Chapter 4 Solving Problems Efficiently

Up to this point, we have learned how to **represent**, **store**, **organize**, and **search** through a collection of possibly complex data items. We know what an **Abstract Data Type** is and why they are useful for coming up with solutions to problems in a way that makes the solution independent from a specific implementation. We studied **lists**, and their implementation as **linked lists**, and we used these for building a simple database able to answer a few queries on items in our collection.

We will no doubt use lists in the future for a variety of applications. So before we close our discussion on lists, it's worth taking time to think about **how efficient** they are as a solution to problems that require frequent look-up of items in the collection (whether it is to look at their content, update the data stored there, or carry out aggregate computations).

Suppose we wanted to use linked lists to implement a fully functional database engine. The database should be able to store information for a large number of data items (think big - tens of millions, hundreds of millions, or even billions of nodes in the linked list). The question is, **what can we expect in terms of the time it takes for us to perform a search on the database?**

Note

Having such a large number of data items in a collection is very common.

- IBM's Watson used millions of documents to answer game questions for Jeopardy
- Google Image Search was indexing well over 10 billion images by 2010
- Facebook had about 1.5 billion daily active users in September 2018
- Amazon sells over 3 billion different products worldwide

So, thinking about what is the most efficient way to maintain a very large collection is of great importance.

To fully understand this question, we have to think a bit about **how data was stored in the list**. A common assumption, and one that is reasonable for many database applications, is that the **data was added in no particular order**, or what is the same thing the **entries** in the list **are randomly ordered** with respect to any data fields we may want to use for answering database queries.

We also typically assume that **queries arrive in random order**. This is a reasonable assumption - think about Google searches arriving over a span of 1 minute from everywhere in the world - chances are, there will be no order or pattern to the searches carried out by people from Mexico, Australia, Singapore, and Finland, all of whom happen to be online at that particular time.

Finally, we have to provide some **definition of efficiency** so we can evaluate how well our list is doing. For a database engine, one reasonable measure is **how many data items have to be inspected** before we find the one we are looking for.

Now, recall from the last unit that **to perform a search** in a linked list, **we have to do a list traversal**. This means starting at the head of the list, and traveling along the list's nodes until we find the one that contains the information we are looking for.

Question: How many nodes do we need to look at before we find the one we want?

- If we are **super lucky**, the data we want will be in the head node and thus **we only look at 1 node** to find what we want, but with randomly ordered data, the chance of this happening **is 1/N**, where **N** is the **number of nodes** in our list. For a list with 1,000,000,000 entries, the odds are very large against this ever happening.
- If we are **super unlucky**, the data we want will be at the tail of the list (or not in the list at all), and we have to look at **all N nodes** before we find it or can be sure it's not there.
- Most of the time, **we are neither super lucky, not super un-lucky**. The data is somewhere in the list, sometimes closer to the head, sometimes closer to the tail. Because the data is randomly ordered, the number of nodes we need to examine **averages out to N/2** after we run many, many queries.

This makes sense: Sometimes we find the data closer to the head, sometimes closer to the tail, but these two balance each other out so on average we have to look at about half of the linked list to answer any given query.

This is really not too bad if our list is a few thousand entries long. But once our linked list grows into the millions of items and beyond, the cost of answering a query becomes too large. Simply put, our linked list has to look at too many data items in order to find a specific one. This translates into a longer wait time for a program requesting information from the database. At some point, the list is so long that the wait time becomes impractically long.

4.0.1 Why we need to look through all that data to find something

We may be inclined to think that the root cause of our problem is the structure of our linked list. And indeed, our **linked list** has a major limitation in that **we can not access an element inside the list without doing list traversal**. However, the root cause of our problem is not due to this limitation of the linked list.

Consider for example **an array** that contains the names of all of the hit songs from 2017 and 2018 as shown in Fig. 4.1 (it's a small list so we can show an example here, but you should be thinking this applies to an array of any size).

I'm the One
Despacito
Look What You Made Me Do
Bodak Yellow
Rockstar
Perfect
Havana
God's Plan
Nice for What
This Is America
Psycho
Sad!
I Like It
In My Feelings
Girls Like You
Thank U, Next

Figure 4.1: A **simple array** that contains the names of hit songs from 2017 and 2018.

The question we want to answer is: Is searching for a specific item in an array with randomly ordered entries

more efficient than searching in a linked list with the same (unordered) entries? Or, what amounts to the same thing, we are asking if using an array allows us to find what we want with fewer than **N/2** items examined on average.

Looking at the array in 4.1, it should be fairly clear that **there is no pattern to how the data is ordered**. If we need to find a specific song name, we have to start at the top and look through the array until we find what we want. This is called **linear search**. Just like a linked list, there is a chance we get super lucky and our query is right at the first entry in the array, we can be super un-lucky so the query item is at the end, or not in the array; and on average we have to look through half the array before we find our information. If the array has **N** entries, we're back to looking through **N/2 entries on average**.

This means that **from the point of view of the number of items we need to look at** to answer a query, **a linked list and an array are equally efficient**. While there may be a slight performance difference in terms of actual run-time (because there is more overhead to a linked list than an array), **any small difference becomes irrelevant as the collection size grows**: both of these containers for storing information become inadequate for quickly answering queries.

4.0.2 Organizing our data for efficient search

If we are to find a faster way to answer queries, **we need to deal with the random order of our data**. We need to **sort it**. Once our data is in sorted order, we can do a much more efficient job of searching for specific items. Fig. 4.2 shows a sorted version of the array from Fig. 4.1.

Bodak Yellow
Despacito
Girls Like You
God's Plan
Havana
I Like It
I'm the One
In My Feelings
Look What You Made Me Do
Nice for What
Perfect
Psycho
Rockstar
Sad!
Thank U, Next
This Is America

Figure 4.2: A **sorted array** that contains the names of hit songs from 2017 and 2018.

For a person, **looking up a song in the sorted array above is much easier**. We understand the entries are in order, so we can easily find the spot where a particular song should be. This is why all content indexes in the backs of books, library shelves, or the phone directory (back in the days when it was actually a printed book) are sorted.

In programming terms, **sorting data has the immediate benefit of making that data much easier to search**, with the result that on a sorted array of pretty much any size, we only have to examine a few items in order to find what we're looking for.

4.0.3 Binary Search

The process of finding an item in a sorted array is called **binary search**. Suppose we want to find the song **"I'm the One"** in the array above. The binary search process is illustrated in Fig. 4.3.

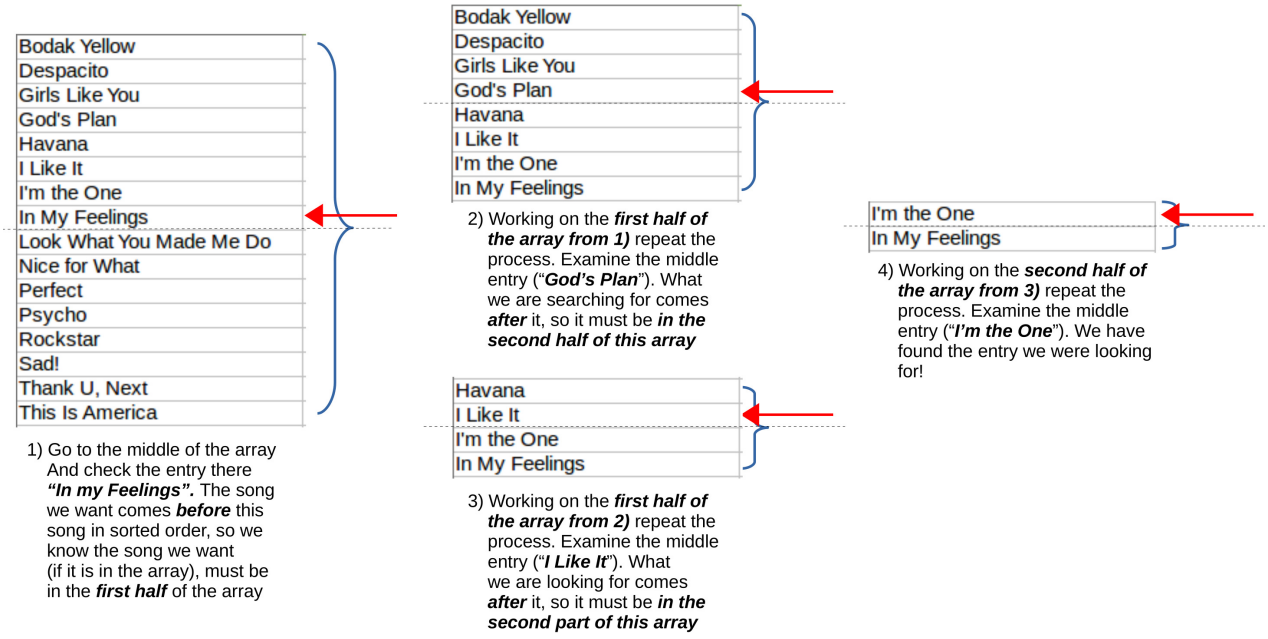

Figure 4.3: The **binary search** process illustrated on a small sorted array.

Binary Search Algorithm

- 1) Go to the entry at the **middle of the current array** - this is the entry at index $\lfloor N/2 \rfloor$ (the **floor of N/2**).
- 2) **Compare** the middle entry with the **query term**, if they match **return the index of the matching item**.
- 3) If the **current array** has only one entry, the **query term** is **not in the array**, return **-1** to indicate the term was not found.
- 4) If the **query term** is **less than** the **middle entry in the current array**, then it has to be **in the first half of the array**, so **the first half becomes the current array, go back to 1)**. Otherwise, the **query term** must be in the **second half of the array**, so **the second half becomes the current array, go back to 1)**.

Binary search is a straightforward process, and it is **incredibly efficient**. This is not evident in the example from Fig. 4.3 because the array shown there is tiny. Let's think about what happens with larger arrays.

Key ideas:

- The **binary search** process **uses the fact that the array is sorted** to **predict** in which half of the array the data we want has to be.
- This means that **for every item we check, we can discard from the search half of the remaining entries**. When we choose which half of the array to search next, we can be sure we will never have to look at any items in the other half.

- Thus, every round of binary search that we complete, the number of items that remain to be checked is cut in half.

This very quickly reduces large collections to very manageable sizes. For example, if our initial array has a length of **1024** entries

- After the first round of binary search, we are left with **512** entries to check
- After the second round of binary search, we have **256** entries left to check
- After round 3, we have **128** entries left to check
- After round 4, we have **64** entries left
- After round 5, we have **32** entries left
- After round 6, we have **16** entries left
- After round 7, we have **8** entries left
- After round 8, we have **4** entries left
- After round 9, we have **2** entries left
- After round 10, we have only **1** entry left to check

This is quite remarkable: In an array with over one thousand entries, we can find any item we want with **10 or fewer** checks. Compare that with the **un-sorted array** or the **linked-list**, in which case we would expect to have to check on average **512 entries**.

So **binary search** is **a lot more efficient** than **linear search** for finding information in large collections, but **just how efficient is binary search?**

This is equivalent to asking: Given an initial value for **N**, the **number of entries in the array**, how many steps are needed to reach the point where only one element is left? (this tells us the maximum number of entries we need to examine to find what we want).

The answer turns out to be $k = log_2(N)$.

For the array with **1024** entries, $k = log_2(N) = 10$, which is exactly the number of steps we had to do above to get to a single item. To get a sense of just how big a difference this makes, Fig. 4.4 shows a comparison of the number of checks that are performed by **binary search** compared with **linear search** as the size of the collection grows, and showing three cases:

- a) The maximum **(worst case)** number of items we need to look at **for binary search**
- b) The **average number** of items we need to look at **for linear search**
- c) The maximum **(worst case)** number of items we need to look at **for linear search**

From Fig. 4.4 it should be very clear to you that **binary search is incredibly efficient**. Even in the worst possible case, with only **25 checks** we can find any one item in a collection with **over 33 million entries**! Conversely, **linear search** on an un-sorted array or linked list would be expected to have to check **over 16 million items**, on average, and all **33 million** if we are unlucky.

N	*Binary Search: $log_2(N)$*	*Linear Search (avg): N/2*	*Linear Search (worst): N*
2	*1*	*1*	*2*
4	*2*	*2*	*4*
8	*3*	*4*	*8*
16	*4*	*8*	*16*
.			
.			
1,024	*10*	*512*	*1,024*
2,048	*11*	*1,024*	*2,048*
4,096	*12*	*2,048*	*4,096*
.			
.			
1,048,576	*20*	*524,288*	*1,048,576*
.			
.			
33,554,432	*25*	*16,777,216*	*33,554,432*

Figure 4.4: Comparison of the amount of work done by **binary search** (worst case) and **linear search** (on average, and worst case) as the size of the collection grows.

4.1 Computational Complexity

The ideas above can be refined into a **concrete, general framework** for comparing different algorithms in terms of the **computational cost** of carrying out a task.

One key idea in understanding how we can compare different algorithms is that we want to find **a measure of the amount of work a particular algorithm has to do as a function of N**, the number of data items the algorithm is working on. We call this measure the **computational complexity of the algorithm**.

In the examples above, we saw that linear search has to examine **N** items in the worst case, while binary search only has to look at $log_2(N)$. The reason for wanting to look at a function of **N** is that **it allows us to predict how an algorithm will perform for increasingly larger data collections**. In our search example, we have two different functions:

$$f_{BinSearch}(N) = log_2(N)$$
$$f_{LinSearch}(N) = N$$

So we say that **binary search has a complexity of** $log_2(N)$ and that **linear search has a complexity of N**. Because the value of $log_2(N)$ grows much more slowly than **N**, we can conclude (without having to test on every possible array) that **binary search is much more efficient than linear search for large collections**. We can visualize this by plotting both functions for increasing values of **N** and comparing the **predicted** amount of work they will have to do, this is shown in Fig. 4.5.

It should be obvious from the plots in Fig. 4.5 that one of these algorithms is a whole lot more efficient than the other at finding information in a large collection. The key observation we can make from this is:

Given two algorithm and the functions of **N** that describe their computational complexity, the **function that grows more slowly will always win-out as the number of data items increases**. So the algorithm with the

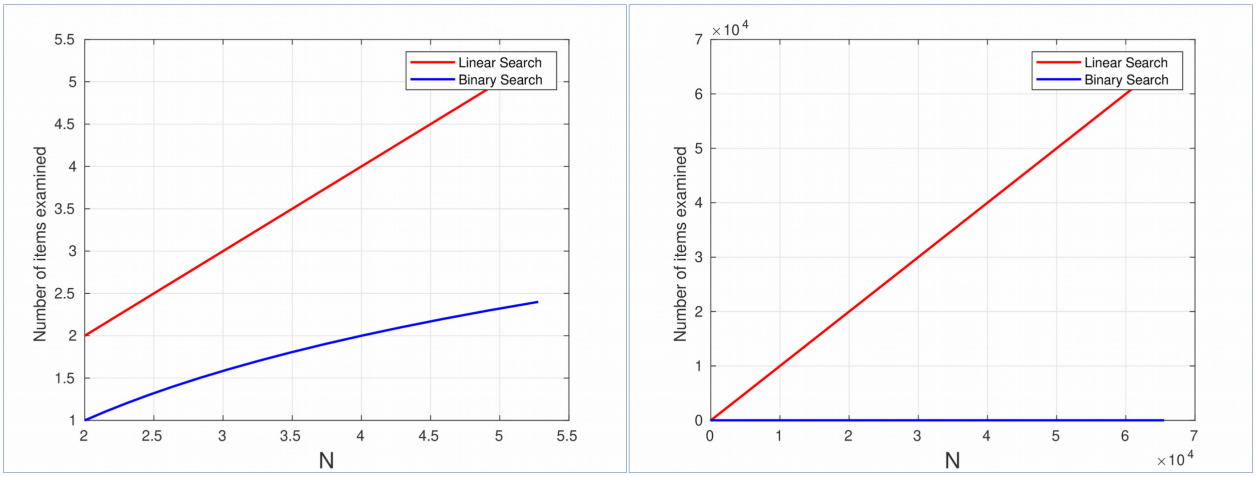

Figure 4.5: Visualization of the **complexity of two different search algorithms**: **Linear search** (red) and **binary search** (blue). Binary search is incredibly efficient, to the point of looking like it does no work at all compared to linear search on large arrays. The shape of each function's curve is easier to compare on a graph for smaller values of N (left side graph)

slower-growing function is said to have a **lower complexity**, and is therefore shown to be **more efficient** for large data collections.

Question: What can we conclude at this point about the run-time of two programs that do search on the same data, but where one uses binary search, and the other uses linear search?

Being able to compare different possible ways of carrying out some task is essential for implementing good solutions to any problem. In computer science, **we compare algorithms by studying and measuring their computational complexity**. Computational complexity is expressed as a function of **N**, the number of data items the program has to work on.

4.1.1 The Big O Notation

In order to be able to better understand and compare the complexity of different algorithms, we use a special notation called the **Big O** notation. The importance of the **Big O** notation is that it allows us to reason about the efficiency of algorithms in terms of general classes of functions. Here's what we mean by that:

Suppose that we have **different implementations of linear search** on arrays (perhaps written by different developers, in different programming languages), and we also have **different implementations of binary search**. We then set out to **carefully measure their actual run-time** for arrays of different sizes, on a particular computer, and we find that they behave as follows:

- a) $.75 \cdot N$ (e.g. for N=10, this implementation runs in 7.5 seconds)
- b) $1.25 \cdot N + .25$ (this one has a .25 second start-up time that is not dependent on **N**)
- c) $2.43 \cdot N$
- d) $1.15 \cdot log_2(N) + 1.12$ (this one also has a start-up time, in this case 1.12 seconds independent of **N**)

- e) $1.75 \cdot log_2(N)$
- c) $15.245 \cdot log_2(N) + 15.25$ (large start-up time! maybe it needs to load libraries before it runs)

The results above require some thought - we know how much work linear search has to do to find an item in the array, and that in the worst case that would be looking at all **N** entries in the array. But **we hadn't considered what could happen when variations between implementations are introduced**.

Perhaps an implementation in **C** will be slightly faster than one in **Java**, and perhaps the one in **Java** will be somewhat faster than one written in **Matlab**. But the important thing to note is that **all of them are linear functions of N**. It is a **property of the linear search algorithm**. The only thing that can change among (correct) implementations is the **constant that multiplies N**, and maybe the presence of a **constant term independent of N**.

We can say exactly the same in a different way: Any correct implementation of the linear search algorithm will have a complexity that is characterized by $c \cdot N + k$, where **c** and **k** are constants, for large values of **N**.

The same is true for different implementations of binary search. They are all **logarithmic functions of N** multiplied by some constant that is implementation dependent, and perhaps including a constant term independent of N.

We want a way to compare algorithms in an implementation independent way. We need to know which is the better algorithm, the one requiring less work to solve a given task. And if we can analyze algorithms in this way, we can always pick the most efficient one. The **Big O** notation makes explicit the **slowest-growing** function of **N** that characterizes the **complexity of some algorithm**. The definition of **Big O** states that:

Definition 4.1 (Big O Notation)

To state that $f(N) = O(g(N))$, which is to say that **a function** $f(N)$ *has a* **Big O** *complexity of $g(N)$ means that $f(N) \leq c \cdot g(N)$ for a sufficiently large value of N (usually stated as $N > N_0$). It is assumed that $f(n)$, c, and $g(n)$ are positive.*

♣

As an example, $f(N)$ could represent the run-time of **binary search**. We know that a correctly implemented binary search will have to check **at most** $log_2(N)$ entries to find the one we need, so we can state $f(N) = O(log_2(N))$ - this immediately tells us that we expect $f(N) \leq c \cdot log_2(N)$ for sufficiently large **N**. We are categorically stating that **whatever the implementation of binary search** (regardless of programming language, which computer it's running on, or who wrote the program), we can **always find a function** $g(N) = c \cdot log_2(N)$ such that the run-time of the algorithm $f(N)$ is always less than, or equal to $g(N)$.

We say that **binary search has a worst-case complexity of** $O(log_2(N))$. Similarly, we can state that **the worst-case complexity of linear search is** $O(N)$ because whatever the implementation, we can always find a function $g(N) = c \cdot N$ such that the runtime $f(N)$ of linear search is always less than or equal to $g(N)$.

From the above, we can categorically conclude that **binary search** is significantly more efficient than **linear search** for large **N**, and this is regardless of any possible differences in the implementation of the algorithms, or the hardware they are running on. We can visualize why this statement is true by looking at the **run-time plots** for the different implementations of **linear search** and **binary search** listed above. This is shown in Fig. 4.6.

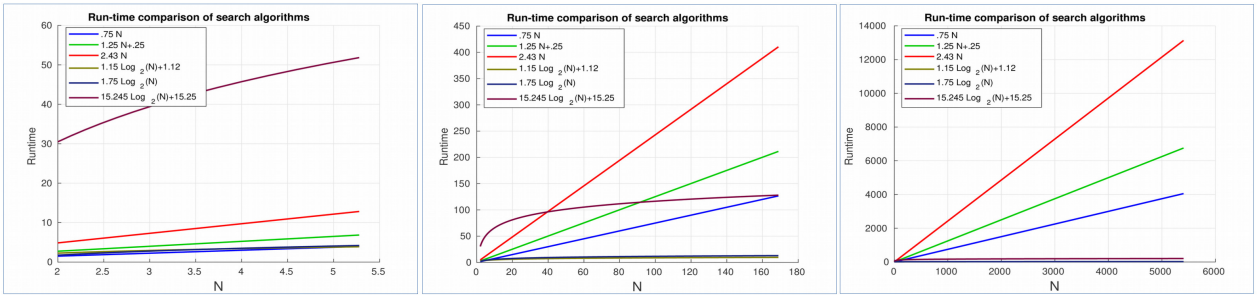

Figure 4.6: Plots of **run-time** for the different versions of **linear search** and **binary search**. Each graph shows what happens for different ranges of values of **N**, increasing from left to right. For small values of **N**, linear search would appear to be more efficient. However, by the time **N=170** all the linear search implementations have overtaken the slowest binary search implementation, and by the time **N=1000** it's **all over for linear search**. The conclusion is: **Binary search will win out in the end, regardless of how it's implemented, if N is large enough.**

The point not to be missed in the plots shown in Fig. 4.6 is that **the algorithm with the smallest Big O complexity will win for large enough N**. The $log_2(N)$ function (or in fact, **any log function, regardless of the base**) is **much slower growing than any linear function**, so we will always expect binary search to be faster on a large enough array. The note about any log function being slower growing than any linear function means we usually **ignore the base of the** log **function**, and we simply state that **binary search is** $O(log(N))$.

Note

Why do we want the slowest-growing function that characterizes the complexity of an algorithm? Consider the graph shown in Fig. 4.7 that plots the runtime of an implementation of **binary search** written in **C** for various values of **N**.

The plot also shows two functions both of which can be used as **worst case upper bounds** on the algorithm's run-time. From the definition of **Big O** complexity, if we take $f(N)$ to be the run-time for the binary search program, both of the following statements are correct:

- $f(N) = O(log_2(N))$
- $f(N) = O(N)$

The Figure clearly shows we can find a **logarithmic** function such that $c_1 \cdot log_2(N) \geq f(N)$ for all values of N in the plot. We can also find a **linear** function such that $c_2 \cdot N \geq f(N)$ for all N in the graph. However, notice that the **linear** function really does not provide a good **prediction** for what the run-time of the program will be for any particular value of N. Indeed, for large N, the **linear** function **over-estimates run-time by a very large amount**, whereas the **prediction from the logarithmic function is much closer** to the real-world behaviour of the program.

Therefore, in order to be able to reason about, or compare the amount of work that different algorithms will have to do while solving a problem, we need to find **the slowest-growing function of N** that provides a bound on how much work the algorithm does as **N** grows.

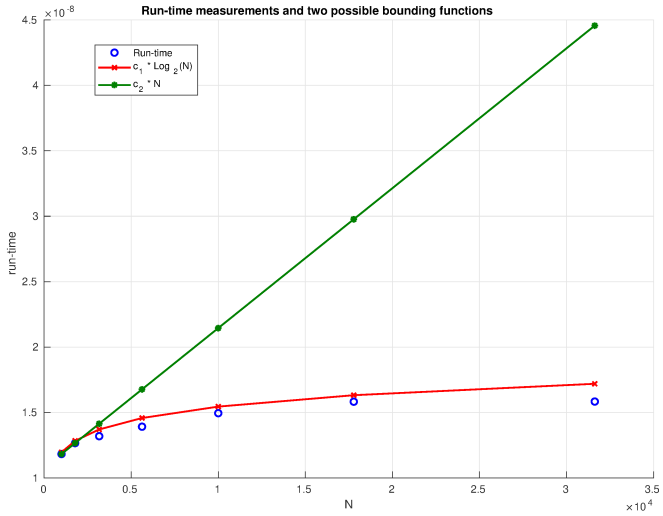

Figure 4.7: Run-times for a **C** implementation of **binary search** (blue markers) for different array sizes (x-axis). The two functions shown can both be used as bounds for the algorithm's run-time. However, the **linear** function is not useful for **predicting** what the run-time should be for a particular value of **N**.

4.1.2 From algorithm complexity to problem complexity

We started with the goal of **understanding how efficient** two different algorithms for finding a particular item in an array can be. Complexity analysis is a wonderful tool for doing this. However, there is an even more powerful idea behind the use of complexity analysis in computer science:

We can **study, quantify, and prove** results regarding the **complexity of solving a given problem**. What is the difference? Let's take an example: **Sorting an array** (and we will come back to this example soon in more detail).

- **Algorithm complexity** means we can look at different algorithms to sort an array, and figure out which one is going to be more efficient as the array size grows.
- **Problem complexity** means we study the actual problem of sorting, and we try to figure out what is the **theoretical lower-limit** on how much work the best possible algorithm has to do to sort an array.

For example, **proving that linear search has a complexity of** $O(N)$ is equivalent to showing that it is **not possible** to find an algorithm that can do linear search with complexity less than $O(N)$.

This is a powerful and important idea because it **allows us to classify problems** by **how hard** (in terms of complexity) they are to solve computationally. Harder problems are characterized by a **Big O** complexity given by fast-growing, or very-fast growing functions of **N**. As a developer, it's important to become familiar with the different classes of problems you may have to work with one day, and to consider carefully what is reasonable to expect of an algorithm, given the complexity of the problem it is working on. A comparison of a handful of important algorithms and problems in terms of their computational complexity is shown in Fig. 4.8.

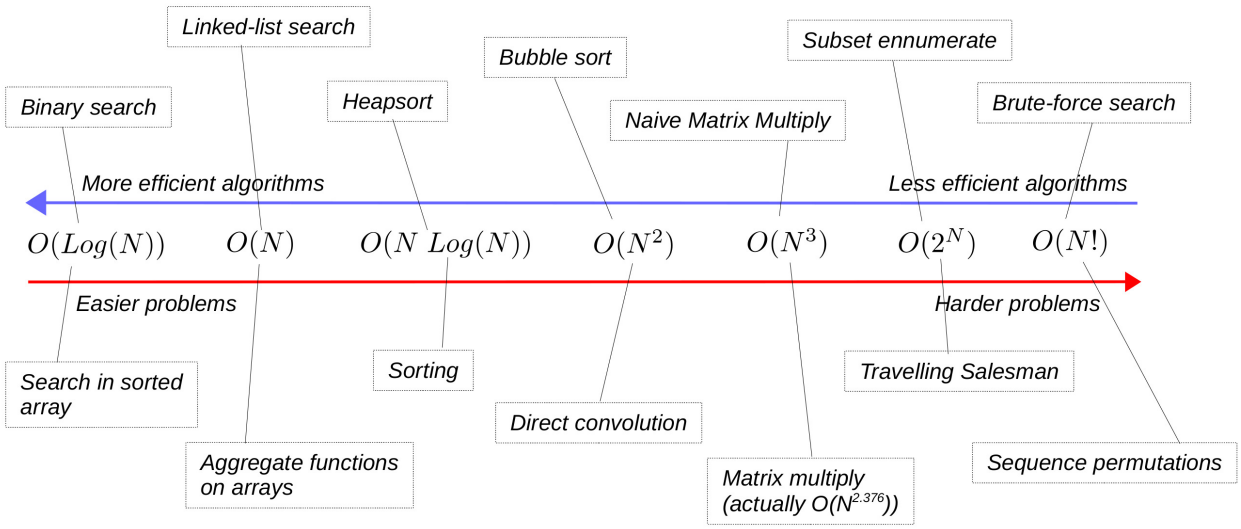

Figure 4.8: The plot above shows examples of **algorithms** (on top) and **problems** (at the bottom) with different **Big O** complexity. Note that for a given problem (e.g. sorting), you can easily find algorithms whose complexity is greater (bubble sort) than the best known complexity for the problem. Knowing what the best known complexity estimate is for a given problem allows you to evaluate how good an algorithm is at solving that problem.

Note

We can measure complexity in terms of different quantities: **the number of times we have to check an item** in a collection, **the run-time** of an algorithm, **the number of mathematical operations** a certain function has to perform. Which measure to use **depends on what problem we are solving**, but the analysis, and what it tells us about an algorithm or problem is stated in the same way using the **Big O** notation.

Computational complexity analysis applies to **every type of problem**, not only searching for information in a collection. For example, it is incredibly important in all areas of computer science that deal with numerical computation, for example machine learning and artificial intelligence. We can learn a lot more about computational complexity, and about how to study, analyze, and prove estimates of a problem's complexity by reading up on the topic. It requires a deeper and more serious study than is appropriate for this book.

4.1.3 So this means the implementation doesn't matter right?

Not quite - solving a problem efficiently requires us to **first think of the complexity of the algorithm** for solving the problem. Once we have selected an algorithm with the best known complexity for solving a particular problem, we have to write a good implementation of it.

Going back to our original example of search algorithms, the difference between the implementations of algorithms that have **the same Big O complexity** becomes important once **N** is large enough, e.g. **N=1000000**:

- $1.15 \cdot log_2(N) + 1.12 = 24.041$
- $15.245 \cdot log_2(N) + 15.25 = 319.11$

So the implementation makes all the difference between us having to wait around **20 sec.** for the result, or

having to wait **over 5 minutes**!

✍ **Exercise 4.1** Consider a function that takes 2 input arrays:

```
void multiply_accumulate(float input[N], float output[N])
```

The function computes each value in the **output** array as $output[i] = \sum_j input[i] \cdot input[j]$

That is, the i^{th} entry in the output array is the result of multiplying the i^{th} entry in the input array with every other entry in the input array (including itself), and adding all of these up.

One way to implement this function is shown below:

```
void multiply_accumulate(float input[N], float output[N])
{
  int i,j;

  for (i=0; i<N; i++)
  {
    output[i]=0;
    for (j=0; j<N; j++)
    {
      output[i]=output[i]+(input[i]*input[j]);
    }
  }
}
```

Question: What is the **Big O** complexity of the function shown above? For this exercise, define the complexity in terms of **how many multiplications are carried out**. If you're having a tough time figuring that out from the code, try and list the operations the loops are doing for a small value of **N** and see if that suggests something to you.

Question: Would the **Big O** complexity change if **instead of multiplications** we defined complexity in terms of the **number of additions** that are carried out by the function?

Question: Is the complexity result we found above the best that we can do for this problem? Is there a better algorithm? And if there is, what is the **Big O** complexity of that algorithm?

4.2 How to make search more efficient

We started this unit trying to understand how efficient linked lists are in terms of letting us search for items in our collection. We have now seen that searching in linked-lists has a **Big O** complexity of $O(N)$, and because of that, for large collections, they are not the optimal way of storing information that will need to be searched frequently.

As a side question: Why is it not possible to perform binary search on a linked list whose items are in sorted order?

We also saw that **binary search** has a complexity of $O(Log(N))$ which makes it incredibly efficient for finding information even for very large collections. So based on this we may conclude that we should give up on linked-lists and that we should simply stick to sorted arrays of items so we can have efficient search.

However, we left arrays behind when we realized that their limitations (fixed capacity, either insufficient for data, or wasteful of space) made them hardly a good solution for the management and storage of large collections of items whose quantity is not known in advance, and in cases where the size of the collection can change a lot over time.

So we have a serious problem now:

At this point it looks like we can have either

1) A **dynamic data structure** that allows us to organize and maintain a large collection of items without wasting space.

or

2) A sorted array with fixed capacity (and the associated known limitations), but where we can perform binary search to efficiently search for information.

What are we missing?

Suppose we decided that efficient search is more important to us than not wasting space, so we are willing to use an array large enough for our collection, so we can do binary search and find items quickly. Sounds like a good plan, except that we haven't accounted for the fact that **we expect items to be added to our collection in random order**. Binary search requires our array to be sorted.

So before we decide that our best bet is to use a sorted array along with binary search, **we have to consider the computational cost** (and you know now that by this we mean the computational complexity in terms of **Big O**) **of sorting** the array in the first place.

4.2.1 The computational cost of sorting

Consider an (unsorted) array with **N** entries. Let's consider the simplest sorting algorithm we can come up with: **bubble sort**. If you haven't seen bubble sort before, here's a simple implementation:

```
void BubbleSort(int array[], int N)
{
   // Traverse an array swapping any entries such that
   // array[i] < array[j]. Keep doing that until the
   // array is sorted (at most, N iterations of the
   // loop on i)

   int t;

   for (int i=0; i<N; i++)
   {
     for (int j=i+1; j<N; j++)
     {
       if (array[i]<array[j])
       {
         t=array[j];
         array[j]=array[i];
```

```
        array[i]=t;
      }
    }
  }
}
```

In **bubble-sort**, the larger elements **float to the top of the array** like bubbles. After the first iteration of the loop on **'i'**, the largest number in the array is in the correct location (the first entry in the array), after the second iteration of the loop on **'i'** the next largest number is at the correct location (second entry in the array), and so on.

The question is: what is the worst-case **Big O** complexity of **bubble sort**? In this case, let's measure work in terms of **how many comparisons** of pairs of entries the algorithm has to perform - which is a common measure of work for **sorting methods**. Let's take a look at how many swaps would be needed to sort the array:

- For the first iteration of the loop on **'i'**, the loop on **'j'** would perform **N-1** comparisons
- For the second iteration of the loop on **'i'**, the loop on **'j'** would perform **N-2** comparisons
- For the third iteration of the loop on **'i'**, the loop on **'j'** would perform **N-3** comparisons
- ... and by now you can see the pattern arising ...
- For the $(N-1)^{th}$ iteration of the loop on **'i'**, the loop on **'j'** would perform **1** comparison
- And finally, for the N^{th} iteration of the loop on **'i'**, the loop on **'j'** would perform **0** comparisons

The total amount of work performed by **bubble sort** is the sum of the comparisons performed for the N iterations of the loop on **'i'**. We can figure out what that is by noting that matching pairs of iterations combine to perform a constant number of comparisons. The **first** and **last** iteration combined carry out $N-1$ comparisons ($N-1$ from the first iteration, 0 from the last one). The **second** iteration and the **next-to-last** iteration combined also carry out $N-1$ comparisons ($N-2$ and 1, respectively). The **third** iteration and the **second from last** combine to carry out $N-1$ comparisons as well, and so on. In total, there are $N/2$ pairs of iterations each of which combines to perform $N-1$ comparisons. So the total number of comparisons performed by the **bubble sort** algorithm is:

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

Question: The total work done (in the worst case) by **bubble sort** has two different terms both of which involve N, so what is the worst-case **Big O** complexity of **bubble sort**?

To answer that question, all we have to do is consider what happens to each of these terms as N grows. The quadratic term $N^2/2$ grows **a whole lot faster** than the linear term $N/2$. For instance, for $N = 1000$, we have that $N^2/2 = 500,000$ and $N/2 = 500$ so the linear term is 1000 **times smaller**. As N **grows to** 10000 we have that $N^2/2 = 50,000,000$ and $N/2 = 5000$ which is $10,000$ **times smaller**.

From this you can glean an important principle: Whenever you have a complexity result that involves **different terms involving N**, the **fastest growing term** will **dominate the complexity of the algorithm as** N **grows**.

In the case of **bubble sort**, that means we need only consider the **quadratic term** $N^2/2$. And we say that worst-case **Big O** complexity of **bubble-sort** is $O(N^2)$.

Note

Don't forget that this is appropriate for comparing and choosing among algorithms that have different **Big O** complexity. If what we are doing is **trying to choose between algorithms that have the same Big O complexity** (e.g. the different versions of binary search we discussed earlier on) then we have to include **everything**. That means, any terms that depend on N as well as the constants that may appear in our estimate of how much work the algorithms are doing.

Houston, we have a problem:

- We want to use a **sorted array** and **binary search** so that searching for items in our large collection has a complexity of $O(log(N))$
- But in order to do that, we have to first sort the array, which (if done with bubble sort) has a complexity of $O(N^2)$

that is not a nice result. The **quadratic cost of sorting** will **dominate** the complexity of the process, and this cost grows **a lot faster** than even the **linear search** complexity which is only $O(N)$. Suddenly, using a **sorted array** doesn't seem like such a good idea.

4.2.2 It gets more interesting

Bubble sort is definitely not the best sorting algorithm out there (President Obama knows this, see Fig. 4.9) - we sometimes use it for small arrays because it's so simple and quick to implement, but we can do much better. If you recall from our discussion above on the **computational complexity of problems**, the best known sorting algorithms have a complexity of $O(N \cdot log(N))$. There are several algorithms that can reach this level of performance, including a couple we will study in detail in the next Chapter. For now let's see how far we can get if we replace our costly bubble sort with the more efficient **merge sort**.

Figure 4.9: "I think the bubble sort would be the wrong way to go". *Photo: U.S. Federal Government – public domain*

How would this change our decision on whether or not **sorting an array** and then using **binary search** to find information efficiently is reasonable? What we have now is the following situation:

- We have to sort the collection, which has a **worst case Big O** complexity of $N \cdot log(N)$ using **merge sort**
- Thereafter we can find any item in the collection in $O(log(N))$ time

Unfortunately, $N \cdot log(N)$ is still bigger than N, so we could be led into thinking that doing **linear search** remains the best option. But it's actually more complicated than that.

Suppose we are working with a collection that doesn't change much - for example, suppose we are maintaining a database of all the streets in a particular city. New streets don't appear every day, and existing streets don't disappear very often either. Under these conditions **we could do the sorting once**, and once we have a sorted collection we can carry out **many, many searches efficiently** using **binary search**.

In this situation, whether we should use **sorting and binary search** or **simple linear search** in an unsorted linked list or array comes down to **how many searches are we going to do?**. The table below illustrates the total cost of **carrying out a sequence of searches** using either **sorting and binary search** or an **unsorted array**.

Example 4.1

```
In the table below, N is the number of entries in the collection, M is the
number of searches we want to perform, 'A' is the cost of
sorting the array (it's done only once), 'B' is the total cost of doing M
searches using binary search (M*log(N)), and 'C' is the cost of
linear search (N*M). The line marked [+] indicates where the cost of
using linear search (C) becomes greater than the cost of sorting and using
binary search (A+B).

N=1000

                  A                              B                                C
M          Sorting Cost (N * log(N))    Searching Cost (M*log(N))    A+B           N*M

1                6907.8                          6.9                 6914.7         1000
10               6907.8                         69.0                 6976.9        10000 [+]
100              6907.8                        690.7                 7598.6       100000
1000             6907.8                       6907.8                138165.6      1000000

N=10000

                  A                              B                                C
M          Sorting Cost (N * log(N))    Searching Cost (M*log(N))    A+B           N*M

1                92103                           9.2                 97112          10000
10               92103                          92.1                 97195         100000 [+]
100              92103                         921.0                 98024        1000000
1000             92103                        9210.3                106313       10000000
```

The table above is interesting because it shows that even though the initial cost of sorting can be significant (as seen, for instance, for $N = 10000$), that cost is the same whether we do 10, or 1000 searches. Because **binary search** is so efficient, it only takes a few searches to make **linear search** more costly than **sorting once and searching many times**. In both of the cases above, it takes only 10 searches for **linear search** to do more work than our solution using **merge sort** and **binary search**.

So, you should keep in mind that **Big O** complexity results have to be used carefully - the actual decision on **what algorithm or combination of algorithms** you should use for working with a particular collection involves

not only the **Big O** complexities for each of the components of the algorithms you're comparing, but also **the type of operations** you are expecting will be carried out on the collection, and **their frequency** (how often they take place).

If our collection had frequent insertions and/or deletions, we would need to **re-sort** the collection (or use **insertion sort**) to keep the array in sorted order, and if this happens often that would quickly make the solution that uses sorting much worse than the **linear search**.

✍ **Exercise 4.2** Build a table like the one shown in Example 4.1, But this time, in addition to the searches, there are **Q** insertions, where **Q** can be **10** or **100** and after each insertion **the array must be re-sorted** (i.e. the cost of **each insertion** is $N \cdot log(N)$).

✍ **Exercise 4.3** Build a table like the one shown in Example 4.1, but instead of **merge sort**, we are using **bubble sort** for sorting the array before doing **binary search**. Fill in the table and find out **how many searches are needed** for **linear search** to become less efficient than the combination of **bubble sort** and **binary search**. Your table should show what happens for **N=1000** and **N=10000**.

So have we found an efficient way to make a large collection searchable?

Not quite...

- The array solution has space limitations (fixed size can be either insufficient, or wasteful of space).
- It requires sorting. With the best sorting algorithm this has complexity $O(NLog(N))$ which is already worse than linear search (it can still be worth it if we sort once and then search many times).
- But, we expect most collections to change over time - insertions, deletions, and modifications will require work to ensure the array remains sorted.
- The conclusion is that a **sorted array with binary search is not necessarily the best solution** for organizing, storing, and searching over a large and changing collection of items.

What do we need?

- It is clear that for efficient search **data has to stay organized in some way**. We need an **ADT** than can facilitate this without requiring a separate sorting step.
- The **ADT** must support efficient **search**, **insertion**, and **deletion**; and these operations must remain efficient (in terms of **Big O** complexity) as the collection grows in size.
- The implementation of this **ADT** as a **data structure** should be **dynamic** and request/use space only as needed by the items in our collection.

We will now learn about an **ADT** and associated data structure that can do just that. We will study its properties, learn how it handles searches, insertions, and deletions, figure out the **Big O** complexity of the typical operations it supports, and look at some of its applications.

4.3 Trees, Binary Trees, and Binary Search Trees

Recall that a linked-list is a data structure in which nodes contain one item from a collection, and one link to a successor node which is the next node in the list.

A **tree is a generalization** of this idea, **it consists of nodes**, each of which contains **one or more data items** from a collection, and **one of more links** to children nodes (the equivalent of the successor node, but now we can have many).

Trees are extremely common structures in computer science, used for a wide range of purposes.

A particularly common type of tree is the **binary tree**, which has the property that **each node has two child nodes**, the left child, and the right child.

The diagram in Fig. 4.10 illustrates a binary tree. Note the following:

- Each node has **two spaces for links** - one for the left child (shown with red arrows), and one for the right child (shown with blue arrows).
- Nodes may have zero, one, or two children.
- The node at the top of the tree is called the **root node**. Similarly to the head of a linked list, we manage a tree by keeping a pointer to the root node.
- Each level in the tree consists of **nodes that are at the same distance or depth** with regard to the **root node**. Each level contains **up to 2 times** the number of nodes of the previous level.
- Leaf nodes are nodes with **no children**.

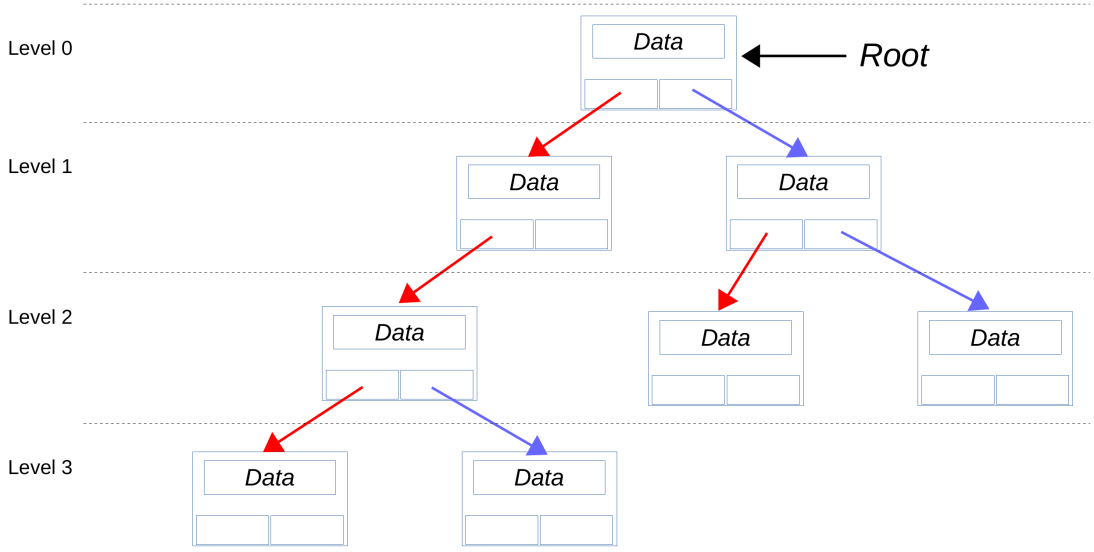

Figure 4.10: Structure of a **binary tree**, each node has up to **two children** and contains data for one item in a collection.

Without any additional refinements, binary trees may be useful for various tasks. However, for efficient management of a data collection, we have to satisfy the requirement that the tree keeps data organized in a way that makes the various operations efficient. With an additional constraint on how the tree is built we can achieve this last requirement.

4.4 Binary Search Trees

A **Binary Search Tree (BST)** is binary tree such that for each node in the tree, the **BST property** holds:

- Data in the **left sub-tree** of a node have value **less than, or equal to** the value of the data in the node
- Data in the **right sub-tree** of a node have value **greater than** the value of the data in the node

This means that at each node, we can quickly determine which of the following three situations is true:

- The data is in the current node
- The data is not in the current node, but if it is in the tree, it must be in the **left sub-tree**
- The data is not in the current node, but if it is in the tree, it must be in the **right sub-tree**

The above should make you think about binary search! The **BST** is intended to allow us to organize a large collection in such a way that we can quickly search through it. Indeed, under the assumption that our data is inserted into the tree in random order, the **BST** can provide an **average case** search complexity of $O(Log(N))$.

4.4.1 Search in a BST

The purpose of a **BST** is to support **efficient search** for data stored in the tree. Let's consider the search process in a **BST** as shown in the pseudocode below:

```
  // This function takes the root of a subtree <subtreeRoot>
  // and a key value we are searching for <queryKey>. If an
  // item in the tree contains a matching key, the function
  // returns the data contained in that node.

searchForKey <-- <subtreeRoot>, <queryKey>

if the <queryKey> is equal to the <key> at <subtreeRoot>
  // Found the key: Return the data stored at this node

  otherwise
      if the <queryKey> is less than the <key> in <subtreeRoot>
          // The node we want must be to the left of <subtreeRoot>
          searchForKey <-- <leftSubtreeRoot>, <queryKey>
      otherwise
          // The node we want must be to the right of <subtreeRoot>
          searchForKey <-- <rightSubtreeRoot>, <queryKey>
```

The search process starts at the **top of the tree**, checking the query value against the value stored at a node. If the value is found the search succeeds and returns the requested data item, otherwise, it **checks whether the query value should be in the left or the right subtree** (which is possible because of the **BST property**), and continues searching on the corresponding subtree.

The search process is illustrated in Fig. 4.11.

Note

With **BSTs**, we normally call the values used to search for information in the tree **search keys**. If the tree contains data items which are simple data types then the keys and the data values are the same. But remember that we intend to use these data structures to organize and maintain large collections of compound data types. So in general, **the key will be a suitable field**, or a **subset of fields** from the **CDTs**. **Keys have to be comparable** (i.e. besides checking for equality, **we must be able to tell which of the two keys is greater and which is lesser** according to some ordering that makes sense for our data). This often involves **writing a comparison function** that works with our data type. Finally, as we discussed in the previous Chapter, **search keys must uniquely identify each item in a collection**.

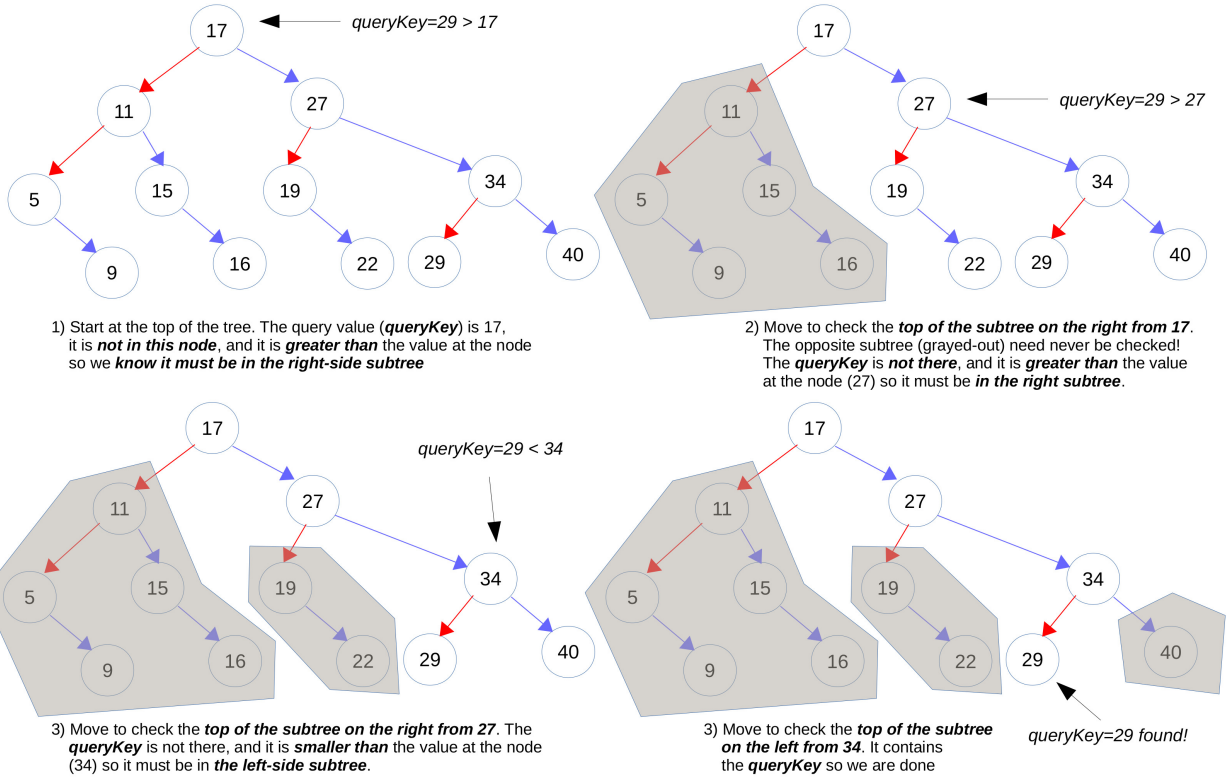

Figure 4.11: Search in a **BST** involves checking nodes starting at the top, at each step deciding whether we have found the data we are looking for (**the queryKey**) or whether we should look for it in the **right subtree** or **the left subtree**. Nodes shaded in gray will never be checked because they belong in a subtree where we know the **queryKey** could never be placed.

You can see how, similarly to binary search, the search process in a **BST** quickly discards from search a large portion of all the values stored in the tree. The question is **how many items need to be examined for us to find the query key (or determine it's not in the tree)?**.

The search moves **down one level in the BST for every comparison** between the **queryKey** and values in the tree. This means that at most, the **search has to examine a number of nodes equal to the height of the BST** - the number of levels in the tree. The **height** of a **BST** is defined as **the length of the longest path from the root of the tree to a leaf node**. This is the number we need to know in order to estimate how muck work is needed to find a

specific item in a **BST**.

So, **how many levels are there in a full BST?** To answer that question we need to figure the **maximum height** and **the minimum height** that a **BST** with **N** items in it could have.

The **height** of the tree depends on where data items are relative to each other. To understand how the tree is shaped, we need to take a look at how the tree is built by **inserting data items** in some order into an initially empty **BST**.

4.4.2 Inserting nodes into a BST

For **linked lists**, we had to keep around a pointer to the **head** of the list. In the case of a **BST**, we will need to keep around a pointer to the **root** of the tree (the node at the very top). The insert process **must ensure that the BST property is enforced** when a new item is inserted in the tree.

The pseudocode for the **BST insert** operation is shown below:

```
// Given the <root> of the BST
// and a new data item whose key is <newKey>
// to be inserted in the tree.

if the BST is empty (root is NULL)  // The first key inserted in the
    <root> := <newKey>              // tree becomes ROOT
otherwise
    BST_insert <-- <root>, <newKey>

// The insert function takes the root of some
// subtree (or the whole tree) and the newKey
// and inserts the new node in the right place
// so the BST property is preserved

BST_insert <-- subtreeRoot, newKey

    if the <newKey> is equal to the <key> at <subtreeRoot>
        then this is a duplicate key: Return without inserting anything

    otherwise

        if the <newKey> is less than the <key> in <subtreeRoot>
            // <newKey> should be on the left-side from <subtreeRoot>
            if the <leftSubtree> of <subtreeRoot> is empty
                    <leftSubtreeRoot> := <newKey>
            else
                BST_insert <-- <leftSubtreeRoot>, <newKey>

        otherwise
            // <newKey> should be on the right-side from <subtreeRoot>
            if the <rightSubtree> of <subtreeRoot> is empty
                    <rightSubtreeRoot> := <newKey>
            else
                BST_insert <-- <rightSubtreeRoot>, <newKey>
```

The **insert** operation works very similarly to the **search** operation. It starts at the top of the tree, working its way downward and choosing either the left or the right subtree at each level depending on whether the **new key** is **less than** or **greater than** the key at each node visited. Once it finds **an empty subtree**, it inserts the new node there. Fig. 4.12 shows several examples of this process.

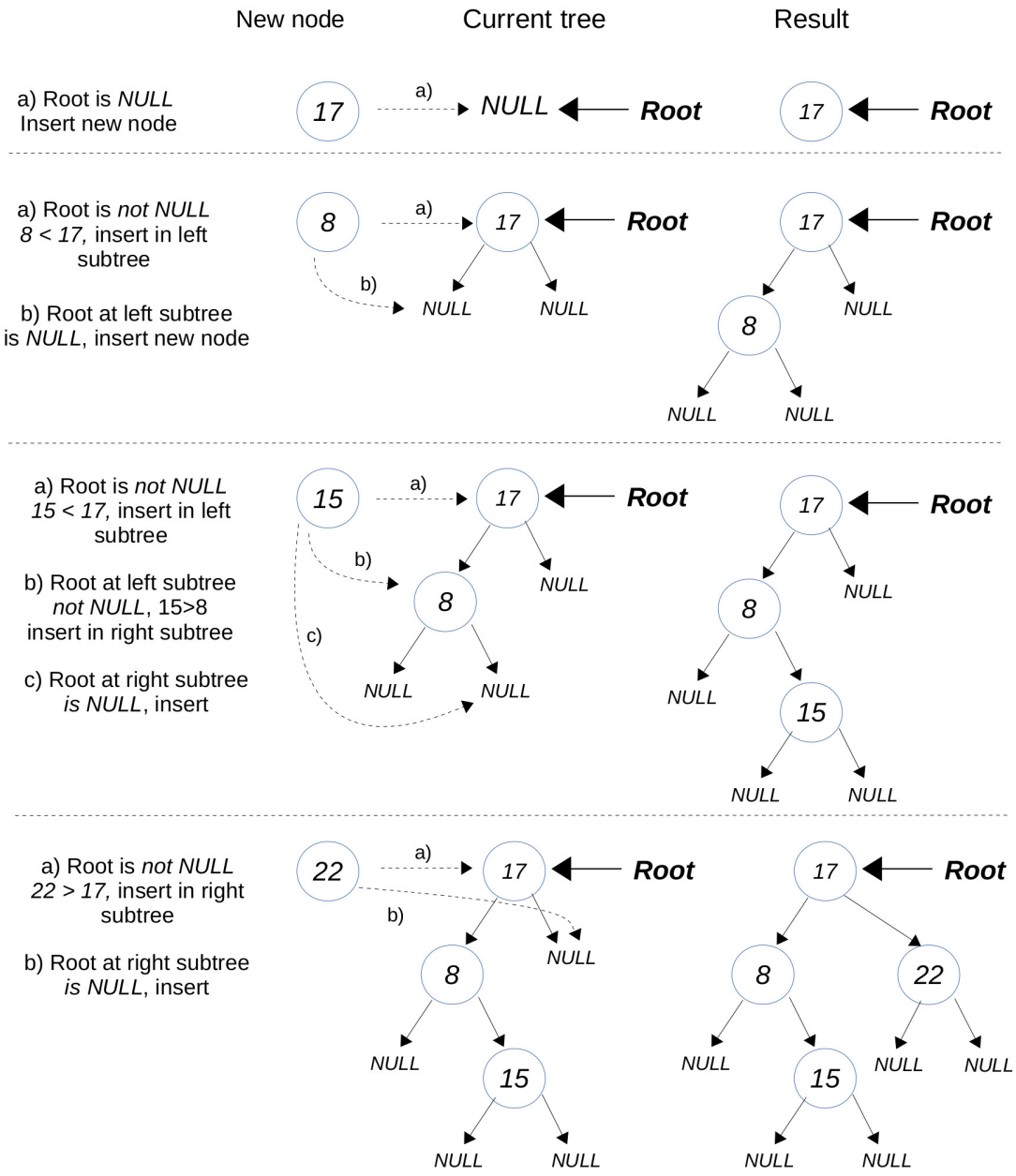

Figure 4.12: Examples of **BST insert** starting with an empty tree. The process works its way down from the top of the tree, choosing the left or right subtree as is appropriate at each level, until an empty subtree is found where the new data item should be stored.

✍ **Exercise 4.4** Starting with the last BST in Fig. 4.12 (after 22 is inserted), list the steps carried out, and draw the resulting tree, after inserting: 19, 4, 1, 6 , and 21 (in that order)

With this in mind, let's revisit our question regarding the possible values for the **height** of a BST that contains **N** items. Because of how the insertion process works, the **shape** of the **BST** (the location of the nodes) will be dependent on **the order** in which keys are inserted. To see this, have a look at the example below, which inserts the numbers **1** to **7** in an **initially empty BST**.

Example 4.2

```
Let's first see what happens when we insert the numbers 1-7 in order into an intially
empty BST.

   1      1      1      1    .. etc .. 1
           \      \      \               \
            2      2      2               2
                    \      \               \
                     3      3               3
                             \               \
                              4               4
                                              .
                                               .
                                                \
                                                 7

Because the numbers are inserted in order, each key we add must be placed to the right
of the one that was inserted just before. The resulting BST is completely one-sided
and in fact looks like a linked list!

The height of a BST that results from inserting N items in order is N-1

Now let's see what happens when we insert the same numbers in the order
4, 2, 6, 1, 3, 5, 7

4      4      4       4       4        4         4
      /      / \     / \     / \      /   \     /   \
     2      2   6   2   6   2   6    2     6   2     6
                   /       / \      / \   /    / \   / \
                  1       1   3    1   3 5    1   3 5   7

In this case, the keys are inserted in such a way that the resulting BST is
balanced - which means that there are no large differences in height between
different subtrees at any place. This latter tree is an example of a
'complete' BST.
```

So the **height** of the **BST** depends on the order in which items are added to the tree. From the example above we can immediately tell that **in the worst case** the **height of the BST** will be **N-1** which means that both **search** and **insert** have a **worst case Big O complexity** of $O(N)$. That should not be surprising because in the worst case, the **BST** looks like a **linked list**.

Conversely, when the tree is as **short as possible**, the height of the tree is given by $\lfloor log_2(N) \rfloor$. This follows from the fact that each successive level in the tree contains **twice as many** nodes as the one before - or what amounts to the same thing: **each additional level in the tree doubles the number of nodes it can contain**. This can only happen when the **BST** is **complete**, which means that **every level of the tree is full with the possible exception of the last one**, and **any nodes in the last level are packed to the left, with no gaps in between**.

Therefore the **the Big O complexity** of **search** and **insert** is $O(log_2(N))$ on a **complete BST**.

At this point let us remember one **important assumption** regarding our collection: Items are added **in random order**. This means we would have to be very very lucky (or very very unlucky) to get either of these special cases of **BSTs**. Most of the time we have a tree whose height is between that of a **complete BST** and that of a misshapen tree that looks like a linked list.

In order to get a useful estimage of what we can expect from a typical **BST**, we need to know what is the **average height of a BST** that results from items being added in random order. The surprising, but very helpful result is that **the average height of a BST** built from a random sequence of insertions is $O(log_2(N))$ - it is not **exactly** $log_2(N)$ but it is within some constant factor of it. Detailed proofs for this result can be found in standard textbooks on algorithms.

This is a good result because it means that **on average**, the **Big O** complexity of **search** and **insert** on a **BST** is $O(log(N))$. And now we're getting somewhere! in the average case, **BST search** is as efficient as **binary search**, and **we do not have to sort the data** because the **BST insert** operation (which is also $O(log(N))$) ensures the **BST property holds** everywhere in the tree at all times.

Note

We can not trick the Universe. The total computational cost (on average) of building a **BST with N keys** is **O**($N \cdot log(N)$) because we are performing **N** insert operations, each with a cost of $O(log(N))$. This is just the same as the cost of sorting an array so we can use binary search. **However**, once we have the **BST** we can **insert** new items into it at a cost of $O(log(N))$, whereas to insert a new item into the sorted array we would need to use **insertion sort** which has a complexity of $O(N)$ or re-sort the array at a cost of $O(N \cdot log(N))$. The real advantage of the **BST** is not in the **initial setup** but in the fact it allows us to perform **any succeeding operations efficiently**.

✍ **Exercise 4.5**

Draw the **BST** that results from inserting the following keys in sequence: 49, 85, 22, 15, 97, 67, 4, 71, 35, 8, 99, 64, 2, 17, 69, 50

4.4.3 Deleting items from a BST

The **delete** operation is a bit more interesting than both **search** and **insert**. This is because we must ensure that **after deletion the BST property still holds** everywhere in the tree.

There are three cases we must consider for deletion:

Case a) The item to be deleted is **a leaf node** (it has **no children**). Figure 4.13 shows an example of this case. The process to be carried out is:

- Find the node with the item we want to **remove** - same process as **BST search**.
- Since this a **leaf node**, there is **nothing else** in its **subtree**. Once we remove this node, there is nothing left.
- Therefore, we set the corresponding **pointer in the parent node to NULL** to indicate that there is now an **empty subtree** on the side where the deleted item used to be.

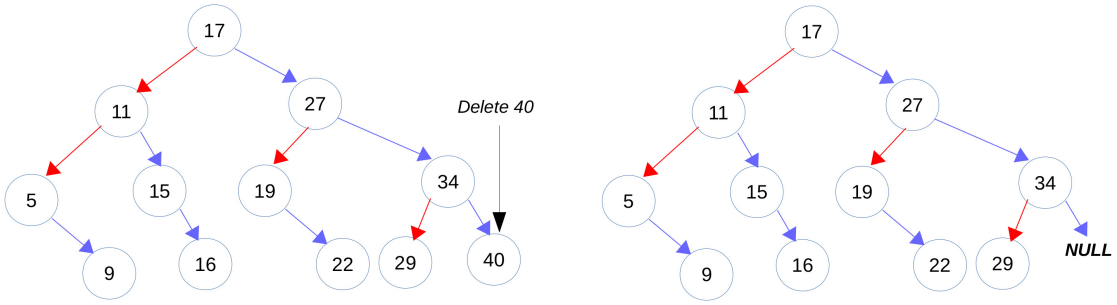

Figure 4.13: Example of **case a)** of **BST delete**. The item being deleted (40) is in a **leaf node**, therefore we simply delete the node and set the corresponding pointer **in the parent node** to **NULL**.

Case b) The item to be deleted has **one child only**. This is very similar to deleting an item inside a **linked list** as shown in Fig. 4.14. The process to be carried out is:

- Find the node with the item we want to **remove** - same process as **BST search**.
- The node's child is **guaranteed to be on the correct side with regard to the node's parent** because **BST insert** ensures the **BST property holds** throughout the whole tree.
- Therefore, we have to link **the node's parent** to **the node's child** and then delete the node.

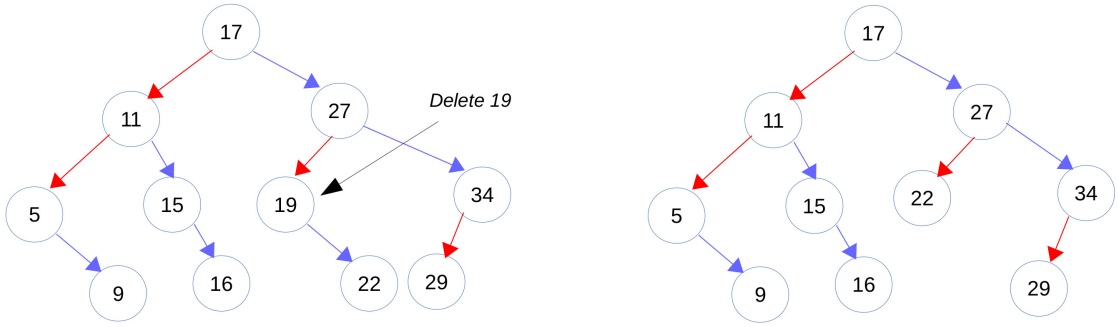

Figure 4.14: Example of **case b)** of **BST delete**. The item being deleted (19) has a single child (22). Linking the **parent node (27)** to the **child node (22)** preserves the **BST property** and we can then delete (19).

Case c) The most general case, the item to be deleted has **two children**. In this case, deleting the node and trying to re-link the tree results in a problem: we would have two dangling nodes (the children of the node we just deleted) and only one link where to attach them. Rather than try to figure out where to attach the two children while preserving the **BST property**, we can solve this problem by carefully **moving data items inside the tree** as shown in Fig. 4.15. The process is as follows:

- Find the node with the item we want to **remove** - same process as **BST search**.
- Search in **the right-side subtree** for the **successor** of this node - this is **the smallest key** in the **right subtree** of the node we are deleting. The **successor** is easily found by starting at the top of the **right subtree** and then traversing **as far down and to the left** as possible (i.e. without following any right-child links).
- **Promote** the **successor** to the **node we are deleting** - this means copying the data (and key) stored at the successor node into the node we are deleting.
- Finally, **delete the successor node** from **the right subtree** - this will be either **case a)** or **case b)**.

The point of this process is that we **do not remove internal nodes** in the **BST**. Instead, we move data around such that we are left with **a valid BST** (the BST property holds everywhere after we have moved the data), and **an easier case of deletion** either **case a)** or **case b)** since the **successor node** can **never have two children** (think through the process of finding the successor to see why that has to be the case).

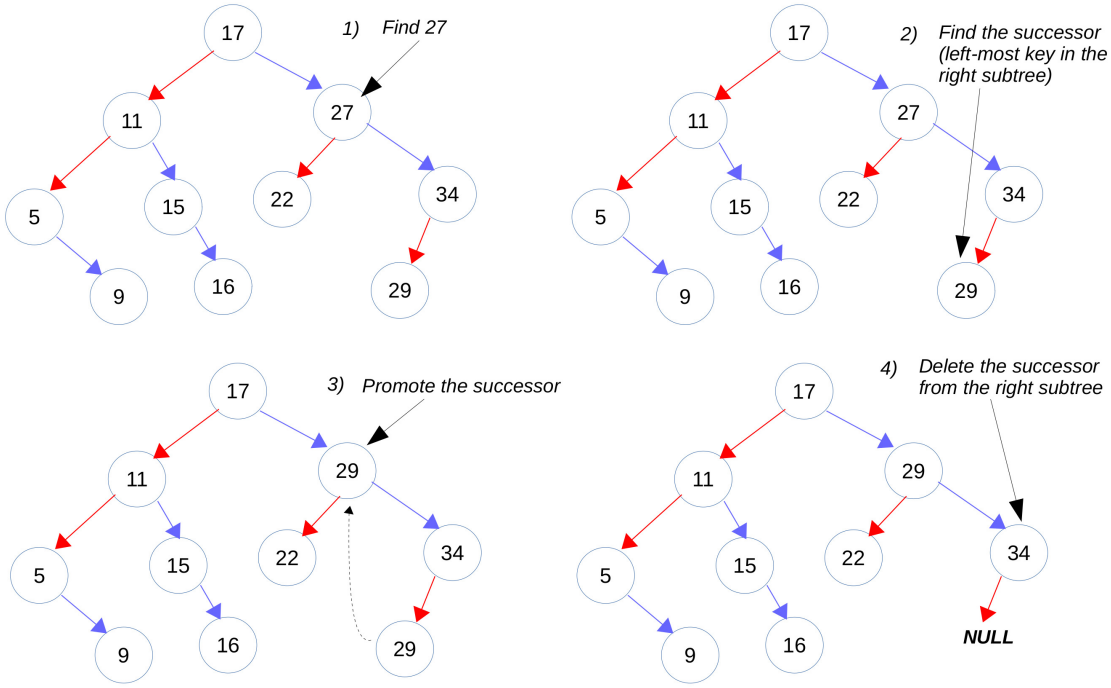

Figure 4.15: Example of **case c)** of deletion. The node has two children so we do not want to remove it, instead we **promote** the **successor** (smallest key in the right subtree) to the node we want to delete, and then **delete the successor** from the **right subtree**.

Note

It should be noted that deletion would work equally well if we used the **predecessor** (largest key in the left subtree) instead of the **successor**. Both of these options will result in a valid **BST** (the **BST property** is maintained throughout the tree), and they involve the same amount of work in terms of computational complexity - there is **no difference**. For **consistency**, in this book we will use the **successor** when deleting nodes from a **BST**.

✍ **Exercise 4.6** Beginning with the tree in Fig. 4.15 after (29) was deleted, draw the trees resulting from deleting: 9, 29, 22, and 11 in that order. Be sure to identify which case of deletion is needed for each number.

Question: What is the **average case Big O complexity** of deleting a node from a **BST**? The process of **finding the node we want to delete** is $O(log(N))$, it is just a regular **BST search**. The actual deletion requires a **fixed number of operations** for cases **a) and b)**. For **case c)** it involves **one additional search with cost** $O(log(N))$ to find the node's **successor**. So just like **search** and **insert**, the **average case Big O complexity of delete** is $O(log(N))$.

This last is important. It means that we can expect all **BST** operations to be **as efficient as binary search** in the average case.

Note

In practice, **BSTs** perform incredibly well under general conditions, but you may be concerned that they perform well only in the **average case**. It's worth pointing out that there are several other types of trees such as **AVL-trees** and **B-trees** that have the property that they **remain balanced** regardless of the order in which keys are inserted. Such **balanced trees guarantee** that the **Big O complexity** of **search, insert**, and **delete** operations remains $O(log(N))$ in the **worst case**. You can learn about these more sophisticated trees in any standard algorithms book.

4.5 Tree traversals

In linked lists, an implicit operation that needs to be carried out frequently is a **list traversal**. A full list traversal involves visiting **each node in the list exactly once**. There are many situations in which we need to visit nodes in a binary tree (for example, in a tree that stores numbers we may want to compute the average of the values in the tree). Because in **BSTs** there is more than one link at each node, that means there is a choice in terms of **the order in which we visit** the nodes to carry out whatever work we need to do.

The process of visiting each node in a **BST** exactly once is known as a **tree traversal**, and because there is a choice in terms of the order in which we visit nodes and work with the data stored in them, there are **3 different types** of **tree traversals** for **binary trees**.

4.5.1 In-order traversal

Possibly the most common type, it specifies nodes will be visited in the order given by the following procedure, starting at the **root** of the **BST**

At any particular node i:

- 1) Perform an **in-order traversal** of the left subtree
- 2) Carry out the specified work at node i
- 3) Perform an **in-order traversal** of the right subtree

The process is illustrated in Fig. 4.16. For this figure, the **work being done** at each node consists very simply of **printing the value in the node**. But in general this could involve more complicated processing of whatever data is stored in the corresponding **BST** node. What is important in the figure is understanding the **how the steps of the process** are applied at each node, and the **order in which nodes are processed**.

In-order traversal visits nodes **in sorted order** with respect to their **key value**. So in the sample case above, the process will print a **sorted list of keys** in the tree: 5, 11, 15, 17, 19, 27, 34.

What is the Big O complexity of in-order traversal? Just like **list traversal**, a **tree traversal** will need to visit each node in the tree. If there are **N** nodes, that means the traversal process will be $O(N)$.

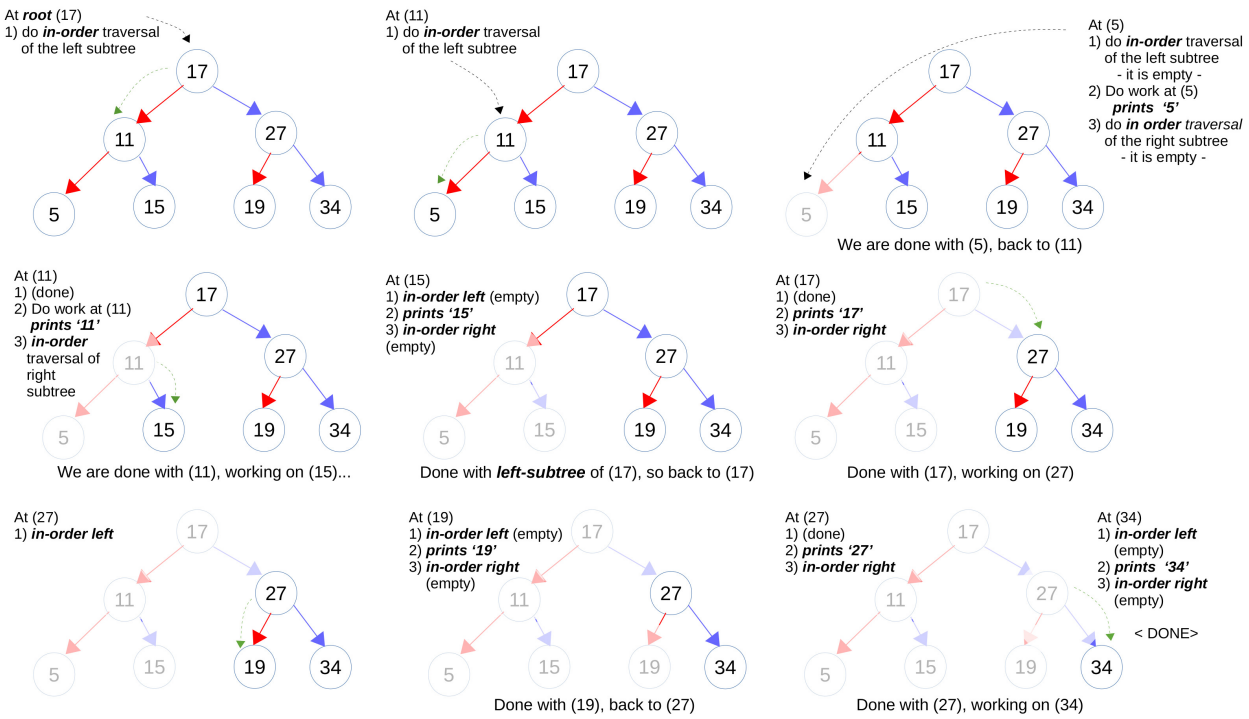

Figure 4.16: Step-by-step **in-order** traversal in a sample **BST**, note the order in which the three steps of the process are applied at each node. In particular, nodes whose **left subtree** is **not empty** have to wait until their left subtree has been processed before they get their turn.

Note

In-order traversal of a **BST** can be used for sorting. The process is called **tree sort**.

- Insert all items to be sorted into the **BST** using as **key** whatever data field or fields we want the data sorted by
- Perform an **in-order** traversal of the resulting tree - this produces the **sorted list of items**

What is the worst-case Big O complexity of tree sort?. We can figure this out by accounting for the **Big O** complexity of each of the two steps in the tree sort process.

- Inserting items into the **BST**: In the **worst case** the items are already sorted, and inserting them into the **BST** produces a structure that resembles a **linked list**. The cost of **each insert** is $O(N)$ because each new item will go **to the bottom of the growing tree**, and we perform N insertions which gives a complexity of $O(N^2)$ for building the **BST**
- The **in-order** traversal has a cost of $O(N)$ regardless of the shape of the tree

This gives a total cost of $O(N^2 + N)$ and as we saw above, the fastest-growing term will dominate this complexity bound, so the **worst case Big O complexity of tree sort is** $O(N^2)$. Which is the same as **bubble sort** and **not very good**.

However, if the input is **not sorted**, as we might expect if data arrives in **random order**, things look a lot better:

- Inserting items into the **BST** will in the **average case** have a computational cost of $O(N \cdot log(N))$
- The **in-order** traversal has a cost of $O(N)$

This gives a total cost of $O(N \cdot log(N) + N)$ and as usual the fastest-growing term dominates the complexity bound. So the **average case** cost of **tree sort** is $O(N \cdot log(N))$ which is on par (in terms of **Big O complexity**) with the best known sorting methods. If instead of a **BST** we use a **balanced tree such as an AVL**, then the $O(N \cdot log(N))$ cost is **guaranteed in the worst case**.

4.5.2 Pre-order traversal

One of the most common uses of a **pre-order traversal** is making an exact duplicate of a **BST**. By traversing the **BST** in **pre-order**, making a copy of each data item, and inserting the new item into another (initially empty) **BST** we obtain two identical trees - each corresponding node is in exactly the same location with respect to the root of the tree. Other applications include **search algorithms** on graphs, which we will study in the next chapter. The **pre-order** traversal is as follows:

At any particular node i:
- 1) Carry out the specified work at node i
- 2) Perform a **pre-order traversal** of the left subtree
- 3) Perform a **pre-order traversal** of the right subtree

The process is illustrated in Fig. 4.17. Same as for the **in-order** example, the work being done at each node is simply printing the value at the node. In this example, the traversal will print: 17, 11, 5, 15, 27, 19, 34. Which is pretty different from what we got out of the **pre-order** traversal.

4.5.3 Post-order traversal

The last type of traversal for binary trees is the **post-order** traversal. It has applications in **compilers** and **parsing languages**. It is also essential for managing **BSTs** because whenever we want to **delete a BST** this has to be done in **post-order**. **Post-order** traversal consists of:

At any particular node i:
- 1) Perform a **post-order traversal** of the left subtree
- 2) Perform a **post-order traversal** of the right subtree
- 3) Carry out the specified work at node i

The process is illustrated in Fig. 4.18. As in the previous traversal examples, the work being done at each node is simply printing the value at the node. **Post-order** traversal will print: 5, 15, 11, 19, 34, 27, 17. Note that **the parent node has to wait** until **both of its subtrees have been processed**, which is what we want when **deleting a BST** - we can not delete a node until both of its subtrees have been removed.

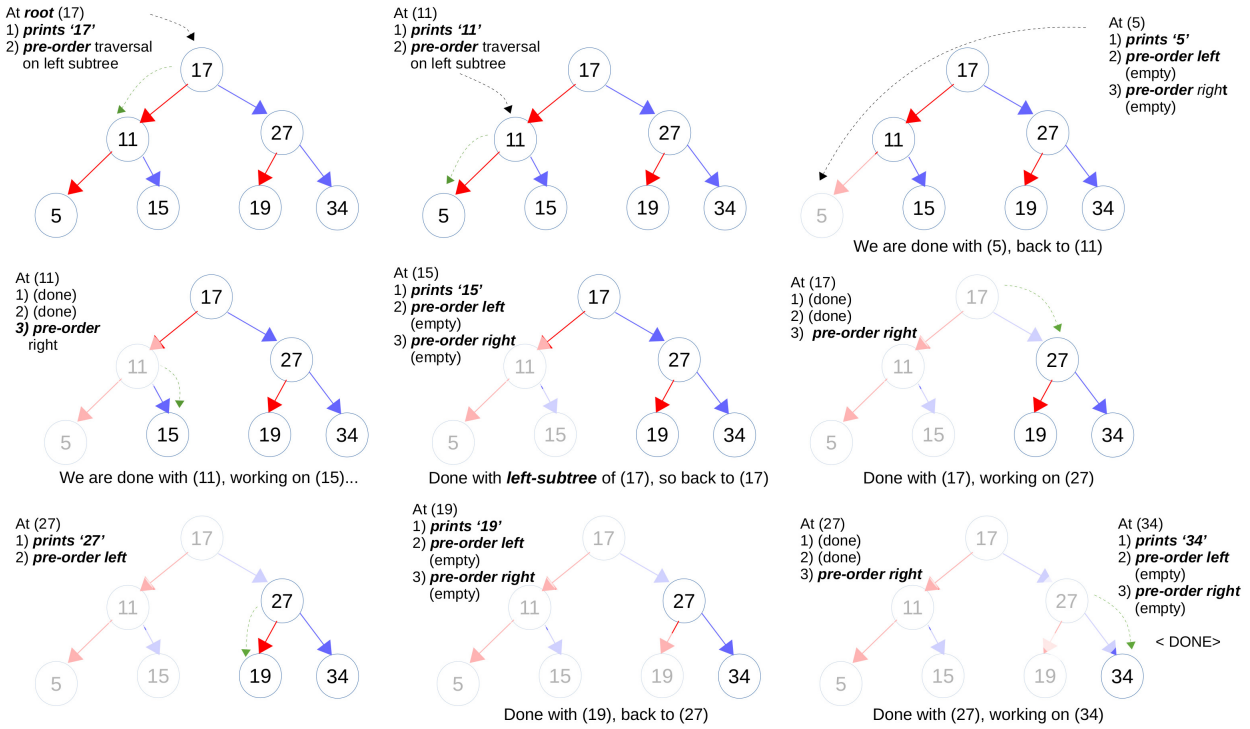

Figure 4.17: Step-by-step **pre-order** traversal in a sample **BST**. In this traversal method, the work at the node is done **before** either of the **subtrees** is processed.

4.6 Implementing a BST

Now that we have learned about all the operations that a **BST** supports, as well as the different **tree traversal** methods we can perform on a binary tree, the last step is to turn our **BST ADT** into a **data structure** by implementing all of the **BST operations** in **C**. The implementation of each function is listed below, you should read through it carefully and review the **pseudocode** and **examples** of the different operations as you work your way through the program code.

Firstly, let's define the **CDTs** we will need. We are going to work with the same example application as in the previous chapter (a collection of restaurant reviews), so we need a **CDT** for the **reviews**, and another one for the **BST nodes**.

```
typedef struct Restaurant_Score
{
  char restaurant_name[MAX_STRING_LENGTH];
  char restaurant_address[MAX_STRING_LENGTH];
  int score;
}Review;

typedef struct BST_Node_Struct
{
  Review rev;      // Stores one review
  struct BST_Node_Struct *left; // A pointer to its left child
  struct BST_Node_Struct *right; // and a pointer to its right child
} BST_Node;
```

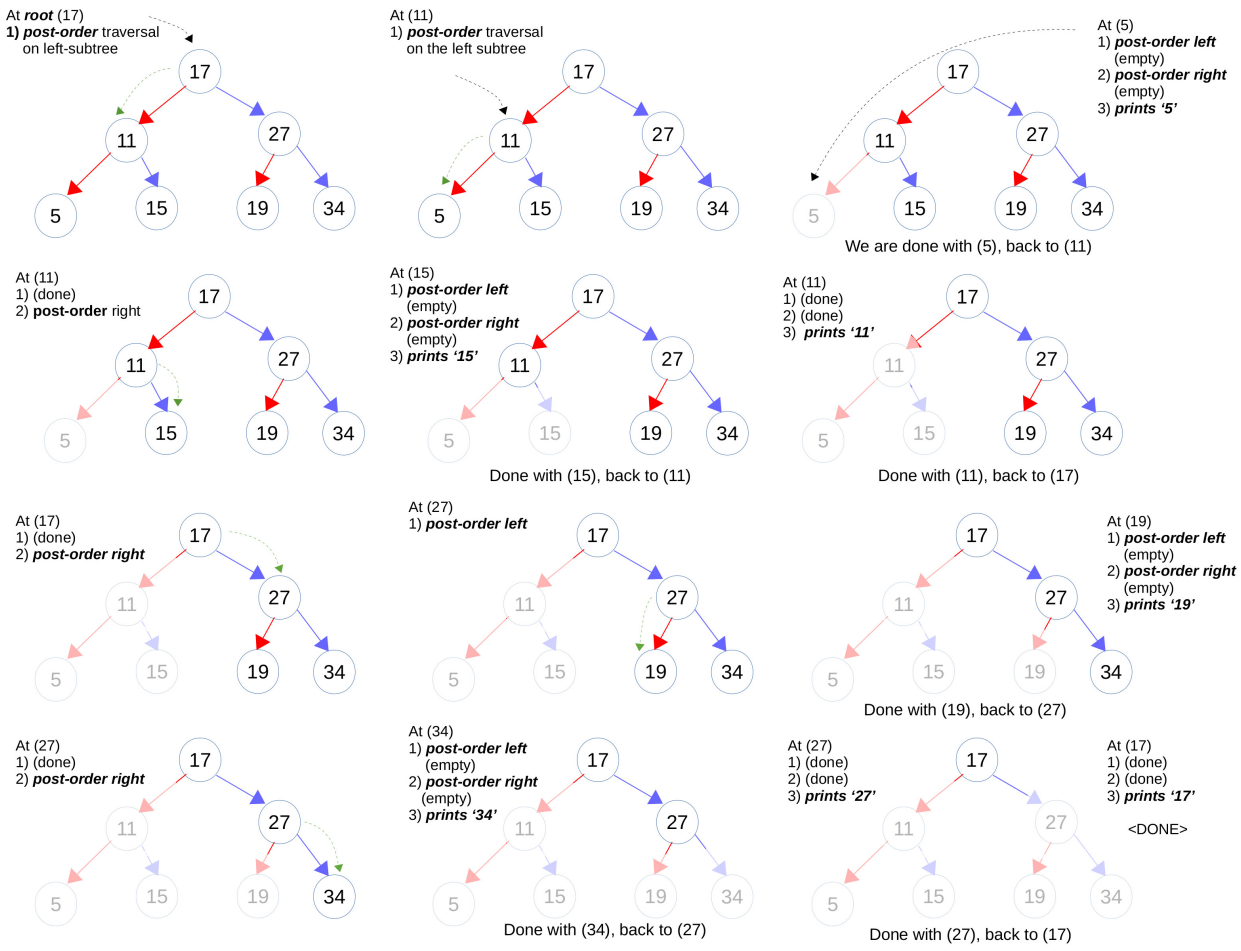

Figure 4.18: Step-by-step **post-order** traversal in a sample **BST**. In this traversal method, **both subtrees of a node** are processed before the node itself.

The **Review CDT** should be perfectly familiar to you by now. The **BST_Node CDT** is just a small modification of the **Review_Node CDT** we used for linked lists. The only difference is that instead of **a single** *next* **pointer**, we now have **two pointers** to **BST_Node CDTs**. One for the **left child**, and one for the **right child**. By convention, in most **BST** implementations you will ever come across, these pointers are called **left** and **right**.

Next we have to write a function to **allocate and return** a newly created **BST_node**.

```
BST_Node *new_BST_Node(void)
{
  BST_Node *new_node=NULL;  // Pointer to the new node

  new_node=(BST_Node *)calloc(1, sizeof(BST_Node));
    if (new_node==NULL)
    {
        printf("new_BST_Node(): Error, there is not enough memory to create new node, returning
            NULL\n");
        return NULL;
    }

  // Initialize the new node's content (same as with linked list)
  new_node->rev.score=-1;
  strcpy(new_node->rev.restaurant_name,"");
  strcpy(new_node->rev.restaurant_address,"");
  new_node->left=NULL;
  new_node->right=NULL;

  return new_node;
}
```

This function is very similar to the one we wrote for **linked lists**, nothing new here except for having to initialize two pointers rather than one.

Next up is the function that inserts a new item into the **BST**:

```
BST_Node *BST_insert(BST_Node *root, BST_Node *new_node)
{
    // This function takes as input the <root> of a subtree
    // (it can be any subtree, or the whole BST), and a
    // <new_node> that contains a review (already filled
    // with valid information).
    // It inserts the <new_node> into the BST

  if (root==NULL)   // Tree is empty, new node becomes
    return new_node; // the root

  // Check for duplicates (and refuse to insert duplicates)
  if (strcmp(new_node->rev.restaurant_name,\
      root->rev.restaurant_name)==0)
    {
        printf("BST_insert(): Duplicate data item detected. Ignoring\n");
        return root;      // The root did not change
    }

  // This node is occupied, we need to figure out on which
  // side of this node the <new_node> needs to go, and then
  // go and insert the <new_node> in that subtree.
```

```
if (strcmp(new_node->rev.restaurant_name,\
    root->rev.restaurant_name)<=0)
{
      // <new_node>'s key is less than the key at <root>,
      // it must be inserted in the left subtree

      root->left=BST_insert(root->left,new_node);

      // The situation at this point is as shown below:
      //
      //             /
      //            /
      //          root      <-- we're working here
      //         /
      //        /
      //    leftRoot        <-- The <new_node> should
      //                        go in this subtree
      //
      // leftRoot is stored in 'root->left'
      //
      // So, we tell the BST_insert() function to go and
      // insert the <new_node> in the subtree that
      // starts at 'root->left'.
      //
      // Why are we updating 'root->left' with the return
      // value of BST_insert()?
      //
      // Because if the situation is as follows:
      //
      //         /
      //        /
      //      root
      //     /
      //    /
      //   NULL
      //
      // Then the first check in BST_insert will simply
      // return <new_node> because this node will
      // become the top of a new subtree. Updating
      // 'root->left' results in
      //
      //        /
      //       /
      //     root
      //     /
      //    /
      //  new_node
      //
      //  If 'root->left' is NOT NULL when we call
      //  BST_insert(), nothing will happen as
      //  BST_insert() will simply return the same
      //  'root->left' it received.
}
else
{
      // <new_node>'s key is greater than the key at <root>
      // it must be inserted in the right subtree

  root->right=BST_insert(root->right,new_review);
```

```
    // Which works exactly as shown above, but on the
    // subtree that is on the right side of <root>
  }

  return root;  // Return the same <root> we received
}
```

The only thing worth commenting in the code above is that the **BST_insert()** function is that we use it **within the function itself** to get the work done. This is an example of **recursion**. We will take a detailed look at **recursion** as a general tool for solving a very large variety of problems that have particular characteristics. In this case, we observe that **inserting a node into a tree** requires us to be able to **insert a node into a subtree** (the left or right one, depending on the key). But every **subtree of a BST** is also **a BST** (see Fig. 4.19).

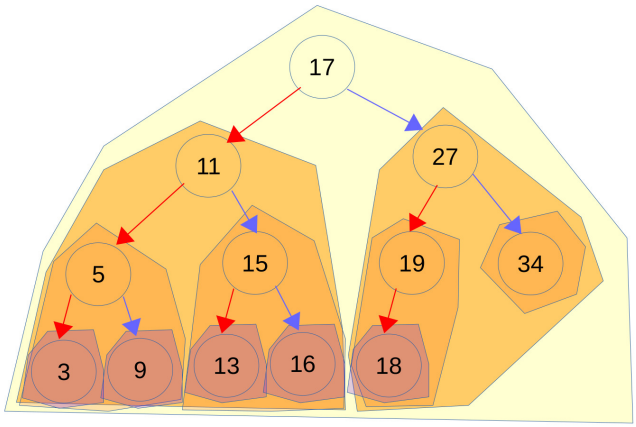

Figure 4.19: Each **subtree** in a **BST** is itself a fully valid **BST** - so inserting a key in a particular **BST subtree** is exactly the same thing as inserting a key in a **BST**.

We take advantage of this property to use the same **BST_insert()** function to handle the process as we traverse from the **root of the BST** to the place where the new node should go.

Because **C** reserves space for a function's variables at the moment we call the function (see Chapter 2), each **call to BST_insert()** will have **its own reserved space for input arguments, local variables, and return value**. The structure of the function shown above directly matches the pseudo-code description of how **BST** insert is supposed to work, there really is nothing special about **recursion**, it is the natural way to implement operations on trees.

The next function to implement is **search**:

```
BST_Node *BST_search(BST_Node *root, char name[1024])
{
    // The function takes as input the <root> of some
    // subtree in the BST (can be the root of the whole
    // tree).
    // It searches for the node that contains the matching
    // restaurant name, and if found, it returns a pointer
    // to the node containing the matching review.

  if (root==NULL) return NULL; // We reached an empty
                               // subtree, the queryKey
                               // is not in the BST
```

```
  // Check if this node contains the review we want, if so, return
  // the pointer to this node
  if (strcmp(root->rev.restaurant_name, name)==0)
    return root;

  // The queryKey is not in the current node, decide
  // if we need to search in the left subtree or
  // in the righ subtree.

  if (strcmp(name, root->rev.restaurant_name)<=0)
  {
        // queryKey is less than the key at <root>
        // so we must go search in the left subtree

        return BST_search(root->left,name);

        // The call above performs a search on
        // the left subtree, and will return
        // whatever we find there i.e. NULL if
        // the queryKey is not found in the
        // left subtree, or the pointer to the
        // node that contains the matching
        // review, whereved it may be in the
        // left subtree.
  }
  else
  {
        // queryKey is greater than the key at <root>
        // so we must go search in the right subtree

    return BST_search(root->right,name);
  }
}
```

Not surprisingly, **BST_search()** is also **recursive**. But if you work your way through it, you will see it is simply a direct implementation of the pseudo-code for **search** that we studied above.

The last operation we need to implement for a fully working **BST** is the **insert** operation:

```
BST_Node *BST_delete(BST_Node *root, char name[1024])
{
  // Takes as input the <root> of a subtree, and
  // the key of a node we want to remove. In this case
  // a restaurant's name.

  BST_Node *tmp;

  if (root==NULL) return NULL; // Tree or sub-tree is empty
                               // nothing to do.

  // Check if this node contains the review we want to delete

  if (strcmp(name,root->rev.restaurant_name)==0)
  {
        // Determine which of the three cases of
        // deletion we are dealing with.

    if (root->left==NULL && root->right==NULL)
```

```
{
  // Case a), no children. The parent will
  // be updated to have NULL instead of this
  // node's address, and we delete this node

  free(root);
  return NULL;

  // The situation here is as follows:
  //
  //          parent
  //         /
  //        /
  //      root      <-- We are at this node
  //
  // We need to delete the current node,
  // and because there are no children, the
  // 'parent' node will have an empty
  // subtree. So, the function returns
  // NULL so the parent can update its
  // subtree to look like so:
  //
  //         parent
  //        /
  //       /
  //     NULL
  //
}
else if (root->right==NULL)
{
  // Case b), only one child, left subtree
  // The parent has to be linked to the left
  // child of this node, and we free this node

  tmp=root->left;    // Store the pointer to the
  free(root);        // child!
  return tmp;

  // The situation is as follows:
  //
  //          parent
  //          /
  //         /
  //       root   <-- We are at this node
  //       /
  //      /
  //    child
  //
  // We need to connect 'parent' to 'child'
  // and remove <root>. So, we store
  // 'child' in a temporary pointer
  // (once we delete <root> that pointer
  // would be gone!)
  //
  //         parent
  //         /
  //        /
  //     (deleted)
  //
  //   tmp-->child
```

```
    //
    // The function then returns 'tmp' which
    // allows the 'parent' node to update its
    // own link so point to 'child'
    //
    //        parent
    //        /
    //       /
    //     child
  }
  else if (root->left==NULL)
  {
    // Case b), only one child, right subtree
    // The parent has to be linked to the right
    // child of this node, and we free this node

    tmp=root->right;
    free(root);
    return tmp;

    // Same as the case above, but the child node
    // is to the right of <root>
  }
  else
  {
    // Case c), two children.
    // We need to find the successor to this
    // node, promote it (copy the data to this
    // node), and then delete the successor
    // from the right subtree.

    tmp=root->right;   // Top of the right subtree
    while (tmp->left!=NULL)   // Successor is the
              tmp=tmp->left;         // leftmost key

          // Promote the successor (copy the data over
          // to <root>)
          root->rev = tmp->rev;

          // And finally delete the successor in the
          // right subtree
          root->right=BST_delete(root->right,tmp->rev.name);

      return root;
  }
}

// The review we want to delete is not in this node,
// determine if it should be on the left or right
// subtree, and call delete on the corresponding
// subtree.

if (strcmp(name, root->rev.restaurant_name)<=0)
{
      // The review we want should be on the left
      // subtree

  root->left=BST_delete(root->left,name);
}
else
```

```
  {
        // The review we want should be on the right
        // subtree

    root->right=BST_delete(root->right,name);
  }
  return root;      // Nothing changed here,
                    // return the <root> we
                    // received.
}
```

Just like **insert**, and **search**, the **BST_delete()** is recursive. It follows the pseudo-code for the **insert** operation as discussed above. While it does a bit more work than either of the other operations, the fundamental process is identical: Starting at the top of the tree, traverse downward choosing either the left or the right subtree until we find the place where work needs to be done.

This completes the implementation of all the **BST operations** required to build, maintain, and use a **BST** in **C**.

✍ **Exercise 4.7** Modify the app you created in the previous Chapter to manage the restaurant reviews so that it uses **BSTs** instead of **linked lists**. The goal here is to see how you can achieve **the same functionality** with **different data structures**.

There are two ways to go about this:

- a) Modify the listing from the previous Chapter to use **BST Nodes** and BST functions using the code above
- b) Expand on the code above by adding a **main()** function providing the same functionality as the program from the previous chapter. You will also need to implement any extra functions (i.e. beyond insert, search, and delete) that were present in the linked-list version

As you are building the application: Follow the **test-as-we-develop** process described at the end of Chapter 3. Make sure each function you add to your program is thoroughly tested before moving on to the next thing.

After your implementation is complete: Add a comprehensive set of **full program tests** to your test driver as discussed in Chapter 3. After all your testing is complete, you should be able to confidently state the program you implemented is solid, works on reasonable input, and handles tricky or erroneous input in a reasonable fashion.

✍ **Exercise 4.8** Implement **the three types of tree traversals**. The traversal functions should print the restaurant name, and the review score for each visited node, in the order specified by the traversal process. Add options in **main()** so that the user can request each of the three types of traversal.

Follow the test-as-we-develop process described at the end of Chapter 3.

✍ **Exercise 4.9 Crunchy!** As we know, if we are unlucky with the order in which items are inserted into a **BST**, we could end up with a **very unbalanced tree** in which some branches are much longer than others (and in the worst case, there is a single branch with all the keys in it).

Write **pseudo-code** for a process that can be used to take as input **an unbalanced BST**, and that will then create a new, **balanced BST** (and delete the old unbalanced one). Make sure that your design **follows the guidelines set at the end of Chapter 2**.

Once you are happy with your design, expand the program that handles restaurant reviews to provide an option to **re-balance the BST** and implement your process into a function that produces a **balanced BST** from whatever the **current BST** looks like.

Test-as-you-develop - write the function that balances a **BST**, and test it thoroughly before integrating it with the rest of your application. Then perform **full program testing**. Follow the testing guidelines at the end of Chapter 3. Your tests should include **verification that the resulting tree is balanced**.

How do we verify that a tree is balanced? there are several ways in which we can define **balanced**. But for simplicity, for this exercise we will define a **balanced BST** as one in which the **difference in height** between the **leaf node that is closest to root** and the **leaf node that is farthest from root** is less than **2**.

How do we find the minimum (or maximum) height of a leaf node? - that's for you to work out, but perhaps tree traversals could be handy...

4.7 Have we solved the problem?

At this point we have a good understanding of what **BSTs** can do, and how to implement them, but we have yet to determine whether all this work has helped us solve the problem of **finding a more efficient way to store, organize, and access a large collection of information**. We may suspect that the **BST** is likely to give us faster search because **its average search complexity is better** than the linked list's average search complexity, but we still have to tie a few loose ends:

a) The time it takes to build the BST compared to the time needed to build a linked list

- For **N** items, the linked list can be built in $O(N)$ time - each item is added at the head so no traversal is needed. That's pretty good!
- For the same **N** items, the **BST** can be built in $O(N \cdot log(N))$ time on average since insertions always happen at the bottom of the tree, and the tree has height $O(log(N))$

Advantage: Linked list

b) The time it takes to build a BST compared with the time needed to sort an array (so we can use binary search)

- Using **merge sort** an array can be sorted in $O(N \cdot log(N))$ time on the worst case
- As we saw above, the **BST** can be built in $O(N \cdot log(N))$ time on average

Advantage: Array - **merge sort guarantees** $O(N \cdot log(N))$ even in the worst case, while for **BSTs** the worst case is $O(N^2)$ - the worst case happens with data that is initially sorted or close to sorted, so it can come up on real-world situations.

c) Search complexity

- Sorted array: $O(log(N))$ worst case using binary search
- Linked list: $O(N)$ both average and worst case
- BST: $O(log(N))$ average case, $O(N)$ worst case

Advantage: The **BST and sorted array** on the average, **sorted array** in worst case. The linked list is **not competitive**, it is simply too slow on average.

d) Storage use

- Sorted array: Fixed size, **not suitable for growing/shrinking** collections as that would require re-allocating the whole array
- Linked list: Space usage is $O(N)$ - one node per item
- BST: Space usage is $O(N)$ - one node per item

Advantage: The **BST and linked-list**, because they only use space for items actually present in the collection, whereas the array has to **pre-reserve** space and we have the problem of estimating in advance **how much we may need** which is not trivial. A lot of the space reserved may go unused.

4.7.1 So what do we choose?

The point of going through all of this work is to help you see that there is a **fair number of factors you have to consider when choosing how to store and organize a collection of data**. Up to this point we have studied 3 different ways of storing items: **arrays**, **linked-lists**, and **BSTs**, and as shown above, each one of them has advantages and disadvantages. So how are we to choose?

a) Consider how large your collection is going to be:

- For very large collections that change frequently, you want to **choose the data structure** that gives you the **best Big O complexity** for common operations like insert, delete, and search.
- Consider the **space requirements**: For **items with small memory requirements** (e.g. just numeric data, like ints or floats, or small compound data types) it's **not unreasonable to pre-allocate a large array** even though entries in it may go un-used. Conversely, for items with a large memory footprint (e.g. compound data types with lots of data, or multi-media content like images, sound clips, etc.) we prefer a dynamic data structure.
- For **smaller collections**, you may want to choose based on **ease of implementation**. There is a trade-off between ease of implementation (arrays require less complicated code to work with than **BSTs**) since it saves developer time, and is likely to produce code with smaller likelihood of having difficult to find bugs.

b) Consider what kind of operations will be performed on the collection's items:

- If you will mostly perform **operations over the entire set**, choose a **linked list or an un-sorted array**. An example of this is the 3D point meshes used in computer graphics which may contain millions of points, but we don't usually perform individual point look-ups; instead, we almost always render the whole mesh. Meshes are often kept in arrays.

- Conversely, if individual item look-ups (**search**), insertions, or deletions make-up the majority of the operations on your collection, then you should **choose a data structure** that gives you the **best Big O complexity** for these operations.

Note

However, keep in mind that there is **no pre-set definition** for what it means for a collection to be **small** or **large** or **very large**. So likely what you will need to do is sit down and **create a table** such as the one shown in Exercise 4.1, then run a few numbers to see what storage method will result in the best performance for the **specific collection you will be working with**, then consider **ease of implementation**. Document your findings, and then make the best choice you can **supported by having thoroughly thought through** the issues mentioned above.

4.7.2 A brief note on databases

Real world systems that manage large collections of data do not rely on **a single kind of data structure**. For example, typical **databases** organize the actual item information into **tables** - these are basically **huge arrays** (stored on disk, not computer memory), where **each row contains all the data fields for one item in the collection** (in effect, one row in a database table is equivalent to a **CDT** we may have in memory). The tables are most often un-sorted.

Separately from the table that contains the actual data, the database keeps an **index in the form of a balanced tree** (typically a **B-tree** or a variation thereof) which contains **search keys** the user may want to run queries with. **Each node in the index contains the location in the original table where the information for the item matching the query key can be found** (this is just like a **pointer**, but instead of containing a memory location, it contains a disk location).

This has several important benefits: Firstly, **the index data structure is very small compared to the size of the collection** because it only stores **keys** and **pointers to entries in database tables**. Often, the whole index can be kept in memory (whereas the entire data collection may be too big for this). This is important because working with information that is in the computer memory is much faster than working with information stored on disk. Secondly, we can define **multiple indexes** over the same collection. For instance, a database that stores information about students at a University may have an index based on **student number**, and another index based on **student name plus date of birth**. This allows the users to perform queries using either the student's name and birth date, or their student number, and both will be equally fast.

Separating the index from the data has the advantage that it allows us to provide different **ways to find information** within the collection efficiently.

The above motivates one final thought regarding the problem of managing a large collection of information: **It involves thinking at multiple levels of detail**. Knowing how particular **ADTs** and their corresponding **data structures** work, as well as the complexity of typical data operations when we use these **data structures** is only the foundation. Once you have understood that part, you can begin thinking at a higher level of abstraction: How will the collection be used? what kinds of operations will be more frequent? what kinds of look-ups may be often required and therefore should be made very fast? and how do we organize the information available into possibly

many different data structures so that the overall system provides excellent performance for all required uses?

This is an extensive and fascinating topic, and you can learn a whole lot more about it by picking up a good book on databases.

4.8 Wrapping up

We have spent some time carefully studying the problem of **how to efficiently manage** large collections of information. Along the way we studied the problem of **measuring, characterizing**, and **reasoning about** the **complexity of algorithms and problems**. We have thought about **how to determine which data structure** will provide a better performance in a particular case (as described by the size of a collection and the types of operations that we will carry out on it), and we have thought about **how and when** we should use the different **ADTs** we have learned up to this point.

Keep in mind that the **theoretical analysis of complexity** is **only one layer** of the difficult problem of figuring out the most efficient way to carry out some task. You need to learn about computer architecture, how modern **CPUs** work, and how to optimize programs for maximum efficiency if you really want to write the best (most efficient, fastest) software to solve any given problem.

As for the **worst-case complexity**. The discussion above may lead you to believe you don't have to worry about it if your data structure has a good average-case complexity. Indeed, for most applications you will find you can get excellent results from using data structures such as **BSTs**. However, **for safety-critical applications**, or for **real-time applications**, examples of which include **industrial control software, medical equipment, electrical power generation, transportation (aircraft flight control), robotics, manufacturing, and so on**; you can **not afford to use a data structure whose worst-case performance is bad**. The point being that even if you need to be very unlucky to get the input that triggers worst-case or close to worst-case performance, you can not afford to take that risk. And this is due to the fact we often have to worry about malicious parties (i.e. hackers and other adversarial entities) who may use knowledge of how your system is built to find a weak point and exploit it. Applications of the type just mentioned **require guaranteed performance bounds** from all the data structures and algorithms used within their software.

4.9 Additional Exercises

✍ **Exercise 4.10** Draw the **BST** that results from inserting the following **keys** in sequence into an **initially empty BST**. Keys are sorted alphabetically.

"Iron Man 3", "Black Panther", "The Avengers", "Captain America", "Iron Man 2", "Aquaman", "Batman Returns", "Thor", "Spider Man: Into the Spiderverse"

✍ **Exercise 4.11** Draw the BST after we delete "Aquaman"

✍ **Exercise 4.12** Draw the BST after we delete "Iron Man 3"

✍ **Exercise 4.13** What is the list of movies generated by a **post-order traversal** of the **BST** from Exercise 4.12?

✍ **Exercise 4.14** It is highly likely that a **BST** whose **keys are movie titles** will run into the problem that there are multiple entries with the same key. As we discussed above this is really not a great situation. Give at least **3 different suggestions** regarding how we can build a **BST** where entries are organized by movie title, yet, there are no duplicate keys.

✍ **Exercise 4.15** Which of the following has the **lowest complexity** for large values of **N**?

- $250000 \cdot log(N)$
- $.001 \cdot N$
- $.000001 \cdot N^2$
- $5000 \cdot log(N^2)$
- $.01 \cdot N \cdot log(N)$

✍ **Exercise 4.16** Which of the following has the **lowest complexity** for small values of **N**?

- $5.25 \cdot N$
- $1.11 \cdot N$
- $5 \cdot log(N)$
- $2.1 \cdot log(N^2)$
- $.00001 \cdot N^2$

✍ **Exercise 4.17** Typically, changes to the **key values** in nodes for a **BST** are not allowed because they could break the **BST property**. However, in practice we may come upon a situation in which **we may have to update a data field used as key**. List the steps we could take to accomplish this without breaking the **BST property** in the tree. Write the process in pseudo-code, and then provide a function in **C** that does this and add it to your **BST** application to handle movie reviews.

Use the test-as-we-develop approach as described at the end of Chapter 3, documenting your tests for the new function and adding them to the **test-driver** you have been working on for the restaurant reviews app.

After your new function is thoroughly tested on its own, **add a set of tests** to verify it works well with the rest of the program, as per the guidelines in Chapter 3.

Finally, add an option to **main()** so the user has access to the new functionality. It should allow the user to change the name of the restaurant (which we are using as key) for any of the reviews **already present** in the **BST**.

✍ **Exercise 4.18** Implement **tree sort** from the description in the text (at the end of the subsection on in-order traversals). Your **tree sort** function should take as input an **un-sorted** array of integers (or floats, up to you), and return the **sorted array** using **tree sort** to carry out the sorting process.

4.10 Building programs that work - Part 4

In the previous Chapter we discussed the process for having a solid **testing strategy** that will allow us to thoroughly check that our program's functions are working as intended, and to verify that the completed software performs its task correctly on any input that may be provided to it.

A **solid testing process** will allow you to find **bugs** - errors in the way the program handles information that result in wrong behaviour. Examples of this can be output that is not correct given the input, it can be behaviour in the application that is not expected, it can be a serious problem that causes the program to break, or it can produce unexpected changes to information being managed by your program. The term **debugging** has been around for a long time (some say since the late 1800's) but you can see the first actual **bug** found inside a computer in Fig. 4.20.

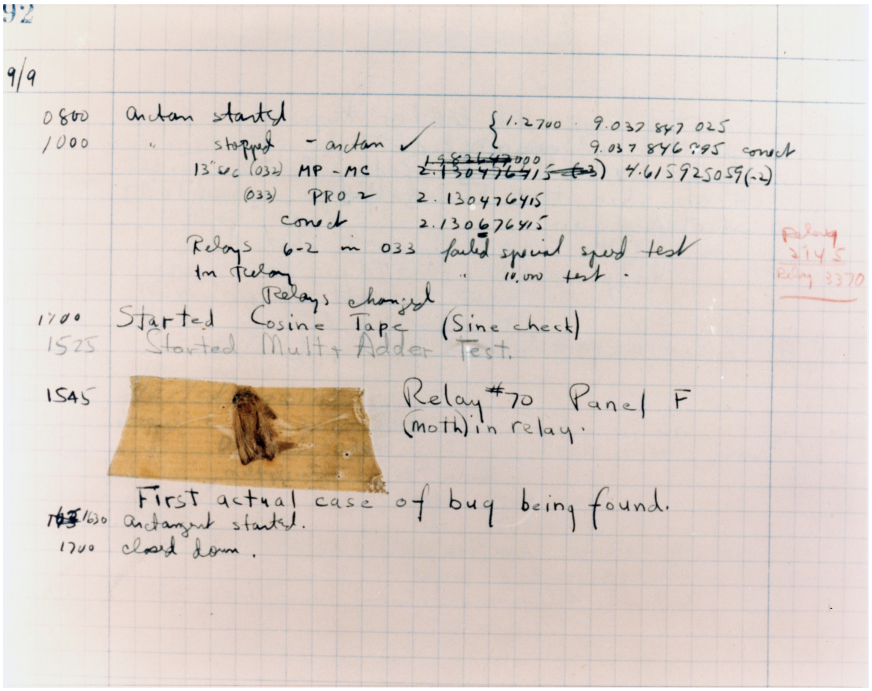

Figure 4.20: The first bug found inside a computer. This is from 1947 and resulted in erroneous results from math computations. *Photo: U.S. Government, Public Domain*

Though the variety of bugs that you will encounter while developing software is very large, the **process for finding and correcting them is the same**. The first step is to identify the existing of a bug. This may occur in different ways:

- The program fails one or more of your tests during development. This is normal while developing software and is the best way of finding and fixing bugs.
- A user reports a problem with the software once it's been released. Which should happen much less often if you have done your work properly. We want to avoid this, it is harder to find and fix bugs that were reported by users.
- Another developer, working with your program, reports a bug. This is common when working on **open source** projects which involve a community of developers working together.

Regardless of how the bug was initially identified, the process for fixing it involves:

- Figuring out how to reproduce the bug. With a failed test this is easy, you know which test failed and can run this test anytime to **trigger** the bug. With user-reported bugs it is a bit more difficult as we often don't have enough information about what was going on when the bug occurred. With bugs identified by a developer, we

usually get more information but we also have the option to contact the developer to figure out what **sequence of inputs or actions** can be used to **trigger** the bug.

- Once we have a way to **trigger the bug**, we need to determine **what is the correct behaviour** the program should have for this **test case**, **input sequence**, or **user action** (whatever the case may be) - we **can not fix a bug if we do not know what the program should do in the correct case**.
- Once we have both **a way to trigger the bug**, and **a thorough understanding of what the program should do in the correct case** it's time to **debug the program**.

4.10.1 What does it mean to debug a program?

The process of fixing a bug, **i.e. debugging**, consists of **tracing through the program** to check **each step the program carries out** to identify **at which point the program produces an unexpected/wrong result or behaviour**. Usually, in a well designed program, we start by **identifying the function** (or perhaps a small sequence of function calls) within which the bug occurs.

Once we have identified the function(s) where the bug occurs:

- We have to check what is happening at each step/line within the function(s) on the **input that triggers the bug**. We need to know what each step in the function is supposed to do, so we can verify each step and find which ones produce the wrong results.
- Once we have identified the line(s) where what the program does is different from what it should be doing:
  - Inspect the content of local variables, input arguments, arrays, and any other data that the program is using.
  - If the information the program is using is correct, then the lines of code that produce the wrong behaviour are likely wrong. Think through them carefully, trace the program on paper for the lines that are the likely problem and figure out which step(s) are incorrect, then fix them.
  - If the data the program is using is **not as expected**, then the source of the bug is somewhere else. You need to **go back to tracing the program** but this time you need to pay attention to where the data you identified as having the wrong value is being manipulated. Once you identify the section of the code where data gets modified or set incorrectly, think through that code, trace what it does on paper, and fix any problems you find.

Keep in mind that **if the problem is with information being used by a function**, the actual **origin of the bug** may be in an entirely different part of the program. However, **once you know which piece of information is getting wrong values** you can trace the program looking at parts that have access and can change that specific information to figure out where things are going wrong.

4.10.2 How do we trace a program?

The simplest way is to **add a lot of print statements**, each of which has to provide you with (at the very least):

- The specific location at which the print statement is taking place - so when you see something wrong you can immediately find what part of the program the corresponding information was printed from.
- Values of any variables, input arguments, arrays, or data structures that are being used at the program at that specific point, and which you need to know in order to determine if the program is doing the right thing at that particular point.
- If the print statement is within a loop, the **iteration of the loop**, so if a problem is detected you know at what point in the loop it happened.

Your task is then to read through the sequence of printed information, to identify where in the sequence something is not correct, and then to go into the program to determine what may be the problem at the location corresponding to where the print statement was produced.

This is a general way to trace through a program, and will work on a wide range of settings. It doesn't require special tools being available. But on the flip side, it doesn't provide you with control over the process of **tracing**. You can not **stop the program** to inspect what is happening in memory, and you have to dig through **possibly very long** sequences of printed information.

A more powerful, but also more complex alternative to using print statements is the use of a **debugger**. This is a specialized tool that allows you to control the process of **tracing through a program** so you can run each step, then have a look at what is going on in memory with the program's variables and data, and then continue with the next step once you have checked everything is ok up to that point.

A standard **debugger** is called **gdb**, and is available in all **Unix/Linux** systems, which includes a large proportion of **computing servers** and other systems you may need to work with. So knowing how to use **gdb** is a valuable skill. The **gdb debugger** is a fairly sophisticated program, so if you want to learn to use it you should spend a meaningful amount of time working through a tutorial on the subject. Many such tutorials are available both as electronic documents and in the form of how-to videos. If you want to follow up on this, here's a good place to get started: `https://users.ece.utexas.edu/~adnan/gdb-refcard.pdf`.

The goal of using a **debugger** is the same as that of using print statements - you want to be able to **inspect what is happening with the program at each step** so that you can **check what is happening against what you know should be happening** and identify the place(s) where something is wrong.

4.10.3 Pitfalls to avoid

Whenever there is a problem with a program it is very important to remember: **computers are deterministic, predictable, and execute the instructions the program contains**. It is easy to fall into one of the following **pitfalls** by thinking that:

- The program is correct, **it must be some random thing with the computer**. This is **incorrect**. Computers are predictable and deterministic. If they do something, it is because the program caused them to do that. Believing the program is correct against the evidence that there is a problem is one sure way to fail to fix the problem.
- It works on a different computer, therefore it's a problem with the computer. This is **most often incorrect**. It can actually happen that a hardware problem may cause a program to fail in weird ways but this is **very rare**.

Most likely, there are differences in how the program is running on these computers and these differences cause the bug to trigger in one but not the other. This requires **very careful investigation** - with very high probability the program has a bug and thorough testing will find it.

- It fails because the test was **tricky** so it is ok, we don't need to fix anything. This is **incorrect**. Thorough testing **requires** that we **try to break the program**. This means testing on **unexpected** or **possibly incomplete or incorrect** inputs and verifying that the program does the right thing and doesn't fail. Our programs should **not fail** because we have not devoted enough thought to making them solid.
- The bug is probably very difficult to trigger by a real user (i.e. it would show very very rarely) so it is ok not to fix it. This is also **incorrect** and **unethical**. If we know there is a problem with the code, **it is our duty to fix it**. Something like this: `https://www.cnn.com/2019/05/05/us/boeing-737-max-disagree-alert/index.html` is bf wrong and should never be accepted.

Some bugs will be much harder to find and fix than others, and the more complex the software is, the more time and effort it will likely take to test and debug thoroughly. However, if you are careful and follow a thorough, logical process to testing and debugging you can **find and fix** any reported bug. **Avoid convincing yourself that it is acceptable to ignore a problem**.

4.11 Time to Practice

In order to practice the process we have established for developing working software, from program design, to testing, to debugging, we will look at one fairly short example that nevertheless will allow us to practice all of the ideas we have covered thus far.

The problem: Our task is to implement a **signal filtering** function. Signal filtering is a common task in applications that deal with **media**, **communications**, and **machine learning** among others. A common example of filtering is **smoothing a signal**, for instance in order to reduce noise. Typically, we will have access to **an input** which usually represents a signal of interest, such as an audio recording; and a **filter** which is a **pattern of values** we will use to combine **data in the input** to produce a result. The filtering process is illustrated in Fig. 4.21.

The **filter** must be **odd-length** since we have to align its central entry with each input entry in order to compute the corresponding output. Normally, the length of the **filter** is much smaller than the length of the signal. For example one second of high-quality sounds data would contain over **40,000** entries, whereas a typical **smoothing filter** would have 5, 7, or 9 entries.

The process itself is straightforward, as shown in the Figure.

4.11.1 Step 1 - Understanding the problem and designing the solution

As discussed at the end of Chapter 2, the first step in implementing a program that works for solving the above problem is to **understand the problem** thoroughly. One way to verify you've understood the process above, is to do an example of the filtering process on paper, with a short filter and maybe one or two different input locations. You can use this later on when designing the tests for functions in the program.

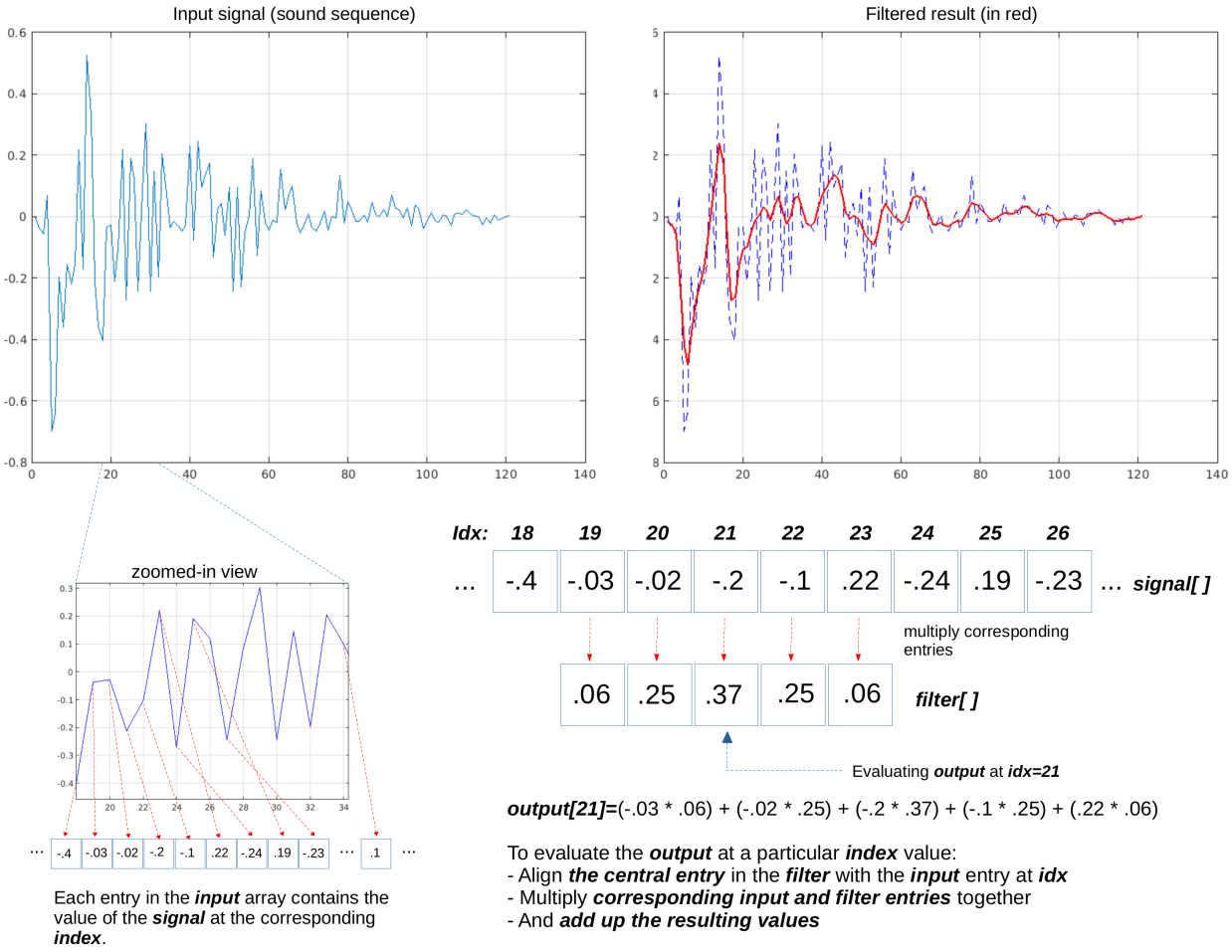

Figure 4.21: The process of filtering a 1D signal. The value of **output[i]** is the result of aligning the center of the **filter** with the entry at **input[i]**, and then adding up the result of mutiplying corresponding values. This has to be done at every **valid** entry in **input**. **Valid entries** are those for which after aligning the filter with the input, there is one input entry to multiply with each filter entry. For simplicity, the **output** value will be set to **zero** for **non-valid** entries.

✍ **Exercise 4.19** Carefully review the description of the problem in the figure, and make notes to yourself that clearly describe the process of filtering a 1D signal, including any conditions placed on the inputs and/or output of the filtering process. Work out at least one short example on paper showing the computation that has to be carried out by the filtering program.

The next step in our process is to come up with a written description of the algorithm, in enough detail that it can be implemented.

✍ **Exercise 4.20** **Before continuing,** write a complete description of the algorithm for 1D filtering. You can do this as a list of steps in bullet point format, or as pseudo code. **Do not attempt to write C code at this point**.

You can compare your description of the algorithm with the **pseudocode** below (take special note of any steps that are different/missing and if needed review the algorithm description to resolve any differences).

```
Algorithm: Filter1D - performs 1D filtering of an <input> array
                    with the specified <filter>.
                    Produces an <output> array with the same
                    length as the <input>, where each entry is
                    the result of applying the <filter> to the
                    corresponding entries in the <input> array.

Procedure:

* Loop over entries in the <input> array, one by one, using index <i>

    - For each value of <i>
        * Align the central entry in <filter> with the <input> entry <i>
        * Check that the every entry in <filter> has a corresponding
          entry in <input>
            - if this is not the case, set <output[i] := 0> and
              continue with the next <i>
        * Compute the result of applying the <filter> to the
          <input> at <i>: Multiply corresponding entries, and
          add up the result of these products (this operation is
          called a cross-correlation)
        * Update <output[i]> to be the result of the cross-correlation
```

Having understood the problem, we have to design our solution. There is **always more than one way, and often many ways** to do this, and in most cases **there is no single correct way** to solve any reasonably interesting problem. So your focus here is not **to find the one right solution** but rather to design a solution that **makes sense to you**, follows **good design practice** in terms of breaking up your algorithm into **self-contained functions**, that you can **test and debug** with reasonable ease, and that, where applicable, **uses data structures and algorithms with good known complexity** for the problem at hand.

✍ **Exercise 4.21** **Before continuing,** come up with your own design for a program that implements the pseudocode shown above. This means deciding how to break the problem into functions, and figuring out what the information flow will be between these functions and **main()**.

In the case above, a reasonable way to break up the problem into functions we have to implement could involve:

- A function called **crossCorrelation()** that computes the **cross-correlation** between an **input** array and a **filter** array at a specified index **i**
- A function called **filter1D()** that **loops over the input array** calling the **cross-correlation** function to obtain the value of the **output**

As noted above, this is only one way to solve the problem, someone else may decide to do everything in the algorithm within as single function. Both of these solutions have their own advantages and disadvantages. We will proceed with the breakdown above.

4.11.2 Step 2 - Testing as we develop

At the end of Chapter 3, we looked at the process of **testing a program as we are developing it**. This involves **choosing the order of implementation** for the various functions, then **implementing them in order** and **thoroughly testing each function** as we complete it before moving on to the next one.

For the problem at hand, we have two functions, and it is fairly clear that **cross-correlation** has to be implemented first because it is used by the function that computes the **output** array. Therefore, we will

- Implement **crossCorrelation()**
- Thoroughly test **crossCorrelation()**
- Implement **filter1D()**
- Thoroughly test **filter1D()**

and once we have completed the steps above, we will then **thoroughly test the complete process** of filtering 1D signals to make sure it works for a variety of possible inputs as well as a variety of filters.

✍ **Exercise 4.22** Implement **crossCorrelation()** and **filter1D()** following the design and pseudocode described above. There is a reference implementation below, which you will use for **testing** and **debugging**, but you should first implement these functions yourself - compare your implementation with the reference below and carefully consider any differences. You can apply the tests and debugging process described below to your own implementation as well as the reference one.

✍ **Exercise 4.23** Write down a set of tests for the **crossCorrelation()** function, the **filter1D()** function, and **the complete working code**. The tests should be thorough and allow you to discover any potential bugs or errors in the computation being performed by the filter.

4.11.2.1 Step 3 - Debugging

In order to practice **debugging**, consider the implementation below for the two functions that comprise our 1D filter:

```
double crossCorrelation(double input[],int n, double filter[], int m, int idx)
{
  // This function returns the cross-correlation between an input
  // array and a filter array at input location 'idx'.
```

```
  //
  // Inputs: input[] - a 1D array of size n
  //         filter[] - a 1D array of size m
  //
  // n>m , and m must be an odd integer number
  // (n-1)-hs >= idx >= hs
  //
  // The cross correlation is defined as
  //
  // cross_correlation=sum_{i = -hs}^(hs) input[idx-i]*filter[i+hs]
  //

  double cross_correlation=0;
  int hs=(m-1)/2;               // Half the filter size
  int i;

  for (i=-hs; i<hs; i++);
  {
    cross_correlation+=input[idx-i]*filter[hs-i];
  }

  return cross_correlation;
}

void filter1d(double input[], int n, double filter[], int m, double output[])
{
 // This function takes a 1D input array of size n, and a 1D filter of
 // size m, and produces an output result (same size as the input) that
 // is the result of evaluating the cross-correlation between the input
 // and the filter at each valid location.
 //
 // Inputs: input[] - a 1D array of size n
 //         filter[] - a 1D array of size m, m<n, m is odd
 //         output[] - a 1D array of size n
 //

 double cc;
 int hs;
 int i;

 hs=(m-1)/2;

 for (i=0;i<=n;i++)    // Initialize output array to zeros
   output[i]=0;

 for (i=0; i<=n; i++)
   output[i]=crossCorrelation(input,n,filter,m,i);
}
```

The code above looks reasonable and correct, but it in fact **contains bugs**. Your task now is to use the tests you developed above to **identify problems** with the implementation, and once you have found a problem, **debug it** by following the process described in this Chapter.

✍ **Exercise 4.24** Use the set of tests you developed to find **bugs** in the implementation above. Once you identify a **bug**, carefully **trace through the code** using either **carefully placed, informative printf() statements**, or a debugger such as **gdb** in order to identify where the problem is, and then correct it. Remember you must be able to follow each instruction in the implementation, and you must know what the correct result of each instruction must be (in

terms of the variables and data involved in them).

If you do not discover any problems with your tests, that would indicate your set of tests was not thorough enough, and you should revisit Chapter 3 and the process of testing, then expand or change your set of tests accordingly. The end result of working your way through this section and completing all of its exercises as well as the debugging task should be an implementation that is thoroughly tested, solid, and that will perform 1D filtering on any input signal and **valid** filter correctly.

✍ **Exercise 4.25** Carry out the testing and debugging process on your own implementation of the functions for the 1D filter. Once you are satisfied that your own implementation works correctly, perform an additional set of **high-level tests** that consist of filtering multiple, different input signals with various filters, using both the (now fixed) reference implementation, as well as your own. **compare the output** from both of these for each input/filter test case, and ensure they are identical for every test. If you find any differences, then either or both of the implementations still contain bugs. Find and correct any problems found in this manner.

The final exercise in this section illustrates one more way in which we can **test** a program we have developed: by **comparing it** with a different implementation (which can be in a different programming language, or simply written by a different developer or team). This is common practice for software that implements functionality that is well known or common, and for which we can easily find reference implementations available for our testing. However, be careful:

- **Do not copy or immitate code from the reference library** - Code that has been developed by someone else is protected by copyright, copying it or using trivial variations of it is an infraction. You are expected to develop your own solution independently and without looking at the work of others. We will discuss how to **incorporate and re-use** code from **open source** or **creative-commons** libraries in a later Chapter.
- **Do not assume** that the reference implementation is **free of bugs**. If you find a disagreement between results obtained with your own code, and that of the reference implementation, all that you can conclude from this is that **at least one of them has a bug**, you have to carefully trace through your program, and if necessary also the reference implementation in order to determine which one(s) have errors.