TIME COMPLEXITY ANALYSIS OF ALGORITHMS

Introduction

We would like to characterise, in a mathematically meaningful way, the complexity of an algorithm. Complexity has many aspects:

- **Conceputal complexity**: that is, the difficulty of writing and verifying the correctness of the algorithm in the first place. We do not have any way of quantifying the complexity of an algorithm in this sense. All other things being equal, we would like to design the simplest and easiest to understand algorithm possible, but we will not formally talk about these notions.

- **The length of the algorithm.** This sense of the complexity of an algorithm can be quantified and is a useful measure in other contexts. This measure is related to Kolmogorov complexity. This measure should not be confused with the “conceptual complexity” of an algorithm: a shorter algorithm is not necessarily easier to understand. We do not consider this measure of complexity in this course.

- **The amount of time the algorithm takes to run.** This is known as the time complexity of the algorithm. This measure, specified in fuller detail below, is the primary focus of the complexity analysis done in this course.

- **The amount of storage used by the algorithm.** This is known as the space complexity of the algorithm. This measure is also a concern in the complexity analysis done in this course, although perhaps not as central as time complexity.

Machine Model and Time Complexity of Algorithms

The worst-case time complexity of an algorithm is expressed as a function

\[ T : \mathbb{N} \to \mathbb{N}, \]

where \( T(n) \) is the maximum number of “steps” in any execution of the algorithm on inputs of “size” \( n \). For example, \( T(n) = 3 \cdot n^2 \) means that on inputs of size \( n \) the algorithm requires up to \( 3 \cdot n^2 \) steps. To make this more precise, we must clarify what we mean by input “size” and “step”.

(a) A general definition of the input size to an algorithm is the number of bits in the input. This is not a very convenient measure for most of the problems we deal with in this course — that is, for the problems considered in this course there is another measure of the input size which is more natural to talk about. When we are dealing with algorithms which solve problems of searching and sorting, the measure of input size we adopt is the number of elements which are to be searched through or sorted. When we are dealing with algorithms which solve graph problems, the measure of input size we adopt is expressed in terms of the number of edges in the graph and the number of nodes in the graph. In this case, the worst-case time complexity of the algorithm is expressed as a function of both of these measures of the input size.

(b) A step of the algorithm can be defined precisely if we fix a particular machine on which the algorithm is to be run. We don’t want to restrict ourselves to any particular machine, so we’ll be a bit vague as to exactly what a step is (in most cases, when we analyze the time complexity of a particular algorithm to solve a particular problem, we will specify in more detail exactly
what the time complexity measures): basically, a step is anything we can reasonably expect a computer to do in a fixed amount of time. Typical examples are:

- Performing an arithmetic operation (+, −, ×, ÷).
- Comparing two numbers.
- Assigning a value to a variable.
- Following a pointer.

There is a problem with assuming that these steps only take a fixed amount of time. For example, an arithmetic operation takes constant time only if the numbers involved are of bounded size.

*Uniform Cost Criterion.* Charge one unit of time for each step.

*Logarithmic Cost Criterion.* Charge log \( n \) units of time for a step, where \( n \) is the maximum number involved in the operation. The rationale for this cost criterion is that the time to perform a simple step is proportional to the length of the numbers involved, which is log \( n \). Even this isn't always quite true. For example, we do not know how to multiply two numbers of length \( n \) this quickly.

The uniform cost criterion makes the analysis simpler but fails to take into account the size of the numbers, which implies that it is a bit unrealistically optimistic. The logarithmic cost criterion is more realistic. In this course, unless otherwise specified, we stick to the uniform cost criterion.

Recall that \( T(n) \) is the maximum number of steps required for inputs of size \( n \), i.e., the worst-case time complexity of the algorithm. Another useful measure of time complexity is *average-case* time complexity. The average-case time complexity of an algorithm, \( T(n) \), is the *expected* number of steps required for inputs of size \( n \). (Thus \( T : \mathbb{N} \to \mathbb{R} \).) For this to be a well-defined concept, we must define a probability space on the set of inputs of size \( n \) for each value of \( n \). The main advantage of measuring average-case time complexity instead of worst-case time complexity is that the average-case time complexity of an algorithm gives a more realistic idea of what to expect when the algorithm is run on a problem of size \( n \), given that it is possible to determine a probability density on the set of inputs to the algorithm which reflects the true probability density on inputs of size \( n \). There are also disadvantages of measuring average-case time complexity instead of worst-case time complexity:

- The probability density on inputs depends on the application where the algorithm is to be used, and is usually not even known.
- Even when the probability density is known, average-case time complexity analysis is usually much more difficult than the worst-case time complexity analysis for the same algorithm.
- In some real-time applications, it is critical that on each input the algorithm be guaranteed to finish within a particular time bound. The worst-case time complexity analysis is useful to see if the criterion is met, whereas average-case time complexity analysis is not at all useful for this purpose.

In this course we concentrate mostly on worst-case time complexity, but we'll see some examples of average-case analysis.

† We will also use a third type of time complexity, called *amortised time complexity*. More on this in good time.

[1]
“Big-Oh” Notation

We are often only interested in obtaining a rough estimate of an algorithm’s time complexity. Since, in the interest of generality, we measure time in somewhat vaguely defined “steps”, there is little point in spending too much time to obtain a precise expression for this number of steps. We are happy with a reasonable approximation.

A style of time complexity analysis that is very common in algorithm analysis is the so-called asymptotic analysis. Except on rare occasions, this is the type of analysis we’ll work with in this course (you’ll be happy to know it’s the easiest!). In asymptotic analysis we drop low order terms and multiplicative constants from the time complexity function \( T(n) \). For example, if the function is \( 3 \cdot n^2 + 7 \cdot \log n \) we are only interested in the “\( n^2 \)” part: \( 7 \cdot \log n \) is a low order term and 3 is “just” a multiplicative constant. Some special mathematical notation has been developed to express asymptotic analysis conveniently. To express asymptotic upper bounds on running times of algorithms, we introduce “big-oh” notation.

**Definition.** Let \( f : \mathbb{N} \to \mathbb{R} \).

\[
O(f) = \{ g : \mathbb{N} \to \mathbb{R} \mid \text{There exist } c > 0 \text{ and } n_0 \geq 0 \text{ such that for all } n \geq n_0, g(n) \leq c \cdot f(n) \}.
\]

In words, \( g \in O(f) \) (pronounced “gee is in big oh of ef”) if for all sufficiently large \( n \) (for \( n \geq n_0 \) \( g(n) \) is bounded from above by \( f(n) \)—possibly multiplied by a positive constant. We say \( f(n) \) is an asymptotic upper bound for \( g(n) \). Figure 1 illustrates graphically what it means for \( g \) to be in \( O(f) \).

![Figure 1. \( g(n) \in O(f(n)). \)](image)

**Example 1.** \( f(n) = 3 \cdot n^2 + 4 \cdot n^{3/2} \in O(n^2) \). This is because \( 3 \cdot n^2 + 4 \cdot n^{3/2} \leq 3 \cdot n^2 + 4 \cdot n^2 \leq 7 \cdot n^2 \). Thus, pick \( n_0 = 0 \) and \( c = 7 \). For all \( n \geq n_0 \), \( f(n) \leq c \cdot n^2 \).

**Example 2.** \( f(n) = (n-5)^2 \in O(n^2) \). This is because \( (n-5)^2 = n^2 - 10 \cdot n + 25 \). Check (with elementary algebra) that for all \( n \geq 3 \), \( n^2 - 10 \cdot n + 25 \leq 2 \cdot n^2 \). Thus, pick \( n_0 = 3 \) and \( c = 2 \). For all \( n \geq n_0 \), \( f(n) \leq c \cdot n^2 \).
Exercise. Prove the following.
(i) \( n \in O(n^2) \).
(ii) \( 3n + 1 \in O(n) \).
(iii) \( \log_2 n \in O(\log_3 n) \).
(iv) \( O(\log_2 n) \subseteq O(n) \). (Hint: \( \log_2 n < n \) for all \( n \geq 1 \)).
(v) \( O(n^k) \subseteq O(n^l) \) for all constants \( k < l \).

Why is this useful? In many cases obtaining the exact time complexity function \( T(n) \) of an algorithm is a very laborious task and yields a messy function. However, it is often relatively easy to find a much simpler function \( f(n) \) that is an asymptotic upper bound for \( T(n) \), prove that \( T(n) \in O(f(n)) \), and leave it at that (without bothering to find an exact expression for \( T(n) \)).

Of course, we want to find the best asymptotic upper bound on \( T(n) \) possible. For example, if \( T(n) = 3 \cdot n^2 + 4 \cdot n^{3/2} + 5 \cdot \log n + 16 \) then \( T(n) \in O(n^2) \). But, \( T(n) \) is also in the sets \( O(n^3), O(n^4), ..., O(n^{100}) \). To say that \( T(n) \in O(n^{100}) \) is true but quite uninformative. We want to find a simple asymptotic upper bound that is as small as possible. The bound \( f(n) \) is as small as possible for \( T(n) \) if \( T(n) \in O(f(n)) \) and for all functions \( g(n) \) such that \( T(n) \in O(g(n)) \), it is also the case that \( f(n) \in O(g(n)) \). In words, \( f(n) \) is as small an asymptotic upper bound as possible for \( T(n) \) if \( f(n) \) is an asymptotic upper bound for \( T(n) \) and all other functions which are asymptotic upper bounds for \( T(n) \) are also asymptotic upper bounds for \( f(n) \). For the \( T(n) \) given above, \( f(n) = n^2 \) is as small an asymptotic upper bound as possible, and so is \( g(n) = 53 \cdot n^2 + 16n \). We much prefer to state \( f(n) \) as an asymptotic upper bound on \( T(n) \) rather than \( g(n) \) because \( f(n) \) is simpler. On the other hand, if \( h(n) = n^2 \log n \) then \( T(n) \in O(h(n)) \), but \( h(n) \) is not as small an asymptotic upper bound as possible (why?).

“Big-Omega” Notation

There is a similar notation for asymptotic lower bounds, the “big-omega” notation.

Definition. Let \( f : N \to R \).

\[ \Omega(f) \overset{\text{def}}{=} \{ g : N \to R \mid \text{There exist } c > 0 \text{ and } n_0 \geq 0 \text{ such that for all } n \geq n_0, g(n) \geq c \cdot f(n) \}. \]

In words, \( g \in \Omega(f) \) (pronounced “gee is in big omega of ef”) if for all sufficiently large \( n \) (for \( n \geq n_0 \)) \( g(n) \) is bounded from below by \( f(n) \) — possibly multiplied by a positive constant. We say \( f(n) \) is an asymptotic lower bound for \( g(n) \). Analogous to the case for asymptotic upper bounds, we are interested in finding a simple asymptotic lower bound which is as large as possible.

Notice the symmetry between “big-oh” and “big-omega”:\( g(n) \in O(f(n)) \) iff \( f(n) \in \Omega(g(n)) \).

Definition. \( \Theta(f) \overset{\text{def}}{=} O(f) \cap \Omega(f) \).

Thus, if \( g(n) \in \Theta(f) \) (pronounced “gee is in big theta of ef”) then asymptotically \( g(n) \) and \( f(n) \) grow at the same rate within a positive multiplicative constant. In terms of this notation our previous concepts simplify to: \( f(n) \) is an asymptotic upper bound (lower bound) on \( T(n) \) which is as small as possible (as large as possible) if and only if \( T(n) \in \Theta(f(n)) \).

Exercise. Prove the following.
(i) \( n^2 \in \Omega(n) \).
(ii) \( \Omega(n \log n) \subseteq \Omega(n) \).
(iii) \( n \in \Theta(n) \).
(iv) \( 3n + 1 \in \Theta(n) \).

(v) \( \sum_{i=1}^{n} \log_2 i \in \Theta(n \log_2 n) \).

(Hint for (v): For \( 1 \leq i \leq n \), \( \log_2 i \leq \log_2 n \). For \( \left\lceil \frac{n}{2} \right\rceil \leq i \leq n \), \( \log_2 i \geq (\log_2 n) - 1 \).

**Useful Rules for “Big-Oh” Notation**

The sets \( O(f) \), \( \Omega(f) \), and \( \Theta(f) \) have the following useful properties.

(i) \( g \in O(f) \) if and only if \( f \in \Omega(g) \).

(ii) \( O(f) = O(g) \) if and only if \( f \in O(g) \) and \( g \in O(f) \).

(iii) \( O(f) = \Theta(g) \) if and only if \( f \in \Theta(g) \).

(iv) If \( f \in O(g) \) and \( g \in O(h) \) then \( f \in O(h) \).

**Exercise.** Prove the above properties.

The following rules are useful for analysis of the running times of algorithms.

*Rule of Sums I:* If \( g_1 \in O(f_1) \) and \( g_2 \in O(f_2) \) then \( g_1 + g_2 \in O(\max\{f_1, f_2\}) \).

*Rule of Sums II:* Let \( f_i, g_i : \mathbb{N} \to \mathbb{R} \), for \( i \in \mathbb{N} \), be such that for some \( c > 0 \) and \( n_0 \geq 0 \), \( g_i(n) \leq c \cdot f_i(n) \), for all \( i \in \mathbb{N} \) and \( n \geq n_0 \). Let \( k : \mathbb{N} \to \mathbb{N} \), and define \( G(n) = \sum_{i=1}^{k(n)} g_i(n) \) and \( F(n) = \sum_{i=1}^{k(n)} f_i(n) \). Then, \( G \in O(F) \).

*Rule of Products:* If \( g_1 \in O(f_1) \) and \( g_2 \in O(f_2) \) then \( g_1 \cdot g_2 \in O(f_1 \cdot f_2) \).

**Exercise.** Prove the above three rules.

Rule of Sums I is useful for finding an asymptotic upper bound on the time complexity of an algorithm which is composed of the execution of one algorithm followed by the execution of another algorithm. Rule of Sums II and Rule of Products are useful for determining the time complexity of loops.

By applying the Rule of Sums I or the Rule of Products repeatedly a constant number of times \( k \), we obtain the following generalisation: Let \( g_i \in O(f_i) \) for \( i = 1, 2, \ldots, k \). Then

\[
\sum_{i=1}^{k} g_i \in O(\max\{f_1, f_2, \ldots, f_k\})
\]

\[
\prod_{i=1}^{k} g_i \in O(\prod_{i=1}^{k} f_i).
\]

It is very important to understand that in these generalisations, \( k \) must be a constant. The generalisations of the two rules obtained by repeatedly applying them a non-constant number of times are not valid. To see this, let \( g(n) = 2 \) for all \( n \in \mathbb{N} \). Thus, \( g(n) \in O(1) \). If we applied the rule of products \( n \) times with respect to \( g(n) \) we would obtain \( \prod_{i=1}^{n} g(n) \in O(1^n) \), i.e., \( 2^n \in O(1) \), which is patently false. What went wrong is that we are applying the rule of products a non-constant number of times.
Example of the time complexity analysis of a simple algorithm

Selection sort is an algorithm for sorting an array. It works roughly as follows: First it finds the smallest element in the array and interchanges it with the first element of the array; then it finds the second smallest element and interchanges it with the second element of the array, etc. Let \( A \) be the array filled with keys that is to be sorted. The procedure \( \text{swap}(a, b) \) interchanges the value of \( a \) with the value of \( b \).

\[
\text{Selection Sort}(A : \text{array}[1..n])
\]

begin
  \( i := 1 \) to \( n - 1 \) do
    begin
      smallest := \( i \)
      for \( j := i \) to \( n \) do
        if \( A[j] < A[\text{smallest}] \) then
          smallest := \( j \)
        end
      end
    end
We first calculate the time for the \( i \)th iteration of the main loop (which starts at line (1)). Then we obtain the time complexity of the entire algorithm by summing up the time complexities of all the iterations \( (i = 1, 2, \ldots, n - 1) \). In the \( i \)th iteration, we have to account for:

- line (2) — takes constant \( O(1) \) time
- line (3) — takes time equal to the number or repetitions of the loop times the time for line (4).
  The time for line (4) is in \( O(1) \). Thus, by the Rule of Products, the time for line (3) is in \( O(n - i + 1) \).
- line (5) — takes constant time.

Thus, the \( i \)th iteration of the main loop is the total time for lines (2), (3) and (5), which, by the Rule of Sums I, is \( \leq c \cdot (n - i + 1) \), for some constant \( c \) (what do you think \( c \) should be?). By Rule of Sums II, the time complexity \( T(n) \) of the entire algorithm is in

\[
O \left( \sum_{i=1}^{n-1} (n - i + 1) \right).
\]

But,

\[
\sum_{i=1}^{n-1} (n - i + 1) = n + (n - 1) + (n - 2) + \cdots + 2 = \frac{n(n + 1)}{2} - 1 = \frac{n^2}{2} + \frac{n}{2} - 1.
\]

Thus, \( T(n) \in O(n^2) \).
AMORTISED COMPLEXITY

Sometimes we are faced with the problem of processing a long sequence of operations that access and modify a data structure. Accordingly, we are interested not just in the time complexity of each operation individually but, more importantly, in the time complexity of processing the entire sequence. The amortised complexity is the worst case time needed to process the entire sequence divided by the number of operations. In this handout we discuss amortised complexity and the techniques used to determine it. We shall make use of two examples to illustrate this: the “list example” and the “stack example”.

The list example: We maintain a set of elements in the form of a linked list and periodically insert new elements (avoiding the insertion of duplicates), search the list to determine if a given element is present, and delete existing elements. Thus we are interested in three types of operations: Insert, Search and Delete. If we implement them in the straightforward way, each of these operations takes $\Theta(n)$ in the worst case, where $n$ is the length of the list at the time the operation is applied.

The stack example: We have a (slightly modified) stack with operations Push($x$) (which puts element $x$ on top of the stack) and Pop($k$) (which removes and returns the top $k$ elements from the stack — if there are fewer than $k$ elements it just returns all that exist). If we use an array to store the stack and a variable to indicate the present top of the stack, we can easily implement Push in $\Theta(1)$ time and Pop($k$) in $\Theta(k)$ time.

In general, we have a sequence of operations $\sigma = a_1, a_2, \ldots, a_m$ which are applied, in order, to a data structure. The data structure is in some initial state $S_0$; after the $i$-th operation the data structure is in state $S_i$. We can denote this process as

$$S_0 \xrightarrow{a_1} S_1 \xrightarrow{a_2} S_2 \xrightarrow{a_3} \ldots \xrightarrow{a_m} S_m$$

where $S_{i-1} \xrightarrow{a_i} S_i$ means that operation $a_i$ is applied to data structure $S_{i-1}$ and modifies it to $S_i$.

In our list example, it could be that the sequence is something like

$$\sigma = \text{Insert 5, Insert 7, Insert 2, Insert 17, Search 5, Delete 2}$$

and initially the list is empty. If we assume that the operations are implemented in the straightforward way, each of the first four operations modifies the list by inserting the indicated number to it; the fifth operation does not change the list (hence, $S_5 = S_4$, in this case); and the sixth operation removes 2, resulting in a list with just 5, 7 and 17.

Let $T(\sigma, S_0)$ be the total time (i.e., total number of steps) that it takes to process sequence $\sigma$, starting with the data structure in state $S_0$. That is, if operation $a_i$ takes $t_i$ steps, then $T(\sigma, S_0) = \sum_{i=1}^{m} t_i$. In many cases, there is a default initial state for the data structure: in the list (and stack) example, for instance, it often makes sense to assume that the initial list (or stack) is empty. In such cases, we shall simply write $T(\sigma)$, instead of $T(\sigma, S_0)$, if $S_0$ is the default initial state.

Consider a specific data structure and operations that can be applied to it. Let $S$ be the set of possible states of the data structure — e.g., the set of all possible lists or all possible stacks in our examples. The (worst case) sequence complexity is the function $C : \mathbb{N} \times S \to \mathbb{N}$, where

$$C(m, S) \overset{\text{def}}{=} \max\{T(\sigma, S) : \sigma \text{ is a sequence of } m \text{ operations}\}$$
The \textit{amortised complexity} is the function \( A(m, S) \stackrel{\text{def}}{=} C(m, S)/m. \) In other words, in amortised complexity we are spreading the cost of processing the entire sequence of \( m \) operations equally among the \( m \) operations. It is important to understand that amortised complexity is different from the worst case complexity of individual operations in the sequence, since different operations may need different amounts of time, and in defining amortised complexity we have “averaged” the total time equally among all operations. It is also important to understand that amortised complexity is different from average-case analysis of the time needed to process a sequence of operations. There is no probability distribution involved in amortised analysis: we are looking at the worst case sequence of operations; we are simply “averaging” the cost of that sequence over all operations.

If \( S \) is the default initial state, we shall write \( C(m) \) instead of \( C(m, S) \), and \( A(m) \) instead of \( A(m, S) \). The difficult part of doing an amortised analysis is determining the sequence complexity \( C(m) \). Once this is done all that we need to do is divide by \( m \) to get the amortised complexity. Thus, in the rest of this handout we focus on the sequence complexity \( C(m) \).

As usual, we do not try to determine \( C(m) \) (and \( A(m) \)) exactly, but characterise it asymptotically. That is, we determine a function \( f \) so that \( C(m) \in \Theta(f(m)) \). This requires us to prove two things:

(a) For every sequence \( \sigma \) of \( m \) operations, \( T(\sigma) \in O(f(m)) \) — this proves that \( f(m) \) is an asymptotic upper bound for \( C(m) \),

(b) For some sequence \( \sigma \) of \( m \) operations, \( T(\sigma) \in \Omega(f(m)) \) — this proves that \( f(m) \) is an asymptotic lower bound for \( C(m) \).

In the rest of this handout we describe techniques that can be used to determine the amortised complexity of data structure problems.

\textbf{Considering operations independently}

One way to find an \textit{upper bound on} \( C(m) \) is to determine the worst case time complexity of the \( i \)-th operation, \( a_i \), and then sum up these times for all \( i \). It is important to realise that this quantity is only an \textit{upper bound on} \( C(m) \). This is because in computing this sum we assume that the \( i \)-th operation is the most time consuming one, \textit{for each \( i \) independently}. It is possible, as we shall see shortly, that performing a time consuming operation may result in the \textit{necessary} reduction of the time required by subsequent operations. In such instances it is not fair to assume that every operation is a worst case possible operation. In other words, the problem with such an analysis is that it ignores the possible effect that one operation may have on subsequent ones. Sometimes this does no harm, but in other cases it is a problem because it gives us an upper bound that is too pessimistic.

Let us consider our two examples. As it will turn out, this approach will give good results in the list example, but not so good ones in the stack example.

\textbf{The list example:} It is not hard to see that the worst case for the \( i \)-th operation is any operation (Insert, Search or Delete) which has to traverse the entire existing list. Since the preceding \( i-1 \) operations can have resulted in a list with at most \( i-1 \) elements, it follows that the maximum time for the \( i \)-th operation is (proportional to) \( i \). Thus, in the case \( C(m) \leq \sum_{i=1}^{m} i = m(m + 1)/2 \in O(m^2) \). However, \( m^2 \) is also an asymptotic lower bound for \( C(m) \): Consider a sequence of \( m \)

Insert operations, inserting \( m \) distinct elements: the \( i \)-th insertion will require time (proportional to) \( i \) and therefore the entire sequence will require time \( m(m + 1)/2 \). Since we have a particular sequence of \( m \) operations that requires \( m(m + 1)/2 \) time, \( C(m) \in \Omega(m^2) \). We already proved that \( C(m) \in O(m^2) \), so \( C(m) \in \Theta(m^2) \), and \( A(m) \in \Theta(m) \).
**The stack example:** Let’s find an upper bound on the time complexity of processing a sequence of \(m\) Push or Pop operations (on an initially empty stack) by summing up the maximum possible cost of each operation. It is clear that the worst case scenario for the \(i\)-th operation (when \(i > 1\)) is to first have \(i - 1\) Push operations, followed by a Pop \((i - 1)\). Then the worst case time complexity of the \(i\)-th operation is (proportional to) \(i\). Thus we have,

\[
C(m) \leq \sum_{i=2}^{m} i = m(m + 1)/2.
\]

Thus \(C(m) \in O(m^2)\), and \(A(m) \in O(m)\).

But how good an asymptotic upper bound is this? Some thought should make us suspect that it might not be a good one. We arrived at it on the assumption that each operation in the sequence of operations is an expensive Pop, i.e., a Pop that removes lots of elements from the stack. But expensive Pop’s can’t be happening all the time: if one Pop removes a lot of elements from the stack, we must perform a lot of (inexpensive) Push’s to rebuild the stack before we can do another expensive Pop.

Now, of course, the preceding argument does not *conclusively* show that \(C(m) \notin \Theta(m^2)\). It is conceivable, for all we know, that there is some way of interleaving \(m\) Push and Pop operations so that the resulting sequence requires \(\Omega(m^2)\) steps to be processed. Since we already have that \(C(m) \in O(m^2)\), the existence of such a sequence would imply that \(C(m) \in \Theta(m^2)\). However, hard as we may try, we won’t be able to come up with such a sequence. This is not coincidental: a different type of analysis of the situation will reveal that, in fact, \(C(m) \in \Theta(m)\).

**Accounting for the dependence among operations**

If summing up the worst case time complexity of each operation independently fails to give a good enough upper bound on the sequence complexity, we must make a more careful analysis of the situation. In particular, we must somehow take into consideration the possible effects of one operation in the sequence on the time complexity of others.

Let’s consider our stack example again. Since we assume that the stack is initially empty, an element cannot be popped from the stack unless it was previously pushed onto it. Furthermore, the time to process a sequence \(\sigma\) of Push and Pop operations is proportional to the number of individual elements pushed and popped in the processing of \(\sigma\). Let \(r\) be the total number of elements ever pushed onto the stack during \(\sigma\). Then the total number of elements pushed or popped is at most \(2r\). Since \(r \leq m\), the number of elements pushed or popped is at most \(2m\) and therefore, the time complexity of processing any sequence of \(m\) operations is \(O(m)\). Of course, each operation takes at least constant time, so there is a sequence of \(m\) operations that requires time in \(\Omega(m)\) (in fact, any sequence of \(m\) operations does). Therefore, \(C(m) \in \Theta(m)\), and \(A(m) \in \Theta(1)\).

This analysis accounted for the dependence among operations by using the fact that the “cost” of a Push operation is, in some sense, due to previous Pop operations. The dependence among operations is not always as direct as in this case, but this example demonstrates the idea in its simplest form.

The argument we used above seems *ad hoc* and specific to the stack example. We shall now repeat this proof in a more systematic fashion in two different ways, which illustrate two different points of view of amortisation: the physicist’s view and the banker’s view. In each case we shall see not only the proof for our specific example but also the general principles involved in such analyses. Almost all analyses that use amortisation adopt, implicitly or explicitly, one of these points of view.
The physicist’s view of amortisation

According to this view we associate with each state of the data structure a potential. More precisely, we define a function $\Phi$ which maps each possible state $S$ of the data structure to a (usually non-negative) real number $\Phi(S)$, called the potential of $S$. Intuitively, we want “good” states of the data structure to have “low” potential, where a data structure state is deemed “good” if operations applied to such a state are cheap (i.e., don’t take a long time). Another way of saying this is that the potential measures the capacity of the data structure to extract work from future operations.

We can actually define the potential any way we wish but the success of the analysis in producing a tight bound for $C(m)$ depends on the choice of $\Phi$. Indeed, almost all the ingenuity needed for a successful analysis based on the physicist’s view goes into the judicious definition of $\Phi$.

Suppose that as before we want to process a sequence $\sigma = a_1, a_2, \ldots, a_m$ of operations starting with a data structure in state $S_0$:

$$S_0 \xrightarrow{a_1} S_1 \xrightarrow{a_2} S_2 \xrightarrow{a_3} \cdots \xrightarrow{a_m} S_m.$$ 

Let $t_i$ be the time required for operation $a_i$ in this sequence. Define the amortised (or adjusted) time of operation $a_i$ (with respect to potential function $\Phi$) as follows:

$$u_i \overset{\text{def}}{=} t_i + \Phi(S_i) - \Phi(S_{i-1}).$$

That is, the amortised time is the actual time plus the difference in the potential of the data structure resulting from performing this operation. The intuition behind this definition is this: The potential of a data structure state is a measure of how “bad” that state is, i.e., how expensive operations applied in that state can be. Now, as we observed before, although a particular operation may be expensive, it can also result in a state of the data structure which makes subsequent operations inexpensive. In other words, such an expensive operation transforms the data structure from one with high potential to one with low potential. That is, the difference in potential is negative and therefore the amortised time $u_i$ is small, even though the real time $t_i$ may be large. By accounting for the beneficial change in the state of the data structure in this manner, the hope is that we will have properly taken into account the effects of operations on each other. It should be clear even from this abstract and general description that everything hinges upon the proper choice of the potential function.

Recall that $T(\sigma, S_0)$ is the time it takes to process sequence $\sigma$, starting with the data structure in state $S_0$. Then we have

$$T(\sigma, S_0) = \sum_{i=1}^{m} t_i = \sum_{i=1}^{m} (u_i + \Phi(S_{i-1}) - \Phi(S_i)) = \sum_{i=1}^{m} u_i + \Phi(S_0) - \Phi(S_m).$$

If the initial state has potential 0 and the potential function is always non-negative, then $T(\sigma, S_0) \leq \sum_{i=1}^{m} u_i$ and we can therefore obtain an upper bound on $T(\sigma, S_0)$ by getting an upper bound on the amortised times for the individual operations.

Let’s apply this methodology to our stack example. A good stack is one in which we can’t perform expensive operations, i.e., one with few elements in it. Thus, a reasonable candidate for the potential function for a stack in state $S$ is:

$$\Phi(S) = \text{number of elements in } S.$$
The amortised time of operation $a_i$ is determined as follows:

If $a_i$ = Push then $u_i = 1 + \Phi(S_i) - \Phi(S_{i-1}) = 1 + (\Phi(S_{i-1}) + 1) - \Phi(S_{i-1}) = 2$.

If $a_i$ = Pop($k$) then $u_i = k + \Phi(S_i) - \Phi(S_{i-1}) = k + (\Phi(S_{i-1}) - k) - \Phi(S_{i-1}) = 0$.

Thus $T(\sigma, S_0) = \sum_{i=1}^{m} u_i + \Phi(S_0) - \Phi(S_m) \leq 2m$, since each $u_i \leq 2$, $\Phi(S_0) = 0$ and $\Phi(S_m) \geq 0$. This is the case for an arbitrary sequence of $m$ operations. Thus, $C(m) \in O(m)$. It is easy to see that any sequence of $m$ operations will require time in $\Omega(m)$ — since each operation takes at least constant time! Thus, $C(m) \in \Theta(m)$, and $A(m) \in \Theta(1)$. This proves that the upper bound of $O(m^2)$ for $C(m)$ that was obtained by summing up the worst case costs of individual operations was, indeed, too pessimistic.

### The banker’s view of amortisation

According to this view, we allocate a certain number of credits to each operation. A credit is supposed to pay for a constant amount of work. The image to keep in mind here is that of a coin-operated computer: Suppose that we get running time on a computer by dropping quarters (representing credits) into a slot. A quarter pays for a fixed amount of time on the computer. When an operation is to be processed, we drop the credits allocated to that operation into the slot. If the operation takes less time than the amount “bought” by its credits, the unused time is saved up and can be used by a future operation that requires more time than can be bought with the credits allotted to it. The credit scheme makes it possible to account for the fact that the running time of one operation in the sequence is dependent on other operations: for example, an expensive operation may be running on saved time — by using credits saved by previous operations; this captures the fact that the time required by that expensive operation is, in some sense, “caused” by the operations whose saved credits it uses. It is in this way that analyses using the banker’s view account for the effects of one operation on the running times of other operations.

The most difficult part of this type of analysis is to devise a suitable credit scheme, i.e., rules for allocating credits to operations. This task is analogous to the choice of a suitable potential function in the physicist’s approach. The success of the analysis depends almost entirely on this choice. In particular, the credit scheme must have the property that, for any sequence of operations, it is possible to process the entire sequence by using only the credits allotted to the operations of the sequence. If we can do this, then we can get an upper bound on the worst case time complexity of any sequence of $m$ operations simply by summing up the number of credits of the operations in such a sequence. This is an upper bound, not the exact worst case time complexity because we typically only show that the number of credits is enough — there may be unused credits left over after the sequence has been processed. If we choose the credit scheme just right, this upper bound will be a tight one, i.e., there will be sequences in which there are no credits (or, at least, not many credits) left over.

Sometimes when we do an analysis based on the banker’s view, it is convenient to think of the credits as being stored in the data structure itself. Of course this is a fictional device to help the analysis. When we actually implement the data structure and its associated operations there is no need to store any credits!

Let’s apply this methodology to our stack problem again. Our credit scheme will be to allocate 2 credits to each Push operation and 0 credits to each Pop($k$) operation. We claim that in this way we can “pay” for processing the entire sequence of any $m$ operations using just the credits allotted to the operations of the sequence. To prove this we argue as follows: A Push operation requires a constant amount of time — let’s say 1 unit. We then use one of the two credits to pay for the Push and save up the remaining one credit. We think of the saved credit as being stored with the
element that was pushed onto the stack. Now a Pop\((k)\) operation requires \(k\) time units. We can pay for this by using the \(k\) credits that are stored with the \(k\) elements that are being popped. (Note the similarity of this argument to the observation we made in our informal analysis, namely that an element can’t be popped unless it was previously pushed.)

We have thus proved that the credit scheme is adequate, i.e., allocates enough credits to permit the processing of any sequence of operations. Hence, we can obtain an upper bound on \(C(m)\) by adding up the number of credits that a sequence of \(m\) operations can have. Since each operation has at most 2 credits, we get that \(C(m) \leq 2m \in O(m)\). As we argued before, it is clear that \(C(m) \in \Omega(m)\), so \(C(m) \in \Theta(m)\), and \(A(m) \in \Theta(1)\). Thus, this analysis based on the banker’s view of amortisation, has also led to a tight bound on \(C(m)\) and \(A(m)\).

### Size of data structure

Sometimes, we need a more refined measure of amortised complexity than the function \(C(m)\) provides. In particular, we want a measure that also takes into account the maximum “size” of the data structure when the sequence is processed.† In that case we can define the sequence complexity as a function \(C : \mathbb{N} \times \mathbb{N} \times \mathcal{S} \to \mathbb{N}\) where

\[
C(m, n, S) \overset{\text{def}}{=} \max \left\{ T(\sigma, S) : \sigma \text{ is a sequence of } m \text{ operations and } n \text{ is the maximum size of the data structure during the processing of } \sigma \right\}
\]

The amortised complexity is defined by \(A(m, n, S) \overset{\text{def}}{=} \frac{C(m, n, S)}{m}\). As before, if \(S\) is the default initial state, we shall write \(C(m, n)\) instead of \(C(m, n, S)\), and \(A(m, n)\) instead of \(A(m, n, S)\).

In general, the maximum size of the data structure in the worst case is a function of the size of the initial data structure and the number of operations in the sequence. For instance, in the list example, the size of the data structure in a sequence of \(m\) operations is at most \(n_0 + m\), where \(n_0\) is the number of elements in the initial list. However, in this new definition of amortised complexity we are free to view the maximum size of the data structure as an independent parameter. This allows us to get better bounds on the amortised complexity of sequences of operations if we have more information about the sequences of operations we are interested in.

An illustration of this can be given using our list example. Suppose that we are interested in sequences of \(m\) operations where the maximum size of the list is \(n\), where \(n\) is viewed as an independent parameter. Assuming, as usual, that the initial data structure is empty we have that \(n \leq m\). Since the list has at most \(n\) elements, any one operation will take time at most (proportional to) \(n\); thus \(C(m, n) \in O(mn)\). Now consider the following sequence of \(m\) operations: first we have \(n\) Insert operations of distinct elements, resulting in a list of size \(n\); the remaining \(m-n\) operations are Searches for the last element in the list. This sequence of operations requires time \(n(n+1)/2 + (m-n)n = mn - n^2/2 + n/2\). Since \(n \leq m\), this is in \(\Omega(mn)\). Therefore, \(C(m, n) \in \Theta(mn)\).

Since \(n \leq m\), \(C(m, n)\) is maximised when \(m = n\), in which case we have \(C(m, m) \in \Theta(m^2)\) which is the same bound as we had when we did not treat \(n\) as an independent parameter. However, if we have some information about the sequences of operations we are interested in, we can get better bounds for the amortised complexity of these sequences. For instance, suppose we somehow know that we are interested only in sequences in which the number of Insert operations is no more than the square root of the total number of operations. In particular, this implies that \(n \leq \sqrt{m}\).

† For each problem there is some reasonable measure of data structure size; for instance, in our list (or stack) example a natural choice might be the number of elements in the list (or stack).
Therefore, the amortised complexity of such sequences is $C(m, \sqrt{m}) \in \Theta(m\sqrt{m})$ — a better bound than $\Theta(m^2)$ which is what we obtained when we did not treat the size of the data structure as an independent parameter in our analysis.

In the stack example, treating the size of the data structure as an independent parameter does not provide a more detailed measure. In other words, in that example, $C(m, n) = C(m)$ for all $n \in \mathbb{N}$. 
LEFTIST TREES

Leftist trees are a data structure for representing priority queues. They were invented by Clark Crane and have the following nice properties:

- **Insert** and **DeleteMin** operations can be performed using time in $O(\log n)$ in the worst case — as with heaps.

- In addition, we can merge two leftist trees with $n_1$ and $n_2$ nodes respectively into a single leftist tree using time in $O(\log(\max(n_1, n_2)))$ in the worst case.

**Definition.** The distance of a node $u$ in a tree, denoted $\text{dist}(u)$, is the length of the shortest path from $u$ down to a descendant node that has at most one child.

**Definition.** A leftist tree is a binary tree such that for every node $u$,

(a) $\text{key}(u) \leq \min(\text{key}(l\text{child}(u)), \text{key}(r\text{child}(u)))$, and

(b) $\text{dist}(l\text{child}(u)) \geq \text{dist}(r\text{child}(u))$.

In the above definition, $\text{key}(u)$ is the key stored at node $u$. We assume that $\text{key}(\Lambda) = \infty$ and $\text{dist}(\Lambda) = -1$.

**Definition.** The right path of a tree is the path $u_1, u_2, \ldots, u_k$, where $u_1$ is the root of the tree, $u_{i+1} = r\text{child}(u_i)$ for $1 \leq i < k$, and $r\text{child}(u_k) = \Lambda$.

Two leftist trees are shown in Figure 1.

Here are a few simple properties of leftist trees that you should be able to prove easily:

**Fact 1.** The left and right subtrees of a leftist tree are leftist trees.

**Fact 2.** The distance of a leftist tree’s root is equal to the length of the tree’s right path.

**Fact 3.** For any node $u$ of a leftist tree, $\text{dist}(u) = \text{dist}(r\text{child}(u)) + 1$.

Tree $T_2$ in Figure 1 illustrates the fact that leftist trees can be unbalanced. However, in **Insert**, **DeleteMin** and **Merge**, all activity takes place along the right path which, the following theorem shows, is short.

**Theorem.** Let $T$ be a nonempty leftist tree, $k$ be the length of its right path, and $n$ is the number of its nodes. Then $n \geq 2^{k+1} - 1$.

**Proof:** By (course-of-values) induction on the height $h$ of $T$. Since $T$ is nonempty, $n \geq 1$ and $h \geq 0$. Suppose the theorem holds for all leftist trees of height less than $h$.

**Case 1:** $k = 0$. We have $n \geq 1 = 2^1 - 1$. So, in this case $n \geq 2^{k+1} - 1$, as wanted.

**Case 2:** $k > 0$. Let $T_L$, $T_R$ be the left and right subtrees of $T$; $n_L$, $n_R$ be the number of nodes in $T_L$ and $T_R$; and $k_L$, $k_R$ be the lengths of the right paths in $T_L$ and $T_R$ respectively. Since $T_L$ and $T_R$ are the subtrees of $T$, they are both leftist trees (by Fact 1), and have height less than $h$. Moreover, since $k > 0$, $T_R$ is nonempty, and since $T$ is a leftist tree so is $T_L$. Therefore, the induction hypothesis applies to $T_L$ and $T_R$, and we have $n_L \geq 2^{k_L+1} - 1$ and $n_R \geq 2^{k_R+1} - 1$. By Fact 3, $k_R = k - 1$. By the definition of leftist tree and Fact 2, $k_L \geq k_R$, and so $k_L \geq k - 1$. We have, $n = n_L + n_R + 1 \geq (2^{k_L+1} - 1) + (2^{k_R+1} - 1) + 1 \geq (2^k - 1) + (2^k - 1) + 1 = 2^{k+1} - 1$, as wanted.

From this we immediately obtain
Two leftist trees. In each node we record its key at the top half and its distance at the bottom half. The right path of $T_1$ is 1, 5, 8 while the right path of $T_2$ is 1.

\textbf{Figure 1}

\textbf{Corollary.} The right path of a leftist tree with $n$ nodes has length at most $\lceil \log_2 (n + 1) \rceil - 1$.

Now let’s examine the algorithm for merging two leftist trees. The idea is simple: if one of the two trees is empty we’re done; otherwise we want to merge two non-empty trees $T_1$ and $T_2$, and we can assume, without loss of generality, that the key in the root of $T_1$ is less than or equal to the key in the root of $T_2$. Recursively we merge $T_2$ with the right subtree of $T_1$, and we make the resulting leftist tree into the right subtree of $T_1$. If this has made the distance of the right subtree’s root longer than the distance of the left subtree’s root, we simply interchange the left and right children of $T_1$’s root (thereby making what used to be the right subtree of $T_1$ into its left subtree and vice-versa). Finally, we update the distance of $T_1$’s root. The pseudocode below gives more details.

We assume that each node of the leftist tree is represented as a record with the following format,

\begin{tabular}{|c|c|c|c|}
\hline
lchild & key & dist & rchild \\
\hline
\end{tabular}

where the fields have the obvious meanings. A leftist tree is specified by giving a pointer to its root. The following algorithm merges two leftist trees whose roots are pointed at by $r_1$ and $r_2$, and returns a pointer to the root of the resulting leftist tree.\footnote{We use “$r_1 \leftrightarrow r_2$” as an abbreviation for “$\text{temp} := r_1; r_1 := r_2; r_2 := \text{temp}$.”} (See Figure 2 for an example of merging.
two leftist trees.)

\begin{verbatim}
MERGE(r_1, r_2)
    if r_1 = \Lambda then return r_2
    elsif r_2 = \Lambda then return r_1
    else
        if key(r_1) > key(r_2) then r_1 \leftrightarrow r_2 end if
        rchil d(r_1) := MERGE(rchil d(r_1), r_2)
        if rchil d(r_1) \neq \Lambda and (lchil d(r_1) = \Lambda or dist(rchil d(r_1)) > dist(lchil d(r_1))) then
            rchil d(r_1) \leftrightarrow lchil d(r_1)
        end if
        if rchil d(r_1) = \Lambda then dist(r_1) := 0
        else dist(r_1) := dist(rchil d(r_1)) + 1
        end if
        return r_1
    end if
end MERGE
\end{verbatim}

What is the complexity of this algorithm? First, observe that there is a constant number of steps that must be executed before and after each recursive call to \textsc{Merge}. Thus the complexity of the algorithm is proportional to the number of recursive calls to \textsc{Merge}. It is easy to see that, in the worst case, this will be equal to \(p_1 + p_2\), where \(p_1\) (respectively, \(p_2\)) is 1 plus the length of the right path of the leftist tree whose root is pointed at by \(r_1\) (respectively, \(r_2\)). Let the number of nodes in these trees be \(n_1\), and \(n_2\). By the above corollary we have \(p_1 \leq \lceil \log(n_1 + 1) \rceil\), and \(p_2 \leq \lceil \log(n_2 + 1) \rceil\). Thus \(p_1 + p_2 \leq \log(n_1 + \log n_2 + 2)\). Let \(n = \max(n_1, n_2)\). Then \(p_1 + p_2 \leq 2 \log n + 2\). Therefore, \textsc{Merge} is called at most \(2 \log n + 2\) times, and the complexity of the algorithm is \(O(\log (\max(n_1, n_2)))\) in the worst case.

Using the \textsc{Merge} algorithm we can easily write algorithms for \textsc{Insert} and \textsc{DeleteMin}:

\begin{verbatim}
INSERT(e, r) { e is an element, r is pointer to root of tree}
1. Let \(r'\) be a pointer to the leftist tree containing only \(e\).
2. MERGE(r', r).
DELETEMIN(r)
1. \(min := \) element stored at \(r\) (root of leftist tree);
2. \(r := \) MERGE(lchil d(r), rchil d(r));
3. \textbf{return} min.
\end{verbatim}

By our analysis of the worst case time complexity of \textsc{Merge}, it follows immediately that the complexity of both these algorithms is \(O(\log n)\) in the worst case, where \(n\) is the number of nodes in the leftist tree.

The \textsc{Insert} operation can also be implemented as in the heap representation of priority queues, by adding the new node at the end of the right path, percolating its value up (if necessary), and switching right and left children of some node (if necessary) to maintain the properties of leftist tree after the insertion. As an exercise, write the \textsc{Insert} algorithm for leftist trees in this fashion. On the other hand, we cannot use the idea of percolating values down to implement \textsc{DeleteMin} in leftist trees the way we did in heaps: doing so would result in an algorithm with \(\Theta(n)\) worst case complexity. As an exercise, construct a leftist tree where this worst case behaviour would occur.
Also, it is possible to implement the **merge** operation in an iterative, rather than recursive, way. Roughly speaking, the idea is this: We traverse the right paths of the two leftist trees we wish to merge, merging them into a single right path, called the *merge path*. The nodes of the two right paths are added to the merge path in increasing key order. As a node is added to the merge path, we also “hang” its left subtree on the left side of the merge path (which, recall, is a right path).

When we have completed the merging of the two right paths in this way, we will have constructed a binary tree containing the nodes of the two trees, and satisfying the heap-order property. However, the tree is not necessarily a leftist tree because for some nodes on the merge path it may be that \( \text{dist(\text{left child}(u))} < \text{dist(\text{right child}(u))} \). So, we traverse the merge path once again but now in *bottom-up* fashion, exchanging the left and right subtrees of nodes when necessary, and updating the \( \text{dist} \) field of each node appropriately, so that at the end we have a leftist tree with the keys of the two trees that were merged. As an exercise, you should write the pseudocode for this algorithm, analyse its complexity, and prove that it produces *exactly* the same leftist tree as the recursive implementation of **merge**.

**Figure 2:** Merging two leftist trees
Suppose we have \( n \) keys, \( 1, \ldots, n \), stored in a list, and we are processing requests to search for various keys. Let \( p_i \) denote the probability of a request being for key \( i \). We assume that \( p_i > 0 \) — i.e., every item is requested with non-zero probability — and that \( \sum_{i=1}^{n} p_i = 1 \). The expected number of comparisons to find a key in the list is simply \( \sum_{i=1}^{n} p_i \cdot (\text{position of } i \text{ in the list}) \). It is not difficult to show that this quantity is minimised when the keys appear in the list in non-increasing probability of being requested. Assuming, without loss of generality, that the keys are numbered so that \( p_1 \geq p_2 \geq \ldots \geq p_n \), the minimum expected number of comparisons to find a key in a static list is \( EC_{\text{opt}} = \sum_{i=1}^{n} i \cdot p_i \).

If we know the probabilities \( p_i \) we can store the keys in the list in this optimal order. If these probabilities are unknown, we can still do something intelligent! We can allow the position of a key to change over time in response to the request pattern. One way to do this is by using the so-called move-to-front policy. According to this policy, every time we find a key we move it from its present position to the front of the list. The idea is that keys that are requested frequently will tend be near the beginning of the list, while infrequently requested keys will tend to be near the end of the list — that is, most of the time, we would expect to find an item near its optimal position.

This sounds like a neat idea, but how do we know that it really works? We shall show that, no matter what the order of the keys in the initial list, after a large number of searches, the list managed using the move-to-front policy is actually quite good. In particular, we shall show that the expected number of comparisons needed to find a key in it is no more than twice the expected number of comparisons to find a key in the optimal list! What is remarkable about this result is the fact that to implement the move-to-front policy we need not know the probabilities of searching for the various keys — while to arrange the keys in optimal order, this knowledge is necessary.

We begin by setting up our probability space. We are given an initial list containing the \( n \) keys, and we are also given a stream of \( m \) requests, for some large \( m \), where each request is to search for some key in the list. Thus, we can view each sample point as a sequence of \( m \) requests, where each request is a key — i.e., a number between \( 1 \) and \( n \). Since, in general, \( m \gg n \), the sequence can contain several requests for the same key. We denote such a sequence as \( \vec{r} = r_1, r_2, \ldots, r_m \), where \( r_i \) is the key searched in the \( i \)th request. The sample space \( \Omega \) consists of all such sequences. The probability distribution \( P \) is defined as \( P(\vec{r}) = p_{r_1} \cdot p_{r_2} \cdot \ldots \cdot p_{r_m} \). (You should verify that this satisfies the axioms of probability distributions; in particular, \( \sum_{\vec{r} \in \Omega} P(\vec{r}) = 1 \).)

Consider a sample point \( \vec{r} \). Suppose we process the requests of \( \vec{r} \) one at a time starting with the original list, and applying the move-to-front policy. That is, every time we search for a key, we move it to the front of the list. After processing all the requests, the keys will appear in some new order in the list, depending on the original order and the sequence of requests. We shall refer to this as the list after \( \vec{r} \).

For our analysis we need to define some random variables and events. For every key \( i \) we have a random variable \( \text{position}_i(\vec{r}) \), defined by \( \text{position}_i(\vec{r}) = \text{position of key } i \text{ in the list after } \vec{r} \). For every pair of keys \( i \) and \( j \) we define the event \( \{ j \text{ precedes } i \} = \{ \vec{r} : j \text{ precedes } i \text{ in the list after } \vec{r} \} \). Finally, we define a random variable \( \text{comparisons}(\vec{r}) \) as the expected number of comparisons needed to find a key in the list after \( \vec{r} \). That is, \( \text{comparisons}(\vec{r}) = \sum_{i=1}^{n} p_i \cdot \text{position}_i(\vec{r}) \). We are interested in the expected value of this random variable. We shall show that as \( m \) gets larger and larger, \( E(\text{comparisons}) \) tends to a value so that \( E(\text{comparisons}) \leq 2 \cdot EC_{\text{opt}} \). We need the following facts.
1. \( E(\text{comparisons}) = \sum_i p_i \cdot E(\text{position}_i) \).

2. \( E(\text{position}_i) = 1 + \sum_{j \neq i} P([j \text{ precedes } i]) \).

3. As \( m \to \infty \), \( P([j \text{ precedes } i]) \to p_j/(p_i + p_j) \).

**Proof of 1.** Using the definition of expected value, and distributivity of multiplication over addition we have

\[
E(\text{comparisons}) = \sum_{\vec{r} \in \Omega} P(\vec{r}) \cdot \text{comparisons}(\vec{r})
\]

\[
= \sum_{\vec{r} \in \Omega} P(\vec{r}) \cdot \left( \sum_{i=1}^{n} p_i \cdot \text{position}_i(\vec{r}) \right)
\]

\[
= \sum_{\vec{r} \in \Omega} \sum_{i=1}^{n} p_i \cdot P(\vec{r}) \cdot \text{position}_i(\vec{r})
\]

\[
= \sum_{i=1}^{n} \sum_{\vec{r} \in \Omega} p_i \cdot P(\vec{r}) \cdot \text{position}_i(\vec{r})
\]

\[
= \sum_{i=1}^{n} p_i \cdot \left( \sum_{\vec{r} \in \Omega} P(\vec{r}) \cdot \text{position}_i(\vec{r}) \right)
\]

\[
= \sum_{i=1}^{n} p_i \cdot E(\text{position}_i)
\]

**Proof of 2.** Let \( \text{precedes}_{ji} \) be the indicator random variable of event \([j \text{ precedes } i]\); that is,

\[
\text{precedes}_{ji}(\vec{r}) = \begin{cases} 
1, & \text{if } j \text{ precedes } i \text{ in the list after } \vec{r} \\
0, & \text{otherwise}. 
\end{cases}
\]

For any \( \vec{r} \in \Omega \), \( \text{position}_i(\vec{r}) = 1 + \sum_{j \neq i} \text{precedes}_{ji} \). Thus,

\[
E(\text{position}_i) = E(1 + \sum_{j \neq i} \text{precedes}_{ji})
\]

\[
= 1 + \sum_{j \neq i} E(\text{precedes}_{ji}) \quad \text{[by linearity of expectation]}
\]

\[
= 1 + \sum_{j \neq i} \left( 1 \cdot P([\text{precedes}_{ji} = 1]) + 0 \cdot P([\text{precedes}_{ji} = 0]) \right)
\]

\[
= 1 + \sum_{j \neq i} P([\text{precedes}_{ji} = 1])
\]

\[
= 1 + \sum_{j \neq i} P([j \text{ precedes } i])
\]

**Proof of 3.** Consider the event \([j \text{ precedes } i]\). Key \( j \) precedes key \( i \) in the list that results after processing the sequence of requests if and only if one of two following disjoint cases holds:

- there was no request for either \( i \) or \( j \), and \( j \) was before \( i \) in the original list — let this be event \( A \), or
- there was a request for \( i \) or \( j \) (or both), and there was no request for \( i \) after the last request for \( j \) (if any) — let this be event \( B \).
Thus, \( P([j \text{ precedes } i]) = P(A) + P(B) \). Since all keys are requested with non-zero probability, as \( m \) gets large, \( P(A) \) tends to 0. Thus, after a large number of requests, i.e., as \( m \to \infty \), \( P([j \text{ precedes } i]) \to P(B) \).

Event \( B \) is the disjoint union of \( m \) events \( B_1, B_2, \ldots, B_m \), where \( B_k \) is the event that the last \( k-1 \) requests were for keys other than \( i \) or \( j \), and the \( k \)th to last request was for key \( j \). (In this way, \( j \) was moved to the front more recently than \( i \) and therefore precedes \( i \) at the end of the request sequence.) Thus, \( P(B) = \sum_{k=1}^{m} P(B_k) \).

But \( P(B_k) = p_j (1 - (p_i + p_j))^{k-1} \), where the first term is the probability that the \( k \)th to last request was for \( p_j \), and the second term is the probability that the last \( k-1 \) requests were for keys other than \( i \) or \( j \). Therefore, \( P(B) = \sum_{k=1}^{m} p_j (1 - (p_i + p_j))^{k-1} = p_j \cdot \sum_{k=1}^{m} (1 - (p_i + p_j))^{k-1} \). As \( m \to \infty \), the sum is an infinite geometric series, and we get \( P(B) \to p_j / (p_i + p_j) \). Recalling that, as \( m \to \infty \), we also have that \( P([j \text{ precedes } i]) \to p_j / (p_i + p_j) \).

We have now proved the three facts we need. The desired result follows with just a bit of algebraic manipulation:

\[
E(\text{comparisons}) = \sum_{i=1}^{n} p_i \cdot E(\text{position}_i) \quad \text{[by (1)]}
\]

\[
= \sum_{i=1}^{n} p_i \cdot (1 + \sum_{1 \leq j \leq n, i \neq j} P([j \text{ precedes } i])) \quad \text{[by (2)]}
\]

\[
\to \sum_{i=1}^{n} p_i \cdot (1 + \sum_{1 \leq j \leq n, i \neq j} \frac{p_j}{p_i + p_j}) \quad \text{[by (3)]}
\]

\[
= \sum_{i=1}^{n} p_i + \sum_{i=1}^{n} \sum_{1 \leq j \leq n, i \neq j} \frac{p_i p_j}{p_i + p_j}
\]

\[
= \sum_{i=1}^{n} p_i + 2 \sum_{i=1}^{n} \sum_{j=1}^{i-1} \frac{p_i p_j}{p_i + p_j}
\]

\[
= \sum_{i=1}^{n} p_i + 2 \sum_{i=1}^{n} \sum_{j=1}^{i-1} \frac{p_i}{p_i + p_j}
\]

\[
\leq \sum_{i=1}^{n} p_i + 2 \sum_{i=1}^{n} p_i \cdot (i - 1) \quad \text{[because } p_j / (p_i + p_j) \leq 1 \text{]}
\]

\[
\leq 2 \sum_{i=1}^{n} i \cdot p_i = 2 \cdot E C_{opt}, \quad \text{as wanted.}
\]
SELF-ORGANISING LISTS
Amortised Analysis of the Move-to-Front Policy

Suppose we have a list of \( n \) keys, against which we want to process an arbitrary sequence \( \sigma \) of search operations, each for some key in the list. If we know the sequence \( \sigma \) ahead of time and we confine ourselves to a static list (i.e., a list in which we are not allowed to change the order of keys as we perform search operations) then obviously the best way to arrange the list in decreasing frequency order: That is, the key most frequently mentioned in \( \sigma \) is kept first, the next most frequently mentioned key is kept second, and so forth.

Now, let us drop the unrealistic assumption of foreknowledge of \( \sigma \) and let us suppose that we will process the sequence of searches against the list using the move-to-front policy. That is, every time a key is found it is moved to the front of the list. (Note that now the list is not static; in general, the order of keys changes as searches are performed.) We are interested to know how much worse the move-to-front will perform, as compared to the optimal arrangement of the keys (in the hypothetical case that we know the sequence ahead of time). One might suspect that foreknowledge of the future would give a tremendous edge and that the move-to-front policy might do arbitrarily worse for some sequence of search operations. It turns out that this is not the case! In particular we shall show that (as the number of searches becomes large), the number of comparisons required by the move-to-front policy is no more than twice that required by the optimal static list.

Let us define some notation to help our discussion. Let \( L \) be any list of \( n \) keys, and \( \sigma \) be a sequence of \( m \) search operations for keys in \( L \). Let \( T_{\text{static}}(\sigma, L) \) be the time needed to process \( \sigma \), using \( L \) as a static list. By “time” we mean the sum of the number of comparisons made over all searches in \( \sigma \). Let \( T_{\text{mtf}}(\sigma, L) \) be the time needed to process \( \sigma \) using the move-to-front policy, starting with \( L \) as our initial list. Again, in the time complexity we only account for comparisons. In particular, we do not account for the cost of moving a key to the front of the list. This is not a big deal since (assuming the list is represented as a linked list), this takes only a constant amount of work and therefore the cost of a search is dominated by the effort to find the key; the cost of moving the key once it is found is negligible.

We shall prove that if \( L \) and \( L' \) are arbitrary lists containing the same \( n \) keys then \( T_{\text{mtf}}(L', \sigma) < 2 \cdot T_{\text{static}}(L, \sigma) + n(n - 1)/2 \). Note that the last term does not depend on the number of operations \( m \) in the sequence \( \sigma \). Thus, as \( m \to \infty \), we can ignore that term (because it will be very small compared to \( T_{\text{static}}(L, \sigma) \)). Thus, we have that as \( m \to \infty \), \( T_{\text{mtf}}(L', \sigma) < 2 \cdot T_{\text{static}}(L, \sigma) \). In particular, if we take \( L \) to be the optimal static list for \( \sigma \) (described above), this says that (as the number of searches becomes large relative to the number of keys \( n \) in the list) no matter what the initial list is, move-to-front will do no worse than twice the best we could possibly have done with a static list even if we knew the sequence of searches in advance in choosing the static list! This is another very impressive result about the move-to-front policy. It shows that move-to-front is competitive to the power of foreknowledge!

Note that this result is, in some sense, stronger than the similar probabilistic result that we proved about the move-to-front policy, where we compared its expected cost (under any probability distribution) to the expected cost of the optimal list (under the same probability distribution). The foregoing result holds about any sequence of searches whatsoever, not about the “average” sequence of searches (which is, in effect, what the probabilistic result was about).

Let \( L = x_1, x_2, \ldots, x_n \) be the static list (the subscripts of the \( n \) keys denote their position in that list). Let \( \sigma = a_1, a_2, \ldots, a_m \) be a sequence of \( m \) search operations for elements in \( L \). Now consider any list \( L' \) containing the same elements as \( L \), possibly in a different order. If we process
the sequence of operations $\sigma$ using the move-to-front policy starting with list $L'$, we get a sequence of lists

$$L_0 \xrightarrow{a_1} L_1 \xrightarrow{a_2} \ldots \xrightarrow{a_m} L_m$$

where $L_0 = L'$, and $L_s$ is obtained from $L_{s-1}$ by moving the key found in search $a_s$ from where it was in $L_{s-1}$ to the front, for each $s, 1 \leq s \leq m$. An inversion in list $L_s$ relative to list $L$ is a pair of elements that are in different order in $L_s$ than they are in $L$, our static list.

Now we define the potential of list $L_s$ as

$$\Phi(L_s) = \text{number of inversions in } L_s \text{ relative to } L$$

To motivate this definition, note that in this case we are not interested in an absolute analysis of move-to-front, but rather a relative analysis, comparing its performance to that of the static list $L$. When we consider the state of the list $L_s$ at some point during the processing using move-to-front, the closer $L_s$ is to the static list $L$, the better move-to-front will fare in the comparison. The number of inversions in $L_s$ relative to $L$ is precisely a measure of how “close” $L_s$ is to $L$.

Recall that the amortised complexity of the $s$th operation $a_s$ is $u_s \overset{\text{def}}{=} t_s + \Phi(L_s) - \Phi(L_{s-1})$. For example, suppose that $L_0 = L$, i.e., the initial state of the list when we use the move-to-front policy is the same as the static list. If the first operation $a_1$ is “search for $x_1$” (recall that $x_1$ is the first element in $L$) then the amortised time $u_1 = 1$, since the time to find $x_1$ is 1, and the state of the list after we move $x_1$ remains the same, so the difference in potential is 0. If, on the other hand, $a_1$ is “search for $x_n$”, then the amortised time $u_1 = n + (n - 1) - 0 = 2n - 1$. This is because the time $t_1$ to find $x_n$ is $n$; furthermore, after we move $x_n$ to the front of the list, the new list $L_1$ has $n - 1$ inversions (all other keys are out-of-order with respect to $x_n$ relative to their position in $L$), while the old list $L_0$ has 0 inversions relative to $L$ (since $L_0 = L$).

Now, let us consider the $s$th operation $a_s$, and suppose that this is the operation “search for $x_i$”. We shall derive an expression for $u_s$, the amortised time of $a_s$. Consider the difference in potential $\Delta \Phi = \Phi(L_s) - \Phi(L_{s-1})$ as a result of this operation. The figure below illustrates the state of the list before and after this operation.

```
position p

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>$x_i$</th>
<th>B</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>L_{s-1}</th>
<th>x_i</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>L_s</th>
<th>A</th>
<th>$x_i$</th>
<th>B</th>
</tr>
</thead>
</table>
```

The only differences in potential between $L_{s-1}$ and $L_s$ are due to inversions that involve $x_i$; all other pairs of keys are in the same order in $L_{s-1}$ as in $L_s$ and do not contribute to any change in the potential. Furthermore, keys that follow $x_i$ in $L_{s-1}$ (i.e., keys in the part of the list marked $B$ in the figure above) are in the same order relative to $x_i$ in $L_s$ as in $L_{s-1}$ so they do not contribute anything to the difference in potential. Thus, to calculate $\Delta \Phi$ it is enough to focus our attention to the keys that precede $x_i$ in $L_{s-1}$, i.e., those in the part of the list marked $A$ in the figure above.

Let $p$ denote the position of $x_i$ in $L_{s-1}$ and $b$ denote the number of elements that precede $x_i$ in both $L_{s-1}$ and $L$ (i.e., the keys $x_j$ with $j < i$ in part $A$ of the list, in the figure above). By
definition, these \( b \) elements are in the same order with respect to \( x_i \) in \( L_{s-1} \) as in \( L \); since \( x_i \) is moved to the front in \( L_s \), these elements are out-of-order with respect to \( x_i \) in \( L_s \) relative to \( L \). Thus operation \( a_s \) creates \( b \) new inversions that did not exist before it. However, the movement of \( x_i \) did not only create new inversions: it also destroyed some inversions that existed in \( L_{s-1} \). In particular, there are \( p - 1 - b \) elements in part \( A \) of the list that are out-of-order with respect to \( x_i \) in \( L_{s-1} \) relative to \( L \), and these are in the same order with respect to \( x_i \) in \( L_s \) as in \( L \). Thus, there are \( p - 1 - b \) inversions that used to exist and which are destroyed after operation \( a_s \) as a result of moving \( x_i \) to the front. Since \( a_s \) created \( b \) and destroyed \( p - 1 - b \) inversions with respect to \( x_i \) and since it did not affect any inversions not involving \( x_i \), we have \( \Delta \Phi = b - (p - 1 - b) = 2b + 1 - p \). So, \( u_s = t_s + \Delta \Phi = p + 2b + 1 - p = 2b + 1 \). Since \( b \) is the number of elements that precede \( x_i \) in both \( L_{s-1} \) and \( L \), and since there are \( i - 1 \) elements that precede \( x_i \) in \( L \), we have that \( b \leq i - 1 \). Thus, \( u_s \leq 2i - 1 \). The time required to search for \( x_i \) in static list \( L \), \( T_{\text{static}}(a_s, L) \), is precisely \( i \). Therefore,

\[
    u_s < 2 \cdot T_{\text{static}}(a_s, L)
\]

Recall from our general discussion of the physicist’s view of amortisation that

\[
    T_{\text{mtf}}(\sigma, I_0) = \left( \sum_{s=1}^{m} u_s \right) + \Phi(I_0) - \Phi(I_m)
\]

The potential of any list is non-negative and the maximum possible potential of any list is \( \binom{n}{2} = \frac{n(n - 1)}{2} \), which occurs in the list in which any pair of keys is in a different order than in \( L \). Plugging these and (1) into (2), and recalling that \( I_0 = L' \), we get

\[
    T_{\text{mtf}}(\sigma, L') < 2 \cdot T_{\text{static}}(\sigma, L) + \frac{n(n - 1)}{2},
\]

as wanted.
AVERAGE CASE ANALYSIS FOR BINARY SEARCH TREES

The worst case number of comparisons to locate a key in a binary search tree (BST) is $O(n)$ where $n$ is the number of keys in the tree. But what about the average case? This is the question we investigate in this handout. To keep things simple, assume the following scenario: $n$ keys are inserted in a BST in some “random” order (resulting in a tree of “random” shape). Then we perform searches on that tree.† Before making this more precise, we introduce some terminology and prove some facts.

Consider a BST $T$. We define a new tree, called the augmented $T$ by adding one or two children to each node of $T$ that has one or no children respectively (see Figure 1). In the special case where $T$ is the empty tree (with 0 nodes), the augmented $T$ is, by definition, also empty. The added nodes are called external nodes and are drawn as boxes — in distinction from the normal nodes that are drawn as circles and are called internal nodes. Internal nodes contain the keys stored in the BST. External nodes represent the intervals “between the keys”. For example, in Figure 2, the external node marked “II” represents the interval of keys $> 8$ but $< 13$. Note that the external nodes are just a crutch to help us in our analysis. They are not part of the BST data structure and are not represented in the computer.

![Figure 1]

**Lemma 1.** If an augmented BST has $n$ internal nodes then it has $n + 1$ external nodes.

**Proof:** $n$ keys define $n + 1$ intervals “between the keys”. Since each external node corresponds to exactly one of these intervals, there must be $n + 1$ such nodes.‡

Let $T$ be an augmented BST. Define the internal path length of $T$, $IPL(T)$, to be the sum of the depths of all the internal nodes of the tree; and the external path length of $T$, $EPL(T)$, to be the sum of the depths of the external nodes of the tree. In the special case where $T$ is empty we define $EPL(T) = IPL(T) = 0$.

† This is not the most general scenario possible in that we do not consider deletions and all searches are performed at the end. Unfortunately there is no conclusive analysis that considers the most general possible scenario. Deletions complicate things enormously and only partial results have been obtained for that case. If you are really ambitious, see Knuth’s *Sorting and Searching*, pp. 428–431 for further discussion on this issue.

‡ A more rigorous proof can be given as follows: First observe that the augmented BST is a strict binary tree, where the external nodes are the leaves. Then prove by induction on $n$ that a strict binary tree with $n$ internal nodes has $n + 1$ leaves. Try it!
Thus in the tree of Figure 1, $IPL(T) = 6$ and $EPL(T) = 16$. The next lemma shows a general relationship that exists between the internal and external path length in any augmented BST.

**Lemma 2.** Let $T$ be any augmented BST with $n$ internal nodes. Then $EPL(T) = IPL(T) + 2n$.

**Proof:** By induction on $n$.

**Basis:** If $n = 0$ then $T$ is empty. By definition, $EPL(T) = IPL(T) = 0$ and, indeed, $EPL(T) = IPL(T) + 2n$.

**Induction Step:** Suppose the lemma holds for augmented BSTs with fewer than $n$ internal nodes, for any $n > 0$. We’ll show that it holds for an augmented BST $T$ with exactly $n$ internal nodes. Let $T_1, T_2$ be the left and right subtrees of $T$ respectively, and let $n_1, n_2$ be the number of internal nodes of $T_1, T_2$. Clearly, $T_1$ and $T_2$ are augmented BSTs, each having $< n$ internal nodes, so by induction hypothesis:

\[ EPL(T_1) = IPL(T_1) + 2n_1 \quad \text{and} \quad EPL(T_2) = IPL(T_2) + 2n_2. \]  

(a)

The external nodes of $T$ are precisely the external nodes of $T_1$ and of $T_2$. However, the depth of each such node in $T$ is one more than the depth of the same node in $T_1$ or $T_2$ (whichever of the two it happens to be in). By Lemma 1, the number of external nodes in $T_i$ is $n_i + 1$, for $i = 1, 2$; therefore,

\[ EPL(T) = EPL(T_1) + EPL(T_2) + n_1 + n_2 + 2. \]  

(b)

Similarly, the internal nodes of $T$ are the internal nodes of $T_1$ and $T_2$, and the root of $T$. The root of $T$ has depth 0 and therefore contributes 0 to $IPL(T)$. The depth of every other internal node in $T$ is one more than that of the corresponding node in $T_1$ or $T_2$. Thus,

\[ IPL(T) = IPL(T_1) + IPL(T_2) + n_1 + n_2. \]  

(c)

Also,

\[ n = n_1 + n_2 + 1. \]  

(d)

Therefore,

\[ EPL(T) = EPL(T_1) + EPL(T_2) + n_1 + n_2 + 2 \quad \text{[by (b)]} \]

\[ = (IPL(T_1) + 2n_1) + (IPL(T_2) + 2n_2) + n_1 + n_2 + 2 \quad \text{[by (a)]} \]

\[ = (IPL(T_1) + IPL(T_2) + n_1 + n_2) + 2(n_1 + n_2 + 1) \]

\[ = IPL(T) + 2n \quad \text{[by (c) and (d)]} \]

as wanted. \qed
The Probability Space

We are interested in studying the average search time for successful and unsuccessful searches in a BST with $n$ nodes. To do this, we define the sample space to be the set of possible sequences of key insertions into the BST, and we define a probability distribution function on this sample space.

Let $x_i$ be the $i$-th key to be inserted into the BST. Let $\Omega_n$ be the sample space consisting of the $n!$ possible different key orderings among the first $n$ keys to be inserted. More precisely, we represent each $\omega \in \Omega_n$ as $\omega = \langle y_1, y_2, \ldots, y_n \rangle$, where $y_1 < y_2 < \cdots < y_n$ (is the relation that we use to compare the keys). Said differently, $\omega = \langle y_1, y_2, \ldots, y_n \rangle$ is a permutation of $x_1, x_2, \ldots, x_n$, so that $y_j = x_i$ means that the $i$-th key to be inserted into the BST is the $j$-th smallest key among the first $n$ keys to be inserted. The probability density function $P_n$ on $\Omega_n$ is the uniform density; that is, for each $\omega \in \Omega_n$, $P_n(\omega) = 1/n!$. This means that each of the $n!$ possible orderings of the $n$ keys to be inserted into the BST is equally likely. We have defined a probability space on key sequences of length $n$ for each positive value of $n$. For consistency, define $\Omega_0$ to consist of the empty sequence, which occurs with probability 1.

An important observation is that we can view choosing a random $\omega_n \in \Omega_n$ as a two step process:

(i) Randomly choose $\omega_{n-1} = \langle y_1, y_2, \ldots, y_{n-1} \rangle \in \Omega_{n-1}$.
(ii) Randomly choose $j$ from the set $\{1, 2, \ldots, n\}$ with uniform probability. The chosen $\omega_n \in \Omega_n$ is defined as $\omega_n = \langle y'_1, y'_2, \ldots, y'_n \rangle \in \Omega_n$, where

$$y'_i = \begin{cases} y_i, & \text{if } 1 \leq i < j \\ x_n, & \text{if } i = j \\ y_{i-1}, & \text{if } j < i \leq n \end{cases}$$

In words, this two-step process says: First randomly choose the relative ordering among the first $n - 1$ keys, then randomly insert the $n$-th key $x_n$ into this ordering, so that $x_n$ is equally likely to be the smallest, the second smallest, \ldots, the largest key among the $n$ keys.

Average Case Analysis of Binary Search Trees

For each $\omega \in \Omega_n$, let $T(\omega)$ be the augmented BST which is constructed by starting with the empty BST and using the insertion sequence $\omega$ to construct $T(\omega)$. For example, if $\omega = \langle x_2, x_3, x_1 \rangle$, the resulting $T(\omega)$ is shown in Figure 2.

For convenience, we shall write $IPL(\omega)$ instead of $IPL(T(\omega))$ (i.e. the internal path length of the augmented binary search that corresponds to $\omega$), and $EPL(\omega)$ instead of $EPL(T(\omega))$ (i.e. the external path length of the augmented binary search tree that corresponds to $\omega$). Define two random variables $S$ and $U$ on probability space $(\Omega_n, P_n)$ as follows:

(1) $S(\omega)$ is the expected number of comparisons to find a key in $T(\omega)$ when the search is equally likely to end successfully at each of the $n$ internal nodes of $T(\omega)$.

$$S(\omega) = \sum_{i=1}^{n} \frac{1}{n} \cdot (\text{depth}_{\omega}(I_i) + 1)$$

where $I_i$ is the internal node that contains $x_i$ and $\text{depth}_{\omega}(I_i)$ is the depth of that node in $T(\omega)$. (The number of comparisons to find a key is one more than the depth of the node where the key is stored.)
(2) $U(\omega)$ is the expected number of comparisons to look for a key in $T(\omega)$ when the search is equally likely to end unsuccessfully at each of the $n+1$ external nodes of $T(\omega)$. More precisely,
\[
U(\omega) = \sum_{i=1}^{n+1} \left( \frac{1}{(n+1)} \right) \cdot \text{depth}_{\omega}(E_i)
\]
where $E_i$ is the $i$-th external node of $T(\omega)$ and $\text{depth}_{\omega}(E_i)$ is the depth of that node in $T(\omega)$. (Note that an unsuccessful search really ends at the parent of an external node — recall that external nodes are fictitious devices we made up to help the analysis, they are not part of the binary search tree. So the number of comparisons for an unsuccessful search is equal to one more than the depth of the external node's parent which is exactly the depth of the external node.)

![Figure 2]

From the above and the definition of internal and external path lengths, we have,
\[
S(\omega) = \left( \frac{\text{IPL}(\omega)}{n} \right) + 1 \quad (1)
\]
\[
U(\omega) = \frac{\text{EPL}(\omega)}{(n+1)} \quad (2)
\]

From Lemma 2, and equations (1) and (2), we get
\[
S(\omega) = (1 + 1/n) \cdot U(\omega) - 1. \quad (3)
\]
Equation (3) is known as Hibbard’s equation, and it relates the average number of comparisons for a successful and unsuccessful search in any augmented BST.

Now we consider the case when the BST is formed by randomly choosing a key insertion sequence $\omega \in \Omega_n$ and constructing $T(\omega)$. We want to determine the expected values of the random variables $S$ and $U$ defined on $(\Omega_n, P_n)$. More precisely, we are interested in $S_n$ and $U_n$ defined as
\[
S_n = E(S) = \left( \frac{1}{n!} \right) \sum_{\omega \in \Omega_n} S(\omega); \quad \text{and}
\]
\[
U_n = E(U) = \left( \frac{1}{n!} \right) \sum_{\omega \in \Omega_n} U(\omega). \quad (5)
\]
Next we’ll establish a recurrence equation for $U_n$. In particular, we shall show that $U_n = U_{n-1} + 2/(n+1)$. To this end, we need the following result.
Lemma 3.

\[ \sum_{\omega \in \Omega_n} EPL(\omega) = (n+1) \sum_{\omega' \in \Omega_{n-1}} EPL(\omega') + 2(n!) \]

Proof: As we discussed in the section "Facts about Probability Spaces", the set of all \( \omega \in \Omega_n \) can be obtained by considering each \( \omega' \in \Omega_{n-1} \) and inserting \( x_n \) in the permutation \( \omega' \) as the first, second, ..., \( n \)-th key. Let \( \omega_1', \omega_2', ..., \omega_n' \) be the \( n \) permutations in \( \Omega_n \) obtained from \( \omega' \in \Omega_{n-1} \) in this manner. So,

\[ \sum_{\omega \in \Omega_n} EPL(\omega) = \sum_{\omega' \in \Omega_{n-1}} \sum_{i=1}^{n} EPL(\omega_i'). \quad (6) \]

Fix any \( \omega' \in \Omega_{n-1} \) and consider \( \omega_i' \) for \( 1 \leq i \leq n \). The external nodes of \( T(\omega_i') \) are the same as the external nodes of \( T(\omega') \) except that the \( i \)-th external node, \( E_i \), in \( T(\omega') \) is now internal (and contains \( x_n \)) and has two children which are external nodes in \( T(\omega_i') \). Thus,

\[ EPL(\omega_i') = EPL(\omega') + depth_{\omega'}(E_i) + 2. \]

So,

\[ \sum_{i=1}^{n} ELP(\omega_i') = \sum_{i=1}^{n} (EPL(\omega') + depth_{\omega'}(E_i) + 2) = n \cdot EPL(\omega') + \sum_{i=1}^{n} depth_{\omega'}(E_i) + \sum_{i=1}^{n} 2 = n \cdot EPL(\omega') + EPL(\omega') + 2n = (n+1) \cdot EPL(\omega') + 2n. \]

Plugging this into (6) we get,

\[ \sum_{\omega \in \Omega_n} EPL(\omega) = \sum_{\omega' \in \Omega_{n-1}} \sum_{i=1}^{n} EPL(\omega_i') = \sum_{\omega' \in \Omega_{n-1}} ((n+1) \cdot EPL(\omega') + 2n) = (n+1) \sum_{\omega' \in \Omega_{n-1}} EPL(\omega') + 2n \sum_{\omega' \in \Omega_{n-1}} 1 = (n+1) \sum_{\omega' \in \Omega_{n-1}} EPL(\omega') + 2n (n-1)! = (n+1) \sum_{\omega' \in \Omega_{n-1}} EPL(\omega') + 2(n!) \]

as wanted. \( \square \)

From (5) and (2) we have,

\[ U_n = (1/n!) \sum_{\omega \in \Omega_n} U(\omega) = (1/n!) \sum_{\omega \in \Omega_n} EPL(\omega)/(n+1) = (1/(n+1)!) \sum_{\omega \in \Omega_n} EPL(\omega). \quad (7) \]
By Lemma 3 then,

\[
U_n = \frac{(n+1) \sum_{\omega \in \Omega_{n-1}} EPL(\omega') + 2(n!)}{(n+1)!}
\]

\[
= \sum_{\omega \in \Omega_{n-1}} EPL(\omega')/n! + 2/(n + 1)
\]

\[
= U_{n-1} + 2/(n + 1).
\]

[see (7)].

By repeated substitution in this recurrence equation we get:

\[
U_n = U_{n-1} + 2/(n + 1)
\]

\[
= U_{n-2} + 2/n + 2/(n + 1)
\]

\[
= U_{n-3} + 2/(n - 1) + 2/n + 2/(n + 1)
\]

\[
\vdots
\]

\[
= U_0 + 2/2 + 2/3 + \cdots + 2/(n + 1).
\]

But \(U_0 = 0\). Thus, we have:

\[
U_n = 2 \left( 1/2 + 1/3 + \cdots + 1/(n + 1) \right).
\]

The quantity \(1/1 + 1/2 + \cdots + 1/n\) is known in mathematics as the \(n\)-th harmonic number and is denoted \(H_n\). Therefore,

\[
U_n = 2(H_{n+1} - 1), \quad (8)
\]

Now we observe that Hibbard’s equation (3), relating the average number of comparisons for a successful and unsuccessful search, holds for every \(\omega \in \Omega_n\). Therefore, it holds “on the average”. More precisely,

\[
S_n = (1 + 1/n) \cdot U_n - 1. \quad (9)
\]

(As a simple exercise, you should prove (9) from (3) and the definitions of \(S_n\) and \(U_n\).)

Using (8) and (9) we get:

\[
S_n = 2(1 + 1/n)(H_{n+1} - 1) - 1 = 2(1 + 1/n)H_{n+1} - 2/n - 3 = 2(1 + 1/n)H_n - 3. \quad (10)
\]

It is known that \(H_n \approx \ln n\). Using this approximation in conjunction with (8) and (10) we get

\[
U_n \approx 1.39 \cdot \log_2 (n + 1) - 2, \quad (11)
\]

\[
S_n \approx 1.39 \cdot \log_2 n - 3. \quad (12)
\]

where, in obtaining (12), we discarded the term \(1/n\) because it gets small as \(n\) gets large.
OPTIMAL BINARY SEARCH TREES

Suppose we are given a set of $n$ keys, $K_1 < K_2 < \ldots < K_n$, which are to be stored in a binary search tree. After the tree has been constructed, only search operations will be performed — there will be no insertions or deletions. We are also given a probability density function $P$, where $P(i)$ is the probability of searching for key $K_i$. There are many different binary search trees in which the $n$ given keys can be stored. For a particular tree $T$ with these keys, the average number of comparisons to find a key, for the given probability density $P$, is

$$
\sum_{i=1}^{n} P(i) \cdot (\text{depth}_T(K_i) + 1),
$$

with $\text{depth}_T(K_i)$ denoting the depth of the node where $K_i$ is stored in $T$. The problem we would like to solve is to find, among all the possible binary search trees that contain the $n$ keys, one which minimises this quantity. Such a tree is called an optimal binary search tree. Note that there may be several optimal binary search trees for the given density function. This is why we speak of an optimal, rather than the optimum, binary search tree.

A simple way to accomplish this is to try out all possible binary trees with $n$ nodes, computing the average number of comparisons to find a key in each tree considered, and selecting a tree with the minimum average. Unfortunately, this simple strategy is ridiculously inefficient because there are too many trees to try out. In particular, there are $(2^n)/(n+1)$ different binary trees with $n$ nodes (if interested in the derivation of this formula, see Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, pp. 388–389). Thus, if there are 20 keys, we have to try out 131,282,408,400 different trees. Computing the average number of comparisons in each at the rather astonishing speed of 1 μsec per tree, will still take 2188 hours or approximately 91 days and nights of computing to find the optimal binary search tree (for just 20 keys)! Fortunately, there is a much more efficient, if less straightforward, way to find an optimal binary search tree.

Let $T$ be a binary search tree that contains keys $K_i, K_{i+1}, \ldots, K_j$ for some $1 \leq i \leq j \leq n$. We shall see shortly why it is useful to consider trees that contain subsets of successive keys. We define the cost of $T$, $c(T)$, as

$$
c(T) = \sum_{l=i}^{j} P(l) \cdot (\text{depth}_T(K_l) + 1).
$$

Hence, if $T$ contains all $n$ keys (so $i = 1$ and $j = n$), the cost of $T$ is precisely the expected number of comparisons to find a key for the given density function. Thus, we can rephrase our problem as follows: Given a density function for the $n$ keys, find a minimum cost tree with $n$ nodes.

Before giving the algorithm to find an optimal binary search tree, we prove two key facts.

**Lemma 1.** Let $T$ be a binary search tree containing keys $K_i, K_{i+1}, \ldots, K_j$, and let $T_L$ and $T_R$ be the left and right subtrees of $T$ respectively. Then,

$$
c(T) = c(T_L) + c(T_R) + \sum_{l=i}^{j} P(l).
$$

† This is not so if $T$ is missing some of the keys, because in that case the probabilities of the keys that are in $T$ do not sum up to 1; then, $P$ is not a proper density function relative to the set of keys in the tree.
Proof: This is an easy consequence of the definition of cost of a tree. You should prove it on your own.  

**Lemma 2.** Let $T$ be a binary search tree that has minimum cost among all trees containing keys $K_i, K_{i+1}, \ldots, K_j$, and let $K_m$ be the key at the root of $T$ (so $i \leq m \leq j$). Then $T_L$, the left subtree of $T$, is a binary search tree that has minimum cost among all trees containing keys $K_i, K_{i+1}, \ldots, K_{m-1}$, and $T_R$, the right subtree of $T$, is a binary search tree that has minimum cost among all trees containing keys $K_{m+1}, K_{m+2}, \ldots, K_j$.

Proof: We prove the contrapositive. That is, if either the left or right subtree of $T$ fails to satisfy the property asserted in the lemma we show that $T$ does not really have the minimum possible cost among all trees that contain $K_i, K_{i+1}, \ldots, K_j$.

Let $T'_L$ and $T'_R$ be minimum cost binary search trees that contain keys $K_i, K_{i+1}, \ldots, K_{m-1}$ and keys $K_{m+1}, K_{m+2}, \ldots, K_j$ respectively. Then, $c(T'_L) \leq c(T_L)$ and $c(T'_R) \leq c(T_R)$.

Further, let $T'$ be the tree with key $K_m$ in the root, and left and right subtrees $T'_L$ and $T'_R$ respectively. Evidently, $T'$ is a binary search tree that contains keys $K_i, K_{i+1}, \ldots, K_j$. If $T_L$ or $T_R$ do not have the property asserted by the lemma, then either $c(T_L) > c(T'_L)$ or $c(T_R) > c(T'_R)$. This, together with the inequalities stated in the previous paragraph, implies that $c(T_L) + c(T_R) > c(T'_L) + c(T'_R)$. From this and Lemma 1 we have

$$c(T) = c(T_L) + c(T_R) + \sum_{i=1}^{j} P(l) > c(T'_L) + c(T'_R) + \sum_{i=1}^{j} P(l) = c(T').$$

Thus, $c(T) > c(T')$, and $T$ is not a minimum cost binary search tree among all trees that contain keys $K_i, K_{i+1}, \ldots, K_j$.  

**Computing an Optimal Binary Search Tree**

Lemma 2 is the basis of an efficient algorithm to find an optimal binary search tree. Let $T_{i,j}$ denote a binary search tree that has minimum cost among all trees that contain keys $K_i, K_{i+1}, \ldots, K_j$. Lemma 2 then says that $T_{i,j}$ must have as its root the key $K_m$ for some $m$, and as its left and right subtrees $T_{i,m-1}$ and $T_{m+1,j}$ — minimum cost subtrees containing the keys $K_i, K_{i+1}, \ldots, K_{m-1}$ and $K_{m+1}, K_{m+2}, \ldots, K_j$ respectively. Thus, tree $T_{i,j}$ has the form shown in Figure 1. Since $T_{i,m-1}$ and $T_{m+1,j}$ are “smaller” trees than $T_{i,j}$, this suggests an inductive procedure, starting with small minimum cost trees (each containing just one key) and progressively building larger and larger minimum cost trees, until we have a minimum cost tree with $n$ nodes — which is what we are looking for.
More specifically, we start the induction with minimum cost trees each containing exactly one key, and proceed by constructing minimum cost trees with 2, 3, \ldots, \( n \) successive keys. Note that there are exactly \( n - d + 1 \) groups of \( d \) successive keys, for \( 1 \leq d \leq n \). Thus, instead of considering all possible trees with \( n \) nodes we consider only \( n \) (minimum cost) trees with 1 node each, then \( n - 1 \) (minimum cost) trees with 2 nodes each, and so on, down to 1 minimum cost tree with \( n \) nodes; that is, we consider a total of \( n(n + 1)/2 \) trees — much fewer than \( \binom{2n}{n} / (n + 1) \) trees.

So now the question is how to construct each \( T_{i,j} \) by induction on \( d = j - i + 1 \), the number of keys it contains. The basis of the induction, \( d = 1 \), is trivial. In this case we have \( j = i \) and the minimum cost binary search tree \( T_{i,j} \) that stores \( K_i \) (in fact the only such tree) is a single node containing \( K_i \). Its cost is \( c(T_{i,i}) = P(i) \).

For the induction step, assume that, for some \( d > 1 \), we have already constructed and computed the costs of all the minimum cost trees with fewer than \( d \) successive keys. Now we want to construct and compute the cost of all minimum cost trees with exactly \( d \) successive keys. Consider such a tree \( T_{i,j} \) (hence, \( j - i + 1 = d \)). Let \( T_{i,m,j} \) be the tree with \( K_m \) in the root, and left and right subtrees \( T_{i,m-1} \) and \( T_{m+1,j} \) respectively. Lemma 2 implies that \( T_{i,j} \) is the minimum cost tree among the \( T_{i,m,j} \)'s. Thus we can find \( T_{i,j} \) simply by trying out all the \( T_{i,m,j} \)'s, for \( m = i, i + 1, \ldots, j \). In fact, Lemma 1 tells us how to compute \( c(T_{i,m,j}) \) efficiently, so that “trying out” each possible \( m \) will not take too long. Since \( T_{i,m-1} \) and \( T_{m+1,j} \) both have fewer than \( d \) keys, we have already (inductively) computed \( T_{i,m-1} \) and \( T_{m+1,j} \) and their costs, \( c(T_{i,m-1}) \) and \( c(T_{m+1,j}) \). Lemma 1 then tells us how to get \( c(T_{i,m,j}) \) in terms of these. Note that when \( m = i \) the left subtree of \( T_{i,m,j} \) is \( T_{i,j-1} \), and when \( m = j \) the right subtree of \( T_{i,m,j} \) is \( T_{j+1,i} \). We define \( T_{i,j} \) to be empty if \( i > j \), and we define the cost of an empty tree to be 0.

Figure 2 shows this algorithm in pseudo-code. The algorithm takes as input an array \( \text{Prob}[1..n] \), which specifies the probability density \( (\text{Prob}[i] = P(i)) \). It computes two two-dimensional arrays, \( \text{Root} \) and \( \text{Cost} \), where \( \text{Root}[i,j] \) is the root of \( T_{i,j} \), and \( \text{Cost}[i,j] = c(T_{i,j}) \), for all \( i \) and \( j \) such that \( 1 \leq i \leq j \leq n \).† To help compute \( \text{Root} \) and \( \text{Cost} \) the algorithm maintains a third array, \( \text{SumOfProb} \), where \( \text{SumOfProb}[i] = \sum_{l=1}^{i} P(l) \) for \( 1 \leq i \leq n \), and \( \text{SumOfProb}[0] = 0 \). Note that \( \sum_{l=1}^{i} P(l) = \text{SumOfProb}[j] - \text{SumOfProb}[i-1] \).

The algorithm of Figure 2 does not explicitly construct an optimal binary search tree, but such a tree is implicit in the information in array \( \text{Root} \). As an exercise you should write an algorithm which, given \( \text{Root} \) and an array \( \text{Keys}[1..n] \), such that \( \text{Keys}[i] = K_i \), constructs an optimal binary search tree.

It is easy to see that the time complexity of this algorithm is dominated by the number of times the innermost (for \( m \)) loop is executed. This is

\[
\sum_{d=1}^{n} d(n - d) = n^2(n + 1)/2 - n(n + 1)(2n + 1)/6 = n(n + 1)(n - 1)/6
\]

which is in \( \Theta(n^3) \). A slight modification of this algorithm leads to an algorithm with complexity in \( \Theta(n^2) \) (if interested, see D.E. Knuth, “Optimum binary search trees”, Acta Informatica, Vol. 1 (1971), pp. 14–25.)

† For technical reasons that will become apparent when you look at the algorithm carefully, we need to set \( \text{Cost}[i,i-1] = 0 \) for \( 1 \leq i \leq n + 1 \). Recall that \( T_{i,j-1} \) is empty and thus has cost 0.
OptimalBST(Prob[1..n])

% Initialisation %
SumOfProb[0] := 0

for i := 1 to n do
    SumOfProb[i] := Prob[i] + SumOfProb[i - 1]
    Root[i, i] := i
    Cost[i, i] := Prob[i]
end for

for i := 1 to n + 1 do
    Cost[i, i - 1] := 0
end for

% Compute information about trees with d > 1 successive keys. %

for d := 2 to n do
    % Compute Root[i, j] and Cost[i, j] for each i and j with j - i + 1 = d. %
    for i := 1 to n - d + 1 do
        j := i + d - 1
        MinCost := +∞
        % Find m between i and j so that c(Ti,m,j) is minimised. %
        for m := i to j do
            c := Cost[i, m - 1] + Cost[m + 1, j] + (SumOfProb[j] - SumOfProb[i - 1])
            if c < MinCost then
                MinCost := c
                r := m
            end if
        end for
        Root[i, j] := r
        Cost[i, j] := MinCost
    end for
end for

Algorithm for Optimal Binary Search Tree

Figure 2

Unsuccessful Searches

In the preceding discussion we have only considered successful searches. However, if we take into account unsuccessful searches, it is possible that the constructed tree is no longer optimal. Fortunately, this problem can be taken care of in a straightforward manner. To find an optimal binary search tree in the case where both successful and unsuccessful searches are taken into account, we must know the probability density for both successful and unsuccessful searches. So, in addition to P(i) we must also be given Q(i) for 0 ≤ i ≤ n, where

- Q(0) is the probability of searching for keys less than $K_1$;
- Q(i) is the probability of searching for keys between $K_i$ and $K_{i+1}$ (exclusive), for 1 ≤ i ≤ n;
- Q(n) is the probability of searching for keys greater than $K_n$. 

36
Thus, $Q$ describes the probability of unsuccessful searches.

In each binary search tree containing $K_1, K_2, \ldots, K_n$, we add $n+1$ external nodes $E_0, E_1, \ldots, E_n$. This is illustrated below, in Figure 3; the external nodes are drawn in boxes, as usual.

![Figure 3](image)

The average number of comparisons for a successful or unsuccessful search in such a tree $T$ is

$$
\sum_{i=1}^{n} P(i) \cdot (\text{depth}_T(K_i) + 1) + \sum_{i=0}^{n} Q(i) \cdot \text{depth}_T(E_i).
$$

The left term is the average number of comparisons for successful searches, and the right term is the average number of comparisons for unsuccessful searches. Now we want to find a tree that minimises this quantity.

We can proceed exactly as before, except that the definition of the cost of a tree $T$ with keys $K_i, K_{i+1}, \ldots, K_j$ is slightly modified to account for the unsuccessful searches. Namely, it becomes

$$
c^d(T) = \sum_{l=i}^{j} P(l) \cdot (\text{depth}_T(K_l) + 1) + \sum_{l=i-1}^{j} Q(l) \cdot \text{depth}_T(E_l).
$$

With this cost function, Lemma 1 is slightly different:

**Lemma 1'.** Let $T$ be a binary search tree containing keys $K_i, K_{i+1}, \ldots, K_j$, and let $T_L$ and $T_R$ be the left and right subtrees of $T$ respectively. Then,

$$
c^d(T) = c^d(T_L) + c^d(T_R) + Q(i - 1) + \sum_{l=i}^{j} (P(l) + Q(l)).
$$

Everything else works out exactly as before. In particular, Lemma 2 is still valid (check this!). As an exercise show how to modify the algorithm in Figure 2 to account for these changes.
Number 8

AVL TREES

Binary search trees work well in the average case, but they still have the drawback of linear worst case time complexity for all three operations (Search, Insert and Delete).

**Definition:** A binary tree of height $h$ is *ideally height-balanced* if every leaf has depth $h$ or $h - 1$, and every node of depth $< h - 1$ has two children.

It would be nice if we could keep the binary search tree ideally height-balanced at all times. Then a tree of $n$ nodes would be guaranteed to have height $h = \lceil \log_2 n \rceil$, so searches would always take time in $O(\log n)$. But insertions and deletions might destroy the ideally height-balanced property, and a reorganisation (to make the tree ideally height-balanced again, while maintaining the binary search tree property) might take as much as linear time.

AVL (or height-balanced) trees are a happy compromise between arbitrary binary search trees and ideally height-balanced binary search trees. The name “AVL” comes from the names of the two Soviet mathematicians, Adelson-Velski and Landis, who devised them.

**Definition:** A binary tree is *height-balanced* if the heights of the left and right subtrees of every node differ by at most one. An AVL tree is a height-balanced binary search tree.

**Note:** By convention, the height of an *empty* binary tree (one with 0 nodes) is $-1$; the height of a tree consisting of a single node is 0.

**Examples:**

![Binary search tree examples](image)

**Non-examples:**

![Binary search tree non-examples](image)

**Good news:**

- The worst case height of an AVL tree with $n$ nodes is $1.44 \log_2 (n + 2)$. Thus, the Search operation can be carried out in $O(\log n)$ time in the worst case.
- Insertions and deletions can also be done in $O(\log n)$ time, while preserving the “AVL-ness” of the tree.
- Empirical studies show that AVL trees work very well on the average case too.

**Bad news:** The algorithms for insertion and deletion are a bit complex.

**Definition:** Let $h_R$ and $h_L$ be the heights of the right and left subtrees of a node $m$ in a binary tree respectively. The *balance factor* of $m$, $BF[m]$, is defined as $BF[m] = h_R - h_L$.

For an AVL tree, the balance factor of any node is $-1$, 0, or $+1$. 

39
• If $BF[m] = +1$, $m$ is right heavy.
• If $BF[m] = -1$, $m$ is left heavy.
• If $BF[m] = 0$, $m$ is balanced.

In AVL trees we will store $BF[m]$ in each node $m$. When we draw AVL trees we will put a “+”, “−”, or “0” next to each node to indicate, respectively, that the node’s balance factor is $+1$, $−1$, or 0.

Next we consider algorithms for the Search, Insert and Delete operations in AVL trees.

THE ALGORITHM FOR Search

We simply treat $T$ as an ordinary binary search tree — there is nothing new to say here.

THE ALGORITHM FOR Insert

To insert a key $x$ into an AVL tree $T$, let us first insert $x$ in $T$ as in ordinary binary search trees. That is, we trace a path from the root downward, and insert a new node with key $x$ in it in the proper place, so as to preserve the binary search tree property. This may destroy the integrity of our AVL tree in that

- the addition of a new leaf may have destroyed the height-balance of some nodes, and,
- the balance factors of some nodes must be updated to take into account the new leaf.

We will address each of these points in turn.

Rebalancing an AVL tree after Insertion

The height-balance property of a node may have been destroyed as a result of the insertion of the new leaf in two ways:

1. the new leaf increased the height of the right subtree of a node that was already right heavy (before the insertion); or,
2. the new leaf increased the height of the left subtree of a node that was already left heavy (before the insertion).

These two cases are illustrated in Figures 1(a) and (b) below. Note that the insertion of the new leaf can affect the balance factors only of its ancestors. To see this, observe that the height of any node that is not an ancestor of the new leaf is the same as before the insertion; consequently the heights of the left and right children of such a node are the same as before the insertion. Node $m$ in Figure 1 is assumed to be the minimum height ancestor of the new leaf which is no longer height balanced as a result of the insertion.

Since the two cases are symmetric (one is obtained from the other by changing every reference of “right” to “left”, and of “+” to “−”, and vice versa), we shall only consider case (1) in detail. There are two ways in which (1) could arise, illustrated in Figure 2(a) and (b) respectively. The balance factors indicated for $A$ and $B$ are after the insertion of the new node.

The subtree shown in Figure 2(a) can be rebalanced by a simple transformation called “single left rotation” on node $m$. This transformation is illustrated in Figure 3: In 3(a) we copied the subtree of Figure 2(a), and 3(b) shows the result of the single left rotation on that subtree.

Note that this transformation has the following properties.
S.1 It rebalances the subtree rooted at node $m$ (so that subtree becomes height-balanced again).
S.2 It maintains the binary search tree property.

S.3 It can be done in constant time: only a few pointers have to be switched around. As an exercise, write a program that implements this rotation, given a pointer to node $m$, assuming the standard representation for binary trees.

S.4 It keeps the height of $m$ equal to its height before the insertion of the new node, namely, height $h + 2$.

Unfortunately the subtree in Figure 2(b) cannot be rebalanced by a single left rotation. You
should check that the subtree resulting from such a transformation is not height-balanced.

Figure 4(a) shows the subtree in Figure 2(b) in a bit more detail. This more detailed picture leads to a different way of transforming the subtree into a height-balanced one. This transformation is called a “double right left rotation” and is illustrated in Figure 4(b). The name comes from the fact that this transformation can be obtained if we rotate first $B$ to the right and then, in the resulting subtree, rotate $C$ left. The balance factors labeled as “*/*” in Figure 4 depend on whether the new node was actually inserted under $T_{22}$ (the first entry of the label) or under $T_{21}$ (the second entry of the label).

The double right left rotation has the following properties.

D.1 It rebalances the subtree rooted at $m$, (so that subtree becomes height-balanced again).

D.2 It maintains the binary search tree property.

D.3 It can be done in constant time: we only have to change a few pointers. As an exercise, write a program that implements this rotation, given a pointer to node $m$.

D.4 It keeps the height of $m$ equal to that node’s height before the insertion of the new node, namely, height $h + 2$.

As we already remarked, the imbalance shown in Figure 1(b) can be fixed in a symmetric way. The two subcases, and the transformations that rebalance the subtrees, called “single right rotation” and “double left right rotation”, are illustrated in Figures 5 and 6 respectively. Remarks analogous to S.1–S.4 and D.1–D.4 apply in these transformations as well.
Figure 4

Figure 5

**Updating the Balance Factors after Insertion**

The balance factors of some nodes may change as a result of inserting a new node. First of all, observe that only the balance factors of the new node’s ancestors may need updating. For any other node \( i \), \( i \)’s left and right subtrees (and, in particular, their heights) have not changed and thus neither has the balance factor of \( i \). But not all of the new node’s ancestors’ balance factors
Figure 6

may need updating. Figure 7 illustrates the issue. Insertion of key 8 to the AVL tree in 7(a) results in the AVL tree in 7(b). Note that only the balance of 9, 8’s parent, has changed. On the other hand, insertion of key 8 to the AVL tree in 7(c) results in the AVL tree in 7(d), where the balance factors of all of 9’s ancestors have changed.

In general, let n be the new node just inserted into the tree and let p be n’s parent. Further, let m be the closest ancestor of p that was not balanced (that is, that had balance factor \pm 1) before the insertion of n; if no such ancestor of p exists, let m be the root of the tree. (Note that m could be p, if BF[p] \neq 0 before the insertion.)

Claim. Only the balance factors of the nodes between p and m (included) need to be changed as a result of the insertion of n.

Justification: Consider the (0 or more) nodes that are ancestors of p and proper descendents of m. By choice of m, all these nodes were balanced before the insertion of n. Thus their two subtrees had the same height and the insertion of n has increased the height of one of the subtrees; hence for each such node, its balance factor must be set to -1 or +1, depending on whether n was inserted to the left or right subtree, respectively. Next consider node m. If m is the root and was balanced before the insertion, similar remarks as above apply to m: in this case the insertion of n has the effect of increasing the height of the entire tree. If m was not balanced before the insertion, we have two possibilities:

- If m was left heavy and n was inserted to m’s right subtree, or if m was right heavy and n was inserted to m’s left subtree, the subtree rooted at m becomes balanced as a result of the insertion (so we must set BF[m] = 0), but its height does not change. Therefore, neither do the heights of m’s ancestors’ subtrees; so the balance factors of m’s proper ancestors do not change, and we can stop the process of balance factor updating here.
- If, on the other hand, m was right heavy and n was inserted to m’s right subtree, or if m was left heavy and n was inserted to m’s left subtree, the subtree rooted at m becomes unbalanced.
(these are the two cases illustrated in Figures 1(a) and 1(b) respectively). We can rebalance the subtree as we discussed previously (by the appropriate type of rotation). After the rebalancing, however, the subtree rooted at \( m \) will have the same height as it did before the insertion of \( m \) (recall Remarks S.4 and D.4). Thus, as argued before, the balance factors of \( m \)'s ancestors do not change. Note, however, that when we rotate, the balance factors of the rotated nodes need updating, so we must do that before stopping.†

The discussion on rebalancing and updating the balance factors after an insertion leads us to the following outline for the AVL tree insertion algorithm.

\[
\text{Insert}(x, T)
\]

1. Trace a path from the root down, as in the binary search trees, and insert \( x \) into a new leaf at the end of that path (the new leaf must be in the proper position, so as to maintain the binary search tree property).

† After a rotation, some of the rotated nodes are no longer ancestors of the inserted node; however, they may still need to have their balance factors updated.
2. Set the balance factor of the new leaf to 0. Retrace the path from the leaf up towards the root and process each node $i$ encountered as follows:

(a) If the new node was inserted in $i$'s right subtree, then increase $BF[i]$ by 1 (because $i$'s right subtree got taller); otherwise, decrease $BF[i]$ by 1 (because $i$'s left subtree got taller).

(b) If $BF[i] = 0$ (so the subtree rooted at $i$ became balanced as a result of the insertion, and its height did not change) then stop.

(c) If $BF[i] = +2$ and $BF[\text{child}(i)] = +1$ then do a single left rotation on $i$, adjust the balance factors of the rotated nodes ($A$ and $B$ in Figure 3(b)), and stop.

(d) If $BF[i] = +2$ and $BF[\text{child}(i)] = -1$ then do a double right left rotation on $i$, adjust the balance factors of the rotated nodes ($A$, $B$ and $C$ in Figure 4(b)), and stop.

(e) If $BF[i] = -2$ and $BF[\text{child}(i)] = -1$ then do a single right rotation on $i$, adjust the balance factors of the rotated nodes ($A$ and $B$ in Figure 5(b)), and stop.

(f) If $BF[i] = -2$ and $BF[\text{child}(i)] = +1$ then do a double left right rotation on $i$, adjust the balance factors of the rotated nodes ($A$, $B$ and $C$ in Figure 6(b)), and stop.

(g) If $i = \text{root}$ then stop.

THE ALGORITHM FOR Delete

To delete a key $x$ from an AVL tree $T$, we first locate the node $n$ where $x$ is stored. (This can be done by using the algorithm for Search.) If no such node exists, we're done (there's nothing to delete). Otherwise we have three cases (as with ordinary binary search trees).

1. $n$ is a leaf: Then we simply remove it. This may cause the tree to cease being height-balanced. So we may need to rebalance it. We also have to update the balance factors of some nodes. These issues will be dealt with shortly.

2. $n$ is a node with only one child: Let $n'$ be $n$'s only child. Note that $n'$ must be a leaf; otherwise the subtree rooted at $n$ would not have been height-balanced before the deletion. In this case we copy the key stored at $n'$ into $n$ and we remove $n'$ as in case (1) (since, as we just argued, it must be a leaf).

3. $n$ has two children: Then we find the smallest key in $n$'s right subtree which, by the binary search tree property, is the smallest key in $T$ larger than the key stored in $n$. To find this key, we go to $n$'s right child (which exists), and we follow the longest chain of left child pointers until we get to a node $n'$ that has no left child. We copy the key stored in $n'$ into $n$ and remove $n'$ from the tree, as in (1), if $n'$ does not have a right child either, or as in (2), if $n'$ has only a right child.

To complete the algorithm we must discuss the conditions under which rebalancing is required and how the rebalancing can be performed. Since cases (2) and (3) ultimately reduce to deleting a leaf, case (1) is the only one we need to consider.
Rebalancing an AVL Tree after Deleting a Leaf

The deletion of a leaf \( n \) will cause the tree to become unbalanced in one of two cases:

(a) It reduces the height of the right subtree of a left heavy node; or,

(b) It reduces the height of the left subtree of a right heavy node.

These two cases, illustrated below in Figure 8, are symmetric (as are the analogous cases in insertion), so we will only consider the first. As an exercise, you should treat the other.

We consider case (a). As with insertion there are two ways this case could arise, shown in Figures 9(a) and 10(a). The subtree in 9(a) can be rebalanced by means of a single right rotation, and the result of this transformation is shown in 9(b). The “\( 0/- \)” next to \( B \) in 9(a) means that...
this case will arise if the balance factor of $B$ (after the deletion) is 0 or $-1$ (that is, the height of $T_2$ is $h$ or $h + 1$). Accordingly, the balance factors of $B$ and $A$ will be $+1$ or $0$, and $-1$ or $0$, after the rotation (see 9(b)).

The unbalanced subtree of Figure 10(a) can be rebalanced by means of a double left right rotation, the result of which is shown in 10(b). The balance factors of the nodes that have a label of the form “$*/+*/$” in Figure 10 depend on the heights of $T_{21}$ and $T_{22}$. Note that at least one of these subtrees must have height $h$ (the other could have height $h - 1$ or $h$). The first entry of the label indicates the balance factor in the event $T_{21}$ has height $h - 1$ and $T_{22}$ has height $h$; the second entry of the label indicates the balance factor in the event both trees have height $h$; and the third entry indicates the balance factor when $T_{21}$ has height $h$ and $T_{22}$ has height $h - 1$.

![Diagram of tree transformations](image)

**Figure 10**

The above two transformations have the following properties.

1. They rebalance the subtree rooted at $m$ (so the subtree becomes height-balanced again).
2. They maintain the binary search tree property.
3. They can be done in constant time by simply manipulating a few pointers. As an exercise, write programs that implement the rotations of Figures 9 and 10, given a pointer to $m$.
4. They may decrease the height of the subtree rooted at $m$, compared to the height of the subtree before the deletion.

Compare 4 with remarks S.4 and D.4 about rotations to restore balance in insertions. The difference is important: In the insertion algorithm just one rotation always rebalances the subtree, and, by maintaining the height of that subtree, it rebalances the entire tree. In the deletion algorithm the rotation balances the subtree, but since the height is decreased, the balance factor of nodes higher up (closer to the root) may change as a result — so we may have to go on rotating subtrees all the way up to the root in order to rebalance the entire tree. Thus in deletion we may have to do as many as $O(\log n)$ rotations. (That’s acceptable though, because each one takes only constant time! We will say more about the complexity of operations shortly.)
Updating the Balance Factors after Deleting a Leaf

We must also address the question of how the deletion of a leaf affects the balance factors of its ancestors (clearly, it doesn’t affect the balance factors of other nodes).

Let \( n \) be the deleted leaf and let \( p \) be its parent. We trace the path from \( p \) back to the root and we process each node \( i \) we encounter on the way as follows:

- If \( i \) was balanced before the deletion (so \( BF[i] = 0 \)) then the left and right subtrees of \( i \) had the same length. The removal of \( n \) shortened one of them (so \( i \)'s balance factor must be updated), but the height of the subtree rooted at \( i \) after the deletion remains the same as before it. This means that the deletion of \( n \) does not affect the balance factors of \( i \)'s proper ancestors. So, in this case, all we have to do is increase \( BF[i] \) by one if \( n \) was in \( i \)'s left subtree (because then the deletion made the right subtree of \( i \) taller than the left), or decrease \( BF[i] \) by one if \( n \) was in \( i \)'s right subtree (because then the deletion made the left subtree of \( i \) taller than the right). After this, we can stop the process of updating balance factors.

- If \( i \) was right or left heavy before the deletion (\( BF[i] = \pm 1 \)), we again update \( BF[i] \) as above. If this balances node \( i \), the deletion of \( n \) shortened one of the two subtrees of \( i \), so we go up the path to consider the next node. Otherwise, the increase or decrease of \( BF[i] \) by one causes the subtree rooted at \( i \) to become (height) unbalanced (\( BF[i] \) becomes \( \pm 2 \)). In this case we need to rebalance the subtree by the appropriate rotation, as discussed previously. If the rotation causes the height of \( i \) to decrease (see Remark 4 above), the process of updating balance factors and, possibly, rotating, must continue with \( i \)'s parent. Otherwise (if \( i \)'s height was not affected by the rotation), the process can stop at \( i \).

- Finally, if the process propagated all the way to the root (\( i = \) root) we can stop.

From this discussion you should be able to distill the outline of an algorithm for AVL tree deletion.

Worst Case Time Complexity for Search, Insert, Delete

Theorem. (Adelson-Velski and Landis) The height of an AVL tree with \( n \) nodes is at most \( 1.44 \log_2(n + 2) \).

Proof: Let \( T_h \) be a height-balanced tree of height \( h \) with the minimum possible number of nodes, and let \( n_h \) be that number of nodes. Since \( T_h \) is height-balanced, one of its left subtrees must have height \( h - 1 \) and the other height \( h - 1 \) or \( h - 2 \). Since we want \( T_h \) to have the minimum number of nodes, we may assume that one of its subtrees is \( T_{h-1} \) and the other is \( T_{h-2} \). Thus, the number of nodes in \( T_h \) is equal to the number of nodes in \( T_{h-1} \) plus the number of nodes in \( T_{h-2} \) plus one (for the root); that is,

\[
n_h = n_{h-1} + n_{h-2} + 1.
\]

Thus \( n_0 = 1, n_1 = 2, n_2 = 4, n_3 = 7, n_4 = 12, \) and so on. Comparing this with the sequence of Fibonacci numbers we see that, in general, \( n_h = F_{h+3} - 1 \) (where \( F_h \) is the \( h \)th Fibonacci number).† From the theory of Fibonacci numbers we know that \( F_h > (\phi^h / \sqrt{5}) - 1 \), where \( \phi = (1 + \sqrt{5})/2 \)‡ (if interested in this and other results on Fibonacci numbers, see Knuth, The Art of Computer Programming, Vol. 1 (Fundamental Algorithms), pp. 78–83.)

† The \( i \)th Fibonacci number is defined inductively as follows: \( F_1 = F_2 = 1 \), and for \( i > 2 \), \( F_i = F_{i-1} + F_{i-2} \).
‡ \( \phi \) is known as the “golden ratio”.

49
Thus for the number \( n \) of nodes in any AVL tree of height \( h \) we must have:

\[
    n \geq n_h = F_{h+3} - 1 > \left( \frac{\phi^{h+3}}{\sqrt{5}} \right) - 2.
\]

Therefore,

\[
    h < \log_\phi \left( (n + 2)\sqrt{5} \right) - 3,
\]

so

\[
    h < \left( \frac{1}{\log_2 \phi} \right) \cdot (\log_2 \sqrt{5} + \log_2 (n + 2)) - 3,
\]

from which the theorem follows by arithmetic.

In the worst case, the algorithms for Search, Insert, and Delete have to process all nodes in a path from the root to a leaf. The above theorem says that this path must involve at most \( O(\log n) \) nodes. Processing a node (be it just comparing the key stored in it to a key we are searching for, updating the balance factor, or performing a rotation on that node) takes constant time. Thus all these algorithms take \( O(\log n) \) time in the worst case.
SELF-ADJUSTING BINARY SEARCH TREES

Another interesting data structure for implementing the dictionary abstract data type is the *self-adjusting binary search tree* or *splay tree*. A self-adjusting BST is a BST maintained in a particular way.

Suppose we are interested in processing a number of dictionary operations (Insert, Delete or Search) starting with an empty dictionary. Throughout these notes \( m \) denotes the number of dictionary operations that are processed and \( n \leq m \) is the maximum number of keys in the dictionary at any time.

In AVL trees we are very careful to maintain the BST height-balanced at all times. This ensures that the tree will have height \( O(\log n) \). Since each operation takes time at most proportional to the height of the tree, each individual operation takes time in \( O(\log n) \). The time needed to process \( m \) operations is therefore in \( O(m \log n) \).

Self-adjusting trees have the interesting property that they achieve the same time bound, even though it is *not* the case that each individual operation takes time in \( O(\log n) \). Some take more (in fact, some may take as long as \( \Theta(n) \)) and some take less, so that the overall time complexity is in \( O(m \log n) \). We cannot hope to derive this bound simply by obtaining an upper bound for the worst-case time complexity of each individual operation and multiplying that bound by the number of operations performed: Since some operations may take as long as \( \Theta(n) \) time, this method would yield an upper bound of \( O(mn) \) — much higher than the claimed \( O(m \log n) \).

The key idea in self-adjusting BSTs is the so-called *splay operation*, which modifies the structure (but not the contents) of a BST. A splay operation at node \( u \) of a BST consists of performing the following actions repeatedly, until \( u \) has become the root of the tree; each application of these actions will be called a *step* of the splay:

- **Case 1**: if \( u \) is the left (respectively, right) child of the root \( r \) then perform a right (respectively, left) rotation at \( r \) (cf. Figure 1).
- **Case 2**: if \( u \) is the left (respectively, right) child of \( v \) and \( v \) is the left (respectively, right) child of \( w \) then perform a right (respectively, left) rotation at \( v \), followed by another right (respectively, left) rotation at \( v \) (cf. Figure 2).
- **Case 3**: if \( u \) is the left (respectively, right) child of \( v \) and \( v \) is the right (respectively, left) child of \( w \) then perform a right (respectively, left) rotation at \( v \), followed by a left (respectively, right) rotation at \( w \) (cf. Figure 3).

After the splay operation at node \( u \) is completed, \( u \) is the root of the resulting tree. An example of the effect of an entire splay is shown in Figure 4. Note the drastic effect of that operation on the structure of the tree.

To process a dictionary operation (Insert, Delete or Search) in a self-adjusting BST, we first perform the operation as in plain BSTs and then we apply a splay operation at the deepest node of the tree that was accessed in performing the dictionary operation.

Specifically, after a Search or Insert operation, a splay is applied at the node inserted or searched (in the event of an unsuccessful search, the splay is applied at the last node accessed during the search).

For Delete operations, the node at which the splay operation is applied is determined as follows: if the key to be deleted is not present, the deletion amounts to an unsuccessful search and thus the last node accessed is splayed. If the deleted key was stored at a leaf, the Delete operation removed that leaf; the splay operation is applied to the node that was the parent of that leaf. If
the deleted key was stored at a node $u$ with just one child $x$, the \texttt{DELETE} operation will replace $u$ by $x$; the \texttt{splay} is applied to $x$. Finally, if the deleted key was stored at a node $u$ with two children, the \texttt{DELETE} operation replaced the key in $u$ by the key in the leftmost node of the tree rooted at the right child of $u$ (call this node $y$) and removed $y$ as in one of the previous two cases, since $y$ is either a leaf or has only a right child; accordingly, the \texttt{splay} operation is applied to the former parent of $y$ or the former right child of $y$.

It is important to note that in self-adjusting BSTs the structure of the tree is modified every time a search is performed. Thus, the shape of the tree depends not only on the order in which keys are inserted or deleted but also on the order in which keys are searched. (In contrast, the structure of AVL trees is modified only when nodes are inserted or deleted.) In this sense, self-adjusting BSTs are reminiscent of self-organising lists. The \texttt{splay} operation can be viewed as a heuristic that makes the most recently searched key the root of the BST, much the way the Move-to-Front heuristic achieves the analogous property in self-organising lists. The amazing property of self-adjusting BSTs is that they achieve the $O(m \log n)$ time complexity for any sequence of $m$ operations as long as there are at most $n$ keys in the tree at any point in time!

\textbf{Some Motivation}

Let $T$ be a BST with $n$ nodes. We will associate with $T$ a positive number which intuitively is a measure of the balance of $T$. A small number indicates $T$ is almost a complete binary tree, with the length of the longest path approximately $\log n$, and a large number indicates $T$ is almost a singly linked list, with the length of the longest path approximately $n$. One such natural measure on $T$ is the following. For each node $u$ in $T$, let $L_T(u)$ be the length of the path from the root to $u$ plus one (thus, $L_T(u)$ is the number of comparisons in a successful search for the key stored at node $u$ in $T$). Define $L(T) = \sum_{u \in T} L_T(u)$ — if $T$ is the empty tree then define $L(T) = 0$. Note that $L(T)$ is the internal path length of $T$ plus $n$ (see the handout titled “The Average Case Analysis for Binary Search Trees”), and that $L(T)/n$ is the average number of comparisons for a successful
search in $T$ assuming every node is equally probable. If $L(T) \approx n \log n$ then $T$ is almost a complete binary tree, whereas if $L(T) \approx n^2$ then $T$ is almost a singly linked list.

A measure equivalent to $L(T)$ is the following. For each node $u$ in $T$, the weight of $u$, $W_T(u)$, is the number of nodes in the subtree rooted at $u$ (including node $u$). Define $W(T) = \sum_{u \in T} W_T(u)$.

**Exercise:** Prove that $W(T) = L(T)$.

Thus, $W(T)$ is also a measure of the balance of $T$. It turns out that for self adjusting trees, the analysis requires a different measure of the balance of $T$.

**Definitions.** For each node $u$ in $T$, the rank of $u$, $R_T(u)$, is the number of bits in the binary representation of $W_T(u)$; that is, $R_T(u) = \lfloor \log W_T(u) \rfloor + 1$. The rank of $T$ is $R(T) = \sum_{u \in T} R_T(u)$ — if $T$ is the empty tree then define $R(T) = 0$.

**Exercise:** Prove that a complete BST with $n$ nodes has minimum rank among all $n$-node BSTs (this is fairly hard); and that a completely degenerate BST with $n$ nodes (i.e., a tree where no node has more than one child) has maximum rank among all $n$-node BSTs (this is easy). Using these facts, prove that the rank of an $n$-node BST is in $\Omega(n)$ and $O(n \log n)$. 

Figure 2

Case 2 of a splay step (two symmetric cases)
The Analysis of Self-Adjusting BSTs

Lemma 1. Let $T'$ be the tree resulting when a node is added to or deleted from $T$, but without performing any splay operations. Then $R(T') \leq R(T) + \log n + 1$, where $n$ is the number of nodes in $T'$.

Proof: The Lemma is immediate in the case of deleting a node, since in that case the weight of each node in $T$ is no greater than in $T'$, and hence the rank of no node increases. Now consider the case of inserting a node. The weight of every ancestor of the new node (including that node) in $T'$ increases by 1 in comparison to its weight in $T$. However, the rank of such a node will increase only if the number of bits in the binary representation of the weight increases. (Recall that the rank of a node is the number of bits in the binary representation of the node’s weight.) Incrementing a number by one causes the number of bits in its binary representation to increase only if the binary representation of the number after the increment consists of a leading 1 followed only by 0’s. Since the maximum weight after the insertion is $n$, the maximum number of bits in the binary representation of the weight of any node requires at most $\lceil \log n \rceil + 1$ bits; thus the number of nodes whose binary representation consists of a leading 1, followed only by 0’s cannot exceed that number. From this we conclude that the number of nodes whose rank increased as a result of the insertion of the new node is at most $\lceil \log n \rceil + 1$. Therefore, $R(T') \leq R(T) + \log n + 1$, as wanted. \qed
Lemma 2. The number of splay steps in a splay operation transforming $T$ to $T'$ is at most $R(T) - R(T') + 3\log n + 1$, where $n$ is the number of nodes in $T$ (and $T'$).

This lemma, whose proof we shall postpone for the moment, is the key to our main result.

Theorem 1. To process an arbitrary sequence of $m$ dictionary operations using self-adjusting BSTs, starting with the empty tree, takes time $O(m\log n)$, where $n$ is the maximum number of keys ever in the tree during this sequence of operations.

Proof: Let $T_0$ be the initial tree which, by assumption, is empty. Let $T_i$ be the tree after the $i$-th dictionary operation (but before the $i$-th splay operation) has been performed, and $T'_i$ be the tree after the $i$-th splay operation. Thus, over the course of time, we get the following sequence of trees:

$$T_0 \rightarrow T_1 \rightarrow T'_1 \rightarrow T_2 \rightarrow T'_2 \rightarrow \cdots \rightarrow T_m \rightarrow T'_m$$

where the $i$-th dictionary operation transforms $T'_{i-1}$ to $T_i$ and the $i$-th splay operation transforms $T_i$ to $T'_i$.

Let $t_i$ be the time needed to process the $i$-th operation; this includes both the time for the dictionary operation itself and the time for the corresponding splay operation. Note that $t_i$ is proportional to the number of splay steps performed by the $i$-th splay operation. This is because (a) the number of splay steps is $\lceil d/2 \rceil$, where $d$ is the depth of the deepest node accessed by the dictionary operation; (b) the time for the dictionary operation is proportional to $d$; and (c) each splay step takes $\Theta(1)$ time. Since we are interested in an asymptotic analysis and therefore ignore constants of proportionality, we shall take

$$t_i = \text{number of splay steps in the } i\text{-th splay operation.} \quad (1)$$
Define the amortised complexity of the i-th operation as usual, taking the potential to be the rank of the tree:

\[
\begin{align*}
    u_i &= t_i + R(T'_i) - R(T'_{i-1}) \\
    &\leq t_i + R(T'_i) - R(T_i) + \log_2 n + 1 \quad \text{[by Lemma 1]} \\
    &\leq t_i + (3\log_2 n + 1 - t_i) + \log_2 n + 1 \quad \text{[by Lemma 2 and (1)]} \\
    &= 4\log_2 n + 2.
\end{align*}
\]

Therefore, the total time needed for a sequence of \( m \) operations is

\[
\sum_{i=1}^{m} u_i + R(T'_0) - R(T'_m) \leq m(4\log_2 n + 2) \in \Theta(m\log_2 n)
\]
as wanted. \( \square \)

For the proof of Lemma 2 we shall need some preliminary facts.

**Lemma 3.** If \( T \) is the tree before and \( T' \) the tree after a splay step at node \( u \) then \( R(T') \leq R(T) + 2(R_{T'}(u) - R_T(u)) \).

**Proof:** This is a proof by cases, depending on the type of the splay step.

**Case 1:** The splay step is of the type depicted in Figure 1. We have,

\[
R(T') = R(T) + [(R_{T'}(u) - R_T(u)) + (R_{T'}(r) - R_T(r))].
\]

But \( R_{T'}(u) - R_T(u) \geq 0 \) (because \( W_{T'}(u) > W_T(u) \)) and \( R_{T'}(r) - R_T(r) \leq 0 \) (because \( W_{T'}(r) < W_T(r) \)), and thus

\[
R(T') \leq R(T) + 2(R_{T'}(u) - R_T(u))
\]
as wanted.

**Case 2:** The splay step is of the type depicted in Figure 2. We have,

\[
R(T') = R(T) + [(R_{T'}(u) - R_T(u)) + (R_{T'}(v) - R_T(v)) + (R_{T'}(w) - R_T(w))].
\]

But \( R_{T'}(w) - R_T(w) \leq 0 \) (because \( W_{T'}(w) < W_T(w) \)) and \( R_{T'}(v) - R_T(v) \leq R_{T'}(u) - R_T(u) \) (because \( W_{T'}(v) < W_T(v) \) and therefore \( R_{T'}(v) \leq R_T(u) \), and \( W_T(v) > W_T(u) \) and therefore \( R_T(v) \geq R_T(u) \)). Hence,

\[
R(T') \leq R(T) + 2(R_{T'}(u) - R_T(u))
\]
as wanted.

**Case 3:** The splay step is of the type depicted in Figure 3. We have,

\[
R(T') = R(T) + [(R_{T'}(u) - R_T(u)) + (R_{T'}(v) - R_T(v)) + (R_{T'}(w) - R_T(w))].
\]

But \( R_{T'}(u) - R_T(u) \geq 0 \) (because \( W_{T'}(u) > W_T(u) \)), \( R_{T'}(v) - R_T(v) \leq 0 \) (because \( W_{T'}(v) < W_T(v) \)) and \( R_{T'}(w) - R_T(w) \leq 0 \) (because \( W_{T'}(w) < W_T(w) \)). Hence,

\[
R(T') \leq R(T) + 2(R_{T'}(u) - R_T(u))
\]
as wanted. \( \square \)
Lemma 4. Let $v$ be the parent and $w$ the grandparent of node $u$ in $T$. Let $T'$ be the tree resulting after a splay step at $u$. It is not possible to have

$$R_T(u) = R_T(v) = R_T(w) = R_{T'}(u) = R_{T'}(v) = R_{T'}(w).$$

**Proof:** Suppose, by way of contradiction, that this was possible. Since $u$ has a grandparent, the splay step must be of the type depicted in Figure 2 or 3.

**Case 1:** The splay step is of the type depicted in Figure 2. Since $R_T(u) = R_T(w)$, we must have $W_T(u) ≥ W_T(w)/2$; and since $R_{T'}(u) = R_{T'}(w)$, we must have $W_{T'}(w) ≥ W_{T'}(u)/2$. But then,

$$W_T(w) = W_T(u) + W_T(w) + 1 \geq W_T(w)/2 + W_T(u)/2 + 1 \geq W_T(w)/2 + W_T(w)/2 + 1 = W_T(w) + 1,$$

a contradiction.

**Case 2:** The splay step is of the type depicted in Figure 3. Since $R_{T'}(v) = R_{T'}(u)$, we must have $W_{T'}(v) ≥ W_{T'}(u)/2$; and since $R_{T'}(w) = R_{T'}(u)$, we must have $W_{T'}(w) ≥ W_{T'}(u)/2$. But then,

$$W_{T'}(u) = W_{T'}(v) + W_{T'}(w) + 1 \geq W_{T'}(u)/2 + W_{T'}(u)/2 + 1 = W_{T'}(u) + 1,$$

a contradiction. □

**Lemma 5.** Let $T$ be the BST before and $T'$ the BST after a splay step at node $u$, where $u$ is not a child of the root in $T$ (hence, the splay step is of the type depicted in Figure 2 or 3). If $R_{T'}(u) = R_{T}(u)$ then $R(T') ≤ R(T) - 1$ (i.e. the rank of the tree actually decreases).

**Proof:** Let $v$, $w$ be the parent and grandparent of $u$, respectively. Then

$$R(T') = R(T) + [(R_{T'}(u) - R_T(u)) + (R_{T'}(v) - R_T(v)) + (R_{T'}(w) - R_T(w))]. \quad (2)$$

We also have,

$$R_{T'}(u) = R_T(u) ≤ R_T(v) ≤ R_T(w) = R_{T'}(u)$$

and hence

$$R_T(u) = R_T(v) = R_T(w) = R_{T'}(u). \quad (3)$$

So from (2),

$$R(T') = R(T) + [(R_{T'}(v) - R_{T'}(u)) + (R_{T'}(w) - R_{T'}(u))]. \quad (4)$$

Clearly,

$$R_{T'}(u) ≤ R_{T'}(u) \quad \text{and} \quad R_{T'}(w) ≤ R_{T'}(u). \quad (5)$$

By Lemma 4 and (3) it cannot be that $R_{T'}(v) = R_{T'}(w) = R_{T'}(u)$. This, together with (5) implies that $[(R_{T'}(v) - R_{T'}(u)) + (R_{T'}(w) - R_{T'}(u))] < 0$ and, together with (4) we get $R(T') ≤ R(T) - 1$, as wanted. □
Now we are fully equipped to give the proof of Lemma 2.

Proof of Lemma 2: Let \( u \) be the node where the splay operation occurs, and \( k \) be the number of splay steps in the splay operation. Consider the path from \( u \) to the root of \( T \) and let \( w_0 = u, w_1, w_2, \ldots, w_k = \) root of \( T \) be the nodes along that path where \( u \) is moved in the successive steps of the splay operation. (Thus, \( w_{j+1} \) is the grandparent of \( w_j \) for all \( j = 0, 1, \ldots, k - 1 \); and \( w_k \) is either the parent or the grandparent of \( w_{k-1} \), depending on whether the length of the path from \( u \) to the root of \( T \) is odd or even, accordingly.)

Let \( T_0 = T, T_1, T_2, \ldots, T_k = T' \) be the trees resulting after each splay step. By a straightforward induction on \( j = 0, 1, \ldots, k \) it is easy to see that \( W_{T_j}(u) = W_{T_j}(w_j) \) and, therefore,

\[
R_{T_j}(u) = R_{T_j}(w_j). \tag{6}
\]

Since the weights of \( w_0, w_1, \ldots, w_k \) in \( T \) monotonically (strictly) increase, we have,

\[
1 \leq R_T(w_0) \leq R_T(w_1) \leq \ldots \leq R_T(w_k) \leq \log n + 1.
\]

Therefore, we’ll have \( R_T(w_j) = R_T(w_{j+1}) \) and therefore, by (6), \( R_{T_j}(u) = R_{T_{j+1}}(u) \), for all but at most \( \log n \)’s, i.e. for at least \( (k - \log n) \)’s. In all of these, except maybe the last, \( u \) is not the child of the root. Thus, by Lemma 5,

\[
\text{for } \geq (k - \log n - 1) \text{ splay steps, the rank of the tree after that splay step will decrease}. \tag{7}
\]

Now,

\[
R(T') = R(T_k) = \left[ \sum_{j=1}^{k} R(T_j) - R(T_{j-1}) \right] + R(T_0)
\]

[by Lemma 3 and (7)]

\[
\leq R(T) + 2 \sum_{j=1}^{k} (R_{T_j}(u) - R_{T_{j-1}}(u)) - (k - \log n - 1)
\]

\[
= R(T) + 2 \sum_{j=1}^{k} (R_T(w_j) - R_T(w_{j-1})) - (k - \log n - 1) \tag{by (6)}
\]

\[
\leq R(T) + 2 (R_T(w_k) - R_T(w_0)) - (k - \log n - 1) \tag{since \( R_T(w_k) \leq \log n + 1 \) and \( R_T(w_0) \geq 1 \)}
\]

\[
\leq R(T) + 2 (\log n + 1 - 1) - (k - \log n - 1)
\]

\[
= R(T) + 3\log n - k + 1.
\]

Hence,

\[
k \leq R(T) - R(T') + 3\log n + 1,
\]

as wanted. \( \square \)
Comparison of AVL trees and self-adjusting BSTs

One advantage of self-adjusting BSTs over AVL trees is that they do not require any balance information to be maintained. This not only saves space but, possibly more significantly, makes the algorithms for the dictionary operations simpler to program.

Another advantage of self-adjusting trees is that, true to their name, their structure adapts to the access pattern dynamically. This means that (as in self-organising lists) keys frequently searched for will tend to be close to the root of the tree. If the access pattern changes over time, so will the structure of the tree. In fact, it is possible to prove that the time to process any (sufficiently long) sequence of search operations on a self-adjusting binary search tree is within a constant factor of the time needed to process that sequence on an optimal (static) binary search tree! (If interested, you can find a proof of this fact in the original paper on self-adjusting trees: Sleator and Tarjan, “Self-adjusting binary trees”, Proc. of the 15th Symposium on Theory of Computing, 1983, pp. 235-245.)

On the negative side, operations on self-adjusting trees cause more rotations than on AVL trees. Another potential disadvantage of self-adjusting trees is that, even though the overall time to perform a sequence of $m$ operations is (asymptotically) the same as in AVL trees, some individual operations may take much longer ($\Omega(n)$, rather than $O(\log n)$). This would be a problem in “interactive” applications where it is important to produce the result of each individual operation quickly because a user is waiting for it; it is not an issue in “batch” applications where the user is only interested in the result of the entire sequence of operations, not each one separately.
ANOTHER EXAMPLE OF UNIVERSAL SET OF HASH FUNCTIONS

Let the universe of keys $U = \{0,1,2,\ldots,2^k-1\}$ and the hash table size $m = 2^\ell$, for some $k, \ell \in \mathbb{N}$, $k \geq \ell$ (typically, $k \gg \ell$). Thus, a hash function must be a mapping of the form $h : U \to \{0,1,\ldots,2^\ell-1\}$. By considering the binary representation of the elements of $U$, we can think of each of them as a $k$-bit vector. Similarly, we can think of each slot $0,1,\ldots,2^\ell-1$ as an $\ell$-bit vector.

The set of $k$-bit vectors is $\mathbb{Z}_2^k$ (i.e., the $k$-fold Cartesian product of $\mathbb{Z}_2 = \{0,1\}$); similarly, the set of $\ell$-bit vectors is $\mathbb{Z}_2^\ell$. Therefore, a hash function in this case is of the form $h : \mathbb{Z}_2^k \to \mathbb{Z}_2^\ell$.

Consider now the set $\mathcal{H}$ of all linear transformations of $\mathbb{Z}_2^k$ to $\mathbb{Z}_2^\ell$. Recall that a linear transformation from $\mathbb{Z}_2^k$ to $\mathbb{Z}_2^\ell$ corresponds to an $\ell \times k$ matrix whose entries are elements of $\mathbb{Z}_2$. Therefore, we may think of the set $\mathcal{H}$ as the set of $\ell \times k$ bit matrices. Thus, a hash function $h \in \mathcal{H}$, maps a key $(x_1, x_2, \ldots, x_k)$ where each $x_i$ is a bit of the key into the slot number whose binary representation is $(s_1, s_2, \ldots, s_\ell)$, computed as

$$
\begin{pmatrix}
 b_{11} & b_{12} & \cdots & a_{1k} \\
 b_{21} & b_{22} & \cdots & a_{2k} \\
 \vdots & \vdots & \ddots & \vdots \\
 b_{k1} & b_{k2} & \cdots & a_{kk}
\end{pmatrix}
\begin{pmatrix}
 x_1 \\
 x_2 \\
 \vdots \\
 x_k
\end{pmatrix}
= 
\begin{pmatrix}
 s_1 \\
 s_2 \\
 \vdots \\
 s_\ell
\end{pmatrix}
$$

where each $b_{ij} \in \{0,1\}$, and the matrix consisting of the $b_{ij}$s corresponds to the hash function $h$.

There are $2^{k-\ell} k \times \ell$ bit matrices, so that $|\mathcal{H}| = 2^{k-\ell} \cdot (It is easy to verify that different matrices define different functions; you should do this.) To pick a random element of $\mathcal{H}$, we simply randomly pick each of the $k \cdot \ell$ bits of the matrix.

Claim. $\mathcal{H}$ is a universal set of hash functions.

Proof: Fix arbitrary $x, y \in U$, where $x \neq y$. We shall prove that there are exactly $2^{(k-1)\ell}$ functions $h \in \mathcal{H}$ so that $h(x) = h(y)$. Since $2^{(k-1)\ell} = 2^{\ell}/2^{k-\ell} = |\mathcal{H}|/m$, it follows that $\mathcal{H}$ is universal, as wanted.

Now suppose that for some function $h \in \mathcal{H}$ we have $h(x) = h(y)$ and therefore $h(x) - h(y) = 0$. Since $h$ is a linear function, $h(x - y) = 0$. (This follows from the fact that matrix multiplication distributes over matrix addition.) Therefore we have that

$$
\begin{pmatrix}
 b_{11} & b_{12} & \cdots & a_{1k} \\
 b_{21} & b_{22} & \cdots & a_{2k} \\
 \vdots & \vdots & \ddots & \vdots \\
 b_{k1} & b_{k2} & \cdots & a_{kk}
\end{pmatrix}
\begin{pmatrix}
 x_1 - y_1 \\
 x_2 - y_2 \\
 \vdots \\
 x_k - y_k
\end{pmatrix}
= 
\begin{pmatrix}
 0 \\
 0 \\
 \vdots \\
 0
\end{pmatrix}
$$

Since $x \neq y$, $x$ and $y$ differ in at least one bit. Without loss of generality, assume that they differ in the first bit, so that $x_1 - y_1 \neq 0$. Since this is subtraction in $\mathbb{Z}_2$, we must have $x_1 - y_1 = 1$. From (1) we have $\ell$ equations (in $\mathbb{Z}_2$), one for each $1 \leq i \leq \ell$:

$$
b_{11}(x_1 - y_1) + \sum_{j=2}^k b_{ij}(x_j - y_j) = 0 \iff b_{11} = -\sum_{j=2}^k b_{ij}(x_j - y_j).
$$

This means that if we fix the bits of all columns of the matrix except for the first column, we will have a unique choice (given by the above equations) for the bits of the first column that will satisfy (1). Therefore, for any choice of the bits of all columns except the first, there is exactly one function in $\mathcal{H}$ that will cause $x$ and $y$ to collide. Hence the number of $h \in \mathcal{H}$ so that $h(x) = h(y)$ is exactly the number of ways we can choose the bits in all columns of the matrix except the first, which is exactly $2^{(k-1)\ell}$, as wanted.

\[\square\]