. Is that going to preserve the properties of the BST? Yes.

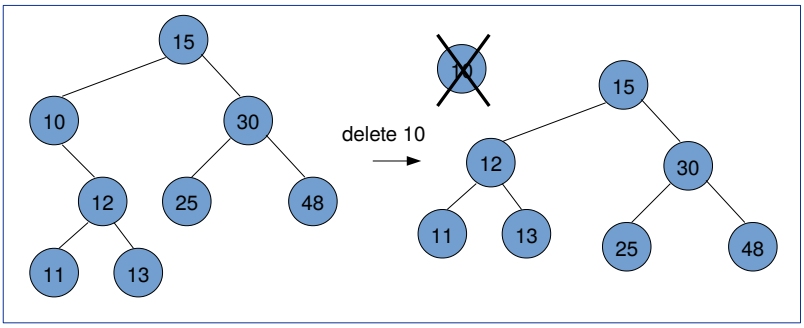


Figure 7.4: Case 2 of the deletion for BSTs.

We can define four sub-cases:

1. The node to be deleted is on the left of its parent, its child on its right.
2. The node to be deleted is on the left of its parent, its child on its left.
3. The node to be deleted is on the right of its parent, its child on its right.
4. The node to be deleted is on the right of its parent, its child on its left.

In all four cases we can link the child of the node to be deleted with the parent of the node to be deleted. The properties will always be preserved. In the two first cases, the keys of the sub-tree are always smaller than the parent, and they should be linked to the left side of the parent. On the last two cases the opposite is true (they are bigger than the parent, Figure 7.4).

7.2.6 - Case 3: two children

This is the most difficult of the three cases. There are two options, one is to change the *links*, the other to replace the *values*. We are going to use the second option. This implies using recursion, as replacing values do not delete any nodes yet. Using recursion, when case 3 occurs then the deletion becomes either cases 1 or 2. Case 3 cannot recur back to case 3 because the largest or the smallest value in the sub-tree *necessarily has only one child or no children*. This is because of the very basic property of the BST. So before tackling the problem of implementing case 3, let us discuss how to find the largest or the smallest key in a sub-tree.

Finding the largest key on the left sub-tree

Finding the largest key on a BST is equivalent to following the right side all the way to the last node. By definition, the largest key should be the last node on the right side. The last node is either a leaf

or has only a left child. A simple code to return the pointer to a Tree node that contains the largest key on the left is shown in listing 7.2.

Listing 7.2: Finding the largest key on the left sub-tree.

```
1 Tree * Tree::SearchLargestOnLeft() {
2   Tree * current=leftprt;
3   if(current==NULL) return NULL;
4   while (current->rightptr != NULL) {
5     current=current->rightptr;
6   }
7   return current;
8 }
```

Finding the smallest key on the right sub-tree

Finding the smallest key on a BST is equivalent to following the left side all the way to the last node. By definition, the smallest key should be the last node on the left side. The last node is either a leaf or has only a right child. A simple code to return the pointer to a Tree node that contains the smallest key on the right is shown in listing 7.3.

Listing 7.3: Finding the largest key on the left sub-tree.

```
1 Tree * Tree::SearchSmallestonRight() {
2   Tree * current=rightprt;
3   if(current==NULL) return NULL;
4   while (current->leftptr != NULL) {
5     current=current->leftptr;
6   }
7   return current;
8 }
```

The reader should use the listings 7.2 and 7.3 to follow different nodes in different BSTs to be convinced that they work accordingly.

Back to the deletion problem

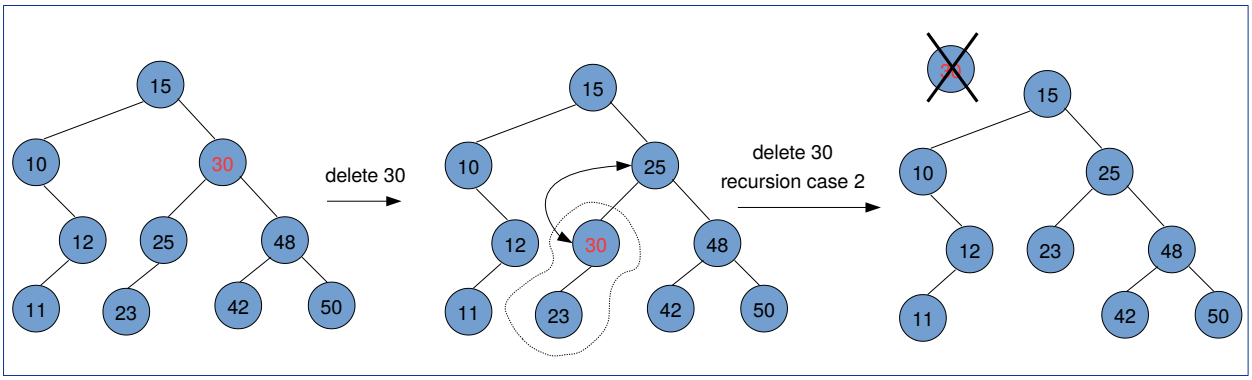


Figure 7.5: Case 3 of the deletion for BSTs: using the largest on the left.

Schematically we can simply use the replacement strategy and recur to cases 1 or 2. Figure 7.5 shows a simple case where the largest value on the left (25) is swapped with the node to delete (30). Temporarily the BST sub-tree with root 30 is violating the BST properties, but if the correct recursion follows the node violating the property is eventually deleted.

Figure 7.6 shows the same deletion of key 30, but this time using the smallest key on the right side of 30. Again, temporarily the BST violates the properties of the keys. This time the recursion gets case 1 and the node with key 30 is deleted.

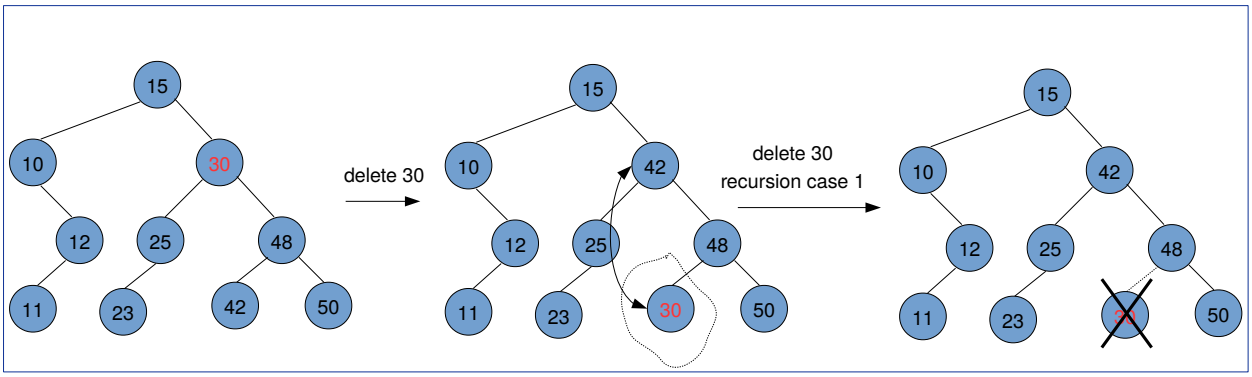


Figure 7.6: Case 3 of the deletion for BSTs: using the smallest on the right.

The recursive deletion function can now be written, although the recursion only occurs in case 3. Listing 7.4 presents the code. The code implies finding (using search) the pointer to the node to be deleted. The pointer is passed to the tree, as well as the pointer to the root of the BST.

Listing 7.4: The BST deletion function covering the 3 cases.

```
1  bool DeleteNode(Tree* nodetodelete,Tree* parent) {
2    if (nodetodelete == NULL) {
3      printf("DEL: didn't find item\n");
4        return false;//if didn't find the match
5    }
6    //case 1: the node is a leaf////////////////////////////////
7    if(nodetodelete->Right()==NULL && nodetodelete->Left()==NULL){
8       if(parent==NULL) {
9         delete nodetodelete;//in this case it is the root
10      }
11      if(parent->Right()){
12        if(parent->Right()->RootData()==nodetodelete->RootData()) {
13          parent->UpdateRight(NULL);
14          delete nodetodelete;//delete the node itself
15          printf("case 1: is a right leaf\n");
16          return true;
17        }
18      }
```

```
19      if(parent->Left()){
20        if(parent->Left()->RootData()==nodetodelete->RootData()) {
21          parent->UpdateLeft(NULL);
22          delete nodetodelete;//delete the node itself
23          printf("case 1: is a left leaf\n");
24          return true;
25        }
26      }
27      return false;//if it reaches here , something went wrong...
28    }
29    //case 2: the node has one child/////////////
30    if (nodetodelete->Right()==NULL && nodetodelete->Left()!=NULL) {
31      printf("DEL:Case 2, has one child on the left\n");
32      if(parent->Right()==nodetodelete){
33        parent->UpdateRight(nodetodelete->Left());
34      }
35      else {
36        parent->UpdateLeft(nodetodelete->Left());
37      }
38      delete nodetodelete;
39      return true;
40    }
41    else if (nodetodelete->Right()!=NULL && nodetodelete->Left()==NULL){
42      printf("DEL:Case 2, has one child on the right\n");
43      if(parent->Right()==nodetodelete){
44        parent->UpdateRight(nodetodelete->Right());
45      }
46      else{
47        parent->UpdateLeft(nodetodelete->Right());
48      }
49      delete nodetodelete;
50      return true;
51    }
52    //case 3: the node has two children/////////////////
53    if(nodetodelete->Right()!=NULL && nodetodelete->Left()!=NULL){
54      printf("DEL:Case 3 two children\n");
55      Tree* largestleft=NULL;
56      Tree* parentlargest=NULL;
57      //get the largestleft pointer and the parentlargest pointer
58      largestleft=nodetodelete->SearchLargestOnLeft(parentlargest);
59      //SWAP values between nodetodelete and largest to the left
60      int temp=largestleft->RootData();
61      largestleft->UpdateValue(nodetodelete->RootData());
62      nodetodelete->UpdateValue(temp);
63      return (DeleteNode(largestleft,parentlargest));//RECURSION!
64    }
65  }
```

7.2.7 - Balance in BSTs

Random insertion and deletions can lead to an unbalanced BST. The definition of a perfectly balanced tree is that the tree is as complete as possible (all leafs in the same level).

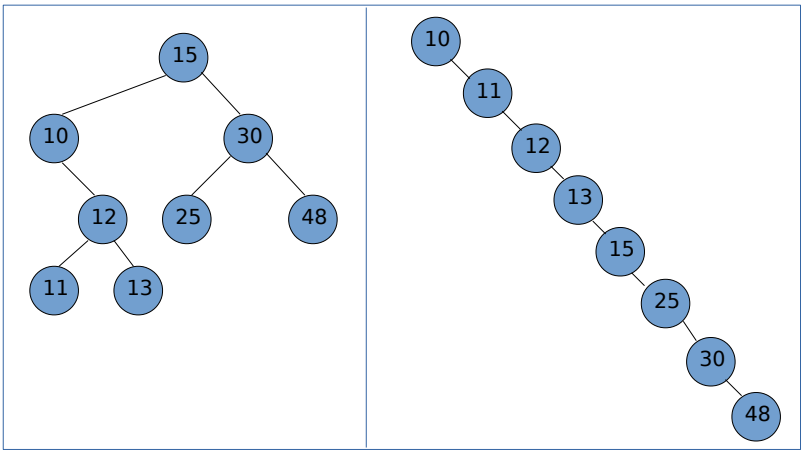


Figure 7.7: BSTs: the left tree is reasonably balanced, while the one on the right is unbalanced. The tree on the right renders a very inefficient search, similar to that of a linked-list.

If the BST is so unbalanced as to look like a linear data structure, the search is going to be as slow as a search in a linked-list. In fact compare the two BSTs in figure 7.7 to visualise that. For every key, try to use the search algorithm and count the number of movements of the current pointer. On average the unbalanced BST will have many more movements than a balanced BST with the same content (same keys).

There are general solutions for balancing BSTs, e.g. Adelson-Vleskii and Landis (so-called AVL trees) present a way to insert and delete nodes while keeping the BST balanced. AVL trees are discussed in the next section.

7.3 - AVL trees

As seen before, BSTs are better when balanced. It is not possible to completely balance a binary tree (unless the tree is complete), but the best possible way is to keep the differences of height between sub-trees to a minimum. Adelson-Vleskii and Landis (AVL) trees are BSTs where the height difference between any sub-trees is at most 1. When inserting or deleting a node, sub-trees need to be rearranged in order to keep the difference at most 1. This implies that insertion is not done in the simplistic way where any new node becomes a leaf (listing 7.1) In order to appreciate the difference, compare the two BST trees in figure 7.8. The one on the left is clearly unbalanced (there is a difference in height of 2), while the one on the right side has no more than difference of 1 in any sub-tree.

The insertion problem becomes now a problem of analysing the difference of height between sub-trees and finding a solution with few changes. One can assume that the algorithm starts with a balanced

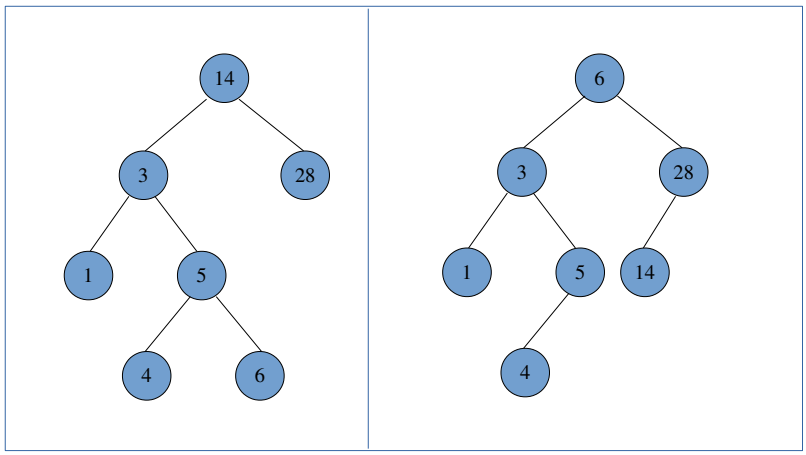


Figure 7.8: AVL trees: a) unbalanced BST b) balanced (AVL tree).

tree (after all this same algorithm would have been used since the first node was inserted), and adding a single node to the tree may make the tree unbalanced. There are three possible general state for the AVL tree that one can encounter after trying to insert a new node. The three cases are summarised in figure 7.9. Note that we do not need all the details of each sub-tree. If there is a difference of one in height, we do not need to know *where* the difference is in the sub-tree because it is assumed that any sub-tree is also an AVL tree. We only need to know if the difference is on the right or on the left.

We want to know what happens to the AVL tree if a node is inserted. We keep the simple approach of inserting as a leaf, but we have to fix the height if the property of the AVL is violated. Suppose that the new node ends up on the left sub-tree in figure 7.9. Three possible cases arise:

1. $HL = HR + 1$ (still an AVL tree)
2. $HL = HR$ (still an AVL tree)
3. $HL = HR + 2$ (**rebalance**)

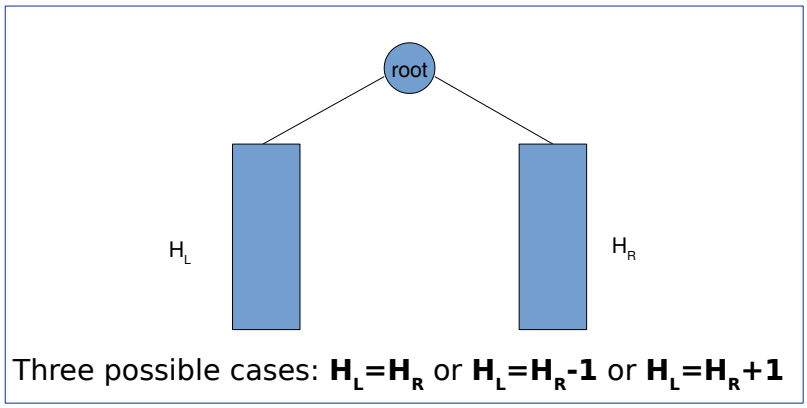


Figure 7.9: AVL trees: three possible cases for a balanced tree.

Only the last case needs action. The other two would keep an AVL tree. The cases for rebalancing can be summarised into two main

cases. Both cases can be resolved by rotating the tree in a certain way in order to rebalance it.

The rebalancing is not necessarily done at the root of the AVL, but at the *deepest root of a sub-tree that becomes unbalanced*. The insertion should insert the new node as a leaf, and climb to its parent to check whether the sub-tree is still balanced. The checking should climb to its parent and so on, until a certain node is found to be unbalanced. Only then should the balancing be applied. The rotation of the deepest unbalanced root of a sub-tree suffices to balance the entire AVL tree.

In terms of programming, there are actually 4 cases, as we only considered one side of the unbalanced situation (the 4 cases can be classified as 2 pairs, each symmetric to each other). We discuss the rebalancing cases below, considering that the left sub-tree in relation to the root is the one receiving the new node (remember that the root is the root of a sub-tree, not necessarily of the entire AVL tree).

7.3.1 - AVL rebalance case 1

The first case is where the new node is created as a leaf of the left sub-tree of G (sub-tree T1). It means that the height of the left sub-tree is one node too deep, so the tree has to be rotated. G becomes the new root, and the sub-trees T2 and T3 become children of the original root, M (figure 7.10).

The other symmetric sub-case 1 (needed for the programming) would occur if the new node is inserted at the other extremity of the tree, so it would be a mirrored version of the AVL in figure 7.10.

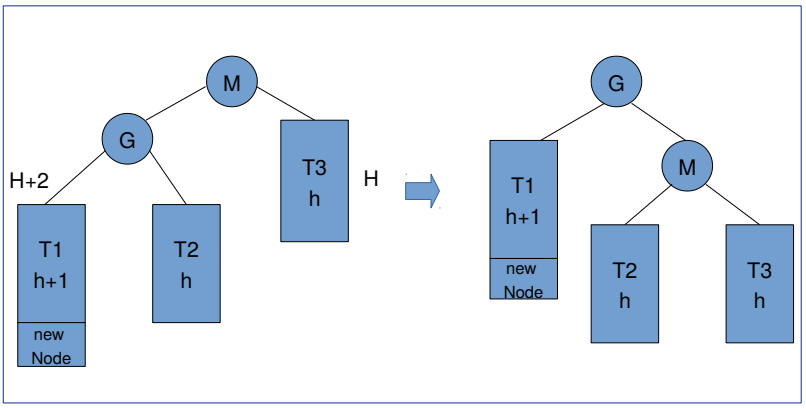


Figure 7.10: AVL rebalance case 1.

7.3.2 - AVL rebalance case 2

Case 2 covers the scenario where the place for the new node is found on the middle of the tree, i.e., either under sub-tree T2 or T3. In this case the rotation is more complicated than case 1. Node K, which is the parent of sub-trees T2 and T3, becomes the new root. As G, T1 and T2 have all keys smaller than K, they are put on the left side of K. On the other hand, M, T3 and T4 have keys that are bigger than K, and therefore they are relocated to the right of K (figure 7.11). Note

that the rotation is exactly the same whether the new node is inserted in T2 or inserted in T3.

The pair sub-case of case 2 is the mirrored image of figure 7.11, when the new node would be located at the right sub-trees of the original root of the AVL tree.

In terms of programming, the rotation involves the update of only four pointers: the two pointers for K, the left pointer for M and the right pointer for G.

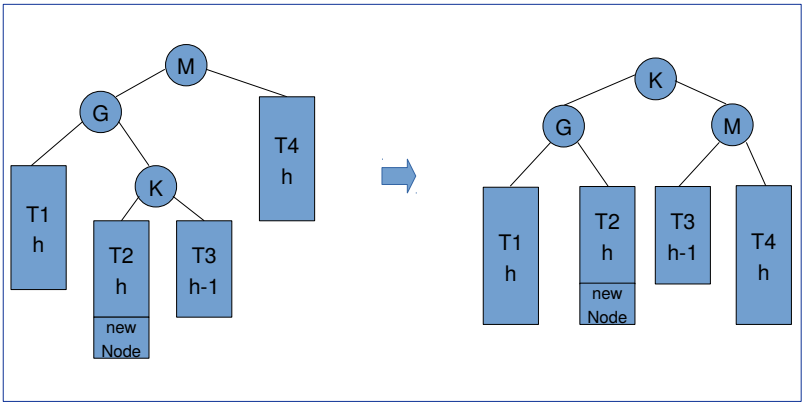


Figure 7.11: AVL rebalance case 2.

7.3.3 - AVL balancing: complete examples with numerical keys

For the first example, the sequence of insertions is: 20, 17, 15, 16, 4, 3, 8, 1, 9, 7, and 5. Figure 7.12 shows the sequence of insertion and rotations. Inserting 20 and 17 does not require rebalancing. When 15 is inserted, it requires case 1 rebalancing. Inserting 16 and 4 does not require rebalancing. When 3 is inserted it requires case 1 again. Finally, inserting 8,1,9 and 7 does not require rebalancing. When 5 is inserted it requires case 2 rotation. Try to compare figures 7.10 and 7.11 with figure 7.12 to see the sub-tree locations.

Sometimes the node that is unbalanced is not the root of the BST, but some other node in the middle. The correct way of rebalancing the BST is to traverse the tree upwards, following the parent of the node that was just inserted. In figure 7.13 the sequence of insertions is: 11, 10, 9, 8, 7, 6, 5, 4, 3, and 2. Note that when the tree becomes unbalanced, nodes in the middle (marked with red) are the parents of parents of parents of the inserted node. In this case, the rebalancing of the tree should be done in the scope of that sub-tree, not for the entire BST.

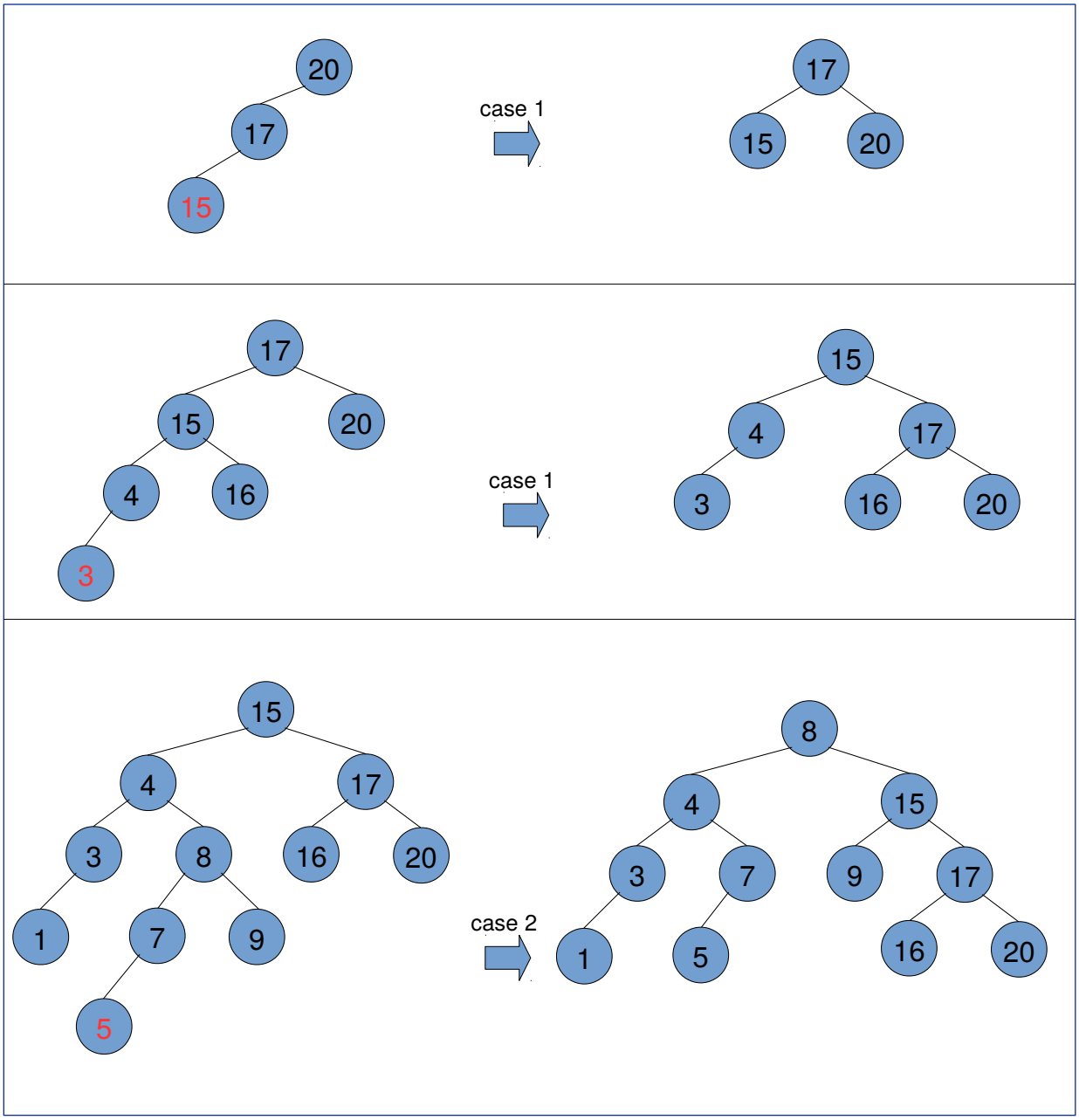


Figure 7.12: AVL insertions and rotations.

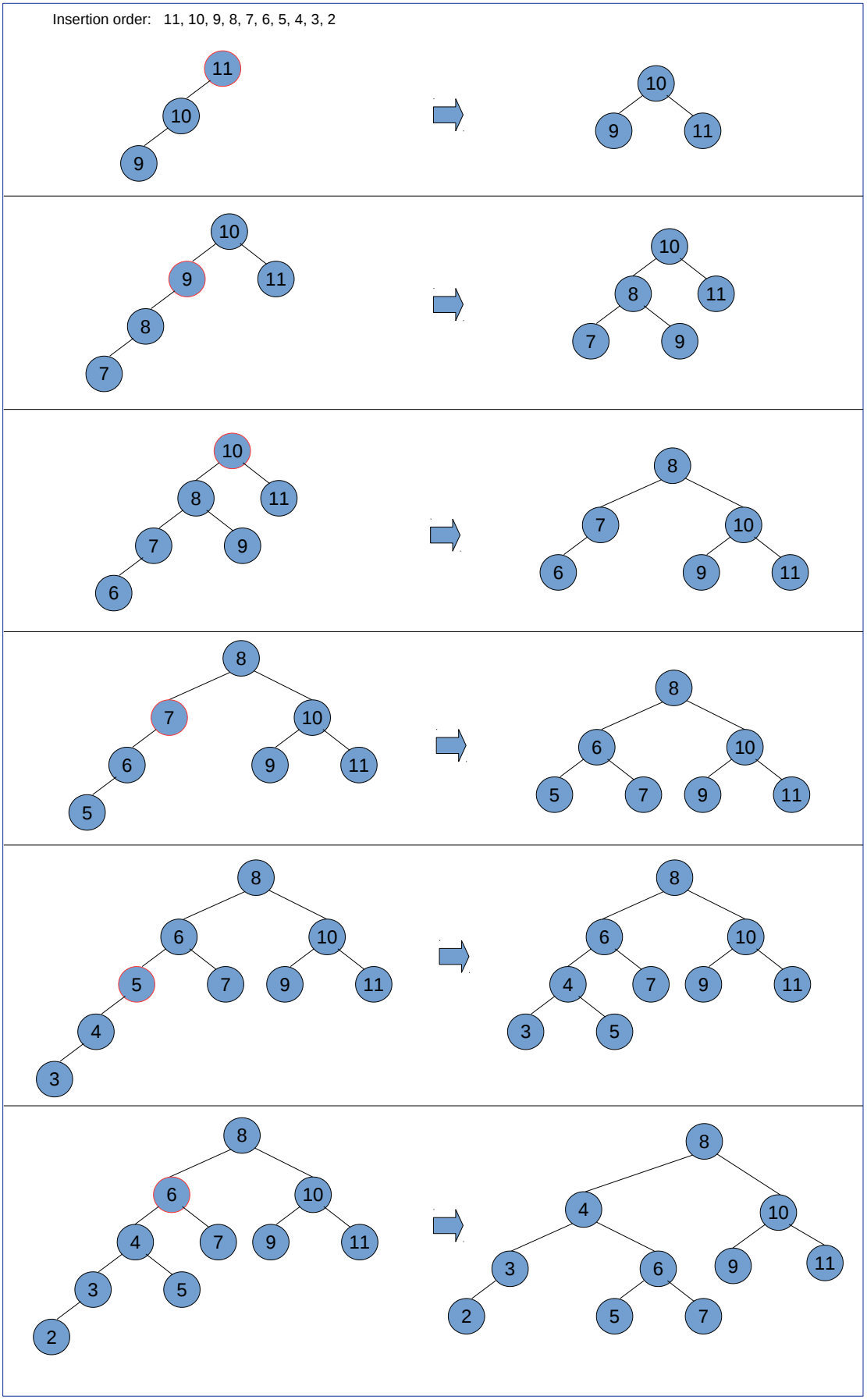


Figure 7.13: Example 2: AVL insertions and rotations.

7.4 - Exercises

1. Using algorithm 8 build the corresponding arithmetic trees (draw them schematically) for the following expressions:

   (a) 3 4 +

   (b) 2 4 + 3 *

   (c) 1 2 3 4 + - * 5 /

   Traverse the arithmetic trees manually using the post-order traversal to see if you get the original RPN expressions.

2. Implement algorithm 8 in C++, using the Stack class from listings 3.5 and 3.6. Modify the Stack to hold a Tree*. The RPN can be furnished to the program as a string.

3. Implement algorithm 9 as a *method* for the BST.

4. Implement algorithm 9 as a *function* for the BST.

5. Modify algorithm 9 to use recursion. Implement the recursive version of the algorithm as a *method* for the BST.

6. Based in listing 7.1 write a function for the insertion of a new node to an existing BST.

7. Modify listing 7.1 to an iterative method (rather than recursive).

8. Consider a BST. What traverse should be used if you want to print all the keys in sorted order (increasing keys)?

9. Implement a *method* for the BST based on the deletion *function* in listing 7.4. The search can be incorporated into the method itself, or a search method that returns the pointer of the node to be deleted can be implemented first.

10. In the AVL tree of figure 7.8, insert the following new nodes in this sequence: 10, 20, 25, 2, 7, and 35. Discuss the need for rotation in each case, and fix the tree with the appropriate rotations.

11. Construct an AVL tree using the following sequence: 1,3,4,5,6,14,28. Is the shape of the tree different than figure 7.8?

12. Construct an AVL tree using the following sequence: 28,5,3,4,6,1,14. Is the shape of the tree different than figure 7.8?

8 Special Binary Trees II: Heaps and B-Trees

This chapter covers another two types of special trees: Heaps and B-Trees.

Heaps are also binary trees, but with special rules about the key order, and requirements about the completion of the tree. Due to these requirements it is possible to implement Heaps using *arrays* or *vectors*.

B-trees are a generalisation of BSTs for cases where each node contains more than two keys. The B-Tree nodes can contain multiple records instead of a single record. Each node can have multiple children rather than only 2.

8.1 - Heaps

Heaps [1] are binary trees where the tree has to be as complete as possible. Therefore, nodes need to be inserted from top down, from left to right. The order of the keys is also a requirement: either the heap keys are always larger or equal to the keys of its own children, or they are smaller or equal depending on the order we want. This makes an ordered way of traversing the tree, but most importantly the root always contains the biggest (or the smallest) key. In this study guide we are always going to deal with decreasing keys, i.e., the largest key is in the root of the heap. Note that keys are allowed to repeat (unlike in a BST). In summary a Heap can be defined as:

[1] Heaps are sometimes called priority queues. There are many types of heaps, but they all follow the same definitions that we explore in this chapter. In the context of this study guide, the heaps are going to have the maximum key value located in the root of the tree. This form of heap is also called *max-heap*.

- the value of any key $\geqslant$ value of the children's keys.
- the tree is perfectly balanced (almost complete).
- any new node is inserted far left within the same level until that level is complete.

Because of the completion requirements one can implement Heaps using arrays or vectors. This is due to the fact that no gaps will exist between the root and the last element. Moreover, it is possible to get simple formulas to find out the indices of parents and children. Without the need to relocate pointers, Heaps can be very fast to insert and delete, but there is a specific way in which to implement these functions. Figure 8.1 shows a heap implemented as an array. Note that the root is always index 0, followed by its children indices 1 and

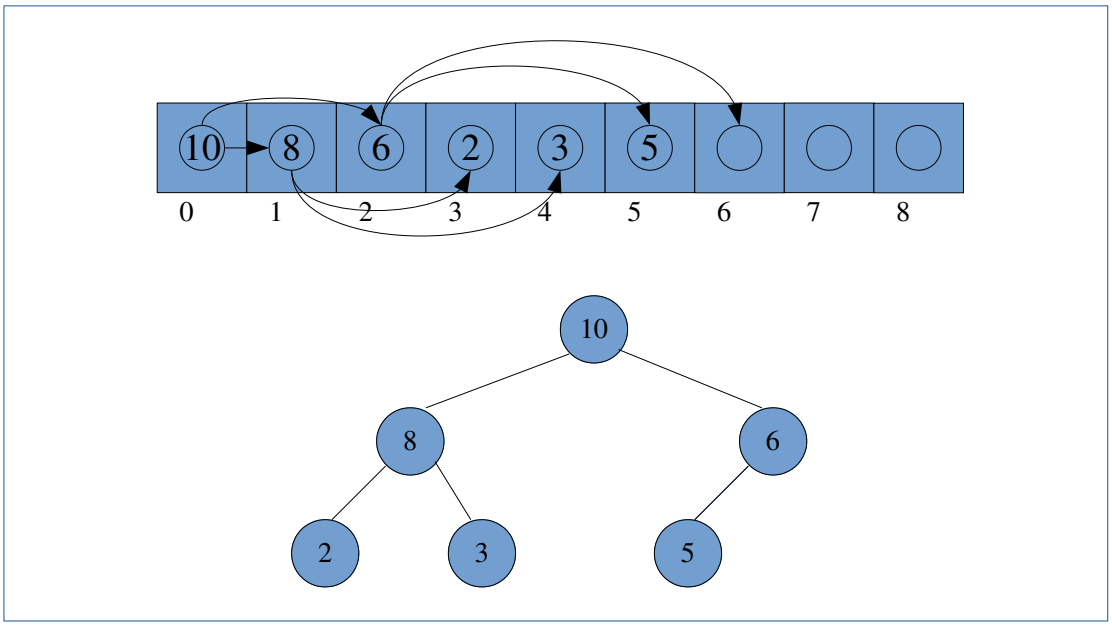


Figure 8.1: Heap implemented as an array. Alternatively it could be a vector instead, with very little modification on the methods shown in the next sections.

2. The children located at 1 has two children, 3 and 4. The node located at 2 has two children too , 5 and 6. Following the same rules, it is trivial to prove that given an index i, its children indices are:

$$leftchild = 2i + 1 \tag{8.1}$$

$$rightchild = 2i + 2 \tag{8.2}$$

Moreover, given a child index i it is easy to find its parent by (integer division):

$$parent = \frac{i-1}{2} \tag{8.3}$$

For example, inserting any new node into the Heap in figure 8.1 would have to use index 6 because this is the next index available (in the tree it is the right child of node 6). However, this could violate the Heap requirements if the new node has a key that is larger than 6. In this case the Heap needs to be fixed [2]. The fixing procedure is very different than the one adopted for an AVL tree. The Heap is always balanced, the problem lies with the order of the keys. The culprit of violating the properties is the key in the node that was inserted or removed. The easiest way to fix the heap is to swap the key of the new node with the key of its parent. A similar procedure can be used to resolve issues when deleting nodes from the Heap. Both algorithms are very fast, as the tree only requires a few key swaps to fix it.

[2] Fixing the heap after inserting or deleting an element is also called "to heapify" the heap, or even re-heapnise.

There is no efficient search procedure for the Heap. The searching involves traversing the array itself. However, searching a heap's key in this manner is rarely needed. Usually when using a Heap we are only interested in getting the largest key value, and its index is always zero (because the node is the root).

The Heap can be implemented as a class, with either an array or a vector inside (listing 8.1).

Listing 8.1: The Heap class.

```
1  class Heap {
2    private:
3      int data[10];//can change that to dynamic
4      int last;//last index
5    public:
6      Heap(){last=-1;};  // constructor
7      ~Heap() { };//destructor
8      void InsertHeap( int newthing);
9      bool Delete(int item);
10 };
```

8.1.1 - Insertions

Insertions require that the new key is checked with its parent. If it is bigger then a swap is carried out. However, the swapped key might still be bigger than its parent, requiring another swap. The algorithm stops when we reach the root of the Heap, in which case the key is the biggest in the Heap. The implementation of a simple insertion with the fixing is in listing 8.2

The reason the fixing is fast is the fact that we can easily find the parent index from any point in the tree. [3] Also, climbing the tree all the way to the root would require no more than $log(N)$ swaps.

Listing 8.2 works as follows: initially, the new key is inserted to the next available position in the array. The variable **last** is updated and the variables for the indices of the new node and the parent are updated (statements 5 and 6) . The while loop repeatedly works out if the variable swapping is true or false, i.e., if there was a swap

[3] In this case:
$parentindex = (6-1)/2 = 2$

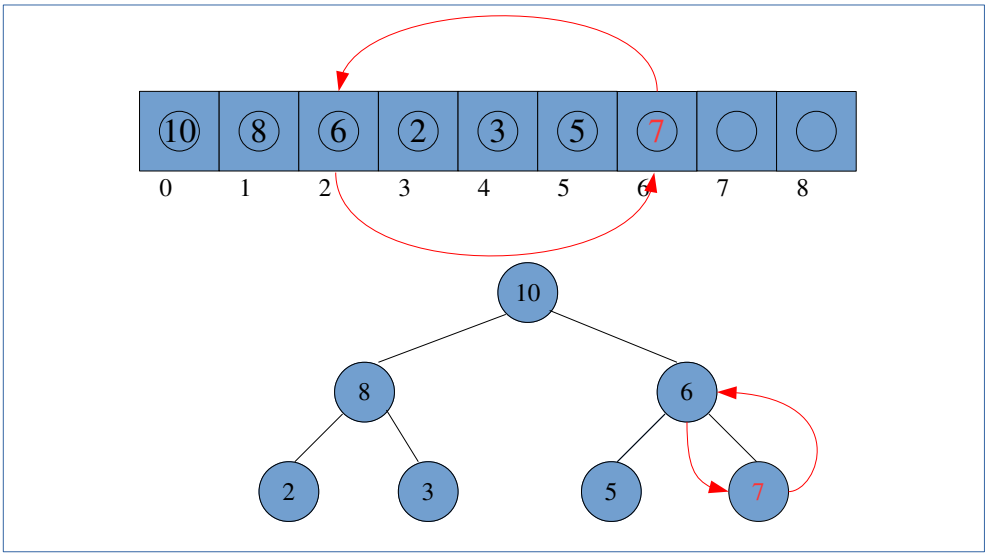


Figure 8.2: Inserting a new key into the Heap. The diagram shows Key 7 being inserted in the heap of figure 8.1.

Listing 8.2: The Heap insertion method.

```
1  void Heap::InsertHeap( int newthing){
2    data[last+1]=newthing;//add to the end
3    last=last+1;
4    if (last==0) return;//only one item in Heap
5    int child_index=last;
6    int par_index=0;
7    bool swapping=true;
8    while (swapping==true ){//fix the heap
9      swapping=false;
10     if(child_index%2==0) par_index=child_index/2-1;//right
11     else par_index=child_index/2;//left
12     if(par_index>=0){
13       if(data[child_index]>data[par_index]) {
14         swap(data[child_index],data[par_index]);
15         swapping=true;
16         child_index=par_index;
17       }
18     }
19   }
20 }
```

between keys of by checking the new key against the key of its parent. The loop continues while the swapping variable is true, making the new key climb levels of the tree. If the index of the parent becomes negative, it means that the new key reached the root of the heap and cannot climb any further. This is checked by statement 12.

See figure 8.2 for an example of insertion.

8.1.2 - Deletions

As mentioned before, there is usually no search involved in a Heap. In the same way, it is more common to delete the root and nothing else. The reader should convince himself that to delete at random, a search would have to be carried out first. Only deleting the root would not require any search. To recover the information from the root and delete the root would be analogous to the Stack `Top()` and `Pop()`, as the index of the root is always `[0]`.

In listing 8.3 the implementation of a method to delete the root is presented. The procedure is simple: keep the index of the last valid element at the variable called `last`, it is easy to swap the element from the root (at index `[0]`) with the last element (at index `[last]`).

Once this is done, we can make `last = last - 1;`. This effectively deletes the last element, keeping the Heap balanced. However, the root's key is now violating the Heap property. We need a loop that keeps swapping the parent (starting from the root) with its *biggest*

child, until done. This is achieved with the `while` loop in listing 8.3.

Listing 8.3: The Heap deletion method.

```
1  void Heap::DeleteRoot(){
2    if(last<0) return;
3    unsigned int deletedvalue=data[0];//the root
4    data[0]=data[last];
5    data[last]=0;//deleting...
6    last=last-1;
7    int parindex=0;//root at the moment
8    int leftindex=parindex*2+1;//left child
9    int rightindex=parindex*2+2;//right child
10   //Fixing the Heap
11   while (data[parindex]<data[leftindex]||data[parindex]<
         data[rightindex]){
12   //PROBLEM: what happens if last=1 or last=0? One or
         both indices for the children may be invalid
13     if (data[rightindex]<data[leftindex]){//follow left
14       swap(&data[leftindex],&data[parindex]);
15       parindex=leftindex;
16     }
17     else{//else follow right
18       swap(&data[rightindex],&data[parindex]);
19
20       parindex=rightindex;
21     }
22     leftindex=parindex*2+1;
23     rightindex=parindex*2+2;
24     if(leftindex>last) { break;}
25     else {
26       if(rightindex>last){
27         if (data[parindex]<data[leftindex]) {
28           swap(&data[parindex],&data[leftindex]);
29         }
30         break;
31       }
32     }
33   }
34 }
```

The listing 8.3 is not perfect. If the variable **`last`** is 1 or 0, the indices for one or for both children of the root may be invalid. This is not a problem when using arrays because the values are only deleted "virtually" by the state of the variable **`last`**, but when using a vector this may become a problem. Some extra checks are needed to improve this code, and this is left as an exercise.

An example of deletion is shown in figure 8.3. In this example the root (key=10) is deleted. Firstly 10 is swapped with 6 (the last element), then `last=last-1;`. Now we need to fix the heap starting from the root and working down. Key 6 is smaller than both 7 and 8. To delete properly, the key 6 requires to be swapped with its largest child, 8. After this swap, the key 6 is checked against 2 and 3. Both are smaller than 6, terminating the loop.

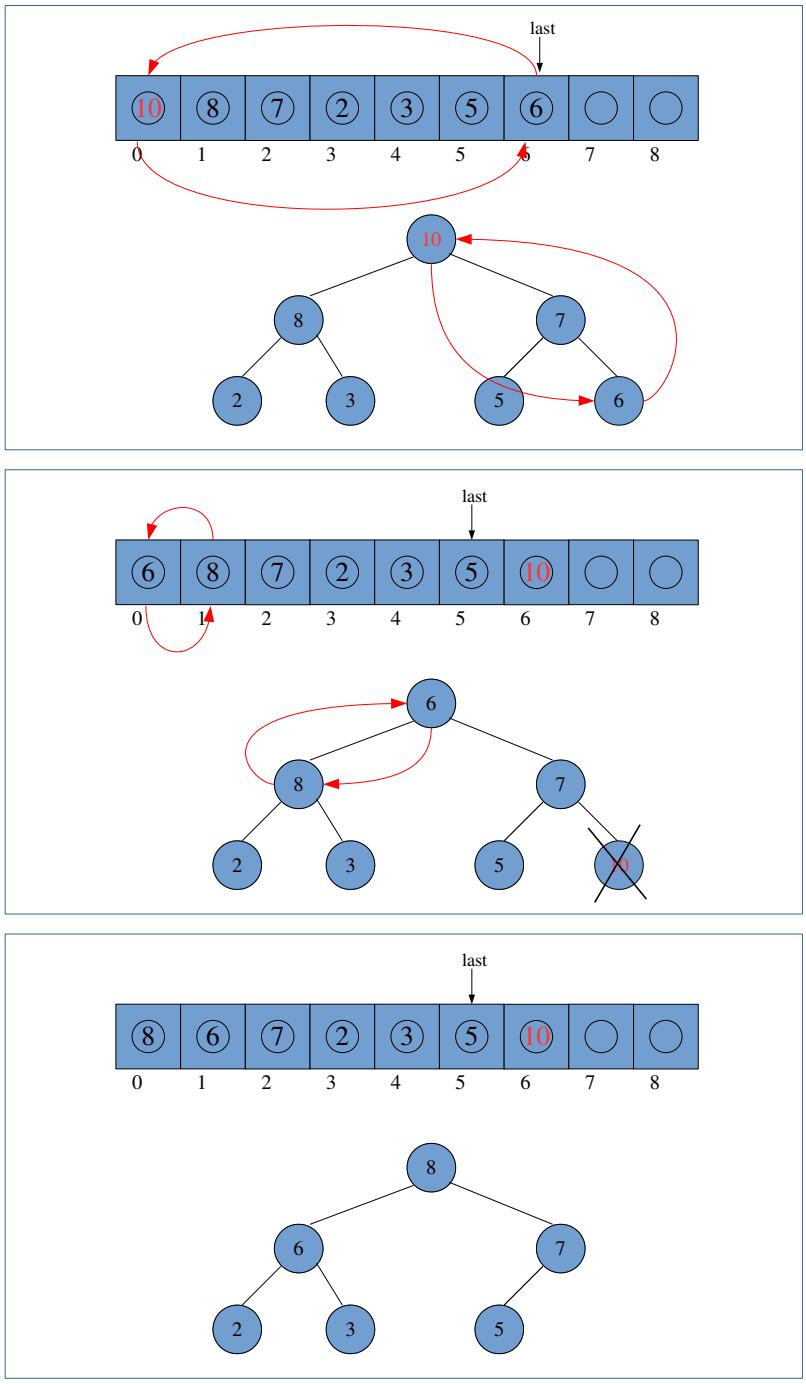


Figure 8.3: Deleting the root from a Heap. The root key is swapped with the last key and `last` is moved one position back. The key in the root has to go down several levels until its children's key are smaller. When deleting, it is important that the key going down the tree (in this case key=6) is swapped with the *largest* of its children.

8.2 - B-Trees

B-Trees are a generalisation of BSTs. B-Trees are commonly used in operating systems, for example in file systems. B-Trees are also used in databases. In a B-Tree, not only the nodes can have multiple children, but also each node has multiple keys (so the records inside the node can be a record array rather than a simple record). There are advantages in using B-Trees, as it can be fast to store and access new

information. However, the form of B-Trees that we are going to show here may be on average half-full.

B-Trees can be of different orders. The order of a B-Tree is an indication of how many children per node it has. A B-Tree of order m has m children and each node holds $m-1$ keys (and records behind its keys). An example of an order 4 node is shown in figure 8.4.

A more formal definition for an m-order B-Tree is:

1. the root of B-Tree has at least two sub-trees and one key, unless it is a leaf.
2. Each non-leaf node holds k-1 keys (and k children) with $\frac{m}{2} \leq k \leq m$.
3. Each *leaf* node holds k-1 keys with $\frac{m}{2} \leq k \leq m$.
4. all leaves are at the same level.
5. the keys follow a similar order to BSTs, smaller to the left, bigger to the right. [4]

[4] This property holds both inside a node (with multiple keys) as well as considering the pointers within that node. Each pointer points to a sub-tree.

According to this definition, the B-Tree is always at least half full and balanced. Compared to a BST with the same number of keys, the B-Tree has fewer levels.

To implement a B-Tree in C++, it is convenient to use arrays inside each node, as the order m has to be fixed before the tree is created (so there is not much point in using vectors in this situation, as the nodes with the array inside will be dynamically allocated anyway). This allows for the generalisation of m, so the insertion, searching and deletion method do not need to be modified for different orders [5].

[5] The order should be determined before compilation, so the order m can be declared in a `#define` statement. Alternatively, the order could be passed as a parameter to the constructor, and in this case we would have to use a vector rather than an array inside the node.

Each B-Tree node needs:

1. one array for the keys ($m-1$ keys per node)
2. one array for pointers (m pointers)
3. the current number of keys in the node (anything from 0 to $m-1$)
4. a flag leaf/non-leaf (the flag is not mandatory, but it makes the methods much easier to implement)

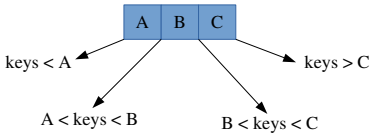


Figure 8.4: Node for a B-Tree of order 3. The figure shows the relationship between the position of the keys inside the node, the pointers and the keys of the children of this node. Note that a search function has to look inside the node for its keys (the keys are in sorted order), and then traverse to the appropriate sub-tree.

Listing 8.4 shows the implementation of a class BTreeNode. The class only includes two methods, Insert and Search. The reader can check a more complete B-Tree class example available on Stream.

An example of an order 3 B-Tree is shown in figure 8.5. Note that every node holds no more than 2 keys, and up to 3 valid (not NULL) pointers. When the node holds 1 key, it only needs two valid pointers (the last pointer is NULL). The B-Tree is self-balancing if we insert and delete according to the rules above. All the leaves are at the same level. However, inserting and deleting at random will cause the B-Tree to be half-full, i.e., there is some unused space in the nodes. This may not be a problem if long records are in the form of a key plus a pointer to the rest of the record, so if the key is empty there is no excessive waste of space apart from a NULL pointer. In the

examples in these sections the records composed by the keys only (according to listing 8.4).

Listing 8.4: The B-Tree node class.

```
1  class BTreeNode {
2  private:
3    int keys[M-1];
4    int number_of_keys;
5    BtreeNode *pointers[M];
6    bool isleaf;
7  public:
8    BTreeNode();//constructor
9    ~BTreeNode() { }//destructor
10   bool Insert(int data, BTreeNode *& currentroot);
11   bool Search(int data);//search by value
12 };
```

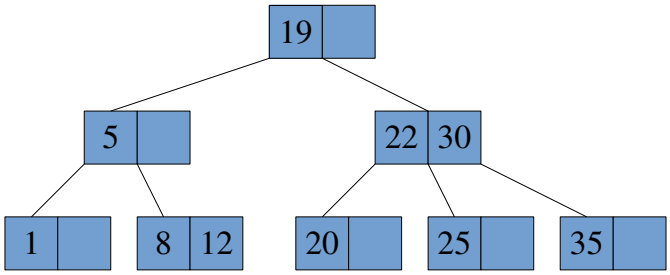


Figure 8.5: B-Tree of order 3. Note that the tree is balanced, but the nodes are not fully utilised (the tree is half empty). The searching of keys work similar to a BST: start from the root and follow to the right or left depending on the key value. The appropriate pointer has to be followed, sometimes the one in the middle of the node. For example, the node containing 22 and 30 has three pointers. Key 25 is placed in the middle pointer because it is smaller than 30 but bigger than 22.

8.2.1 - Insertion

Insertion in B-Trees are initially similar to BSTs in the sense that we need to search the place for a new key, and this place is a leaf. The difference is that in the BST insertion we simply created a new node with the new key. In B-Trees, we have to use an *existing* node to insert the new key. Only in the case that the leave is full do we need to create a new node.

Also, the search happens in a slightly different way. Given a key, the search will follow the same rules (if smaller *go to the left*, if larger *go to the right*), but of course one has to sweep through all the keys in a node before proceeding through the pointer to the next child. In an order 3 B-Tree, any of the 3 pointers might be followed depending on the value of the argument. For example, in figure 8.5 if we want to insert 21, the search would see the node with 19 and two pointers. The pointer on the right of 19 (the middle pointer) should be followed. Key 21 is smaller than 22, therefore the leftmost pointer should be followed. Finally we arrive at a node containing only 20, and this is a leaf. Key 21 can be inserted in that leaf, as there is one more space for it. The reader might have realised already that there may be cases

in which the leaf is already full. What do we do then? Insertions in B-Trees fall into three cases. The cases are:

- **Case 1**: the leaf where the new key is to be inserted has an empty place for it.
- **Case 2**: the leaf where the new key is to be inserted is full. The full node has to be **split** into two, distributing the pointers among them. The middle key (explained below) is "promoted" to the parent of the leaf. Case 2 can repeat again if the parent is also full. Every repetition of case 2 makes one new allocation.
- **Case 3**: the root of the B-Tree is full. In this case it will require another split/promotion. Case 3 makes two new allocations (the split node and a new root).

Figure 8.5 was created with the keys in the following order: 19, 5, 22, 30 , 1, 8, 20 , 25, 35 and 12. Figure 8.6 shows the same B-Tree after inserting key 21.

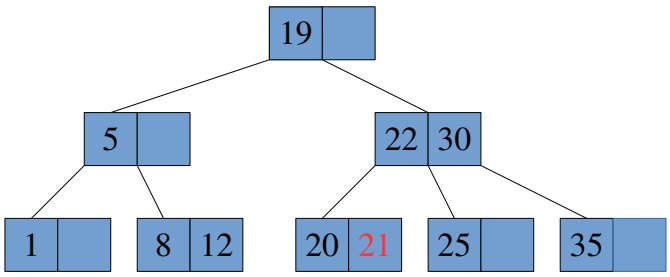


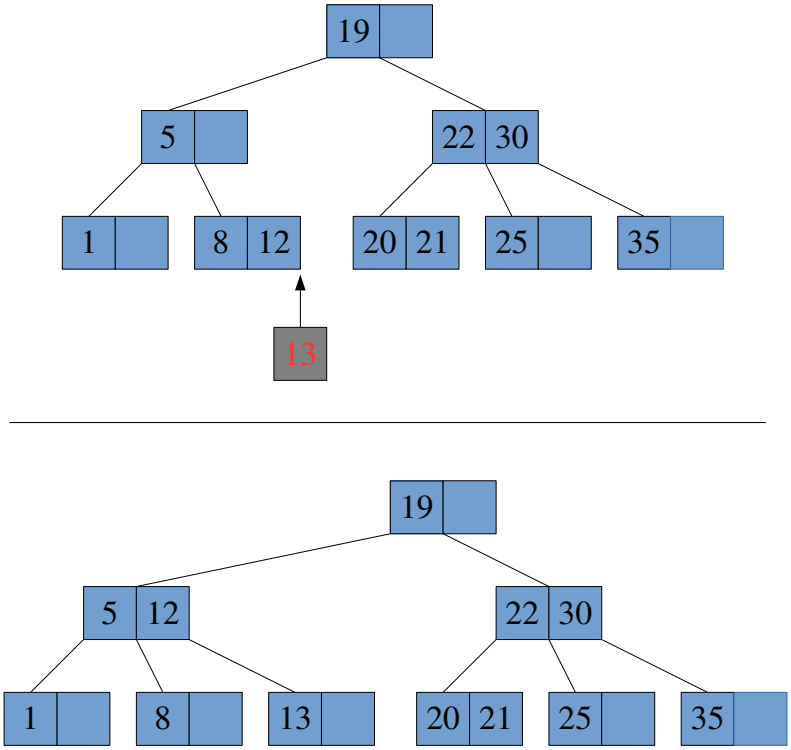
Figure 8.6: B-Tree insertion case 1.

The example for case 2 is trying to insert key 13. The correct place for it would be the leaf containing 8,12. However, it is full. The procedure is to split the leaf into two nodes (create one new node) and distribute the keys. It is important to promote the middle key, not the inserted key. In this example, keys 8, 12 and 13 are involved in the split. Key 12 is promoted to the parent of the leaf, and 8 stays in the old node, while 13 goes on the new node. The pointers from the parent of the leaves are updated accordingly (figure 8.7).

A more complex case 2 example is shown in figure 8.8. Here we attempt to insert key 26. Its correct place would be node 24,25. It is full, and therefore we have to split the node and promote 25 (middle of 24,25,26). 25 should be copied to node 22,30, which is also full. It needs to be split again, and 25 is promoted to the root, which has a place for it. Note that in this case two new nodes were allocated (the newly allocated nodes are shown in grey).

Finally, an example of case 3 is shown in figure 8.9. This for an attempt to insert key 42 after keys 36,40 and 44 were successfully inserted. Initially we don't know that the root is going to be used for the insertion. The insertion of 42 starts as a case 2, where the leaf should be the one containing 40,44. As the leaf is full, it is split and 42 is promoted (middle of 40,42,44). However, the parent is also

Figure 8.7: B-Tree insertion case 2.



full, so it is also split and 36 is promoted (middle of 30,36,42) to the root node. Now it is clear that the root node is full and needs to be split, and 25 is promoted (middle of 19,25,36). But only splitting the root node does not solve the problem, as there is no node where 25 can fit. So this last operation (lines 14 to 17 in algorithm 10) shows that another node needs to be allocated to receive 25 as a new root of the B-Tree. Case 3 makes two allocations, with the two previous allocations for case 2 makes 4 allocations in total (indicated in grey in figure 8.9).

A general algorithm for the insertion of the B-Tree is shown in algorithm 10. The three cases are indicated in the algorithm itself. Note that case 1 always returns immediately. The same for case 3, where a new root is allocated. Case 2 however can go through the loop several times, climbing the B-Tree (if the nodes found to fit the key are full) until reaching the root, when case 3 is called.

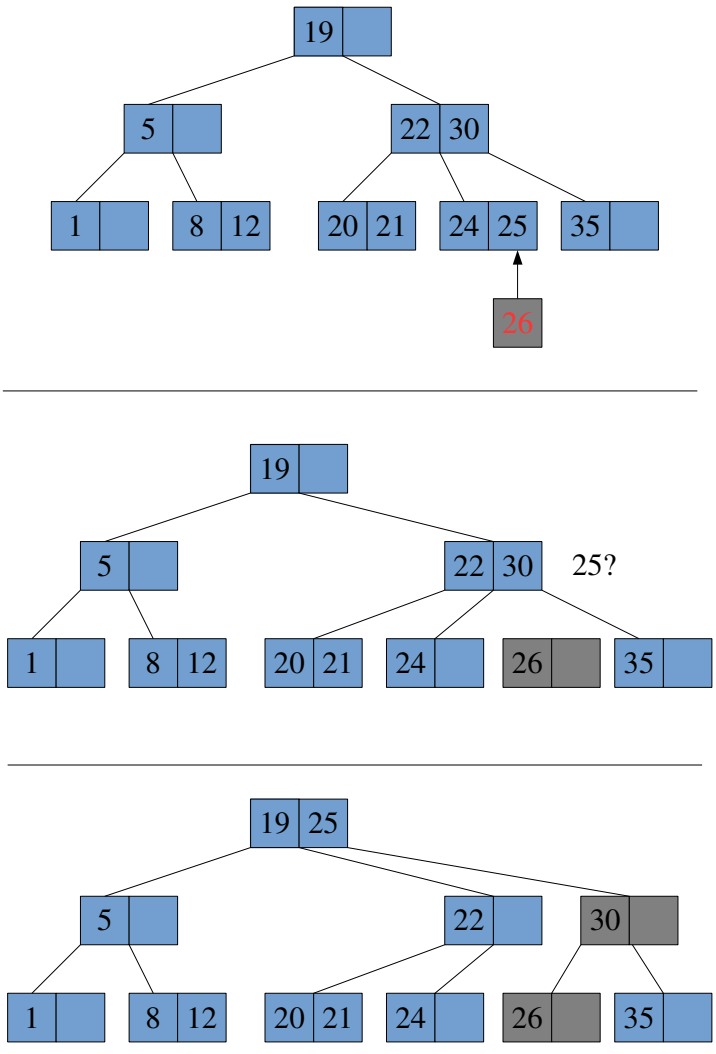


Figure 8.8: B-Tree insertion case 2 (twice).

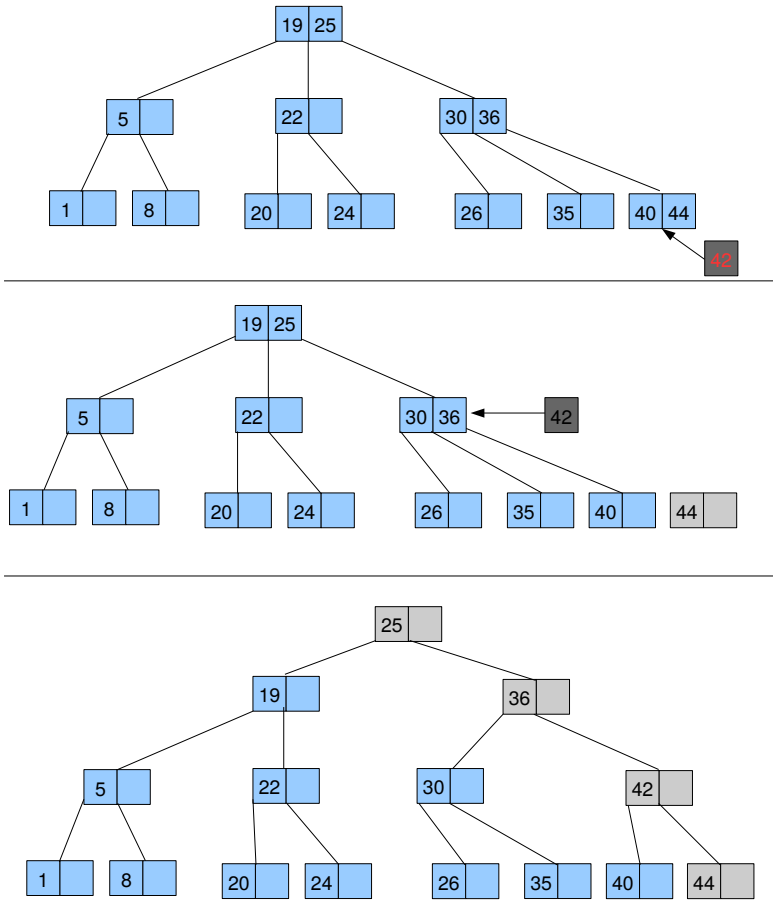


Figure 8.9: B-Tree insertion case 3.

Algorithm 10 B-Tree insertions.

```
1:  function InsertBTree(BTree B, key K)
2:      node = leaf to insert K;          ▷ separate function to search for the place of the new key
3:      while (true) do
4:          if (node is not full) then                                          ▷ CASE 1
5:              insert K, increment key counter;
6:              return;
7:          else
8:              split node into node1 and node2          ▷ allocate new node2, use old node1
9:              distribute keys and pointers;                  ▷ look at the example on Stream
10:             K = middle;          ▷ change the key, as the original K is in its place already
11:             if (node not root) then                                         ▷ CASE 2
12:                 node = its parent;
13:             else                                                            ▷ CASE 3
14:                 create new root                                ▷ parent of node1 and node2
15:                 copy K to root;
16:                 distribute pointers;
17:                 return;
18:             end if
19:         end if
20:     end while
21: end function
```

8.3 - Exercises

1. Implement the Heap class and methods as seen in listings 8.1, 8.2 and 8.3.

2. Build the Heap (schematically) for the following keys in this order: 3,5,1,8,6,9,10,2.

3. Delete the root of the previous Heap until it is empty again. Make sure that the properties are fixed for every deletion.

4. Build an order 3 B-Tree (schematically) for the following keys: 8, 24, 26, 25, 19, 5, 35, 1, 22, 30, 2, 3. Be careful with the considerations for each insertion case.

5. Finish the implementation of the B-Tree in listing 8.4, including the search and the insertion methods. Use your implementation to test the B-Tree of the previous exercise. (Stream has a fully implemented B-Tree example, including a print-readable B-Tree function).

6. Build a B-Tree of order 5 where the following keys are inserted: 8, 24, 26, 25, 19, 5, 35, 1, 22, 30, 2, 3. How different is the tree when comparing with the order 3 B-Tree? You can check the shape of the tree by running the complete example of a B-Tree insertion available on Stream (remember to change the order of the B-Tree in the code).

9 Sets and Bags

Sets and Bags are important ADTs. They can be used as auxiliary data structures in many different algorithms.

9.1 - Sets

A set is a collection of unique items of the same type. Each item could represent a record with a key. In a set the keys cannot be repeated. Also, in a set there is no particular order to the elements, i.e., there is no previous or next, nor there is a specific index in which an element can be found. Any operation typically implies in a traversal of the set. For example, one could define maximum or minimum items, and these would have to be found by traversing the set. An alternative is to keep checking for maximum and/or minimum every time an item is inserted or deleted.

An example of representation of a set containing 2,3,4 and 8 is:

$S = \{2, 3, 4, 8\}$

Among the important operations for a set are insertion, deletion membership and checking whether the set is empty. Two crucial operations for sets are `union` and `intersection`. An example of the last two operations can be found in figure 9.1.

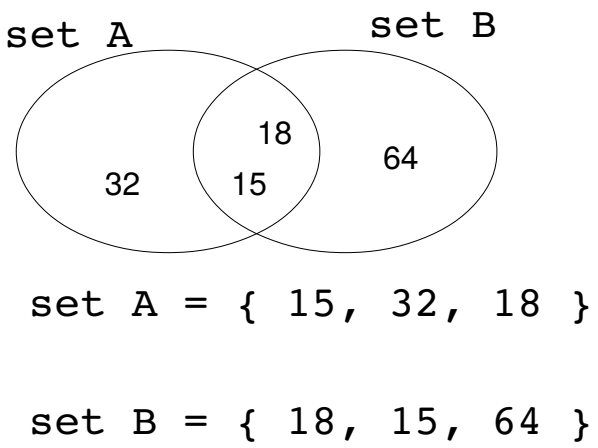


Figure 9.1: Sets A and B.

9.2 - Bags

A bag is similar to a set, with the difference being that the item can be repeated (contains many instances of the same item). It is easy to remember the difference between sets and bags using coins. A set of New Zealand coins would be:

$Set\_Coins = \{10c, 20c, 50c, \$1, \$2\}$

A bag of coins could be represented by:

$Bag\_Coins = \{10c(10), 20c(4), 50c(5), \$1(2), \$2(3)\}$ with the number of coins in parenthesis.

9.3 - Implementation

There are many forms in which sets and bags can be implemented. Here we just show four examples: using linked-lists, arrays, trees and bit vectors. For the scope of the course, the bit vector implementation is the most relevant, although the other implementations are widely used in certain applications.

9.3.1 - Linked-lists

The nodes of a linked list could represent a set or a bag. In the set, the keys are the elements themselves. In the bag, an additional member to the `struct` of the linked list could represent the number of the item in the bag (see figure 9.2).

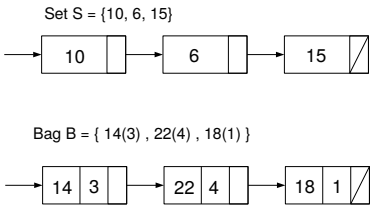


Figure 9.2: Implementation of sets and bags using linked-lists.

9.3.2 - Arrays or Vectors

Using arrays have the usual problem with fixed size. This can be overcome by using vectors. However, the problem of insertion and deletion in the middle of a set or bag remains. The space between elements is wasted memory, but this may be desirable if the performance is the main target.

9.3.3 - Tree

Using trees can be quite efficient to search. However it tends to be overly complex. There are special cases of sets where it is desirable to represent the set as a tree (e.g. disjoint sets), but usually the tree can also be implemented as an array or vector, similar to a Heap.

9.3.4 - Bit vectors

Sets can be implemented as bit vectors. This is a very economic representation of sets where integers can occur at random (i.e., any number can appear as an item of the set). The economy comes from representing the number by position rather than by value. For example, the position of the bit inside a certain type could represent the item itself, with the bit *on* if the item is present, or the bit *off* it the item is not present. The other advantage is that bit-wise operators

can be used to achieve the basic operations: insertions, deletions, membership, unions and intersections.

AND (&)

X	Y	output
0	0	0
0	1	0
1	0	0
1	1	1

OR (|)

X	Y	output
0	0	0
0	1	1
1	0	1
1	1	1

XOR (^)

X	Y	output
0	0	0
0	1	1
1	0	1
1	1	0

SHIFT (<< and >>)

```
<< multiply by 2

>> integer division
 by 2
```

Figure 9.3: Bit-wise operators.

An example of bit vector can be an `unsigned char`, with the bit position indicating which is the integer in the set.

The set $A = \{5, 1, 6, 3\}$:

Position from right : `7 6 5 4 3 2 1 0`

presence of the item: `0 1 1 0 1 0 1 0`

Any C/C++ type could be used for the purpose of a bit-vector. If more than 64 elements are needed, than an array (or vector) of bit vectors can be used, with an implicit multiplier for the different indices of the array.

Recall how the bit-wise operators work and examine their truth tables (figure 9.3).

Given a bit-vector in the form of an unsigned char, an example of a set is:

$Set = \{1, 3, 5, 6\}$

and the bit-vector state: `0 1 1 0 1 0 1 0`

In order to change or read individual bits in the bit vector, one can use a mask. The mask can be initialised to **1** and then shifted to the position required. The mask is then used with one of the bitwise operators to achieve the desired effect on the bit vector.

The following listings shows how to insert, delete and check for the membership of elements in a bit vector.

Insertion

To insert 2 to the set represented by `bitvector` we use the bit-wise operator OR (|). An unsigned char temp is used to get the third bit from the right on (this represents 2). As temp starts with 1, a shift of 2 will make temp = 4 (in binary is `00000100`)

Listing 9.1: Set insertion.

```
1 unsigned char bitvector;
2 unsigned char temp=1;
3 temp=temp << 2;
4 bitvector = bitvector | temp;
```

Membership

To find whether 2 is in the set, the bit-wise operator AND (&) is used:

Listing 9.2: Set membership.

```
1 unsigned char bitvector;
2 unsigned char temp=1;
3 temp=temp << 2;
4 temp = temp & bitvector;
5 if(temp) return true;
6 else return false;
```

Deletion

To delete element 2 from the set the bitwise operator XOR (^) is used, however the XOR bitwise operator only toggles the bit. If the element was not in the set in the first place, the XOR operator would insert it, and this is not what the function was supposed to do. We need to ask about the membership before trying to delete, so an `if` statement before the XOR is included:

Listing 9.3: Set deletion.

```
1 unsigned char bitvector;
2 unsigned char temp=1;
3 temp=temp << 2;
4 if(temp & bitvector) bitvector = bitvector ^ temp;
```

The operations *union* and *intersection* between two sets can also be achieved with the use of bitwise operators. There is no need for a mask in this case.

Union

Listing 9.4: Set union.

```
1 unsigned char bitvector1;
2 unsigned char bitvector2;
3 unsigned char unionset;
4 unionset = bitvector1 | bitvector2;
```

Intersection

Listing 9.5: Set intersection.

```
unsigned char bitvector1;
unsigned char bitvector2;
unsigned char intersectionset;
intersectionset = bitvector1 & bitvector2;
```

9.3.5 - *Printing bit vectors for debugging purposes*

Sometimes it is difficult to visualise and debug code with bit-vectors. A simple print function can be very helpful. The function in listing 9.6 prints a bit-vector using an `unsigned char`. It is easy to modify the function to cater for larger types [1].

Listing 9.6: Printing bit-vectors in readable format.

```
void printbits(unsigned char bitvector){
  for(int i=(sizeof(char)*8-1); i>=0; i--){
    if(i==3) printf(" ");
    unsigned char temp=1;
    temp=temp<<i;
    temp=bitvector&temp;
    if(temp) printf("1");
    else printf("0");
  }
  printf("\n");
}
```

[1] A bit vector can be printed as an integer or as a hex using the normal `printf` or `cout` functions. However, this is not very helpful to visualise the bits themselves. For example, the bitvector containing $\{1,2,3,5,6\}$ printed as an integer would be `110` and as a hex `0x6E` Using `printbits` from listing 9.6, it would print `0111 0110`.

One can upgrade the printing function by changing the types and the `for` loop limits.

9.4 - *Exercises*

1. Implement two sets as bit-vectors and populate them with: set A = 1,2,3,4 and set B = 5,6,7,8.
2. Make a third set, and print its union with set A and then set B.
3. Print the intersection of sets A and B.
4. Insert 3 in set B.
5. Delete 2 from set A.
6. Check for the membership of 6 in sets A and Set B.
7. Create a bit vector with places for 64 elements (use an `unsigned long int`). Modify the `printbits` function (listing 9.6) to cater for this larger bit vector.
8. Write a new function to print a bit vector in the format $\{0,1,2,3,4,5,6,7\}$. You can start from the `printbits` function.

10 Graphs

Graphs are abstract representations of objects that can be connected to each other. Each object is represented by a **node** (or **vertex**) and the links between objects are represented by **edges** (also called **arcs**). The edges can represent information about the relationship of the linked nodes. For example, the edge could represent the distance between cities, or represent the bandwidth between two computer network routers. Many natural problems can benefit from the use of graphs. Graphs can represent problems in computer science, math, biology, physics and many more areas. There are many algorithms available for applications in these areas. New problems that can be represented as graphs might have a simple solution using an existing graph algorithm.

Before proceeding with implementations and algorithms for graphs, it important to start with some definitions and specific jargon used in graph theory.

A **directed graph** is a graph where the edges have a defined direction. The edges are represented by arrows. On the other hand, in an **undirected graph** the edges link in both directions and have the same cost.

A **labelled graph** uses words in the edges instead of a numeric value. If can represent some relationship between the nodes. For example, if the nodes are atoms, the edges can represent the fact that they can form compounds. On the other hand, if the edges are numeric the graph is called a **weighted graph**.

If two given nodes, say M and T, are connected (there is an edge between them) they are **adjacent**.

If there are more than one link that can connect Y via nodes M and T to S, then there is a **path** between Y and S. The path can be represented by a sequence of nodes. A **simple path** has no repeated nodes in its sequence. The **length of a path** is the sum of the edges of that path. For graphs that are not weighted, one can assume that the edge has a length of 1.

A **connected graph** has at least one path from every node to every other node. A **fully connected graph** has a direct path (from source to destination without any nodes in between) from every node to every other node.

A **cycle** is a simple path that goes back to the source (i.e., no repeated nodes except the source). An **acyclic graph** is a graph that contains no cycles.

The usual operations such as insertion, deletion and search apply to graphs. There are also classical operations that only apply to graphs, such as finding a path between two nodes, finding the shortest or the longest path between two nodes, finding a cycle etc. We are going to study some of these operations in this chapter, and include two classical algorithms in our discussion, namely Kruskal and Dijkstra algorithms.

10.1 - Data Structures used in Graphs

10.1.1 - Based on sets

Two sets can be used, one representing the nodes, and the other representing the edges. Figure 10.1 shows the representation for weighted directed and non-weighted undirected graphs.

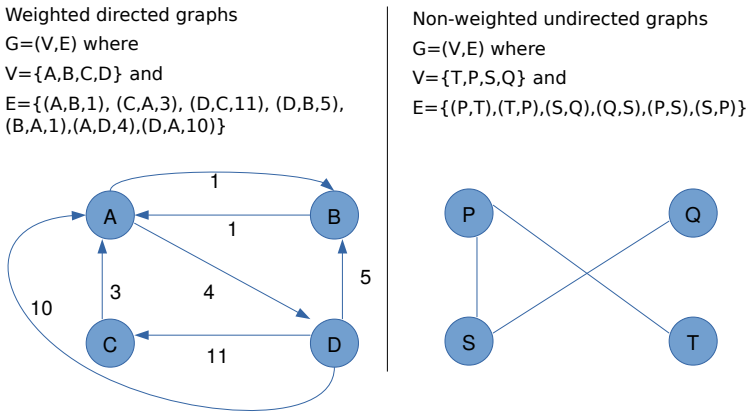


Figure 10.1: Graphs represented by **sets**.

10.1.2 - Based on arrays (or vectors)

Adjacency Matrices can be used to represent a graph. These are easily implemented with arrays or vectors. The advantage of such implementations is that the access to nodes and to edges is very fast (random access), but if the graph is very large and not fully connected there is waste of space in memory, just like a sparse matrix would have. Figure 10.2 shows adjacency matrices for weighted directed and non-weighted undirected graphs. In the matrices note that the origin (from) and destination (to) are clearly marked. This is an arbitrary convention, it could be the other way around (from in the columns and to in the rows). Note that the absence of an edge has to be represented as ∞, not zero. An edge with zero is an immediate link between two nodes (no cost, i.e., the distance is zero). Zeros in the Adjacency Matrix represent the link between a node and itself, hence all the diagonals in the matrix should be filled with zeros.

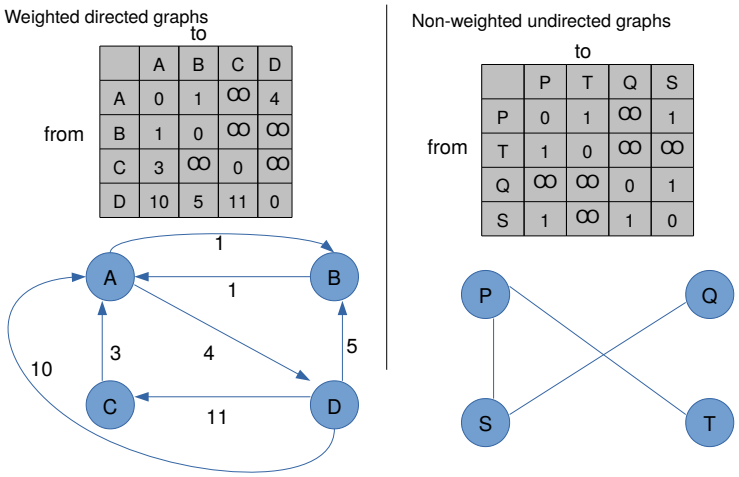


Figure 10.2: Graphs represented by **adjacency matrices**.

10.1.3 - Based on linked-lists

The graph can also be represented by Adjacency Lists. In this case multiple linked-lists are needed. The main linked-list contains the Graph nodes. Each node of the linked-list where the Graph nodes are contained has also a listpointer pointing to a separate linked-list where the edges (including the connection and the distance) are contained. If the graph has N nodes, then it would have $N + 1$ linked-lists. The advantage of this implementation is that nodes and edges can be allocated and deallocated at will. The disadvantage is that the performance is slower than using Adjacency Matrices. Note that the absence of an edge is simply not represented in the Adjacency List, nor is the link between a node and itself. The only way to find out that node A is not linked to node B is by traversing both linked-lists. *Weighted undirected graphs* have duplications of the edges (e.g., from A to B and from B to A if these nodes are connected). Figure 10.3 shows adjacency lists for weighted directed and non-weighted undirected graphs.

Alternatively, the head linked-list (the one representing the nodes) could be a vector instead of a linked-list. In this mixed representation, the nodes would be randomly accessible as long as there is a relationship that can be computed from the node's name into the index (e.g, node A corresponds to index zero, node B corresponds to index 1 and so forth).

10.1.4 - A Graph class based on vectors and linked-lists

Using a mix of a vector (containing vertices or nodes) and linked lists (containing edges), a generic Graph class can be implemented. This class is not ideal for all algorithms. As explained before, finding any cost implies in traversing one of the linked lists, and if the graph is too large this has performance implications. However, for the algorithms that we are going to use this implementation is good enough because the number of nodes that can be used in practise is limited.

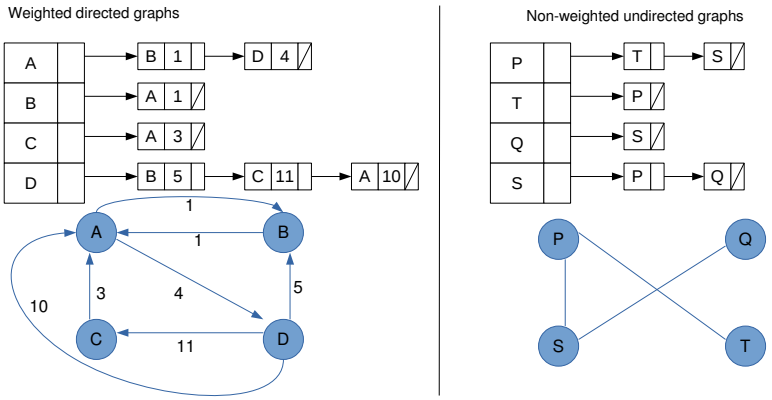


Figure 10.3: Graphs represented by **adjacency lists**.

Also, this naive implementation is only useful for inserting nodes. Deletion would require a shuffle of all the other nodes to avoid getting a gap in the vector. The edges can be inserted and deleted as we would with linked-lists. Listing 10.1 shows the code.

Listing 10.1: The Graph class.

```
1  struct GraphNode{//used by the vector
2    char key;
3    Node *listpointer;//node points to a linked-list with
         its links
4  };
5
6  struct Node {//used by the linked-lists
7    char key;
8    int distance;
9    Node *next;
10 };
11
12 #include <vector>
13 using namespace std;
14 class Graph{
15 private:
16   vector<GraphNode> adjlist;
17 public:
18   Graph(){};
19   ~Graph(){};
20   void AddNewGraphNode(char newgraphnode);
21   void AddNewEdgeBetweenGraphNodes(char A, char B, int
         distance);
22   void PrintAllGraphNodesWithCosts();
23 };
```

The only three methods available are insertion of vertices (`AddNewGraphNode`), insertion of edges (`AddNewEdgeBetweenGraphNodes`) and a printing method (`PrintAllGraphNodesWithCosts`). The methods are in listing

10.2. The methods rely on the previously studied linked-list functions `AddNode` and `PrintLL`.

Listing 10.2: The Graph methods.

```
void Graph::AddNewGraphNode(char newgraphnode){
  GraphNode temp;
  temp.key=newgraphnode;
  temp.listpointer=NULL;//important
  adjlist.push_back(temp);
}

void  Graph::AddNewEdgeBetweenGraphNodes(char A, char B,
    int distance){
  //find which node A is
  int a;
  for (a=0;adjlist.size();a++){
    if (A==adjlist[a].key) break;
  }
  AddNodetoFront(adjlist[a].listpointer, B, distance);
}

void Graph::PrintAllGraphNodesWithCosts(){
  for (int a=0;a<adjlist.size();a++){
     printf("From Node %c: \n", adjlist[a].key);
     PrintLLnodes(adjlist[a].listpointer);
  }
}
```

To create the graph seen in the figure 10.3, the following main() function can be used:

Listing 10.3: The main function to create and print a Graph.

```
Main(){//creates the graph and traverses it
 Graph mygraph;
 mygraph.AddNewGraphNode('A');
 mygraph.AddNewGraphNode('B');
 mygraph.AddNewGraphNode('C');
 mygraph.AddNewGraphNode('D');
 mygraph.AddNewEdgeBetweenGraphNodes('A', 'D', 4);
 mygraph.AddNewEdgeBetweenGraphNodes('A', 'B', 1);
 mygraph.AddNewEdgeBetweenGraphNodes('B', 'A', 1);
 mygraph.AddNewEdgeBetweenGraphNodes('C', 'A', 3);
 mygraph.AddNewEdgeBetweenGraphNodes('D', 'A', 10);
 mygraph.AddNewEdgeBetweenGraphNodes('D', 'C', 11);
 mygraph.AddNewEdgeBetweenGraphNodes('D', 'B', 5);
 mygraph.PrintAllGraphNodesWithCosts();
 }
```

10.2 - Graph Traversals

In a graph traversal we should visit all vertices only once. Depending on the data structure used to implement the graph, one can find a specific way to traverse a graph. For example, in the implementation used in this chapter it suffices to traverse the vector where the nodes are declared. There is no need to use the linked-lists. This is not an entirely satisfactory solution (even though it can work for specific cases such as this) because we want a generic solution for any graph, including the case where we need to work with a graph schematically. For that we need a proper traversal. In this section we are going to explore the *depth-first search* (**DFS** for short).

The idea with the depth-first search is to start at a node and visit all the other nodes linked to it. But this is not as simple as in the case of a tree. The graph can have cycles, in which case the DFS would have an infinite loop. The graph can also have sub-graphs that are disconnected from the other nodes, in which case we would not be able to visit them all. The simplest way is to mark the nodes that were already visited and keep track of them. If we visit a node twice we know that there is a cycle and we can avoid it. If there are still nodes that were not visited then we have to restart the algorithm from the sub-graph. The algorithm needs an auxiliary structure to hold the marks to the already visited nodes. It also needs to be able to back track previously visited nodes, so we can visit its other neighbours. As we have seen before with trees, a convenient way to backtrack nodes is using a stack of pointers to nodes. [1] Algorithm 11 shows one of many possible ways to achieve what was discussed above, but it is *restricted to connected graphs*.

It should be clear that the `for` loop in line 10 algorithm 11 involves the traversal of the linked-list associated with node `v`.

[1] The stack could contain nodes of GraphNode type or pointers to the same type. The important thing is for the stack to have access to both the name of node (`char`) and its linked-list.

Algorithm 11 DFS.

```
1:  function DFS(Graph G, node v)
2:      declare a boolean vector M;   ▷  All elements of M initialised to false
3:      declare stack S;              ▷  S is a stack of nodes.
4:      S.push(v)
5:      while (S is not empty) do
6:          v = S.top()
7:          S.pop()
8:          if (if M[v] is false) then
9:              M[v] = true;
10:             for each neighbour x of v do
11:                 S.push(x)
12:             end for
13:         end if
14:     end while
15: end function
```

It is interesting to actually implement the DFS algorithm to see

how much it changes from algorithm 11. There are steps that we take for granted, but need to be coded when implementing it in C++ due to the decisions we made when implementing the graphs. Note that we needed to search for the correct position of the keys of the nodes (lines 7, 12 and 21 of listing 10.4). These lines were not explicitly written in the line 10 of algorithm 11. Alternatively, the letters can be related to their positions within the vector, but this would require inserting the nodes in a certain order.

The code stops short of doing the traversal if there are islands of nodes that are disconnected from the rest of the graph. The only way to avoid that would be to mark each visited node, and call the function again for yet to be visited nodes, until all nodes are marked.

Listing 10.4: A DFS implementation using stacks.

```
1  void Graph::DFS_iterative(char vertex){
2    int a=0;
3    vector<bool> M;
4    for(a=0;a<adjlist.size();a++){M.push_back(false);}
5    stack<GraphNode> S;
6    GraphNode v;
7    for(a=0;a<adjlist.size();a++){if(adjlist[a].key == vertex) break;}
8    S.push(adjlist[a]);
9    while(S.empty()==false){
10     v=S.top();
11     S.pop();
12     for(a=0;a<adjlist.size();a++){if(adjlist[a].key == v.key) break;}
13     if(M[a] == false){
14       cout << "Visited " << v.key << endl;
15       M[a]=true;
16       //traversing the linked-list
17       Node *temp;
18       temp = v.listpointer;
19       while(temp!=NULL){
20         cout << "neighbour of " << v.key << " is " << temp->key << endl;
21         for(a=0;a<adjlist.size();a++){if(adjlist[a].key == temp->key) break;}
22         S.push(adjlist[a]);
23         temp=temp->next;
24       }
25     }
26   }
27 }
```

In order to avoid not traversing islands of nodes, one can use a start function and modify the DFS function (listing 10.5). A new vector is added to the class, called `num`. This vector keeps track of the order in which the nodes are being visited (the sequence is given by another auxiliary integer called `cyclenumber`). Also, a method called `ReturnIndex` returns the index of a node given its name (`char`).

Listing 10.5: A DFS that can go through islands.

```
void Graph::Start_DFS_iterative(){
  cout << "DFS recursive " << endl;
  if (adjlist.size()==0) return;
  cyclenumber=0;
  num.resize(adjlist.size());
  for(int a=0;a<num.size();a++) {num[a]=0;} //initialise to zero;
  //check if there are still vertices not visited
  bool isThereNonVisitedVertex=true;
  while (isThereNonVisitedVertex) {
    isThereNonVisitedVertex=false;
    for(int a=0; a<num.size(); a++){
      cout << "num["<<a<<"] " << num[a] << endl;
      if (num[a] == 0) {
        isThereNonVisitedVertex=true;
        cout << "Iterative still unvisited nodes " << endl;
        DFS_iterative(adjlist[a].key);
      }
    }
  }
}


void Graph::DFS_iterative(char vertex){
  int a=0;
  stack<GraphNode> S;
  GraphNode v;
  for(a=0;a<adjlist.size();a++){if(num[a] == 0) break;}
  S.push(adjlist[a]);//start from first node that has not yet been visited
  while(S.empty()==false){
    v=S.top();
    cout << "v is now " << v.key << endl;
    S.pop();
    if(num[ReturnIndex(v.key)] == 0){
      cout << "Iterative Visited " << v.key << endl;
      cyclenumber++;
      num[ReturnIndex(v.key)]=cyclenumber;
      Node *temp;
      temp = v.listpointer;
      while(temp!=NULL){
          cout << "neighbour of " << v.key << " is " << temp->key << endl;
          if(num[ReturnIndex(temp->key)]==0) S.push(adjlist[ReturnIndex(temp->
              key)]);
          temp=temp->next;
      }
    }
  }
}
```

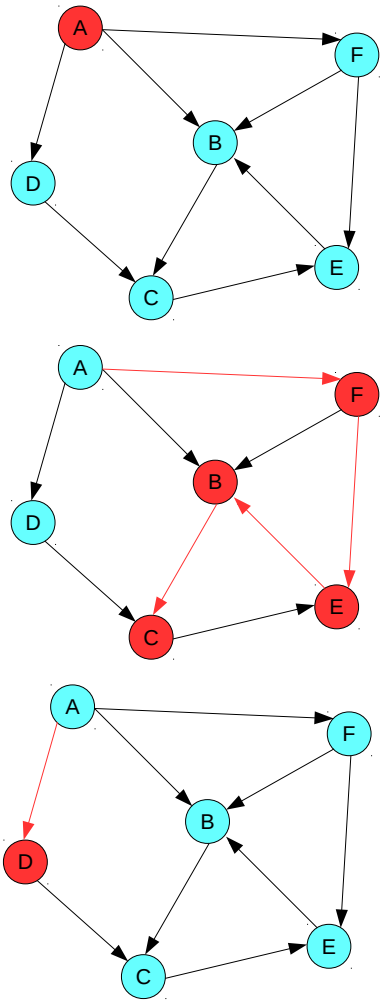


Figure 10.4: Traversing the graph starting with node A. The traversal using the code in listing 10.5 could have more than one answer depending on the order of input. It could be A, F, E, B, C, and D, if F is the first neighbour of A to be traversed. Otherwise, it could also be A, D, C, E, B and F. As the order E, B would also depend on the input order, it could also be A, F, B, C, E and D instead. There is no unique answer for the traversal of a graph.

10.3 - Minimum Spanning Tree

A *spanning tree* `S` of graph `G` is a sub-graph of `G` where all the vertices are copied to `S`, but the only edges copied are the ones that are sufficient to link all vertices without causing cycles. If the sub-graph contains cycles, it would not be a tree.

If the total cost of linking all the vertices is the minimim possible cost (considering all the possible combination of available edges on `G`), then we have the `minimum spanning tree` `M` of a graph `G`. **MST** is the usual acronym for the minimum spanning tree of a graph `G`.

MST can find the best way to connect lines (telephone, networking cables etc) in such a way that the minimum amount of material is used, but all the nodes are connected. There are two important algorithms that can find the `MST` of a graph `G`: Kruskal's algorithm and Prim's algorithm.

10.3.1 - Kruskal's Algorithm

Kruskal's algorithm uses spanning trees to build a graph with the minimum cost for the sum of the edges used for connecting the

vertices, i.e., where there are no cycles and the sum of the edges (the 'cost') is minimum but enough to connect all nodes. Algorithm 12 formalises Kruskal algorithm for an undirected graph.

Algorithm 12 Kruskal.

```
1:  function KRUSKAL(Graph G)
2:      create an empty graph C;
3:      Copy all nodes (vertices) from G to C;
4:      while (There are unconnected nodes) do
5:          Copy the smallest edge from G to C;
6:          if (a cycle is found) then
7:              delete copied edge and mark it not to be used again;   ▷ Backtracking...
8:          end if
9:      end while
10: end function
```

It is interesting to note that running the algorithm will always find at least one answer, as long as the original graph is connected and undirected (all nodes have at least one edge that links with the rest of the graph). However, multiple answers with the equivalent minimum cost might be found (no guarantee of uniqueness). Figure 10.5 shows an example. Can you find an alternative minimum cost spanning tree?

Another interesting aspect of Kruskal's algorithm is that it relies on a cycle detection algorithm. It is possible to modify the DFS algorithm slightly to cater for this purpose. If you pay attention to listing 10.5, you will see that we check whether we already visited a node. If we repeat a visit, it means that we have a cycle in the sub-graph.

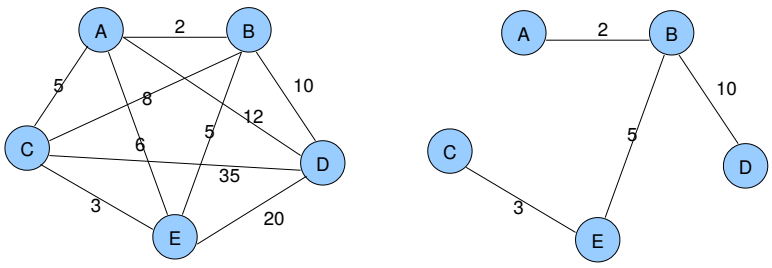


Figure 10.5: Kruskal's algorithm: on the left the original graph, on the right the minimum cost spanning tree. Note that Kruskal's algorithm can generate multiple answers. For example, the edge A→C could have been used, as it has the same cost as edge B→E.

10.3.2 - Prim's Algorithm

Prim's algorithm does not need to rely on a separate function to find cycles in the sub-graph as we start building the MST. Instead, we fix a source node and repeat N times a search for new edges to be included in the MST. Every time the outer loop runs, we look for the minimum cost edge where the combinations of nodes obeys a certain rule. We only look for edges linking nodes that are already in the MST to nodes that are not yet in the MST. Figure 10.6 shows an example to clarify the algorithms internal workings.

Algorithm 13 Prim.

```
1:  function PRIM(Graph G with vertices v() and edges e())
2:      create an empty set of vertices VC
3:      VC={ source }                          ▷  any random node can serve as source
4:      create an empty set of edges EC = {}
5:      for ( N-1 times ) do                   ▷  N = number of nodes in graph G
6:          min = ∞
7:          for ( x = each vertex ) do
8:              if ( v(x) belongs to VC ) then
9:                  for ( y = each vertex ) do
10:                     if ( v(y) does NOT belong to VC ) then
11:                         if ( cost( v(x) → v(y) ) < min ) then
12:                             min = cost( v(x) → v(y) )
13:                             newVertex = v(y)
14:                             newEdge = v(x) → v(y)
15:                         end if
16:                     end if
17:                 end for
18:             end if
19:         end for
20:         insert newVertex into VC
21:         insert newEdge into EC
22:     end for
23: end function
```

10.4 - Dijkstra's Algorithm

The algorithm finds the shortest paths from a source to every other node. This is an important algorithm because many real life problems can be modeled like that, based on a graph where one has to find the shortest path from an origin vertex to a destination vertex. The most obvious application in the modern world is computer networks routing process. The communication between computers (typically in a client/server model) has to rely on the fastest possible paths through different routers that composes the infra-structure of the networks. The protocols rely on algorithms that find the shortest paths very quickly and optimally.

Dijkstra published his (now famous) algorithm in 1959. Since its publication, many different versions of the minimum path algorithms have been developed. In this chapter we are going to study the simplest form of the Dijkstra algorithm for weighted directed (or undirected) graphs where the costs (the edges) are positive integers.

The algorithm is going to be presented considering that the graph is implemented as an adjacency list (see algorithm 14).

The Dijkstra's algorithm starts by initialising all the nodes to state temporary and distance infinite, except for the source node that has the state initialised to permanent (the distance no longer changes) and distance to zero (as the distance from the source to itself is zero).

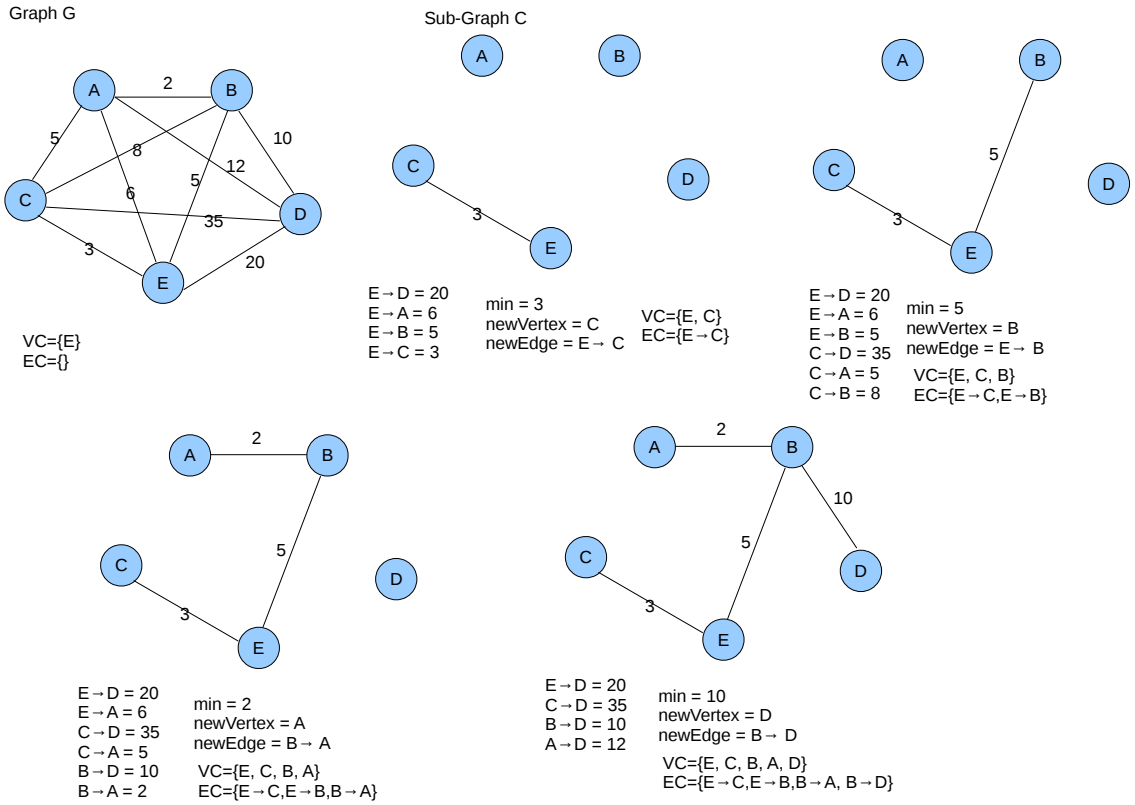


Figure 10.6: The same graph used in figure 10.5 was used in this figure. The same two MSTs can be found (depending on which edge is chosen when the minimum is a draw).

The two arrays (they can be combined into one array for implementation in C++) `d[]` and `s[]` keep the current minimum distance and state during the run for all nodes of the graph. Note that either there is a simple formula to reach the correct index of the arrays, or the arrays have to be traversed [2].

The main `while` loop is used to update the distances and states. The most important part of the algorithm is how it keeps the current costs calculated to the current minimum distances. The `for` loop inside the `while` loop does the job of updating the distances for all the neighbours of the current node. Initially the current node is the source (in figure 10.7 the source is `A`). Only the nodes `B` and `E` are updated because they are the only ones reachable from `A` directly (`B` and `E` are neighbours of `A`). The updates are done with the direct distance between `A` to `B` and `A` to `E` (figure 10.7 b).

In line 15 of the algorithm, there is a function called for the cost between two nodes (`cost(current,v)`). This cost can be found by traversing the vector until we find `current` key. The record contains a pointer to a linked-list, we then traverse this corresponding linked-list to find the cost to node `v`.

The next two statements are crucial for understanding Dijkstra. Statement 17 searches for a node `v` with state `t` that has the smallest possible distance. At this stage of the algorithm, there are still four nodes with state `t` (`B`,`C`,`D` and `E`). The smallest distance at this moment is with node `B`, which becomes the current node for the next round. Statement 18 of the algorithm makes the state of node `B` permanent.

[2] If the input of the keys is randomised, it will require a traversal of the vector to find the corresponding key. However, if there is a relationship between the key and the index, then it is possible to search for the key using a simple formula. For example, if A corresponds to index [0], B to [1], C to [2] and so on, given a char with the key we can find the index as: $index = key - 65$.

Algorithm 14 Dijkstra.

```
1:  function DIJKSTRA(Graphs G and source SourceNode)
2:      declare an array of distances d[N];
3:      declare an array of states s[N];
4:      declare a graph node current = SourceNode;
5:      d[SourceNode] = 0;      ▷  the distance from the source to itself is zero
6:      s[SourceNode] = p;      ▷  The state of the source node is permanent
7:      for (each vertex v of G) do
8:          if (v != SourceNode) then          ▷  Initialisation
9:              d[v] = ∞;          ▷  all other nodes distance is infinite
10:             s[v] = t;          ▷  all other nodes start with state temporary
11:         end if
12:     end for
13:     while (there is a vertex v with state s[v]==t) do
14:         for (each neighbour v of current) do
15:             d[v] = min (d[v], d[current] + cost(current,v);
16:         end for
17:         find minimum d[v] with state s[v]==t, make current=v;
18:         s[current] = p;          ▷  the minimum cost was found for this node
19:     end while
20: end function
```

This means that the minimum distance between A and B is 2, and cannot possibly change in future rounds. Why is that so? If we attempt any other alternative path to reach B it will cost more than 2, as any other node that is involved will start with a distance that is already more than 2.

Following the next round of the while loop, we have to update all the neighbours from B. The only reachable nodes from B are C and E, so these two nodes have their distances updated. Now the equation to update is:

$$d[v] = \mathbf{min}(\, d[v]\,,\, d[current] + cost(current, v\,)) \tag{10.1}$$

In other words, the current distance of a certain neighbour node is compared to the distance of going through the current node. If reaching v through a path that goes via the current node is cheaper (smaller distance), then we update d[v]. Otherwise, we leave the distance unchanged. Why does that work? Because the current node is guaranteed to have the minimum distance from the source. If there is an alternate path that goes via some other nodes to reach v, then there is no point in changing the path. On the other hand if the distance is smaller, then we rely on the new path to find a new distance. The new calculated distances might not be the minimum yet and may still change. At the end of the for loop (the second for loop at line 14 in algorithm 17), the smallest of the d[] for which s[] is still t is guaranteed to be minimum. This is true because there is no possible alternate path, as every other possibility would already have

a larger initial distance than the one we just updated. So among the choices for the third round are nodes `C`, `D` and `E`. As `d[C]=5`, `d[D]=∞` and `d[E]=9`, we choose node C as the next current node (figure 10.7 c).

The next round updates the neighbours of node `C`. The only node reachable is `D`, so the new `d[D] = 2+5 = 7`. `D` becomes the next current node because it has a smaller `d[]` than node `E` (figure10.7 d).

Finally, node `E` is updated to `d[E] = 7 + 1 = 8`. It is the only node reachable from the `current` (`current` is node `D`). Now the only choice for the next round would be node `E`, and it becomes permanent. The next round of the `while` loop is now false, as every node is at state `p`, and the algorithm terminates. The final answers are the distances stored in the array `d[]`.

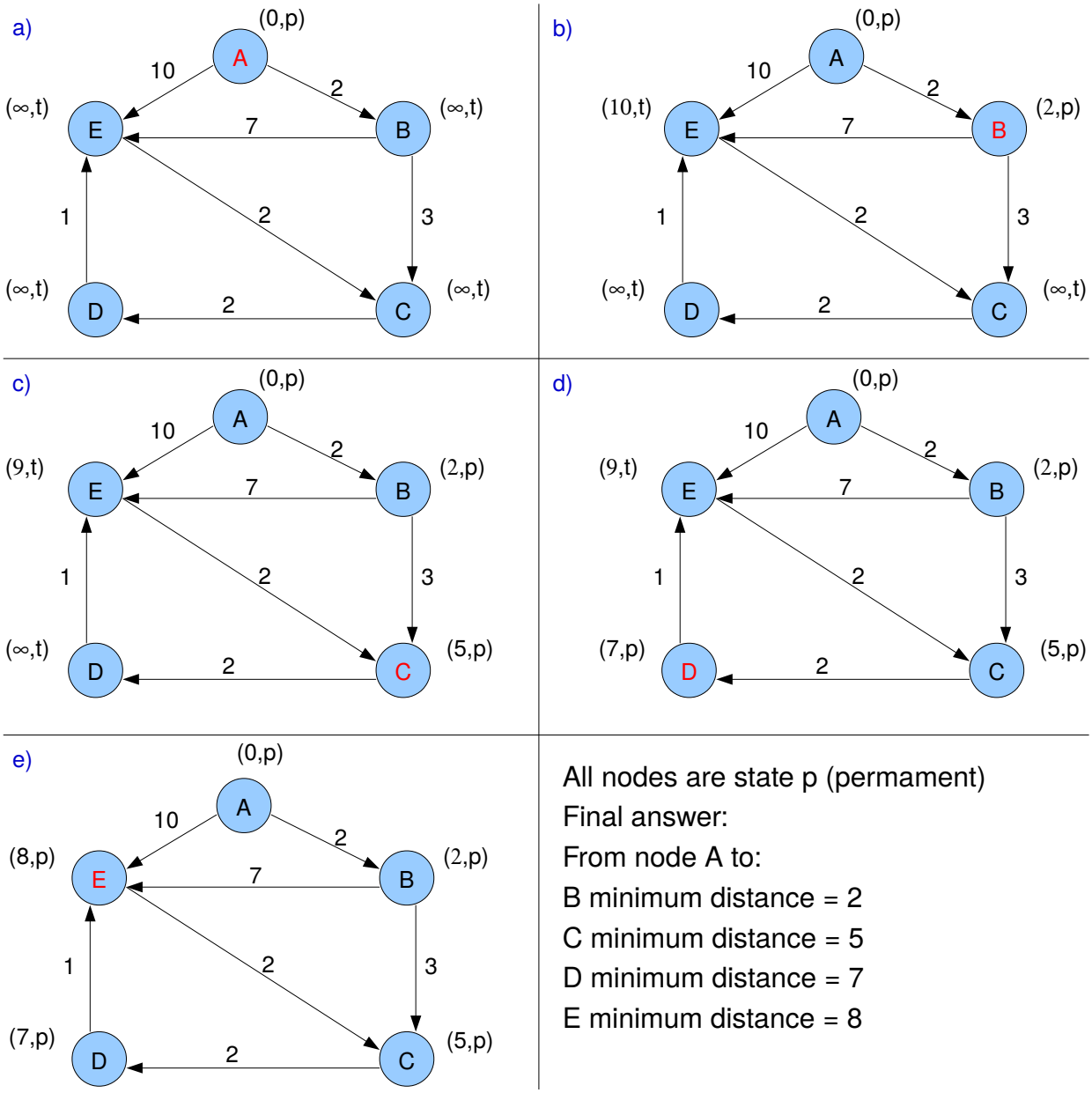


Figure 10.7: Dijkstra algorithm in action.

10.5 - Exercises

1. Draw the sets for the graph in figure 10.8.
2. Draw the adjacency matrix for the graph in figure 10.8.
3. Draw the adjacency list for the graph in figure 10.8.
4. Schematically run Kruskal's and Prim's algorithms in the graph of figure 10.8.
5. Schematically run Dijkstra's algorithm in the graph of figure 10.8 with source A. Repeat the run with every other node as the source.
6. Implement Prim's algorithm using the class studied in the previous sections.
7. Implement the Graph class studied in the previous sections. Consider how you would implement Dijkstra using this class. If the algorithm is implemented as a function, what considerations should you have about additional methods needed?

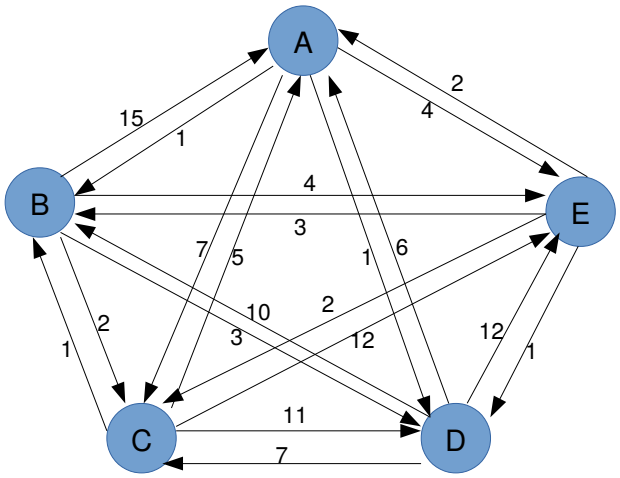


Figure 10.8: A graph for the exercises.

11 Sorting Algorithms

In this chapter we are going to study some of the well known sorting algorithms. The list of algorithms explored in this guide is not an exhaustive one, and there are many other alternatives for sorting.

Sorting is the arrangement of keys or indices (or anything that can be put in a certain order) in either ascending or descending order. In this chapter we are always going to sort in the ascending order. The algorithms we are going to look at in detail are: *selection sort*, *insertion sort*, *bubble sort*, *merge sort*, *quicksort*, *radix sort* and *heap sort*. In the next sections we describe each algorithm, present a simple implementation based on arrays and discuss their computational complexity ($O()$, or worst case scenario complexity). The somewhat simplistic complexity analysis is based on the *number of comparisons* used to sort.

All the listings in this chapter use simple arrays to explain the concept. The attentive reader will know that if the data size is not known in advance, one can easily change the container of the data to a vector. In some cases one can even use linked-lists to do the sorting, although this is not applicable to every sorting algorithm.

11.1 - Selection sort

In selection sort, we firstly mark an element as the `front`. We then traverse the rest of the array to find the smallest number. The smallest number is swapped with the `front`. The `front` index moves one position forward in every cycle (represented by `pass`). Figure 11.1 shows how the algorithm works schematically for the sequence 12, 4, 3, 9, 1. Listing 11.1 shows a simple C++ implementation.

Listing 11.1: Selection sort.

```
1 for (pass = 0; pass < n - 1; pass++) {
2   min = pass;  // min is an index
3   for (i = pass + 1; i < n; i++) {
4     if (data[i] < data[min]) { min = i; } // comparison
5   }
6   swap(data[min], data[pass]);
7 }
```

Analysing the number of comparisons (this is the same as the number of cycles we have to go through), we can say that the worst case scenario complexity is $O(N^2)$ (N is the number of elements in

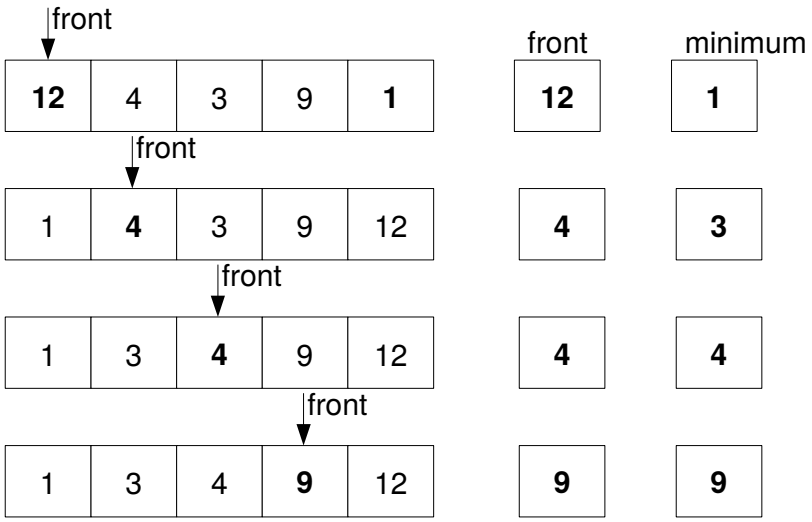


Figure 11.1: Selection sort.

the array to be sorted). Usually when we have double `for` loops as we have in listing 11.1 the algorithm is $O(N^2)$, but not always. One has to carefully analyse the `for` loop conditions to make sure that the algorithm is indeed quadratic.

In the case of the selection sort, the outer `for` loop repeats $N-1$ times. The inner loop changes the number of cycles depending on the `pass` value. This could give the impression that the algorithm is something less than quadratic, as the inner loop runs smaller and smaller cycles until at the last element it only runs once. However, the number of cycles can be seen as a triangle. The first run of the inner loop happens $N-1$ times also. The total number of cycles can be found by:

$$\frac{(N-1)(N-1)}{2} = \frac{N^2}{2} - N + \frac{1}{2}$$

As discussed before, for the complexity function we are only interested in the worst scenario trend. The equation above shows that the factor $N^2/2$ dominates the growth of the function, and therefore the complexity is $O(N^2)$.

11.2 - Insertion sort

Insertion sort uses a different mechanism. We only traverse the array once. We collect one number from the array and copy it. The next number is collected, but it is copied before or after the first one depending on their relative values. We then collect the third number and copy it, but carefully putting it in its right place in the copies array.

In practise it is possible to implement insertion sort using a single array and an extra variable to hold the current number (figure 11.2 and listing 11.2).

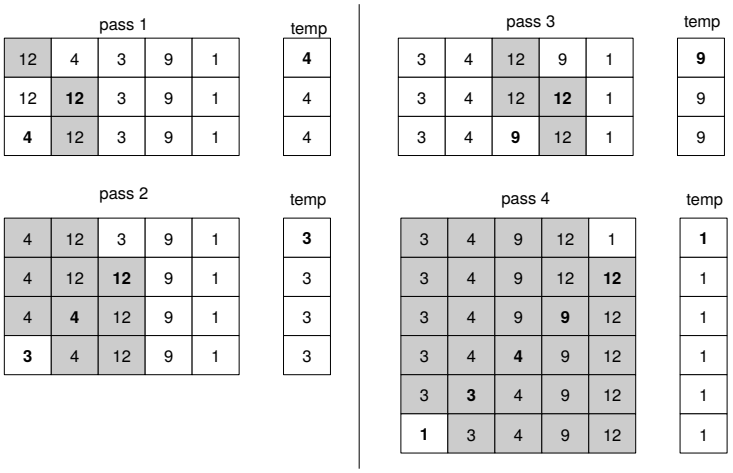


Figure 11.2: Insertion sort.

Listing 11.2: Insertion sort.

```
1 for (pass=0; pass < n-1; pass++) {
2   temp=data[pass+1];//note comparison in next line
3   for (i=pass + 1; i > 0 && data[i-1] > temp; i--)
4     {
5       data[i] = data[i-1];  // shuffling
6     }
7   data[i] = temp;
8 }
```

The inner loop shuffles portions of the array that are larger than `temp` to make space for it. The content of `temp` is then copied to the space left by the shuffling (where there is a repeated value). This algorithm could work really well for linked-lists. However, it would not make any difference in the complexity.

The outer loop runs $N-1$ times. The inner loop runs an unknown number of times (it will depend on the initial order of the array), as it depends on the comparison between the data and `temp`. In the worst case scenario, the inner loop will run $N-1$ times at some point (in figure 11.2 this happens in the last pass, four numbers are shuffled). This is a similar situation to what the selection sort had. The trend is still quadratic for insertion sort.

It is interesting to note that although selection sort and insertion sort have the same worst case scenario complexity of $O(N^2)$, they are going to have completely different number of comparisons for different inputs. It is easy to see that selection sort runs the same number of comparisons disregarding the initial order of the array. The insertion sort runs less comparisons if the input array is already sorted, but more comparisons if the input array is in reversed order. If the input array is in random order the number of comparisons stays between the number of comparisons when running the sorted and the reversed array.

11.3 - Bubble sort

Bubble sort uses pairs of numbers and drags the larger one all the way to the end of the array. It is called bubble because the algorithm works as if an imaginary bubble would be dragged all the way to the end, bringing with it the largest number for that pass.

Listing 11.3 shows a simple implementation of the bubble sort. Being that this is a simple code, it is not optimised yet. Figure 11.3 shows a running example with the array 12, 4, 3, 9, 1. The figure is based on what happens with the array and variables in listing 11.3. In this figure the bubble is represented by either a red pair or a grey pair. The red pair shows when a swap needs to be carried out. One red bubble in the pass suffices to change the state of the variable `swapping = true;`.

Note that there are five passes in this algorithm (N passes). In the first one, the number 12 is dragged all the way to the end of the array, placing 12 in the right place. In the second pass the bubble goes through the same positions, dragging with it the number 9 to the position before 12. The first round of pass 2 has also dragged 4 one position forward. So the bubble can modify the order of more than one element in one given pass. At pass 4 the only swapping occurs between 3 and 1. This causes the algorithm to go through yet another pass, even though the array was already sorted in pass 4.

The listing 11.3 can be improved in two ways. Firstly, one comparison in each pass can be eliminated. For example the comparison between 9 and 12 in pass 2 should not happen, as 12 was the largest number in pass 1 to be dragged all the way to the end. In pass 3, two comparisons (between 4 and 9 , and between 9 and 12) can be avoided. Secondly, pass 5 can be eliminated because in the worst case scenario at pass $N-1$ we only have the first pair of numbers out of order (see figure 11.3 pair 3-1 in pass 4).

Listing 11.3: Bubble sort.

```
1  bool swapping = true;
2  while (swapping) {
3    swapping = false;
4    for (i = 0; i < n-1; i++) {//N-1 bubbles
5      if (data[i] > data[i+1]) {//comparison
6         swap(data[i], data[i + 1]);
7         swapping = true;
8      }
9    }
10 }
```

Listing 11.4 implements the modified version of the Bubble sort. The two improvements described above where implemented by creating a variable `k` and modifying the `for` loop to keep track of how far the bubble needs to go within a pass. This also avoids the last pass, as when `k=1` the `for` loop is not going to happen.

Figure 11.3: Bubble sort.

What is the complexity of the Bubble sort algorithm? Let us first analyse Listing 11.3. The outer `while` loop makes a maximum of N passes, and the inner `for` loop happens $N-1$ times, so the number of comparisons is:

$$N(N-1) = N^2 - N$$

That would imply $O(N^2)$ complexity for the worst case scenario. The number of comparisons for listing 11.3 is $5(5-1) = 20$. Do the improvements implemented in listing 11.4 modify the complexity of the algorithm? After all, the number of comparisons is significantly smaller. For listing 11.4 the number of passes is $N-1$ and the `for` loop happens $N-1$, then $N-2$, then $N-3$ etc times. Indeed, applying the improved algorithm in figure 11.3 only makes 10 comparisons, half of that achieved with the first version of Bubble sort. However, we argue that the complexity of the algorithm remains the same. The algorithm is essentially the same on the two versions, but in the second the number of comparisons are optimised. The trend of the growth of the number of comparisons is still quadratic.

The complexity of listing 11.4 can be found by analysing the area of the triangle given by the number of comparisons in each pass:

$$\frac{(N-1)(N-1)}{2} = \frac{N^2}{2} - N + \frac{1}{2}$$

The complexity is indeed $O(N^2)$. The lesson here is that one can optimise an algorithm to minimise the number of cycles or operations, but this is not going to change the overall nature of the algorithm itself. Of course it is worth carrying out optimisations, but for large amounts of input data the growth of the runtime is still going to

be quadratic, although the runtime is perhaps half of that for the non-optimised version.

Listing 11.4: An improved Bubble sort.

```
1  bool swapping = true;
2  int k=n;//MODIFIED HERE: //marker for what is already sorted
3  while (swapping) {
4    swapping = false;
5    for (i = 0; i < k-1; i++) {//MODIFIED HERE: //don't look at
         any item > k, with k=k-1 at every step
6      comparisons++;
7      if (data[i] > data[i+1]) {//comparison
8         swap(&data[i], &data[i + 1]);
9         swapping = true;
10     }
11   }
12   k=k-1;
13 }
```

11.4 - Merge sort

Merge sort is composed of two algorithms, one that splits the set into two smaller parts, and the other that merges them in the sorted order. Merge sort is a very good example of the so-called "divide and conquer" strategy.

Let us start with the merge algorithm. If two sets, already sorted, are available, it is easy to merge them while guaranteeing that the resulting set is also sorted. The algorithm is very simple (algorithm 15). We stress the point that both arrays `A` and `B` have to be pre-sorted, otherwise the merged array `C` is not going to be sorted.

Algorithm 15 Merge

```
1: function MERGE(array A, array B, array C)        ▷ A and B already sorted
2:     compare the first item of A with the first item of B
3:     while (while there are elements in A or B) do
4:         choose the smallest and move it to C
5:         move to the next item of either A or B (from which the item was chosen)
6:     end while
7: end function                                      ▷ array C is now a sorted merge of A and B
```

A possible implementation of the merge algorithm as a C++ function is in listing 11.5. In this implementation, a single array is used and the sub-arrays A and B are defined by the variable `mid`. Note that in this arrangement, the sub-arrays are already sorted somehow, and the sub-arrays are combined using the same array (see figure 11.4). The algorithm uses three temporary variables (`i1`,`i2` and `i3`) to hold indices that will be compared. A temporary array called `temp[N]` is

used as the resulting array (`C` in algorithm 15). At the end, the values are copied back to the original array. The first `while` loop merges the numbers if there are still valid number in both sub-arrays. The second and third `while` loops are used if one sub-array is longer than the other (so to copy the remaining numbers to the resulting array).

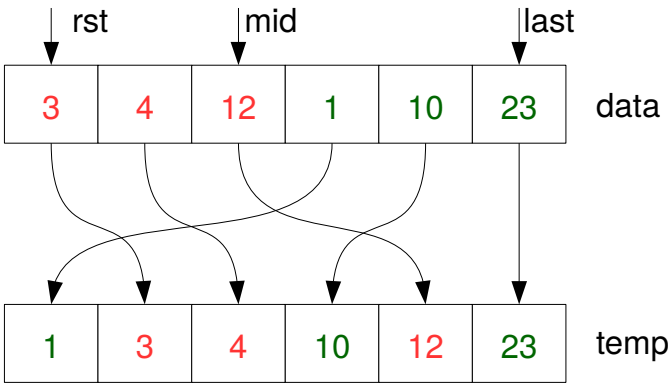


Figure 11.4: Merge example. The two sub-arrays are $\{3, 4, 12\}$ and $\{1, 10, 23\}$. The resulting array is sorted. Note that for the merge function to work both sub-arrays have to be pre-sorted.

Listing 11.5: Merge C++ implementation using a single array.

```
1  void Merge(int first, int last) {
2     int mid = (first + last)/2;
3     int i1 = first - 1, int i2 = first, i3 = mid + 1;
4     int temp[N];
5     while ((i2 <= mid) && (i3 <= last)) {
6       if (data[i2] < data[i3]) {
7          i1++;  temp[i1] = data[i2];  i2++;
8       } else {
9          i1++;  temp[i1] = data[i3];  i3++;
10      }
11    }   // may still be items in the first sub-array
12    while (i2 <= mid) {
13      i1++;  temp[i1] = data[i2];  i2++;
14    }   // may still be items in the second sub-array
15    while (i3 <= last) {
16      i1++;  temp[i1] = data[i3];  i3++;
17    }   // copy from temp back to data
18    for (i = first; i <= last; i++) { data[i] = temp[i]; }
19 }
```

The reader might think initially that the algorithm 15 is not very useful: after all, who is going to sort `A` and `B` beforehand? The trick to make it work is to split larger sets into smaller ones, to the point where a single number is left in `A` and `B`. Then the merging can occur without any previous sorting effort. To achieve that, it is convenient that the merge sort is implemented as a recursive function. The code is in listing 11.6. In the listing, the `MergeSort` function is called recursively. After doing the two halfs, the Merge is called, coalescing the two sub-arrays sorted previously. It is easy to follow the recursions and

to find out which parts are computed before and which parts are merged (figure 11.5). The figure shows the two sub-arrays in the first split as red (A) and green (B).

Note that in the `MergeSort` function whenever a recursion is called where `first = last` then it is going to return without going any further in the recursion. In other words, when the sub-array splits into a single item, there is no other sorting to be done and it can be immediately merged with its sub-array pair. The order in which the `MergeSort` and `Merge` occur is as follows: `MergeSort(0,4)`, `MergeSort(0,2)`, `MergeSort(0,1)`, `MergeSort(0,0)`, `MergeSort(1,1)`, `Merge(0,1)`, `MergeSort(2,2)`, `Merge(0,2)`, `MergeSort(3,4)`, `MergeSort(3,3)`, `MergeSort(4,4)`, `Merge(3,4)`, `Merge(0,4)`.

Listing 11.6: Merge sort (recursive).

```
1  void MergeSort(int first, int last) {
2     int mid = (first + last)/2;
3     if (first < last) {
4        MergeSort(first, mid);
5        MergeSort(mid + 1, last);
6        Merge(first, last);
7     }
8  }
```

Finally we need to analyse the complexity of Merge sort. How many comparisons do we have to do? The clue lies with the recursive nature of the algorithm, where the large problem is divided into halves, so at every recursion the problem is smaller. We are only interested in the number of comparisons between elements (data), not between the indices in the `while` loops. This gives an indication of how many cycles we have to go through before completing the task. If the two sub-arrays have the same size, the first loop in the `Merge` function will happen N times. Otherwise, if one sub-array is shorter one of the other `while` loops will compensate for it. Because MergeSort is recursive, the process will happen twice with a sub-array

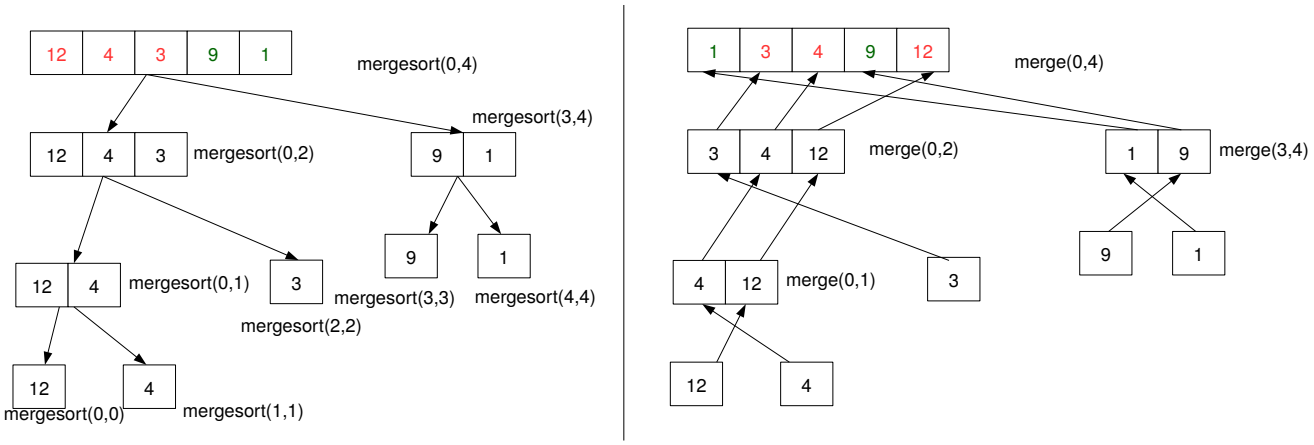


Figure 11.5: Merge sort. The figure on the left shows the `mergesort` function calls. The figure on the right shows the sub-arrays being merged together.

that is half of the size of the original (see lines 4 and 5 in listing 11.6). Each half is also divided in half, so in relation to the original size we have 4 times a quarter of the cycles and so on and so forth. In the worst case scenario with N a power of 2, the number of cycles is $2^k N/2^k$, with $N = 2^k$. So we can say that the number of cycles or comparisons is approximately:

$$N + 2N/2 + 4N/4 + 8N/8 \ldots 2^k N/2^k$$

We already explored a similar geometric sequence in the introduction section. The number of comparisons above form a geometric series (note the coefficient values) and it amounts to a multiple of $Nlog(N)$, so the worst case complexity is $O(N\,log(N))$. [1]. This complexity is slightly better than the quadratic complexity of the three previous algorithms we have studied (compare the curves in figure 1.4). Although `MergeSort` might be slower in runtime for small arrays, it is much faster than the quadratic algorithms when very large arrays have to be sorted. There is plenty of evidence for that, including the fact that Merge sort became popular when mainframes were still the prevalent hardware available to companies: the Merge sort could indeed save hours of computational time when compared to any other known algorithm at the time. Typically an algorithm that uses recursion to divide its data is going to have multiples of logarithmic complexity (in this case, multiplied by N).

[1] also called *linearithmic*

The huge advantage of Merge sort is its low complexity. However, depending on how Merge sort is implemented, it needs a copy of the data. This can became a problem for very large amounts of data. One alternative is to use a linked-list, so the elements are not reallocated (no copy) but just moved around. The implementation of the algorithm for a linked-list is very different than the one studied in this section. It is also possible to implement Merge sort with a non-recursive approach.

11.5 - Quicksort

Quicksort (one word) is also a divide-and-conquer algorithm. However, the approach is quite different than that of the Merge sort. The procedure consists of finding the correct place of one value (called the *pivot*) and running Quicksort separately on the left and right sides of the pivot. Eventually the entire array is sorted. The advantage of Quicksort is that it does not need a copy of the data.

One disadvantage of Quicksort is that the worst case scenario could be quadratic. We will see that the worst case scenario can only happen with a fraction of the possible initial states of the array (e.g. already sorted) and the choice of the pivot. Otherwise, the complexity should be linearithmic.

To illustrate the works of the algorithm, we have to make assumptions about the position of the pivot. Let us assume that the pivot is always the first element. The algorithm starts by searching for the

first number that is larger than the pivot and storing its position. A second search is carried out, but this time the search is done from the higher index to the lower indices. In the second search, the first number that is smaller than the pivot has its position stored (figure 11.6). As the larger-than-pivot is larger than the smaller-than-pivot (in the figure, 37 > 19), then they have their positions swapped. The search up continues, and it finds that 61 is the next larger than the pivot. In the other direction, it finds that 11 is the next smaller than the pivot. The reader should be convinced that once the larger from the left and the smaller from the right meet (they have neighbouring indices), then there is no point in continuing the search. Indeed, by now all the items on the left of 11 (including itself) are smaller than the pivot, and all the items that are larger than the pivot are on the right side of 11. The last step for this part is to fit the pivot between the smaller and the larger. Although the array is still not sorted, we can guarantee that the pivot (26) is in its right position in relation to all other items.

Once the pivot changes to its correct position within the array, it is time to call Quicksort recursively. Note that now the two halves (the left and right of the pivot 26) can run independent instances of Quicksort. They are not going to change the fact that 26 is positioned in the correct index. The example of the recursive Quicksort continues

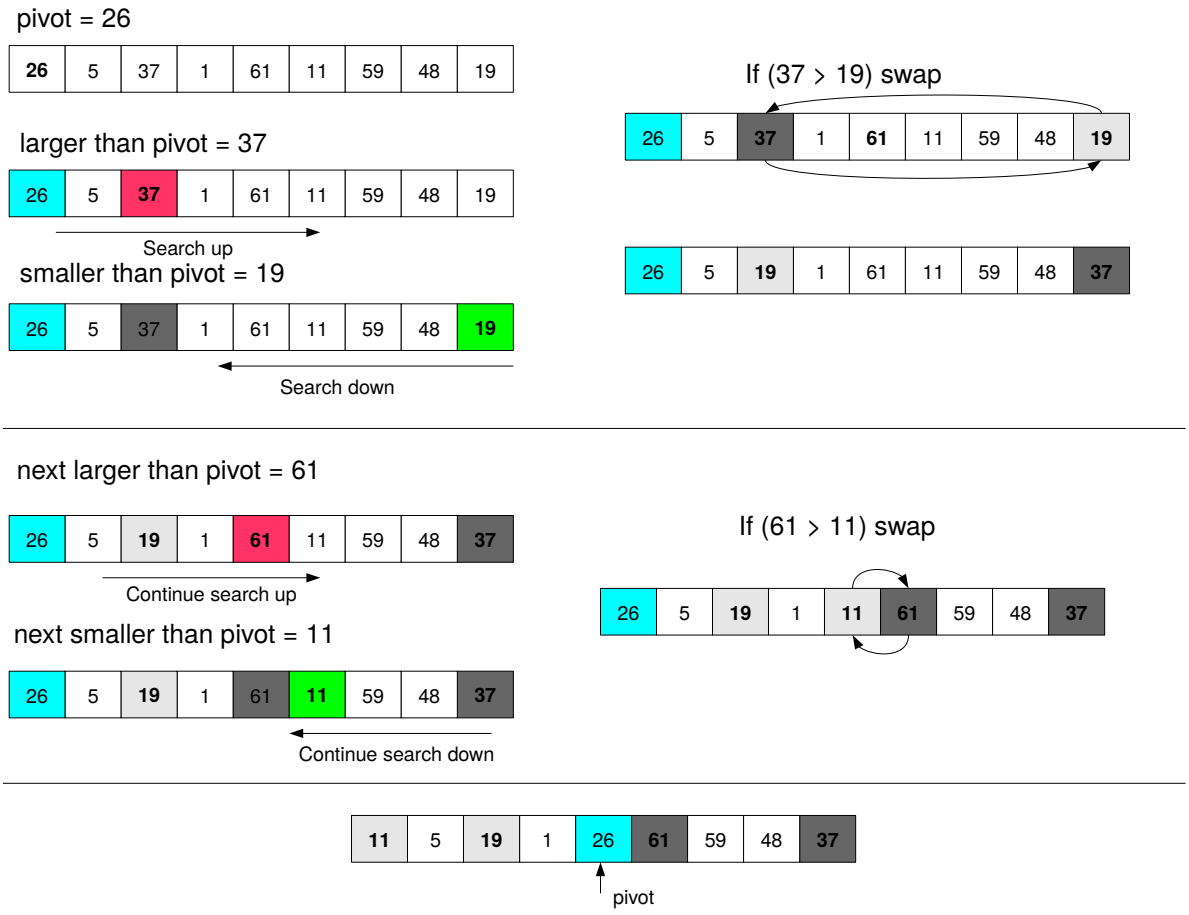


Figure 11.6: Quicksort: searching larger and smaller values in relation to the pivot. At the end the pivot is in the correct position.

in figure 11.7. Note that for every Quicksort call one pivot (different for every instance) will be positioned correctly. In theory one would have as many recursive calls as N. However, when the sub-array has only one item it no longer needs any sorting, and it has to be in the correct position within the original array. So after only 6 Quicksort calls, the array with 9 numbers is sorted.

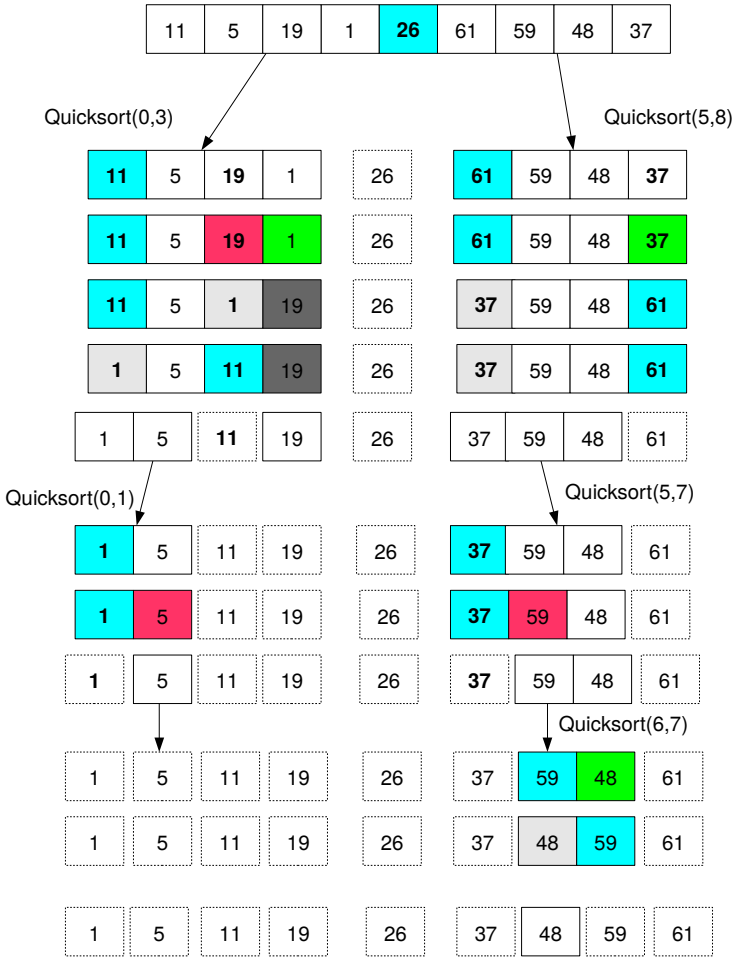


Figure 11.7: Quicksort: further recursive calls continuing from figure 11.6. This figure shows that when the pivot is the median value of the set, it splits the array into two sub-arrays of equal sizes. The choice of the pivot has a critical influence on the performance of Quicksort.

Listing 11.7 shows a simple implementation where the pivot of the sub-problems is always the first element. The code is illustrated by figures 11.6 and 11.7. In the listing, the outer `while` loop controls the position of the search (index `i` cannot be larger than index `j`) and the inner `while` loops makes the search upwards or downwards. The `if` statement in the outer loop swaps the larger with the smaller to position them in one of the halves. The first `if` statement after the outer `while` loop takes care of moving the pivot to its position. The two other `if` statements decide if there is going to be a recursive call or not. Note that whenever `first=j-1` or `j+1=last`, then the Quicksort recursion is *not* called (that would be calling the Quicksort with a sub-array of one item).

Listing 11.7: Quicksort.

```
1  void Quicksort(int first, int last) {
2   int i = first+1, j = last, pivot = data[first];
3   while (i < j) {
4     while ( (data[i] < pivot) && (i<last) ) { i++;}
5     while ( (data[j] > pivot) ){ j--; }
6     if (i < j) {swap(&data[i], &data[j]); }
7   }
8   if(pivot>data[j]){swap(&data[first], &data[j]);}
9   if(first < j-1) { Quicksort(first, j-1); }
10  if(j+1 < last) { Quicksort(j+1, last); }
11 }
```

11.5.1 - Complexity of Quicksort

How many comparisons do we do? Considering the comparisons only of the data (with the pivots), we have a maximum (often less) of $N-1$ comparisons. If the choice of the pivot is bad (for example, always the smallest element), then the partition will always be an empty sub-array on the left and an $N-1$ array on the right. Figure 11.8 illustrates a small example. In this figure, case 1 illustrates an array that is ordered and the pivot is always the first item. In the worst case scenario, the number of comparisons will be:

$$\frac{(N-1)(N-1)}{2} = \frac{N^2-2N+1}{2}$$

We can state that in the worst case scenario, for the worst possible choice of pivot in all cycles, and for a specific input order of the array, the complexity is $O(N^2)$. However, this is rarely the case with Quicksort. Experiments with the runtime show that Quicksort is twice as fast than any other $O(N\,log(N))$ algorithm such as Merge sort. This demonstrates that the worst case scenario is rare for Quicksort, and with the right choice of pivot the average case is more likely.

To make the best case complexity analysis simple, let us assume that the pivot is always the median value of the sub-array [2] The assumption is just to get an idea of the complexity, as this would be the ideal position for the pivot because it would split the rest of the array into two halves of equal size (the left and the right would have approximately the same number of items). This assumption implies having divided the sub-arrays into approximately $N/2$ size. So the number of comparisons (cycles) could be approximate to:

[2] Trying to use the median value of the entire set to choose the pivot is not practical, as it would imply in sorting before sorting. There are ways to randomise the position of the pivot. Yet another way is to use the median value of the set $\{first, middle, last\}$ positions within the array to be partitioned.

$$(N-1)+2(N-3)/2+4(N-5)/4+8(N-9)/8\ldots 2^k(N-2^k+1)/2^k$$

The equation above also amounts to a multiple of $N\,log(N)$ as it happened with Merge sort. So the best case scenario for Quicksort is also $O(N\,log(N))$. It is out of the scope of this guide, but the average case can also be proven to be $O(N\,log(N))$.

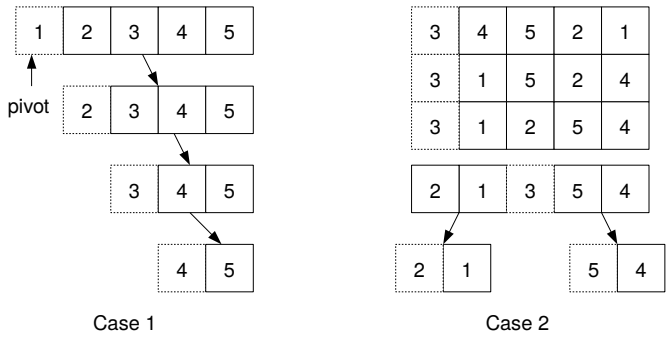


Figure 11.8: Quicksort: two cases where the pivot is chosen differently. Case 1 takes 4 splits, while case 2 only takes 2 splits to sort.

11.6 - Radix sort

Radix sort is an interesting algorithm because it is very intuitive. When trying to sort a pile of magazines by date for example, you may have had the idea to pile them up by year first. Then going into each pile of year, sort the month. This idea is very easy to extend if you think that decimal numbers have a positional approach, with each digit representing powers of 10. A similar idea for sorting magazines can be applied to decimal numbers: we start by making a pile (or buckets) for each one of the units (0 to 9). We then visit each pile taking elements from units and putting them into piles that represent the tens. We repeat the process, this time looking at the third significant digit (the hundreds) and so on and so forth. At the end of the process, the numbers are sorted.

Radix sort requires a separate data-structure to copy the elements to. This could be a stack or a queue. In listing 11.8 an example of a queue implementation is presented. This implementation does not use comparisons (not between elements of the array, that is). Instead the place on each queue (the queues are acting as "buckets") is found by a simple operation, in this case dividing by `factor` and getting the remainder of the division by `radix`. The queues are continuously filled up with the entire array, and then emptied again. Also, we assume

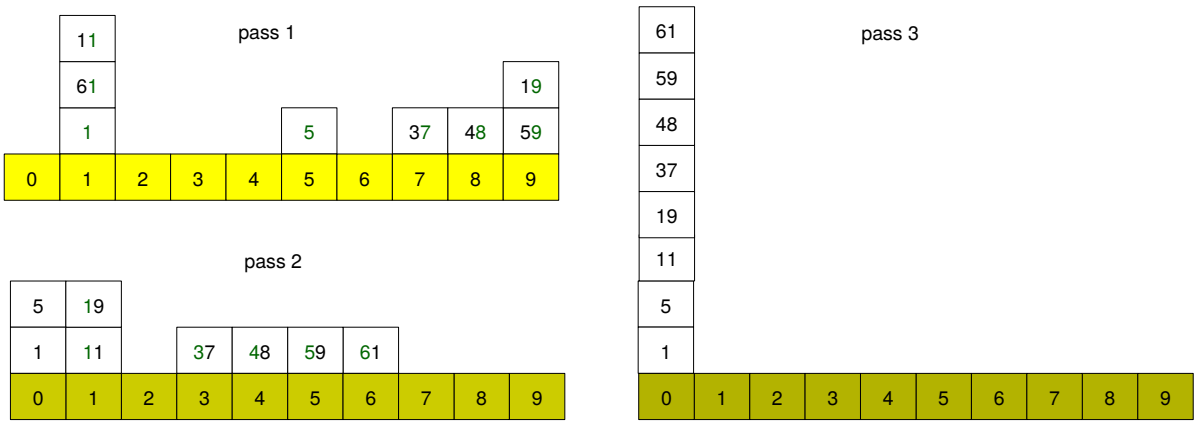


Figure 11.9: Radix sort. The indices for the queues are indicated by the yellow squares. The `front` of each queue is the closest number to the yellow square.

that each number has zeros on the left, so for example number 5 is going to queue 0 in pass 2. After three passes, all the numbers were copied to the first queue ('0' for the hundreds, or the third most significant digit). Note that reading the queue in order (`Front()` followed by `Leave()`) prints the numbers in sorted order. The last pass can be avoided, reading the queues from pass 2 and directly storing them into the array will also solve the sorting problem.

Now we have the problem of finding the complexity of Radix sort. If we look at listing 11.8, it is not very clear how many comparisons are actually made and what the role of the queues in the complexity is. Any algorithm's complexity is at least the complexity of the largest $O()$ of one of its components. Analysing the queue's operations, we know that if this is implemented as a linked-list the Join, Front and Leave are all constant (represented by $O(1)$). So the queue cannot influence the complexity, although it can influence the run-time of a certain implementation. The outer `for` loop (statement 4 in listing 11.8) will go through the same number of cycles as the number of digits, but it is not dependent on N. The first inner `for` loop (line 5) runs N times. The second for loop (line 8) runs `radix` times, and the `while` loop (line 9) runs the same number of elements in the queue `j`. In other words, the second inner `for` loop with the `while` loop are going to call Front and Leave N times, and that is the number of data values copied from the queues.

Listing 11.8: Radix sort.

```
1  void Radixsort(int *data, int n) {
2    int i,j,k,factor,radix=10,digits=3;
3    Queue queues[radix];//array of our Queue class...
4    for (i=0,factor=1;i<digits;factor*=radix,i++){
5      for(j=0;j<n;j++) {
6        queues[(data[j]/factor)%radix].Join(data[j]);
7      }
8      for (j=k=0;j<radix;j++){
9        while (!queues[j].isEmpty()){
10         data[k++]=queues[j].Front();
11         queues[j].Leave();
12       }
13     }
14   }
15 }
```

So it seems that in terms of cycles, the complexity is going to be a multiple of N, or $O(N)$, or linear complexity. However, the runtime might not be very good when compared to other sorting algorithms because the computational cost of dealing with queues (where every element is allocated and deallocated repeatedly) may hinder a good runtime. Also, a copy of the data is required when using the queues, so it has a similar problem to Merge sort in terms of memory use. In the available literature sometimes the linear complexity of the Radix

sort is shown as $O(wN)$, indicating that there might be a significant multiplier to N. If N is small in relation to w, this can tend to become quadratic. In other words, if the number of queues (variety of the items) is large in relation to N, then the complexity is somehow more expensive than a plain linear algorithm.

Note that the radix and the number of digits somehow limit the variety of numbers that show up. With only three digits per number, if the array contains thousands of items many of them are going to be repeated. It is interesting to note that using a different radix (the base of the number representation) and the number of digits can change the way the algorithm is implemented, resulting in a different runtime for the same set with N items (remember: the complexity is still the same). For example, one could use binary (`radix=2`), which is smaller than 10. But binary numbers have more digits than their decimal representations, so the outer loop runs more cycles. On the other hand, instead of using the modulo to find the corresponding queue, one can use a bit-wise operator, which is faster. Also, in the binary version of Radix sort there are only two queues (either index 0 or 1).

11.7 - Heap sort

Finally, the last of the sorting algorithms in this chapter is the Heap sort. This algorithm transforms the data in a Heap (remember the Heap binary tree from chapter 8?). For the Heap sort, the process consists of building up the Heap (copying the array) and deleting the root of the Heap until the Heap is empty. During the deletion, we store the key of the Heap being deleted. In doing so, we get the data sorted.

One could use a max heap (where the keys of the nodes are always larger than that of its children) or a min heap (where the keys are smaller). Either way, it is a matter of copying the data with the correct index to get it sorted or in reversed sorted order. The following algorithm summarises the procedure:

Algorithm 16 Heap sort

```
1:  function HEAP_SORT(array A, int n)
2:      Declare a new Heap H;
3:      for (i=0; i<n, i++) do
4:          insert A[i] into H      ▷ This can be achieved using listing 8.2
5:      end for
6:      for (i=(n-1); i>=0, i-) do
7:          copy the root's key to A[i];
8:          delete the root of H    ▷ This can be achieved using listing 8.3
9:      end for
10: end function
```

In algorithm 16 the first loop copies the data from the array into a new Heap. The second loop keeps deleting the root of the Heap

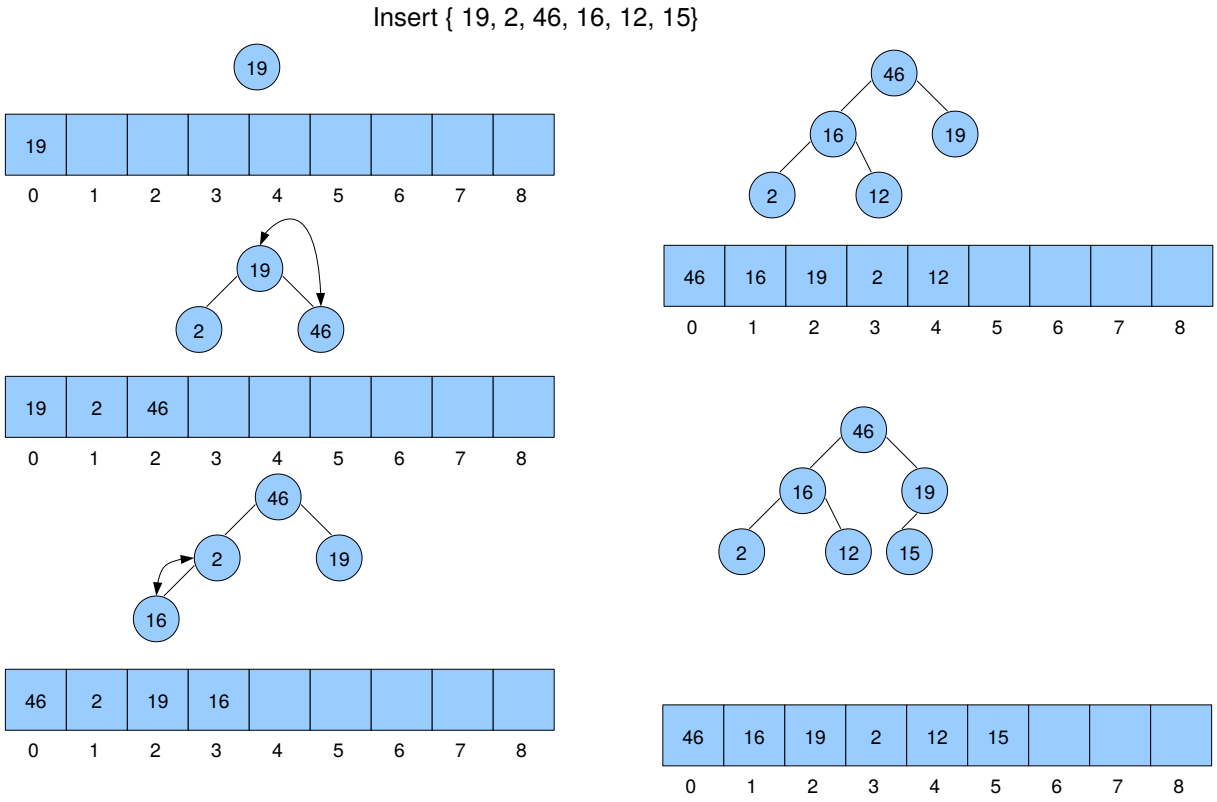


Figure 11.10: Heap sort: copying the input 19,2,46,16,12,15 into a heap.

until it is empty, while copying the data back into the array. The copy uses the reversed order of the index of the array to achieve the correct sorting order. The two main tasks of the Heap sort are the insertion and the deletion, which were already studied in chapter 8.

When deleting from the root, the root is swapped with the last element (lowest right-most leaf), and therefore the key in the root is smaller than it should be. The minimum number of comparisons for a key to go down the heap is two. The key should swap with the largest of its children. So we can compare the children, and than compare the largest child with the parent.

The first phase of algorithm 16 is illustrated in figure 11.10. In this figure, the input array $19, 2, 46, 16, 12, 15$ is used. Note that for every new insertion, the heap has to be fixed by swapping the new node with its parent until the parent is larger than the inserted node. This operation is $log(N)$ because every time the key climbs the number of possible parents is divided by two.

The final phase of the algorithm is illustrated in figure 11.11. In this figure only the first three deletions are represented. Every time a root is deleted, a new root has to replace the old. The most effective way to do that is to initially swap the root with the last node and subsequently delete the last node (e.g., swapped 46 with 15). Due to the properties of the heap, the root is going to be the largest item in the tree. So 46 can safely be copied to position `n-1` in the array (represented in green). After deleting 46, the new root 15 should be swapped with one of its children if it is smaller than one of them. 15

is swapped with 19, the largest of its children. After the swap the heap can be taken to the next loop, which will delete 19.

The analysis of the complexity of the Heap sort might be a bit trickier than what it seems at a first glance. One could be deceived into believing that the complexity is linear. After all, both `for` loops run exactly N times. However, the two functions being used implicitly are not of constant complexity. The insertion requires that the Heap is fixed after inserting the element at the last position. The fixing requires the traversal of the heap upwards. The insertion is $log(N)$, similar to that of a binary search (also for the same reasons). The deletion of the root also has a complexity $log(N)$ because the root is swapped with the last element, and the last element might need to be swapped with one of its children until the Heap is fixed. In the worst case scenario, the total number of cycles required by the Heap sort is:

$$N\,log(N) + N\,log(N) = 2N\,log(N)$$

Therefore the complexity is $O(Nlog(N))$, similar to Quicksort and Merge sort. If we want to analyse how many comparisons we have to do for the insertion and the deletion, there is an easy rule. When

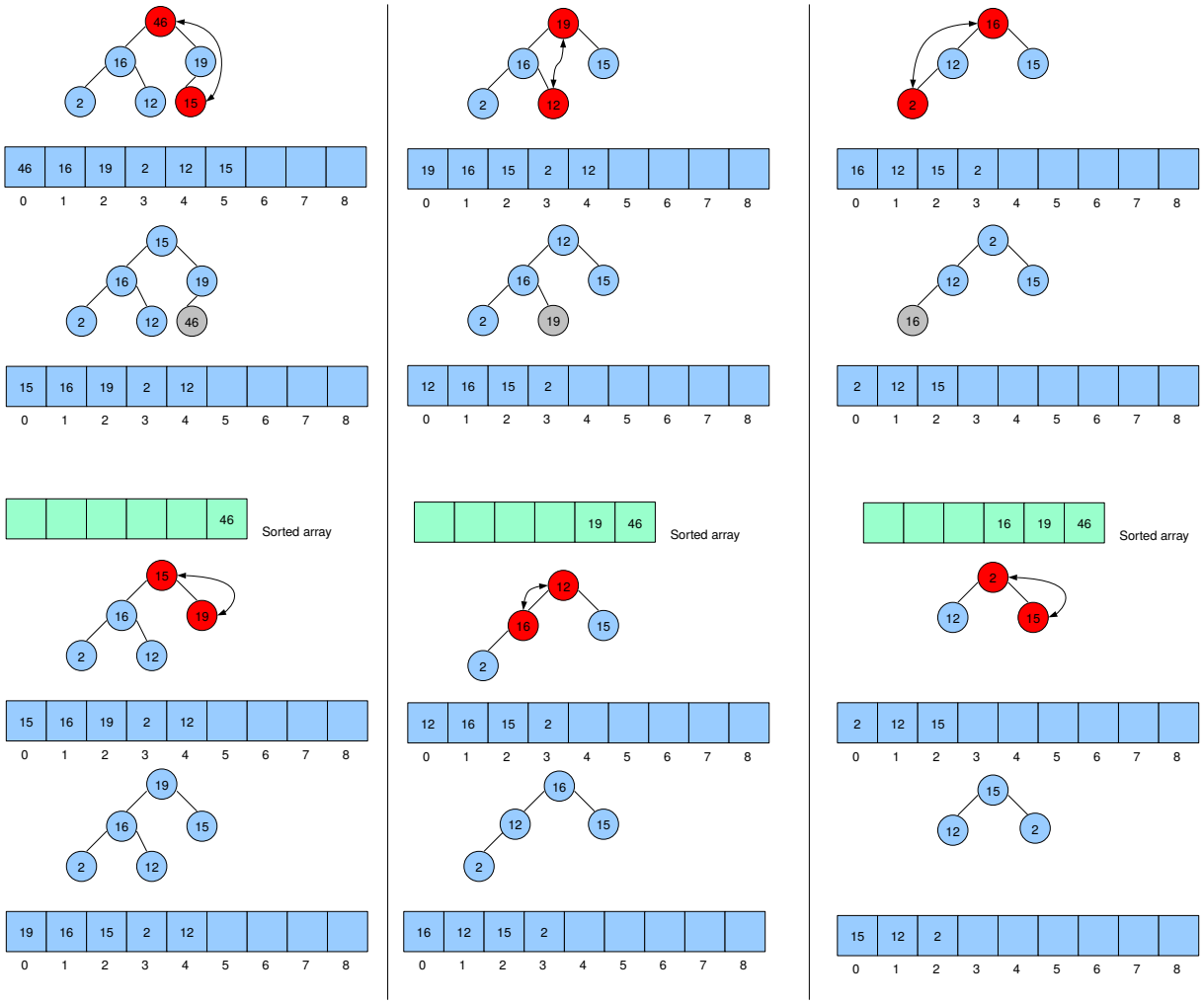


Figure 11.11: Heap sort: deleting the root of a heap while copying the key to the array.

inserting, there is a comparison between the inserted node and its parent. If they swap places, then the parent has to be compared to its parent and so on and so forth, until the either the swap does not occur, or the swap reaches the root of the Heap. So for every attempt to get the new key upwards there is one comparison.

11.8 - Summary

The time complexity of each of the algorithms studied in this chapter is summarised in table 11.1. Note that the only algorithm that has different worst-case complexity different than average-case complexity is Quicksort. Typically Quicksort is the algorithm of choice for numerical sorting of small and large data. For very small amounts of data, especially if the data storage is static, sometimes it is convenient to use one of the $O(N^2)$ algorithms because in that case the runtime may be faster than Quicksort using recursion.

Sorting algorithms	**worst-case complexity**	**average-case complexity**
Selection sort	$\mathbf{O(N^2)}$	*same*
Insertion sort	$\mathbf{O(N^2)}$	*same*
Bubble sort	$\mathbf{O(N^2)}$	*same*
Merge sort	$\mathbf{O(N \log(N))}$	*same*
Quicksort	$O(N^2)$	$\mathbf{O(N \log(N))}$
Heap sort	$\mathbf{O(N \log(N))}$	*same*
Radix sort	$\mathbf{O(wN)}$	*same*

Table 11.1: *Time* complexity for different sorting algorithms. These complexities can be used to help to choose algorithms depending on how large the datasets are. The lower complexities are preferable if the sets are large. The runtime of the quadratic algorithms are not going to be any worse in performance when compared to the lineartithmic ones for small sets. For practical purposes we should consider the complexities shown in bold (Quicksort is usually linearithmic and the worst case can be avoided with a clever implementation).

11.9 - Exercises

1. Write a complete program that implements all the sorting algorithms seen in this chapter. Include a form of measurement of cycles (e.g., count the number of data comparisons) and store different results for different data. Vary the size of the data and also the initial state of the array (e.g., ordered, reversed and random). Build a table using a spreadsheet and compare the results of the different algorithms.

2. Implement the radix sort using binary numbers and avoiding integer divisions (use the bit-wise operators to figure out which queue the numbers go).

3. Compare the runtime between the bubble sort in listing 11.3 and the improved bubble sort in listing 11.4. Run both versions with increasing N and plot the runtime to see what is the trend. Use curve fitting to get an equation for the runtime as a function of N.

4. Write a non-recursive C++ code for Quicksort. Is there any advantage in doing so?

5. For the array 1,2,3,4,5,6,7,8,9,10, we run Quicksort. In algorithm 1 the pivot is always the first element. In algorithm 2, the pivot index is in the middle of the array (e.g., if N=5 pivotindex =2, if N=4 pivotindex is also 2). Show how many recursive calls we would carry out for both algorithms. Repeat the exercise with 5,7,9,1,6,2,3,10,4,8 and 10,9,8,7,6,5,4,3,2,1.

6. A list of different algorithms and their O( ) time-complexities are shown in table 11.2. Assuming that the run-time (in seconds) with a problem size of N=100 are known, how long could we expect them to compute a problem size of N=1000? And N=10000?

	Measured runtime $N = 100$	**Measured runtime** $N = 1000$	**Measured runtime** $N = 10000$
$O(N)$	3		
$O(Nlog(N))$	4		
$O(N^2)$	2		
$O(N^3)$	2		

Table 11.2: Estimating runtime given the complexity.

12 Searching Algorithms

In this chapter we are going to compare simple forms of searching (seen before when we dealt with linked-lists and BSTs) with a more sophisticated form of search called *hashing*.

12.1 - Searching Algorithms

When we needed to search for keys in a linked-list, it was clear that in the worst-case scenario we would have to traverse the entire linked-list, just to find out that the key is either in the last element or not in the linked-list at all. This is clearly a $O(N)$ approach for searching.

The scenario changed when we used a BST for searching. Due to the way the keys are organised inside a BST, the complexity for the worst-case scenario is $O(log(N))$, as the problem is divided by two every time the current pointer moves to another level of the binary tree.

Another approach to the binary search where the complexity would also be $O(log(N))$ is when using data that is already sorted, and accessible randomly (via an index). In this case, the search of a key starts in the middle of the structure (it could be implemented as an array or vector). If the data is already sorted, the search key is compared to the key in the middle. If it is smaller, we only need to look at the left side of the array, otherwise look at the right side. It is easy to implement a recursive function to achieve just that, without the use of a BST.

There is another way to search for keys that has an even better complexity than the previously discussed algorithms: hashing. In principle, hashing's worst case scenario is constant $O(1)$. However, there are strict conditions for the state of the data and the hashing table in order for that to happen. In practise only static data can achieve such a low complexity. It is very difficult to keep the searching cost low when the data changes fast. This is discussed in the next section.

12.2 - Hashing

When using a hashing algorithm, the assumption is that the key is unique. The key an be defined in a very loose way, which can include parts of the data of the record itself, as long as it is unique. Similarly to a BST, non-uniqueness would cause the search to not find every

instance of a possibly duplicated key.

Hashing relies on a *hash function* that computes the index where the record is located. The hash function receives the key as a parameter. In order to locate the record on that particular index, hashing algorithms also use *hash tables* to help placing the data. The hash table can be a very simple form of lookup table, or a more complex entity that contains a link to a list of records.

12.2.1 - Some Examples

The simplest example of hashing uses parts of the data itself to find the location of records. In this simple approach keys are unique, but the index might not be unique. Therefore, the hash table may have empty entries, and multiple entries for certain indices.

Let us start with a simple example using part numbers for some equipment. The keys are numerical in this example, and the has function is simply the keys themselves. Let us suppose that there are only 100 possible keys, and at the moment the content is of four part-numbers:

part-number	48	26	99	58

Table 12.1: Random keys to be searched.

The table in this case needs at least 100 entries (size 100) to accommodate the records. The *occupation* of the hash table is low (only 4% in this case), and if we need more records than 100 we would have to recreate the table. On the other hand, in order to make the table more occupied to avoid wasting space in the hash table, we would need some simple hash function that makes it smaller. For example, the function could be the integer division by 10. Using the same example and a new hash function:

$$index = \texttt{(int)}\frac{partnumber}{10} \tag{12.1}$$

However, this would create a new problem: a *collision*. It is easy to see why. Given the keys in the above example, the items in the hash table are 4, 2, 9 and 5. The problem is if a new unique key is inserted, e.g., 57, now two distinct keys will yield the same index 5. Although the occupancy of the hash table is 40%, the collision poses a problem for the search accuracy.

12.2.2 - Choosing a Good Hash Function

One of the problems in finding good hash functions is that nobody knows the distribution of the keys in advance. New keys can be inserted at random, and keys may be deleted at random. Even knowing the current set of keys would not help in getting a perfect hash function.

One of the ways to illustrate the last paragraph is the so called *birthday surprise*. What is the probability of finding two people in a room who have the same birthday? If the room was populated at

random, intuition tells us that this would be a rare event. However, statistical analysis of the phenomenon shows us that it is more common than one would expect, especially if the group of people is small. It is trivial to understand that if 367 people are in the room, then the probability of finding two with the same birthday is 100%. But how many people do we need in the room to reach a 50% probability? Remarkably only 24 people. The birthday surprise tell us that collisions are very likely to happen.

We follow the discussion with some examples of hash functions that are used in practise.

The simplest one was to use the key itself as an index, and we saw the drawbacks of doing that. Other functions are: modulo, truncation, folding and multiplication.

Listing 12.1: Hash functions.

```
index = hash(key); //hash returns key
```

The modular arithmetic hash function is simply:

Listing 12.2: Hash functions: Modular arithmetic.

```
int modular_hash(key){
  return key % N; //N is the number of entries in the
      hash table
}
```

It seems that modular hashing works better on average if the divisor N is a prime number.

Truncation uses parts (digits) of the original key. This is very useful if the key is alphanumeric. For example, a key `ba_2_3_4_part` would get a hash index of `234` in listing 12.3.

Listing 12.3: Hash functions: Truncation.

```
int truncation_hash(&key){ //key is a string
  char temp[3];
  temp[0] = key[3];
  temp[1] = key[5];
  temp[2] = key[7];
  return atoi(temp);
}
```

Folding is the recombination of parts of keys into an index. For example, if the key is `124381920` we can combine the number in groups of three digits and add them up to make the index. In this case, we always assume that the keys have 9 digits (e.g., pad with zeros on the left if the key has less digits). In the example, the index becomes:

$$index = 124 + 381 + 920 = 1425$$

It is trivial to compute the table size, the maximum value would be $999 + 999 + 999 = 2997$ entries if we consider that the keys start with `000000001`. Yet another way of recombining parts of the key is the use of bit wise operators, e.g. see listing 12.4.

Listing 12.4: Hash functions: Folding.

```
int folding_hash(&key){ //key is a string
  int index = 0;
  index = key[0] ^ key[1] ^ key[2];
  return index;
}
```

Multiplication uses the fractional part of a key multiplied by a number `T` (where `T` is between zero and one). It seems that a good value for `T` is the golden ratio (0.618033). It tends to distribute random keys evenly. In listing 12.5, N is the table size and `fraction()` is a function that returns the fractional part of a float number.

Listing 12.5: Hash functions: Multiplication.

```
int multiplication_hash(&key){ //key is a string
  int index = 0;
  index = (int) (N * fraction (key*T) );
  return index;
}
```

12.2.3 - Resolving Collisions

So far we have been trying to find a good hash function that is able to distribute the keys evenly and without too much wasted space in the hash table. If the table is occupied more than 50% it would still be feasible to use such a simple approach. However, with more keys inserted and deleted, it is unlikely to stay at this level of occupancy. In order to keep the hash table organised and avoid collisions two main methods could be used: open hashing and closed hashing.

Open Hashing

Chaining is a common way of avoiding collisions. In chaining a linked-list could be used to hold records for which the keys generated the same index. This avoids collisions but it does not guarantee any occupancy rate. Also, it means a compromise in the search worst case scenario: it might no longer be of constant time. An additional linear search is required once the index is found, and if many keys fall into the same linked-list this can be significant to the runtime.

Closed Hashing

The simplest way to avoid collisions without using linked-lists is to use a *rehash* function. When a collision occurs (easily detected by checking an existing key), then one could select a new index. Moving

the item to the next index is a common strategy (see listing 12.6). This is called "moving on" or linear probing. Another common way is to generate a random number (k) to move the item away from the collision. This imposes a problem though, because either the number k has to be stored in the search rehash or the approach will cause more delays on the search.

Listing 12.6: Rehash functions.

```
index=hash(key);
if(collision) index=rehash(index,key);

int rehash1(index, key){
  while (!collision){
    index=index+1;
  }
}

int rehash2(index, key){
  while (!collision){
    index=index+k;
  }
}
```

12.3 - A Simple Hashing Implementation Example

In this section a very simple example of hashing is presented. The hash function is a simple modular arithmetic function, followed by a re-hash that keeps moving one position until no collisions occur. In this example, we also try to search for each one of the keys inserted, as well as searching for some key that is not in the table. In the worst case scenario (the key does not exist) we still have to do a linear search. However, for most of the keys the search cost is constant (see listing 12.7)

Listing 12.7: A complete hash example.

```
#define HASHSIZE 11

struct somedata {
  int id;
  char name[100];
};

class Hash{
private:
  somedata hashtable[HASHSIZE];
  int size;
public:
```

```
  Hash();
  int hash(int &id);
  int rehash(int &id, int &i);
  int insertdata(somedata &data);
  int removedata(somedata &data);
  void printtable();
  bool searchtable(int id);
};

int Hash::hash(int &id){
  return (id % HASHSIZE);
}

int Hash::rehash(int &id, int &i){
  return ( (id+i) % HASHSIZE);
}

int Hash::insertdata(somedata &data){
  printf("inserting %d %s \n",data.id, data.name);
  if(size<HASHSIZE){
    int hashed=hash(data.id);
    if(hashtable[hashed].id==-1){//empty position
      hashtable[hashed].id=data.id;
      strcpy(hashtable[hashed].name,data.name);
      size++;
      return 0;
    }
    else {//rehash ...
      while(i<HASHSIZE){
        hashed=rehash(data.id,i);
        if(hashtable[hashed].id==-1){
          hashtable[hashed].id=data.id;
          strcpy(hashtable[hashed].name,data.name);
          size++;
          return 0;
        }
        i++;
      }
      return -1;//could not find an empty slot
    }
  }
  return -1;//Table is already fully occupied
}

bool Hash::searchtable(int id){
  int attempts=0;  int hashed=hash(id);
  if(hashtable[hashed].id==id) {
    printf("found record in first attempt\n");
    return true;}
```

```
62    else{
63      int i=0;
64      while(i<HASHSIZE){
65        attempts++;
66        hashed=rehash(id,i);
67        if(hashtable[hashed].id==id){
68          printf("found record %d attempts\n",attempts);
69          return true;
70        }
71        i++;
72      }
73    }
74    printf("record is NOT in the table (after %d attempts)\
          n",attempts);
75    return false;
76  }
```

12.4 - Cuckoo Hashing

This is a relatively novel technique (2001) where two different hashing functions are used. Each item has two possible positions in the table, depending on which hash function is used. When a new item arrives in the table it "kicks out" the item currently in that spot. The displaced item moves to an alternative position (hopefully vacant). If not, kick out the item, so the next displaced item moves too.

Sometimes an infinite loop starts (we know that because a cycle of kick-outs start). A new pair of hash functions is needed in this case. Another way to look at Cuckoo hashing is to use two tables, as shown in the following pseudo-code (algorithm 17).

Figure 12.1 a) shows an example of Cuckoo hashing using one table. A moves to B, which moves to a vacant place. C and P could also move. However, H and W are on a cycle, so a new hash() has to be found.

Figure 12.1 b) shows the use of two tables. Example with T1 and T2: K replaces A, A replaces B, B finds a new place in T1.

12.5 - Perfect Hashing

There are cases where all the data is known in advance, e.g.: dictionaries, data on read-only devices, a table of reserved words for a compiler etc.

In a minimal perfect hash there are no collisions (no waste of time resolving it) and no waste of space in the hash table. In other words, the number of keys is the table size. Every key will find a unique position within the table. This is the only case where the complexity $O(1)$ for the searching can be guaranteed. Note that this only happens with static data. One could build a perfect table for dynamic data, but as soon as a new element is inserted the table would have to be

Algorithm 17 Cuckoo Hashing

```
1:  function CUCKOO_HASHING(K) ▷ Inserting key K ▷ assume that
    two tables, T1 and T2, are available
2:      if (K is in T1[h1(K)] or in T2[h2(K)]) then
3:          do nothing
4:      end if
5:      for (i=0 to maxLoop-1) do              ▷ avoids infinite loops
6:          swap(K, T1[h1(K)]);;               ▷ displace position in T1
7:          if (K==-1) then
8:              return;                        ▷ success, empty slot in T1
9:          end if
10:         swap(K, T2[h2(K)]);                ▷ collision, displace in T2
11:         if (K==-1) then
12:             return;                        ▷ success, empty slot in T2
13:         end if
14:     end for
15:     rehash();              ▷ could not get a place, new T1 and T2
16:     insert(K);
17: end function
```

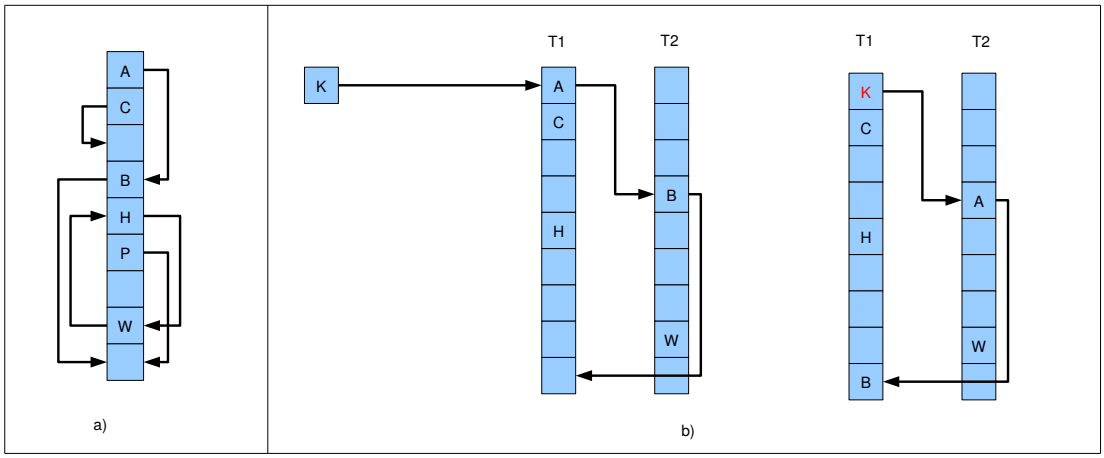


Figure 12.1: Cuckoo hashing. a) Example of a cycle. b) Using two tables

rebuilt. The rebuilding of the table would cost more than constant time.

12.5.1 - Cichelli's Method

Cichelli's method is an algorithm devised in 1985. He used it to produce a faster PASCAL compiler. This method uses the characteristics of the words that compose the keys. The simplest example could use: the length of the word, its first letter, and its last letter. Therefore, the hash function is:

$$h(key) = (length + g(word[1]) + g(word[last]))\ mod\ Tsize \qquad (12.2)$$

Where: `g()` is the function for each letter, `Tsize` is the size of the hashing table (`Tsize`=number of keys). One needs to determine g(letter) depending on the contents of the set of keys. This is the

merit of Cichelli's method. He found a systematic way of building the hashing table.

The method consists of:

1. Determining the occurrence of first and last letters in all keys.
2. Getting the sum of the occurrences, then sort
3. Finding `g(word[first])` and `g(word[last])` in such a way that *no collisions occur*.
4. If collisions still occur after the whole round, removing the word by *backtracking* and putting it at the end of the list.

A pseudo-code of the method is presented in algorithms 18 and 19.

In order to get an understanding of the method, it is better to have a clear example. Using the example from the literature, lets assume that the list contains the following words:

Calliope, Clio, Erato, Euterpe, Melpomene, Polihymnia, Terpsichore, Thalia and Urania (the nine Greek muses).

Considering only the first and last letters of each word, we can count the number of occurrences for each letter:

- E = 6
- A = 3
- C = 2
- O = 2
- T = 2
- M = 1
- P = 1
- U = 1

According to the occurrence of each letter, we compute the sum of the first and second letter occurrences for each word and sort them out from largest to smallest. E.g., EuterpE would add to 12 (E=6 occurs twice), and UraniA would add to 4 (A=3 and U=1). Sorting the words, they should be listed as: EuterpE, CalliopE, EratO, TerpshichorE, MelpomenE, ThaliA, CliO, PolyhymniA and UraniA. Note that when two words' sum are equal, their order would not matter.

We can then start to fix the function `g()` for specific letters. Initially, we can hypothesise that `g()` returns zero for every letter, so for example `g(E)=0`. If the hash function returns a unique index, then we adopt the values for g() for the first and last letter of the word without changing the functions g() for those letters.

The hash function for the first word EuterpE yields an index of: $7 + 0 + 0$ This is so because the hash function is getting the number of the letters plus `g(E)=0` plus `g(E)=0` (the `g()` function for the first

Algorithm 18 Cichelli

```
1: function ConstructHash(List)                    ▷ without considering backtracking
2:     word = next word from the List;
3:     if ( word[0] and word[last] already have g() functions  then
4:         try(word,-1,-1);                        ▷ -1 meaning g() are assigned
5:     end if
6:     if (success) then
7:         go back to the start                    ▷ go to the next word
8:     else
9:         if (neither have g() ) then
10:            for ( each (n,m) ) do
11:                try(word,n,m);
12:                if (success) then
13:                    go back to the start        ▷ go to the next word
14:                end if
15:            end for
16:        else
17:            for (each n) do
18:                try(word,-1,n) or try(word,n,-1)
19:                if (success) then
20:                    go back to the start        ▷ go to the next word
21:                end if
22:            end for
23:        end if
24:    end if
25: end function
```

and last letter of the word). As 7 was not taken by any other word yet, we can fix `g(E)=0` and include Euterpe in an array at position 7.

Then we get the next word from the sorted list. We again hypothesise that we can keep `g()` returning zero, unless a collision occurs (two words with the same index). When the collision occurs, we keep trying to modify the `g()` function for one of the letters (first or last) of the word where the collision occurred. If we find a new value for the `g()` function that suits the table, then the problem is solved. However, it is possible that the two `g()` functions are already fixed by a previous word. In this case the only solution is to backtrack until the `g()` is freed, and change the order of the words.

Let us complete the example. On the left side we compute the hash functions and fix the `g()` for each letter that occurs in the words. On the right side we mark the indices that are already committed with a word.

$EuterpE\ \ g(E) = 0,\ h = (7 + 0 + 0)\ mod\ 9 = 7$ $\{7\}$

$CalliopE\ \ g(C) = 0,\ h = 8$ $\{7, 8\}$

$EratO\ \ g(O) = 0,\ h = 5$ $\{5, 7, 8\}$

$TerpsichorE\ \ g(T) = 0,\ h = 11\ mod\ 9 = 2$ $\{2, 5, 7, 8\}$

$MelpomenE\ \ g(M) = 0,\ h = 9\ mod\ 9 = 0$ $\{0, 2, 5, 7, 8\}$

$ThaliA \;\; g(A) = 0, \; h = 6$	$\{0, 2, 5, 6, 7, 8\}$
$CliO \;\; h = 4$	$\{0, 2, 4, 5, 6, 7, 8\}$
$PolyhymniA \;\; g(P) = 0, \; h = 1$	$\{0, 1, 2, 4, 5, 6, 7, 8\}$
$UraniA \;\; g(U) = 0, \; h = 6$	*collision*!
$UraniA \;\; g(U) = 1, \; h = 7$	*collision*!
$UraniA \;\; g(U) = 2, \; h = 8$	*collision*!
$UraniA \;\; g(U) = 3, \; h = 0$	*collision*!
$UraniA \;\; g(U) = 4, \; h = 1$	*collision*!
$UraniA \;\; g(U) = 5, \; h = 2$	*collision*!
$UraniA \;\; g(U) = 6, \; h = 3$	$\{0, 1, 2, 3, 4, 5, 6, 7, 8\}$

Algorithm 19 try()

```
1:  function TRY(word, a, b )              ▷ a and b are number for g()
2:      if (a != -1 AND b != -1) then                    ▷ both assigned
3:          compute h(word);
4:          if (h(word)is not claimed) then
5:              reserve (h(word));
6:              return success;
7:          else
8:              return failure;          ▷ only solution is to backtrack
9:          end if
10:     else
11:         if (a == -1 OR b == -1 ) then          ▷ either one assigned
12:             assign g(word[first]=a OR g(word[last]=b;
13:             compute h(word);
14:             if (h(word)is not claimed) then
15:                 reserve (h(word));
16:                 return success;
17:             else
18:                 return failure;
19:             end if
20:         else                                    ▷ none yet assigned
21:             assign g(word[first]=a;
22:             assign g(word[last]=b;
23:             compute h(word);
24:             if (h(word)is not claimed) then
25:                 reserve (h(word));
26:                 return success;
27:             end if
28:         end if
29:     end if
30: end function
```

The algorithm could get stuck without finding a perfect hash table. In this case, one solution is to backtrack and try a brute force approach changing the order of the words. If the number of words is too big for a brute force approach, then a g() function in the middle of the word could be added to the hash function.

There is a complete implementation example on Stream. You can experiment with different sets of words to see how the algorithm produces a perfect hash table for most cases. In this implementation the g() function is simply filling up an array, so given a letter it will return the corresponding value in from the array. This is very efficient, as when the table is constructed the array is also built, so it can be incorporated into the program later for searching purposes only. This is the ideal approach for a static dictionary (for example, it could be embedded on a device).

12.6 - Exercises

1. Assume that N = 16, the hash function is (i % 16), and collisions are handled by closed hashing using the "moving on" method. Show the resulting table if the following numbers are inserted in this order: 1 4 9 16 25 36 49 64 81 100 121

2. Repeat question (1) using the rehashing function index = (index + 4) % 16, instead of linear probing. What problem arises?

3. Use Cichelli's method to devise a perfect hash table (and function) for the following words: DO, WHILE, FOR, END, THEN, OR, AND, LABEL.

13 Special Problems

13.1 - Strategies for Algorithm Development

There are many ways of writing algorithms. In this section a few strategies (or design paradigms) are briefly presented.

13.1.1 - Brute Force

Also called exhaustive search. In this approach we devise an algorithm that computes all the possible solutions for a problem and then the best solution is selected. The advantage of such an approach is the guarantee of optimisation. However, if the problem size is large the runtime might become infeasible.

13.1.2 - Divide-and-conquer

In this approach the problem is divided into smaller problems of the exact same type. We already studied two algorithms that use this approach, Merge sort and Quicksort. Binary search also uses the same strategy. The advantage of the divide-and-conquer approach is the smaller complexity (and corresponding runtime), but that sometimes comes with an extra cost: memory. Often the runtime complexity is lowered at the same time as the memory cost is increased. This is the case with Merge sort.

13.1.3 - Dynamic Programming

In dynamic programming, one starts from a partial solution and keeps modifying the solution until it achieves a better one. [1]

One of the simplest algorithms that use this approach is Dijkstra's algorithm. Note that in Dijkstra's method the cost of a path is temporarily larger than the optimal until it is modified, and eventually minimised. Dijkstra proved that the algorithm always reaches the minimum cost for every destination.

However, algorithms using dynamic programming often reach the so-called *local minimum* (there might be multiple local minima, hence the reference to local minima). Therefore, despite the fact that this approach is usually fast, one cannot guarantee optimisation until analysing the problem to see whether local minima can possibly occur.

[1] The name *dynamic programming* was coined by Richard Bell in the 1950s. It has nothing to do with *programming* in the sense that we understand it nowadays. It would better fit the concept of planning. At the time computers were not widespread, and programming languages were very primitive and tied to the specificities of the hardware available. The same happens with the name *linear programming*.

Sanjoy Dasgupta, Christos Papadimitriou, and Umesh Vazirani. *Algorithms*. McGraw-Hill Higher Education, Boston, MA, 2008

13.1.4 - Greedy Algorithms

Greedy algorithms take a short-cut by eliminating solutions that would not be solving the problem, or that would be sub-optimal solutions. It is not easy to find greedy rules that can help an algorithm to take these short-cuts. A simple example is the computation of primality of a certain integer number. The exhaustive search would include trial divisions for every number less than N. It is easy to see that a short cut would be to limit that to numbers less than $\sqrt{N}$ rather than up to $N-1$. It turns out that there are number theory rules that can shorten the primality test even further, although primality tests remain a very difficult task when N has a few hundred digits.

13.1.5 - Linear Programming

If a system of simple linear equations can describe a problem, optimisations can be done in a way to solve these equations.

A classical algorithm using this approach is called *Simplex*. The algorithm is simple to implement and works fast. The issue with linear programming is that often solutions that are inherently non-linear are approximated to a linear problem. This could lead to very bad solutions.

13.1.6 - Reduction

Reduction uses very sophisticated ways of transforming the original problem into one where a lower complexity algorithm is known. A good example is the multiplication of large integers. The grade school method takes $O(N^2)$ for the multiplication of two numbers, each with N digits (note that the base does not matter). Multiplication of very large numbers might take too long. A solution was found by using Fourier transforms. Transforming the way the numbers are represented and using Fast Fourier Transforms (FFT) takes the complexity of the multiplication to $O(Nlog(N)log(log(N)))$, a bit less than quadratic. The time saved is only worth for very large numbers (hundreds of thousands of digits). For small numbers the grade school method is still the fastest one.

13.1.7 - Search and Enumeration

Search and enumeration is a generic paradigm that uses enumeration structures (where the solutions or partial solution can be stored) and rules to cut sets of solutions out of the analysis. In this category, genetic algorithms, Monte Carlo algorithms (probabilistic), approximation algorithms using branch-and-bound etc are included. The search and enumeration method can solve a very broad category of problems in computer science. However, algorithms may still be slow and there is no guarantee of optimality.

13.2 - Traveling Salesman Problem (TSP)

This is an example of a classical problem in computer science and mathematics. In fact, this is one of the most studied problems in this class of algorithms. It involves the optimisation of the sum of the costs of travelling through a graph.

The problem was formulated mathematically in the 1930s, although forms of the problem were known since the 1800s. The problem consists of finding the shortest path that visits all the nodes in a graph (only once) and returns to the origin node. The problem is deceitfully hard to solve. It does not seem related in any way with Dijkstra's solution for finding the shortest path from one node to another.

There are many real-life applications in logistics, but also in computer networks, electronics, DNA sequencing etc. Simple applications in logistics include delivery, e.g., a postman has to leave office, deliver all letters, and goes back to the office. A simple example in electronics involves robotic machines that need to test circuits by probing specific points of the chips. One wants the robot probe to take as little time as possible to test a chip, and therefore one needs to solve the TSP for that circuit considering that each probing point is a vertex of a graph.

It would be nice if we knew a low complexity solution, but we do not. There are many exact solutions that are slightly better than brute force (exhaustive method) such as dynamic programming, but all these solution have much worse than polynomial complexities. There are lots of approximate solutions though (so-called heuristics), and these are usually good enough for practical purposes. Although in practice we can find optimal solutions for graphs with thousands of vertices, these solutions are still not satisfactory because of the lack of a polynomial exact solution. Engineers and computer scientists are usually happy with these approximations, but not mathematicians.

13.2.1 - Exhaustive Method

One algorithm that can solve the TSP is an exhaustive search of all possible paths that go from the origin, pass through all nodes and come back to the origin. In a fully connected graph this is very easy to code. However, are we ever going to find a solution? Does the runtime gets so big that we have to wait a lifetime for the answers?

Let us analyse the case of a fully connected graph with N nodes. Starting from a certain node, we can only get $N-1$ possible paths from the origin. Every one of these will arrive at a second node, and we have $N-2$ possible paths for each case, so the total is so far $(N-1)(N-2)$ paths. The next part of the paths are multiplied by $N-3)$ and so on, so we have:

$$(N-1)(N-2)(N-3)...1 = (N-1)! \tag{13.1}$$

Some of these paths will be repetitions. In fact, the $(N-1)!$ paths pair up, so the total number of paths is divided by 2. The complexity of such an algorithm is $O(N!)$. This does not look very hard until you

do the maths for the number of paths for a relatively small number of nodes.

For the example, let us use the case of the major urban areas of Europe. There are about 79 urban areas in the European Union that can be recognised as major urban areas (economic importance, population, infra-structure, transport hub etc). Suppose that these 79 cities can be linked to each other directly (that is not the case, but just for the sake of using a fully connected graph we can assume this). The number of possible TSP paths is $78!/2$. This is approximately 5.66×10^{114} paths. We should estimate the runtime of an exhaustive algorithm. Being very optimistic, suppose that one could evaluate one path per CPU cycle. A 1 Gflop CPU would be able to run $365.25 \times 24 \times 60 \times 60 \times 10^9 = 31557600000000000$ paths per year. The estimate runtime for running the algorithm on a desktop machine is:

$$\frac{78!/2}{31557600000000000} \approx 1.79 \times 10^{98}\ years \approx 1.79 \times 10^{89}\ billion\ years \tag{13.2}$$

The result is not very encouraging, one would wait quite a long time for the result. How about using a supercomputer? The sum of all top 500 supercomputers yields about 1 ExaFlop (equivalent to 1000000000 times 1 GigaFlop computers), so we can divide the number of years by that:

$$\frac{1.79 \times 10^{89}\ billion\ years}{10^9} \approx 1.79 \times 10^{80}\ billion\ years \tag{13.3}$$

It is intuitive to conclude that no amount of computational infrastructure would help to alleviate the problem. When the runtime of an algorithm running a problem of size N is too long, we say that it is *infeasible* to compute (while theoretically possible, the runtime is not practical in any sense). And yet you can check on the Internet and realise that solutions with thousands of nodes have been achieved. How is that so? Firstly it is possible to solve the TSP for larger graphs that are not fully connected, and where the edges of the graph are a representation of the Euclidean distance between the nodes. Also, it is possible to have approximate solutions even for the fully connected graphs. One of the simplest and easiest to implement is called branch-and-bound.

13.2.2 - Finding approximate solutions for the TSP using Branch-and-Bound

TSP paths can be created with a general Tree. Each node of the tree generates children of nodes that were not previously used in this path. This is best illustrated in an example of a fully connected graph with few nodes, as in figure 13.1.

Note that the number of paths for a 4 nodes graph would be $\frac{3!}{2} = 3$. The figure shows 6 paths, but they pair up, For example, the path A-B-D-C-A is the same as A-C-D-B-A, just in opposite order.

The Branch-and-Bound method uses a simple approach of starting with an exhaustive analysis of a partial solution and pruning the solutions that do not look promising. Instead of taking every path to the end, the branch-and-bound approach trims the tree of solutions, eliminating the longer ones so far. Using a fixed threshold of 6, we can limit the cost at a certain level to multiples of 6, so at level one we only keep costs that are less than or equal 6. At level 2, the limit would be 12, at level 3 the limit would be 18 and so on.

Figure 13.2 shows the same graph using the branch-and-bound algorithm. The nodes in grey indicate that the path was abandoned, saving runtime for other more promising sub-paths. Eventually we get a short path, but there is no guarantee that it is indeed the shortest. We might have abandoned a path that did not look promising (above the threshold), but it could be the shortest in the long run. With small graphs of figure 13.2 this may not happen, but it is easy to produce a simple graph with more nodes where this is the case. In the branch-and-bound method, if the threshold was high enough to eliminate all the paths, we could backtrack and recompute with a smaller threshold until at least one path could be generated.

13.2.3 - Dynamic Programming

This solution makes a small change in the way of thinking. Instead of considering the many possible paths that exist, it find combinations of vertices with minimum cost. It eliminates many duplications of partial paths that would otherwise be used in an exhaustive approach. In order to organise these partial solutions it is convenient to use recursion.

According to [2], the complexity of the classical dynamic programming solution by is $O(N^2 2^N)$. The solution needs a lot of memory space: $O(N2^N)$. Despite the fact that the complexity is smaller than the brute force approach, it still can only solve relatively small problems. For example, for a 30 cities problem the amount of memory needed would be approximately 4GB. With only 35 nodes this amount

[2] David L. Applegate, Robert E. Bixby, Vasek Chvatal, and William J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, Princeton, NJ, USA, 2011

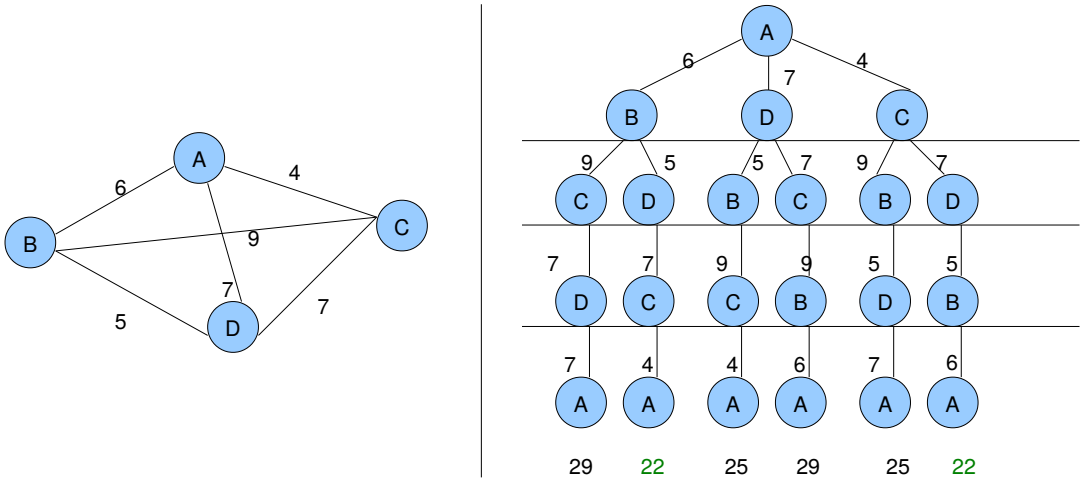


Figure 13.1: Exhaustive search for a solution for the TSP.

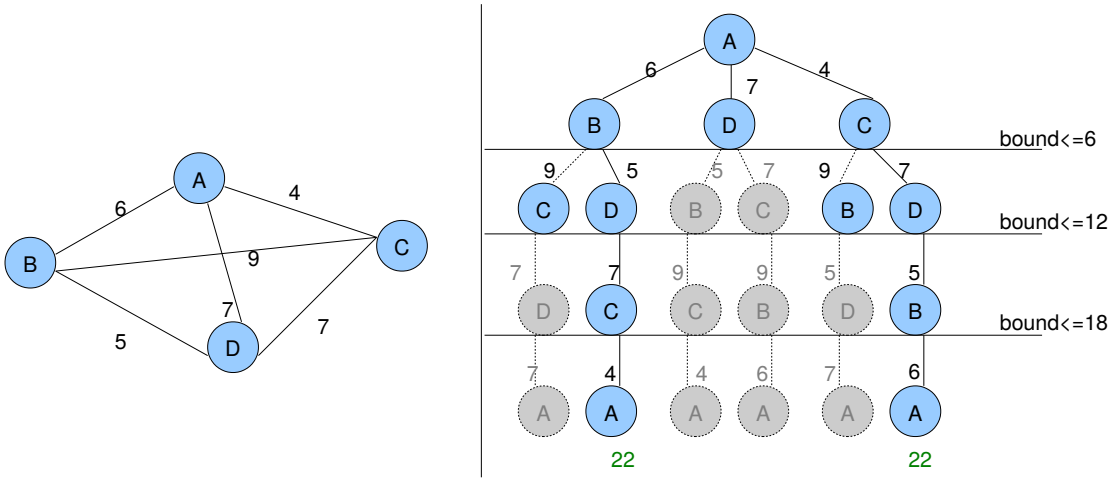


Figure 13.2: Branch-and-bound search for a solution for the TSP.

would grow to 128GB, above the capacity of ordinary computers.

In order to solve small problems with dynamic programming the process should start by finding the minimum path for every node $i \neq A$, and each path should include all other nodes in the set. [3] As we want the path to go back to A, and considering a minimum cost from A to i using all other nodes, the total cost for a certain minimum path will be $cost(i) + dist(i, A)$, where $cost(i)$ is the path from A to i and $dist(i, 1)$ is the direct cost to go from i back to A.

[3] It is tempting to think that Djisktra's algorithm would help to determine this minimum cost. This is a different path though, it should involve ***all*** the nodes in the set S.

One can use recursion to compute all the possible paths from 1 to i. The recursion eventually leads to subsets of only 2 nodes, for which the cost is the edge between these two nodes (the direct path from A to i). The recursive calls return their minimum to compose all possible subsets for 3 nodes, then 4 nodes and so on, until we have paths with all the nodes in the graph. This explains why dynamic programming is faster than brute force. Instead of considering all possible paths, the minimum cost of partial paths (involving a certain subset of nodes) is computed and stored in memory, cutting many possible paths that would cost more than other paths involving the same subset of nodes.

Lets call the minimum cost using a certain subset $C_{min}(S, i)$. Once the minimum cost for this subset is found, it locks the path for future use. The minimum implies in using the nodes belonging to sub-set S in a certain order. The subset includes the origin and several nodes all the way to node i. For every one of these sets, we have to keep adding nodes, as long as they are different than A and i. For every node j added to the subset, the minimum cost is:

$$C_{min}(S, i) = min\{C(S - \{i\}, j) + dist(j, i)\} \tag{13.4}$$

where j belongs to S, $j! = i$ and $j! = A$.

In other words, equation 13.4 states that the minimum cost of going from A to i using the nodes in set S can be computed using the minimum cost of going from A to some other node j plus the distance from j to i. This equation can be broken down all the way until the subset S contains only two nodes, for which the answer is known.

For a practical example, lets consider the same undirected graph of figure 13.1. However, note that the order in which the calculation is done in this example is exactly the opposite order that happens in a recursive algorithm.

$$\begin{bmatrix} 0 & 6 & 4 & 7 \\ 6 & 0 & 9 & 5 \\ 4 & 9 & 0 & 7 \\ 7 & 5 & 7 & 0 \end{bmatrix}$$

We start with all the possible i nodes with sets of two nodes. The minimum path for them all is the direct cost from A to i and back.

The subsets are $\{A,B\}$, $\{A,C\}$ and $\{A,D\}$:

$C_{min}(\{A,B\},B) = 6$
$C_{min}(\{A,C\},C) = 4$
$C_{min}(\{A,D\},D) = 7$

These partial results can be used to compute paths with more nodes. For every node j added to the subset, the minimum cost of a path including can be broken down using equation 13.4.

Lets generate all the possible sets for subsets of size 3:

$C_{min}(\{A,C,B\},B) = C_{min}(\{A,C\},C) + dist(C,B) = 4+9 = 13$
$C_{min}(\{A,D,B\},B) = C_{min}(\{A,D\},D) + dist(D,B) = 7+5 = 12$
$C_{min}(\{A,B,C\},C) = C_{min}(\{A,B\},B) + dist(B,C) = 6+9 = 15$
$C_{min}(\{A,D,C\},C) = C_{min}(\{A,D\},D) + dist(D,C) = 7+7 = 14$
$C_{min}(\{A,B,D\},D) = C_{min}(\{A,B\},B) + dist(B,D) = 6+5 = 11$
$C_{min}(\{A,C,D\},D) = C_{min}(\{A,C\},C) + dist(C,D) = 4+7 = 11$

Note that the minimum costs $min\{C(S-\{i\},j)$ for sets of size 2 were determined by the previous recursion, so they were available for the computation of size 3. Now we can generate all the sets of size 4, using the previously computed minimum costs for size 3.

$C_{min}(\{A,C,D,B\},B)$ has two possibilities:

$C_{min}(\{A,D,C\},C) + dist(C,B) = 14+9 = 23$
$C_{min}(\{A,C,D\},D) + dist(D,B) = 11+5 = 16$

Therefore the minimum path from A to B using C and D is 16. The TSP solution would involve going from B to A to close the cycle. We still have to see if the last node before A could be some other node.

$C_{min}(\{A,B,D,C\},C)$ has two possibilities:

$C_{min}(\{A,D,B\},B) + dist(B,C) = 12+9 = 21$
$C_{min}(\{A,B,D\},D) + dist(D,C) = 11+7 = 18$

$C_{min}(\{A,B,C,D\},D)$ has two possibilities:

$C_{min}(\{A,C,B\},B) + dist(B,D) = 13+5 = 18$
$C_{min}(\{A,B,C\},C) + dist(C,D) = 15+7 = 22$

For the size 4 sets, the result is:

$C_{min}(\{A,C,D,B\},B) = 16$
$C_{min}(\{A,B,D,C\},C) = 18$
$C_{min}(\{A,B,C,D\},D) = 18$

Finally, we can test the final 3 possibilities including the path from i back to A:
$C_{min}(\{A, C, D, B\}, B) + dist(B, A) = 16 + 6 = 22$
$C_{min}(\{A, B, D, C\}, C) + dist(C, A) = 18 + 4 = 22$
$C_{min}(\{A, B, C, D\}, D) + dist(D, A) = 18 + 7 = 25$

As before, we found that the minimum is 22. This approach does not seem better than the exhaustive one, but this is just because the example is too small. Using larger N, one can verify that the growth of the exponential function is much slower than the factorial function.

Implementation of this algorithm usually uses memoisation. [4] This is because some of the calls may be repeated (e.g., notice that we could have used the same $C_{min}(S, i)$ in more than one line in the example above). This costs memory, and in the case of TSP it can cost significant amounts of memory. Indeed, using an auxiliary matrix to store partial paths, the solution needs a lot of memory space: $O(N2^N)$. Despite the fact that the time complexity is smaller than the brute force approach, this algorithm can only solve relatively small problems. For example, for a 30 cities problem the amount of memory needed would be approximately 4GB. With only 35 nodes this amount would grow to 128GB, above the capacity of ordinary computers.

It is possible to minimise the amount of memory by replacing the auxiliary matrix by an alternative data structure that only stores calls that were ever used. However, this does not change the memory complexity bound of $O(N2^N)$.

[4] Memoisation (not memorisation) is a term coined by Donald Michie in the 1960s. Michie worked with Alan Turing during the war. Memoisation is the strategy of storing the results of function calls for a specific parameter. For example, in the dynamic programming algorithm for TSP a certain recursion could be called more than once. Instead of recomputing the result for that set of parameters the algorithm simply searches for the answer in memory.

13.2.4 - Christofides Algorithm

This is an interesting approach that was devised in 1976 by Nicos Christofides. The approximation is proved to be within 1.5 times of the optimal solution, and it is still the best one known (that can be proved within a certain margin). However, it only works for the special cases where the graph obeys the triangle inequality. This means that the algorithm works for the case of Euclidian graphs (symmetric, where the triangle inequality applies).

Algorithm 20 Christofides Algorithm

```
1: function TSP(graph G)
2:     Create a minimum spanning tree S of G ▷ See Kruskal algorithm 10.3.1
3:     Get a set of vertices with odd degree
4:     Find a minimum perfect match M from the vertices in line 3
5:     Combine the edges of M and S into a graph H
6:     Find an Eulerian circuit in H
7:     Shortcut                    ▷ Eliminate repeated nodes in the sequence
8: end function
```

13.3 - Problem Classification

Real computers are deterministic, i.e., one can predict what the next step is knowing the state of the registers, memory etc. The next CPU cycle is predictable.

An imaginary non-deterministic computer would have a list of possible actions for the next cycle, but it is unpredictable what the action is going to be. That can be simulated by randomising parts of algorithms, however even random numbers in digital computers are pseudo-random. The concept of non-deterministic computers are used to prove certain theorems in computer science.

Any problem in computer science can be classified as: **P** (polynomial time algorithms), **NP** (non-deterministic polynomial), **NP-complete** (there is also **NP-hard**, which is the class of problem at least as hard as the hardest of the NP-complete problems).

P problems have known polynomial time algorithms on a real computer. Many of the problems that we have studied fall into this class (e.g., sorting, searching, insertion at trees and linked-lists etc). For certain problems we even have proofs that the complexity cannot be lowered. Many of the problems however have functional algorithms, but we might not know a better algorithm with an even lower complexity.

NP problems is the class of problems that could be solved by a (non existent!) non-deterministic computer in polynomial time.

NP-complete problems is the class of problems in which we don't know a method to locate the solution in polynomial time. We may be able to verify solutions in polynomial time. Classes of problems can be defined using the concept of reduction. If a problem can be solved by a different algorithm, requiring some transformation of the data, we say that the initial problem reduces to the latter. As long as the reduction is within polynomial time, then the original problem can be solved within the complexity of algorithm to which it was reduced. Based on the concept of reduction, one can say that the hardest problems within a set are complete. So the **NP-Complete** class contains the hardest problems in **NP**, and are likely not to be in **P**.

Figure 13.3 shows the relationship between NP, P and NP complete classes.

In summary, **NP** problems can be verifiable in polynomial time. **P** problems can find a solution for the verifiable problems in polynomial time in a real computer. **NP-complete** problems have no known method to find a solution in polynomial time in a real-computer.

For example, factorising a large number with two prime factors of same N number of digits is a problem for which we do not have a polynomial time algorithm. After we have the solution, we can check by multiplying the numbers, in $O(N^2)$ or even $O(Nlog(3))$ using a multiplication algorithm devised by Karatsuba in the 1960s. It would take 2^{N-1} multiplications to find a solution by brute force, or $O(2^N)$. This is hard enough so we use the infeasibility of factorising large

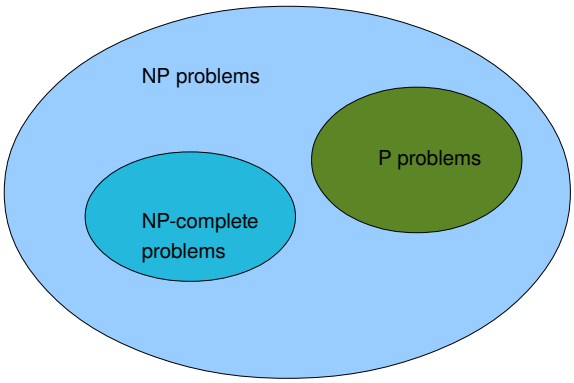


Figure 13.3: Problem classification.

integers for cryptography purposes (the RSA algorithm is based on that).

Remember that "hard" problems might be cracked by a real computer, it just means that large hard problems would take a very long time. The concept of hardness tells us about the limits for solving certain classes of problems.

Nobody knows if we are ever going to find polynomial time algorithms for some classical problems. The analysis of these different classes raised a subtle question: is **P = NP**? In other words, if a problem can be verified in polynomial time, is there an algorithm to find the solution in polynomial time? The question became so important that it is one of the Millennium prize problems, with a million dollars as the prize. So far no one has been successful in claiming the prize.

13.4 - Exercises

1. Use the graph below to solve the Travelling Salesperson Problem, starting at A.

   - a) Use the exhaustive search approach to find the minimum path.
   - b) Use the Branch & Bound method. Choose a fixed bond (threshold) for all steps. You may need to retry a few times.
   - c) Use a Greedy Algorithm which always takes the shortest available path.
   - d) Compare the three results and discuss the implications on optimisation and performance when choosing different paradigms to solve the TSP for a small graph.

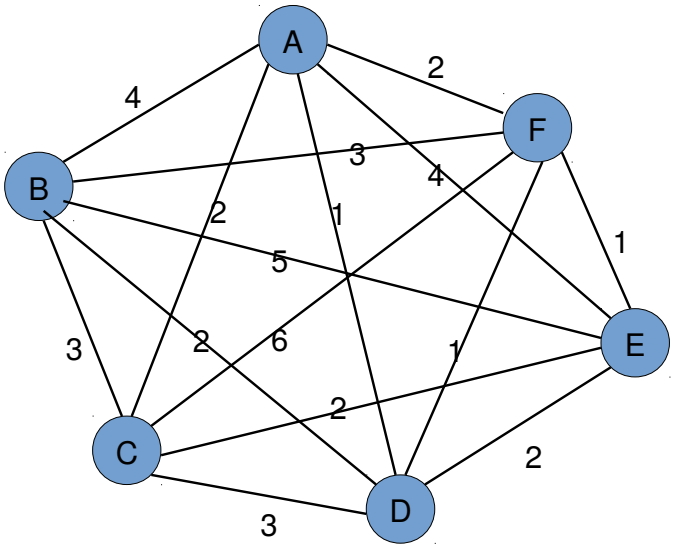


Figure 13.4: Graph for the TSP exercise.

Appendix A C++ I/O

A.1 - Screen printing and getting user input

C and C++ rely on libraries to do anything that is beyond the basic syntax. The I/O library that is the default for any C++ program is the `iostream`. You need to include that library in any program where you want to do simple I/O.

A.1.1 - Output

Using `cout` it is easy to do printouts. As an example, listing A.1 shows the comparison between `cout` and `printf`.

Listing A.1: Printing with cout.

```
1  #include <stdio.h>
2  #include <iostream>
3  using namespace std;
4
5  int main(){
6  int A=10;
7  char B = 'a';
8  char C[10]="testing";
9  float D = 1.56789;
10
11 printf("Printf: %d %c %s %f\n",A,B,C,D);
12
13 cout << "Cout: " << A << " " << B << " " << C << " " << D << endl;
14 }
```

The advantage of the C++ `cout` is clear in the simple syntax. It takes time to get used to it, but it is simpler. Later we will learn that the operator « can be overloaded, allowing us to customise the printout for our own classes.[1]

[1] This is achieved through overloading, a capability that allows programmers to change the behaviour of certain operators.

A.1.2 - User input

Input in C++ is done via `cin`. This is equivalent to `scanf()` function used in C. Both scanf and cin halt the program and wait until the user types in something and press enter.

Listing A.2: Getting an input with cin.

```
1  #include <stdio.h>
2  #include <iostream>
3
4  using namespace std;
5
6  int main(){
7      int a;
8      float b;
9      char c[100];
10
11     printf("Type an integer, a float and a string \n");
12     scanf("%d %f %s",&a, &b, c);
13     printf("You typed: %d %f %s \n",a,b,c);
14
15     cout << "Type an integer, a float and a string" << endl;
16     cin >> a >> b >> c;
17     cout << "You typed: " << a << " " << b << " " << c << endl;
18 }
```

There are a few things to note in listing A.2. In `scanf`, you have to know when to include `&` before the variable (you need to get its address). In `cin`, you just type the variable name and the compiler makes the necessary arrangements.

Unfortunately strange things can happen with a simple input program like that. If the user types in the wrong format, the answer is completely wrong (you should try that by yourself). One needs to check that every variable receive the appropriate input. Ideally, one variable per input makes it safer. Inputs like that are rarely used in practice though, as graphical GUI libraries are available and are much more attractive for the users.

A.2 - C strings and C++ strings

This is usually a source of confusion to the programmer who knows C and is learning C++. C strings are simply arrays that contain type char. Functions that were written for it rely on the fact that a terminator symbol shows where the string ends. This symbol is `\0`.

For example, a string where we want to store ABCDE needs at least 6 chars rather than 5 (one extra for `\0`).

C++ strings are different as they are a proper class. There are advantages in using them because they are safer, and generally easier to use.

In listing A.3 you can see how a C string was declared (line 7). It is just an array of chars. It is unsafe because if we wanted more characters on the string we would have to now its size and make sure that we are not copying more than it can fit.

The C++ string class has many constructors, including one where

you can copy a constant `char` pointer. So in line 20 of listing A.3 we create an instance of a string containing "AB". And in line 24 we create another one. Note that for printing out each character of a C++ string we need to limit the loop to `Cppstring_B.size()` rather than using `sizeof()`. One of the advantages of using C++ strings is the fact that operators can be overloaded to fulfil the role of functions such as concatenate. In line 25 we can use + to concatenate two strings.

Listing A.3: Strings in C and C++.

```
1  #include <stdio.h>
2  #include <iostream>
3  using namespace std;
4
5  int main(){
6      //C string
7      char Cstring[10]="ABCDEFGHI";
8      printf("sizeof(Cstring) is %ld \n",sizeof(Cstring));
9      for(int i=0;i<sizeof(Cstring);i++){
10         printf("char is %c \n",Cstring[i]);
11     }
12     for(int i=0;i<sizeof(Cstring);i++){
13         printf("value of the char is %d \n",Cstring[i]);
14     }
15     //change the fifth symbol to \0
16     Cstring[4]='\0';
17     printf("now the string is printed as %s \n",Cstring);
18
19     //C++ string
20     string Cppstring_A="AB";
21     cout << sizeof(Cppstring_A) << endl;
22     cout << Cppstring_A.size() << endl;
23     cout << Cppstring_A << endl;
24     string Cppstring_B("CDEFGHI");
25     cout << Cppstring_A+Cppstring_B << endl;
26
27     //printing one character at a time:
28     for(int i=0;i<Cppstring_B.size();i++){
29         cout << Cppstring_B[i] << " ";
30     }
31     cout << endl;
32 }
```

Sometimes programmers need to convert from C strings to C++ strings and vice-versa. Listing A.4 shows one of the many possibilities for this type of conversions.

We can copy the content of a C string to a C++ string by using one of the many constructors (see line 10). To copy the content of C++ strings to a C string we can use the `c_str()` method (line 16). This method returns the equivalent of a C string, so that we can use the

function `strcpy()` directly. The programmer has to be careful about the sizes of both strings to avoid errors.

Listing A.4: Copying strings in C and C++.

```
#include <string.h>
#include <stdio.h>
#include <iostream>

using namespace std;

int main(){
    //converting a C string into a C++ string class instance
    char Cstring_B[15]="TEST";
    string Cppstring_C(Cstring_B);
    cout << "C string: " << Cstring_B << "     Cpp string: " <<
        Cppstring_C << endl;

    //converting a C++ string into an array of chars (C string)
    string Cppstring_D="QWERTY";
    char Cstring_C[15];
    strcpy(Cstring_C,Cppstring_D.c_str());
    cout << "C string: " << Cstring_C << "     Cpp string: " <<
        Cppstring_D << endl;
}
```

A.3 - File processing in C++

If you studied C before, you know how to read from files and how to save to files. The next listings (A.5 and A.6) shows a simple comparison to read and to save files in C and C++.

The C version uses `FILE`, `fopen()` and `getline()` to copy each line onto a string. Each line is copied to the string `line` and then printed.

For the C++ version, it uses `ifstream`, which is a class for the file I/O. The method `open()` opens the file for a certain mode (read only in this case). Finally, the `getline()` function (different than the C function with the same name above) copies each line to a C++ string called line.

In fact there is also a method for `cin` called `getline()`. This method can be used to get keyboard input to a C string. There are historical reasons for that, although one has to agree that it is weird that a C++ method stores the line in a C style string.

There are other ways in which a file can be read. For example, if we already know in which format the file is written (e.g., so many integers per line), then we could use `fscanf()` or `istringstream` to parse each line.

Listing A.5: Reading a txt file in C.

```c
#include <stdio.h>
#include <stdlib.h>

int main(){
    char * line = NULL;
    size_t len = 0;
    char file_name[200]="somefile.txt";
    FILE *input = NULL;
    input = fopen(file_name, "r");
    if(input == NULL){
        printf("Cannot open file %s. Exiting.\n", file_name);
        exit(0);
    }
    int linenumber=0;
    while( getline(&line,&len,input) != -1){
        printf("Line %d: %s \n",linenumber,line);
        linenumber++;
    }
    fclose(input);
}
```

Listing A.6: Reading a txt file in C++.

```cpp
#include <iostream>
#include <fstream>
using namespace std;

int main(){
    ifstream input;
    string file_name="somefile.txt";
    input.open(file_name.c_str(),fstream::in);
    if(!input.good()){
        cout << "Cannot open file " << file_name << endl;
        exit(0);
    }
    int c;
    string line;
    int linenumber=0;
    getline(input,line);//read the first line
    while (input.good()){
        cout << "Line " << linenumber << " : " << line << endl;
        linenumber++;
        getline(input,line);
    }
    input.close();
}
```

Bibliography

2019. URL http://www.cplusplus.com/.

David L. Applegate, Robert E. Bixby, Vasek Chvatal, and William J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, Princeton, NJ, USA, 2011.

Sanjoy Dasgupta, Christos Papadimitriou, and Umesh Vazirani. *Algorithms*. McGraw-Hill Higher Education, Boston, MA, 2008.

Adam Drozdek. *Data Structures and Algorithms in C++*. Cengage Learning, Boston, MA, 2013.

Index