CPS 230

DESIGN AND ANALYSIS OF ALGORITHMS

Fall 2008

Instructor: **Herbert Edelsbrunner**
Teaching Assistant: **Zhiqiang Gu**

Table of Contents

1 Introduction

Meetings. We meet twice a week, on Tuesdays and Thursdays, from 1:15 to 2:30pm, in room D106 LSRC.

Communication. The course material will be delivered in the two weekly lectures. A written record of the lectures will be available on the web, usually a day after the lecture. The web also contains other information, such as homework assignments, solutions, useful links, etc. The main supporting text is

TARJAN. *Data Structures and Network Algorithms.* SIAM, 1983.

The book focuses on fundamental data structures and graph algorithms, and additional topics covered in the course can be found in the lecture notes or other texts in algorithms such as

KLEINBERG AND TARDOS. *Algorithm Design.* Pearson Education, 2006.

Examinations. There will be a final exam (covering the material of the entire semester) and a midterm (at the beginning of October), You may want to freshen up your math skills before going into this course. The weighting of exams and homework used to determine your grades is

homework	35%,
midterm	25%,
final	40%.

Homework. We have seven homeworks scheduled throughout this semester, one per main topic covered in the course. The solutions to each homework are due one and a half weeks after the assignment. More precisely, they are due at the beginning of the third lecture after the assignment. The seventh homework may help you prepare for the final exam and solutions will not be collected.

Rule 1. The solution to any one homework question must fit on a single page (together with the statement of the problem).

Rule 2. The discussion of questions and solutions before the due date is not discouraged, but you must formulate your own solution.

Rule 3. The deadline for turning in solutions is 10 minutes after the beginning of the lecture on the due date.

Overview. The main topics to be covered in this course are

I Design Techniques;
II Searching;
III Prioritizing;
IV Graph Algorithms;
V Topological Algorithms;
VI Geometric Algorithms;
VII NP-completeness.

The emphasis will be on algorithm **design** and on algorithm **analysis**. For the analysis, we frequently need basic mathematical tools. Think of analysis as the measurement of the quality of your design. Just like you use your sense of taste to check your cooking, you should get into the habit of using algorithm analysis to justify design decisions when you write an algorithm or a computer program. This is a necessary step to reach the next level in mastering the art of programming. I encourage you to implement new algorithms and to compare the experimental performance of your program with the theoretical prediction gained through analysis.

I Design Techniques

2 Divide-and-Conquer

We use quicksort as an example for an algorithm that follows the divide-and-conquer paradigm. It has the reputation of being the fasted comparison-based sorting algorithm. Indeed it is very fast on the average but can be slow for some input, unless precautions are taken.

The algorithm. Quicksort follows the general paradigm of divide-and-conquer, which means it **divides** the unsorted array into two, it **recurses** on the two pieces, and it finally **combines** the two sorted pieces to obtain the sorted array. An interesting feature of quicksort is that the divide step separates small from large items. As a consequence, combining the sorted pieces happens automatically without doing anything extra.

```
void QUICKSORT(int $\ell, r$)
  if $\ell < r$ then $m = $ SPLIT$(\ell, r)$;
                     QUICKSORT$(\ell, m-1)$;
                     QUICKSORT$(m+1, r)$
  endif.
```

We assume the items are stored in $A[0..n-1]$. The array is sorted by calling QUICKSORT$(0, n-1)$.

Splitting. The performance of quicksort depends heavily on the performance of the split operation. The effect of splitting from ℓ to r is:

- $x = A[\ell]$ is moved to its correct location at $A[m]$;
- no item in $A[\ell..m-1]$ is larger than x;
- no item in $A[m+1..r]$ is smaller than x.

Figure 1 illustrates the process with an example. The nine items are split by moving a pointer i from left to right and another pointer j from right to left. The process stops when i and j cross. To get splitting right is a bit delicate, in particular in special cases. Make sure the algorithm is correct for (i) x is smallest item, (ii) x is largest item, (iii) all items are the same.

```
int SPLIT(int $\ell, r$)
  $x = A[\ell]$;  $i = \ell$;  $j = r + 1$;
  repeat repeat $i$++ until $x \le A[i]$;
         repeat $j$-- until $x \ge A[j]$;
         if $i < j$ then SWAP$(i, j)$ endif
  until $i \ge j$;
  SWAP$(\ell, j)$; return $j$.
```

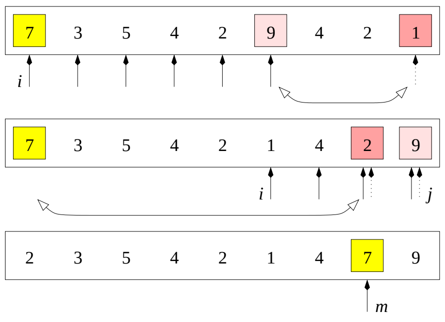

Figure 1: First, i and j stop at items 9 and 1, which are then swapped. Second, i and j cross and the pivot, 7, is swapped with item 2.

Special cases (i) and (iii) are ok but case (ii) requires a stopper at $A[r+1]$. This stopper must be an item at least as large as x. If $r < n-1$ this stopper is automatically given. For $r = n-1$, we create such a stopper by setting $A[n] = +\infty$.

Running time. The actions taken by quicksort can be expressed using a binary tree: each (internal) node represents a call and displays the length of the subarray; see Figure 2. The worst case occurs when A is already sorted.

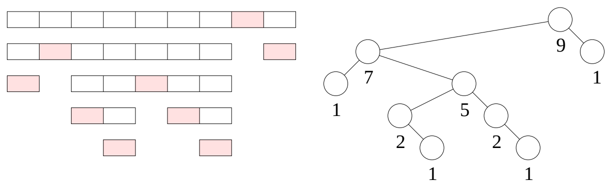

Figure 2: The total amount of time is proportional to the sum of lengths, which are the numbers of nodes in the corresponding subtrees. In the displayed case this sum is 29.

In this case the tree degenerates to a list without branching. The sum of lengths can be described by the following recurrence relation:

$$T(n) \quad = \quad n + T(n-1) \quad = \quad \sum_{i=1}^{n} i \quad = \quad \binom{n+1}{2}.$$

The running time in the worst case is therefore in $O(n^2)$.

In the best case the tree is completely balanced and the sum of lengths is described by the recurrence relation

$$T(n) \quad = \quad n + 2 \cdot T\left(\frac{n-1}{2}\right).$$

If we assume $n = 2^k - 1$ we can rewrite the relation as

$$\begin{aligned} U(k) &= (2^k-1) + 2 \cdot U(k-1) \\ &= (2^k-1) + 2(2^{k-1}-1) + \ldots + 2^{k-1}(2-1) \\ &= k \cdot 2^k - \sum_{i=0}^{k-1} 2^i \\ &= 2^k \cdot k - (2^k-1) \\ &= (n+1) \cdot \log_2(n+1) - n. \end{aligned}$$

The running time in the best case is therefore in $\mathrm{O}(n \log n)$.

Randomization. One of the drawbacks of quicksort, as described until now, is that it is slow on rather common almost sorted sequences. The reason are pivots that tend to create unbalanced splittings. Such pivots tend to occur in practice more often than one might expect. Human and often also machine generated data is frequently biased towards certain distributions (in this case, permutations), and it has been said that 80% of the time or more, sorting is done on either already sorted or almost sorted files. Such situations can often be helped by transferring the algorithm's dependence on the input data to internally made random choices. In this particular case, we use randomization to make the choice of the pivot independent of the input data. Assume RANDOM(ℓ, r) returns an integer $p \in [\ell, r]$ with uniform probability:

$$\text{Prob}[\text{RANDOM}(\ell, r) = p] = \frac{1}{r - \ell + 1}$$

for each $\ell \le p \le r$. In other words, each $p \in [\ell, r]$ is equally likely. The following algorithm splits the array with a random pivot:

```
int rSPLIT(int $\ell, r$)
  $p$ = RANDOM($\ell, r$); SWAP($\ell, p$);
  return SPLIT($\ell, r$).
```

We get a *randomized* implementation by substituting rSPLIT for SPLIT. The behavior of this version of quicksort depends on p, which is produced by a random number generator.

Average analysis. We assume that the items in $A[0..n-1]$ are pairwise different. The pivot splits A into

$$A[0..m-1], \quad A[m], \quad A[m+1..n-1].$$

By assumption on function rSPLIT, the probability for each $m \in [0, n-1]$ is $\frac{1}{n}$. Therefore the average sum of array lengths split by QUICKSORT is

$$T(n) = n + \frac{1}{n} \cdot \sum_{m=0}^{n-1} \left(T(m) + T(n-m-1) \right).$$

To simplify, we multiply with n and obtain a second relation by substituting $n-1$ for n:

$$n \cdot T(n) = n^2 + 2 \cdot \sum_{i=0}^{n-1} T(i), \qquad (1)$$

$$(n-1) \cdot T(n-1) = (n-1)^2 + 2 \cdot \sum_{i=0}^{n-2} T(i). \qquad (2)$$

Next we subtract (2) from (1), we divide by $n(n+1)$, we use repeated substitution to express $T(n)$ as a sum, and finally split the sum in two:

$$\begin{aligned} \frac{T(n)}{n+1} &= \frac{T(n-1)}{n} + \frac{2n-1}{n(n+1)} \\ &= \frac{T(n-2)}{n-1} + \frac{2n-3}{(n-1)n} + \frac{2n-1}{n(n+1)} \\ &= \sum_{i=1}^{n} \frac{2i-1}{i(i+1)} \\ &= 2 \cdot \sum_{i=1}^{n} \frac{1}{i+1} - \sum_{i=1}^{n} \frac{1}{i(i+1)}. \end{aligned}$$

Bounding the sums. The second sum is solved directly by transformation to a telescoping series:

$$\begin{aligned} \sum_{i=1}^{n} \frac{1}{i(i+1)} &= \sum_{i=1}^{n} \left(\frac{1}{i} - \frac{1}{i+1} \right) \\ &= 1 - \frac{1}{n+1}. \end{aligned}$$

The first sum is bounded from above by the integral of $\frac{1}{x}$ for x ranging from 1 to $n+1$; see Figure 3. The sum of $\frac{1}{i+1}$ is the sum of areas of the shaded rectangles, and because all rectangles lie below the graph of $\frac{1}{x}$ we get a bound for the total rectangle area:

$$\sum_{i=1}^{n} \frac{1}{i+1} < \int_1^{n+1} \frac{\mathrm{d}x}{x} = \ln(n+1).$$

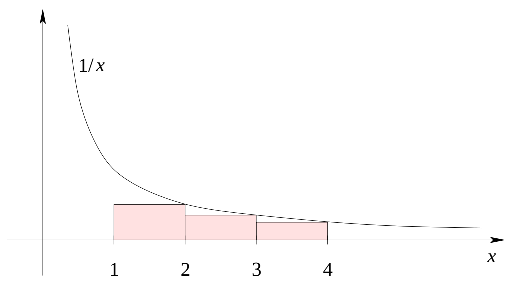

Figure 3: The areas of the rectangles are the terms in the sum, and the total rectangle area is bounded by the integral from 1 through $n+1$.

We plug this bound back into the expression for the average running time:

$$\begin{aligned} T(n) &< (n+1) \cdot \sum_{i=1}^{n} \frac{2}{i+1} \\ &< 2 \cdot (n+1) \cdot \ln(n+1) \\ &= \frac{2}{\log_2 e} \cdot (n+1) \cdot \log_2(n+1). \end{aligned}$$

In words, the running time of quicksort in the average case is only a factor of about $2/\log_2 e = 1.386\ldots$ slower than in the best case. This also implies that the worst case cannot happen very often, for else the average performance would be slower.

Stack size. Another drawback of quicksort is the recursion stack, which can reach a size of $\Omega(n)$ entries. This can be improved by always first sorting the smaller side and simultaneously removing the tail-recursion:

```
void QuickSort(int $\ell, r$)
  $i = \ell$; $j = r$;
  while $i < j$ do
    $m = $ rSplit$(i, j)$;
    if $m - i < j - m$
      then QuickSort$(i, m-1)$; $i = m + 1$
      else QuickSort$(m+1, j)$; $j = m - 1$
    endif
  endwhile.
```

In each recursive call to QuickSort, the length of the array is at most half the length of the array in the preceding call. This implies that at any moment of time the stack contains no more than $1 + \log_2 n$ entries. Note that without removal of the tail-recursion, the stack can reach $\Omega(n)$ entries even if the smaller side is sorted first.

Summary. Quicksort incorporates two design techniques to efficiently sort n numbers: divide-and-conquer for reducing large to small problems and randomization for avoiding the sensitivity to worst-case inputs. The average running time of quicksort is in $\mathrm{O}(n \log n)$ and the extra amount of memory it requires is in $\mathrm{O}(\log n)$. For the deterministic version, the average is over all $n!$ permutations of the input items. For the randomized version the average is the expected running time for *every* input sequence.

3 Prune-and-Search

We use two algorithms for selection as examples for the prune-and-search paradigm. The problem is to find the i-smallest item in an unsorted collection of n items. We could first sort the list and then return the item in the i-th position, but just finding the i-th item can be done faster than sorting the entire list. As a warm-up exercise consider selecting the 1-st or smallest item in the unsorted array $A[1..n]$.

```
$min = 1$;
for $j = 2$ to $n$ do
  if $A[j] < A[min]$ then $min = j$ endif
endfor.
```

The index of the smallest item is found in $n-1$ comparisons, which is optimal. Indeed, there is an adversary argument, that is, with fewer than $n-1$ comparisons we can change the minimum without changing the outcomes of the comparisons.

Randomized selection. We return to finding the i-smallest item for a fixed but arbitrary integer $1 \le i \le n$, which we call the *rank* of that item. We can use the splitting function of quicksort also for selection. As in quicksort, we choose a random pivot and split the array, but we recurse only for one of the two sides. We invoke the function with the range of indices of the current subarray and the rank of the desired item, i. Initially, the range consists of all indices between $\ell = 1$ and $r = n$, limits included.

```
int RSELECT(int $\ell, r, i$)
  $q$ = RSPLIT($\ell, r$); $m = q - \ell + 1$;
  if $i < m$ then return RSELECT($\ell, q-1, i$)
    elseif $i = m$ then return $q$
    else return RSELECT($q+1, r, i-m$)
  endif.
```

For small sets, the algorithm is relatively ineffective and its running time can be improved by switching over to sorting when the size drops below some constant threshold. On the other hand, each recursive step makes some progress so that termination is guaranteed even without special treatment of small sets.

Expected running time. For each $1 \le m \le n$, the probability that the array is split into subarrays of sizes $m-1$ and $n-m$ is $\frac{1}{n}$. For convenience we assume that n is even. The expected running time increases with increasing number of items, $T(k) \le T(m)$ if $k \le m$. Hence,

$$\begin{aligned} T(n) &\le n + \frac{1}{n}\sum_{m=1}^{n} \max\{T(m-1), T(n-m)\} \\ &\le n + \frac{2}{n}\sum_{m=\frac{n}{2}+1}^{n} T(m-1). \end{aligned}$$

Assume inductively that $T(m) \le cm$ for $m < n$ and a sufficiently large positive constant c. Such a constant c can certainly be found for $m = 1$, since for that case the running time of the algorithm is only a constant. This establishes the basis of the induction. The case of n items reduces to cases of $m < n$ items for which we can use the induction hypothesis. We thus get

$$\begin{aligned} T(n) &\le n + \frac{2c}{n}\sum_{m=\frac{n}{2}+1}^{n} m-1 \\ &= n + c\cdot(n-1) - \frac{c}{2}\cdot\left(\frac{n}{2}+1\right) \\ &= n + \frac{3c}{4}\cdot n - \frac{3c}{2}. \end{aligned}$$

Assuming $c \ge 4$ we thus have $T(n) \le cn$ as required. Note that we just proved that the expected running time of RSELECT is only a small constant times that of RSPLIT. More precisely, that constant factor is no larger than four.

Deterministic selection. The randomized selection algorithm takes time proportional to n^2 in the worst case, for example if each split is as unbalanced as possible. It is however possible to select in O(n) time even in the worst case. The *median* of the set plays a special role in this algorithm. It is defined as the i-smallest item where $i = \frac{n+1}{2}$ if n is odd and $i = \frac{n}{2}$ or $\frac{n+2}{2}$ if n is even. The deterministic algorithm takes five steps to select:

`Step 1.` Partition the n items into $\lceil \frac{n}{5} \rceil$ groups of size at most 5 each.

`Step 2.` Find the median in each group.

`Step 3.` Find the median of the medians recursively.

`Step 4.` Split the array using the median of the medians as the pivot.

`Step 5.` Recurse on one side of the pivot.

It is convenient to define $k = \lceil \frac{n}{5} \rceil$ and to partition such that each group consists of items that are multiples of k positions apart. This is what is shown in Figure 4 provided we arrange the items row by row in the array.

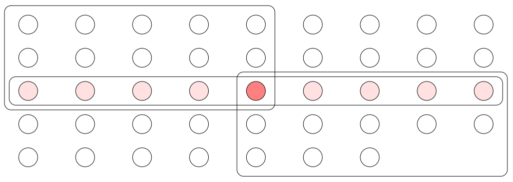

Figure 4: The 43 items are partitioned into seven groups of 5 and two groups of 4, all drawn vertically. The shaded items are the medians and the dark shaded item is the median of medians.

Implementation with insertion sort. We use insertion sort on each group to determine the medians. Specifically, we sort the items in positions $\ell, \ell+k, \ell+2k, \ell+3k, \ell+4k$ of array A, for each ℓ.

```
void ISORT(int $\ell, k, n$)
  $j = \ell + k$;
  while $j \leq n$ do $i = j$;
    while $i > \ell$ and $A[i] > A[i-k]$ do
      SWAP$(i, i-k)$;  $i = i - k$
    endwhile;
    $j = j + k$
  endwhile.
```

Although insertion sort takes quadratic time in the worst case, it is very fast for small arrays, as in this application. We can now combine the various pieces and write the selection algorithm in pseudo-code. Starting with the code for the randomized algorithm, we first remove the randomization and second add code for Steps 1, 2, and 3. Recall that i is the rank of the desired item in $A[\ell..r]$. After sorting the groups, we have their medians arranged in the middle fifth of the array, $A[\ell+2k..\ell+3k-1]$, and we compute the median of the medians by recursive application of the function.

```
int SELECT(int $\ell, r, i$)
  $k = \lceil (r - \ell + 1)/5 \rceil$;
  for $j = 0$ to $k - 1$ do ISORT$(\ell + j, k, r)$ endfor;
  $m' = $ SELECT$(\ell + 2k, \ell + 3k - 1, \lfloor (k+1)/2 \rfloor)$;
  SWAP$(\ell, m')$;  $q = $ SPLIT$(\ell, r)$;  $m = q - \ell + 1$;
  if $i < m$ then return SELECT$(\ell, q - 1, i)$
    elseif $i = m$ then return $q$
    else return SELECT$(q + 1, r, i - m)$
  endif.
```

Observe that the algorithm makes progress as long as there are at least three items in the set, but we need special treatment of the cases of one or of two items. The role of the median of medians is to prevent an unbalanced split of the array so we can safely use the deterministic version of splitting.

Worst-case running time. To simplify the analysis, we assume that n is a multiple of 5 and ignore ceiling and floor functions. We begin by arguing that the number of items less than or equal to the median of medians is at least $\frac{3n}{10}$. These are the first three items in the sets with medians less than or equal to the median of medians. In Figure 4, these items are highlighted by the box to the left and above but containing the median of medians. Symmetrically, the number of items greater than or equal to the median of medians is at least $\frac{3n}{10}$. The first recursion works on a set of $\frac{n}{5}$ medians, and the second recursion works on a set of at most $\frac{7n}{10}$ items. We have

$$T(n) \leq n + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right).$$

We prove $T(n) = \mathrm{O}(n)$ by induction assuming $T(m) \leq c \cdot m$ for $m < n$ and c a large enough constant.

$$\begin{aligned} T(n) &\leq n + \frac{c}{5} \cdot n + \frac{7c}{10} \cdot n \\ &= \left(1 + \frac{9c}{10}\right) \cdot n. \end{aligned}$$

Assuming $c \geq 10$ we have $T(n) \leq cn$, as required. Again the running time is at most some constant times that of splitting the array. The constant is about two and a half times the one for the randomized selection algorithm.

A somewhat subtle issue is the presence of equal items in the input collection. Such occurrences make the function SPLIT unpredictable since they could occur on either side of the pivot. An easy way out of the dilemma is to make sure that the items that are equal to the pivot are treated as if they were smaller than the pivot if they occur in the first half of the array and they are treated as if they were larger than the pivot if they occur in the second half of the array.

Summary. The idea of prune-and-search is very similar to divide-and-conquer, which is perhaps the reason why some textbooks make no distinction between the two. The characteristic feature of prune-and-search is that the recursion covers only a constant fraction of the input set. As we have seen in the analysis, this difference implies a better running time.

It is interesting to compare the randomized with the deterministic version of selection:

- the use of randomization leads to a simpler algorithm but it requires a source of randomness;
- upon repeating the algorithm for the same data, the deterministic version goes through the exact same steps while the randomized version does not;
- we analyze the worst-case running time of the deterministic version and the expected running time (for the worst-case input) of the randomized version.

All three differences are fairly universal and apply to other algorithms for which we have the choice between a deterministic and a randomized implementation.

4 Dynamic Programming

Sometimes, divide-and-conquer leads to overlapping subproblems and thus to redundant computations. It is not uncommon that the redundancies accumulate and cause an exponential amount of wasted time. We can avoid the waste using a simple idea: **solve each subproblem only once**. To be able to do that, we have to add a certain amount of book-keeping to remember subproblems we have already solved. The technical name for this design paradigm is *dynamic programming*.

Edit distance. We illustrate dynamic programming using the edit distance problem, which is motivated by questions in genetics. We assume a finite set of *characters* or *letters*, Σ, which we refer to as the *alphabet*, and we consider *strings* or *words* formed by concatenating finitely many characters from the alphabet. The *edit distance* between two words is the minimum number of letter insertions, letter deletions, and letter substitutions required to transform one word to the other. For example, the edit distance between `FOOD` and `MONEY` is at most four:

$$\texttt{FOOD} \longrightarrow \texttt{MOOD} \longrightarrow \texttt{MOND} \longrightarrow \texttt{MONED} \longrightarrow \texttt{MONEY}$$

A better way to display the editing process is the *gap representation* that places the words one above the other, with a gap in the first word for every insertion and a gap in the second word for every deletion:

```
F O O   D
M O N E Y
```

Columns with two different characters correspond to substitutions. The number of editing steps is therefore the number of columns that do not contain the same character twice.

Prefix property. It is not difficult to see that you cannot get from `FOOD` to `MONEY` in less than four steps. However, for longer examples it seems considerably more difficult to find the minimum number of steps or to recognize an optimal edit sequence. Consider for example

```
A L G O R   I   T H M
A L   T R U I S T I C
```

Is this optimal or, equivalently, is the edit distance between `ALGORITHM` and `ALTRUISTIC` six? Instead of answering this specific question, we develop a dynamic programming algorithm that computes the edit distance between an m-character string $A[1..m]$ and an n-character string $B[1..n]$. Let $E(i,j)$ be the edit distance between the prefixes of length i and j, that is, between $A[1..i]$ and $B[1..j]$. The edit distance between the complete strings is therefore $E(m,n)$. A crucial step towards the development of this algorithm is the following observation about the gap representation of an optimal edit sequence.

PREFIX PROPERTY. If we remove the last column of an optimal edit sequence then the remaining columns represent an optimal edit sequence for the remaining substrings.

We can easily prove this claim by contradiction: if the substrings had a shorter edit sequence, we could just glue the last column back on and get a shorter edit sequence for the original strings.

Recursive formulation. We use the Prefix Property to develop a recurrence relation for E. The dynamic programming algorithm will be a straightforward implementation of that relation. There are a couple of obvious base cases:

- **Erasing:** we need i deletions to erase an i-character string, $E(i,0) = i$.
- **Creating:** we need j insertions to create a j-character string, $E(0,j) = j$.

In general, there are four possibilities for the last column in an optimal edit sequence.

- **Insertion:** the last entry in the top row is empty, $E(i,j) = E(i,j-1)+1$.
- **Deletion:** the last entry in the bottom row is empty, $E(i,j) = E(i-1,j)+1$.
- **Substitution:** both rows have characters in the last column that are different, $E(i,j) = E(i-1,j-1)+1$.
- **No action:** both rows end in the same character, $E(i,j) = E(i-1,j-1)$.

Let P be the logical proposition $A[i] \neq B[j]$ and denote by $|P|$ its indicator variable: $|P| = 1$ if P is true and $|P| = 0$ if P is false. We can now summarize and for $i,j > 0$ get the edit distance as the smallest of the possibilities:

$$E(i,j) \quad = \quad \min \left\{ \begin{array}{l} E(i,j-1)+1 \\ E(i-1,j)+1 \\ E(i-1,j-1)+|P| \end{array} \right\}.$$

The algorithm. If we turned this recurrence relation directly into a divide-and-conquer algorithm, we would have the following recurrence for the running time:

$$\begin{aligned} T(m,n) &= T(m,n-1) + T(m-1,n) \\ &\quad + T(m-1,n-1) + 1. \end{aligned}$$

The solution to this recurrence is exponential in m and n, which is clearly not the way to go. Instead, let us build an $m+1$ times $n+1$ table of possible values of $E(i,j)$. We can start by filling in the base cases, the entries in the 0-th row and column. To fill in any other entry, we need to know the values directly to the left, directly above, and both to the left and above. If we fill the table from top to bottom and from left to right then whenever we reach an entry, the entries it depends on are already available.

```
int EditDistance(int $m, n$)
  for $i = 0$ to $m$ do $E[i,0] = i$ endfor;
  for $j = 1$ to $n$ do $E[0,j] = j$ endfor;
  for $i = 1$ to $m$ do
    for $j = 1$ to $n$ do
      $E[i,j] = \min\{E[i,j-1]+1, E[i-1,j]+1,$
                  $E[i-1,j-1] + |A[i] \neq B[j]|\}$
    endfor
  endfor;
  return $E[m,n]$.
```

Since there are $(m+1)(n+1)$ entries in the table and each takes a constant time to compute, the total running time is in O(mn).

An example. The table constructed for the conversion of `ALGORITHM` to `ALTRUISTIC` is shown in Figure 5. Boxed numbers indicate places where the two strings have equal characters. The arrows indicate the predecessors that define the entries. Each direction of arrow corresponds to a different edit operation: horizontal for insertion, vertical for deletion, and diagonal for substitution. Dotted diagonal arrows indicate free substitutions of a letter for itself.

Recovering the edit sequence. By construction, there is at least one path from the upper left to the lower right corner, but often there will be several. Each such path describes an optimal edit sequence. For the example at hand, we have three optimal edit sequences:

```
A L G O R I   T H M
A L T R U I S T I C
```

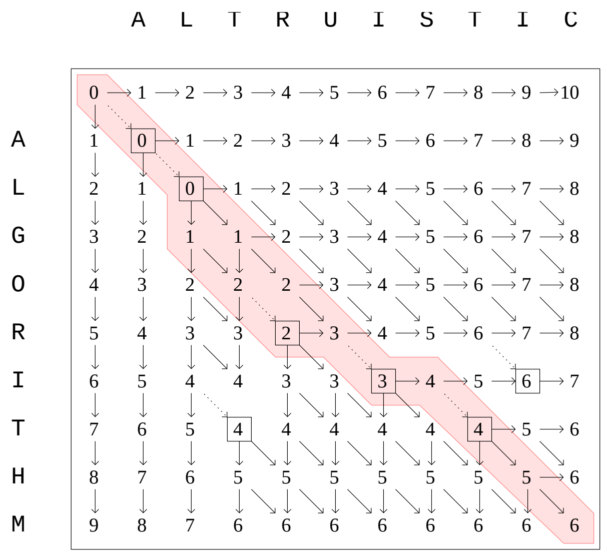

Figure 5: The table of edit distances between all prefixes of `ALGORITHM` and of `ALTRUISTIC`. The shaded area highlights the optimal edit sequences, which are paths from the upper left to the lower right corner.

```
A L G O R   I   T H M
A L T   R U I S T I C

A L G O R   I   T H M
A L   T R U I S T I C
```

They are easily recovered by tracing the paths backward, from the end to the beginning. The following algorithm recovers an optimal solution that also minimizes the number of insertions and deletions. We call it with the lengths of the strings as arguments, R(m,n).

```
void R(int $i, j$)
  if $i > 0$ or $j > 0$ then
    switch incoming arrow:
      case $\searrow$: R$(i-1, j-1)$; print$(A[i], B[j])$
      case $\downarrow$: R$(i-1, j)$; print$(A[i], \_)$
      case $\rightarrow$: R$(i, j-1)$; print$(\_, B[j])$.
    endswitch
  endif.
```

Summary. The structure of dynamic programming is again similar to divide-and-conquer, except that the subproblems to be solved overlap. As a consequence, we get different recursive paths to the same subproblems. To develop a dynamic programming algorithm that avoids redundant solutions, we generally proceed in two steps:

1. We formulate the problem recursively. In other words, we write down the answer to the whole problem as a combination of the answers to smaller subproblems.
2. We build solutions from bottom up. Starting with the base cases, we work our way up to the final solution and (usually) store intermediate solutions in a table.

For dynamic programming to be effective, we need a structure that leads to at most some polynomial number of different subproblems. Most commonly, we deal with sequences, which have linearly many prefixes and suffixes and quadratically many contiguous substrings.

5 Greedy Algorithms

The philosophy of being greedy is shortsightedness. Always go for the seemingly best next thing, always optimize the presence, without any regard for the future, and never change your mind about the past. The greedy paradigm is typically applied to optimization problems. In this section, we first consider a scheduling problem and second the construction of optimal codes.

A scheduling problem. Consider a set of activities $1, 2, \ldots, n$. Activity i starts at time s_i and finishes at time $f_i > s_i$. Two activities i and j *overlap* if $[s_i, f_i] \cap [s_j, f_j] \neq \emptyset$. The objective is to select a maximum number of pairwise non-overlapping activities. An example is shown in Figure 6. The largest number of activities can be scheduled by choosing activities with early finish times first. We first sort and reindex such that $i < j$ implies $f_i \leq f_j$.

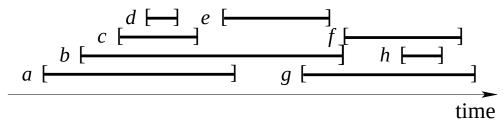

Figure 6: A best schedule is c, e, f, but there are also others of the same size.

```
$S = \{1\}$; $last = 1$;
for $i = 2$ to $n$ do
  if $f_{last} < s_i$ then
    $S = S \cup \{i\}$; $last = i$
  endif
endfor.
```

The running time is O($n \log n$) for sorting plus O(n) for the greedy collection of activities.

It is often difficult to determine how close to the optimum the solutions found by a greedy algorithm really are. However, for the above scheduling problem the greedy algorithm always finds an optimum. For the proof let $1 = i_1 < i_2 < \ldots < i_k$ be the greedy schedule constructed by the algorithm. Let $j_1 < j_2 < \ldots < j_\ell$ be any other feasible schedule. Since $i_1 = 1$ has the earliest finish time of any activity, we have $f_{i_1} \leq f_{j_1}$. We can therefore add i_1 to the feasible schedule and remove at most one activity, namely j_1. Among the activities that do not overlap i_1, i_2 has the earliest finish time, hence $f_{i_2} \leq f_{j_2}$. We can again add i_2 to the feasible schedule and remove at most one activity, namely j_2 (or possibly j_1 if it was not removed before). Eventually, we replace the entire feasible schedule by the greedy schedule without decreasing the number of activities. Since we could have started with a maximum feasible schedule, we conclude that the greedy schedule is also maximum.

Binary codes. Next we consider the problem of encoding a text using a string of 0s and 1s. A *binary code* maps each letter in the alphabet of the text to a unique string of 0s and 1s. Suppose for example that the letter ‘t’ is encoded as ‘001’, ‘h’ is encoded as ‘101’, and ‘e’ is encoded as ‘01’. Then the word ‘the’ would be encoded as the concatenation of codewords: ‘00110101’. This particular encoding is unambiguous because the code is *prefix-free*: no codeword is prefix of another codeword. There is a one-to-one correspondence between prefix-free binary codes and binary trees where each leaf is a letter and the corresponding codeword is the path from the root to that leaf. Figure 7 illustrates the correspondence for the above 3-letter code. Being prefix-free corresponds to leaves not having children. The tree in Figure 7 is not full because three of its internal nodes have only one child. This is an indication of waste. The code can be improved by replacing each node with one child by its child. This changes the above code to ‘00’ for ‘t’, ‘1’ for ‘h’, and ‘01’ for ‘e’.

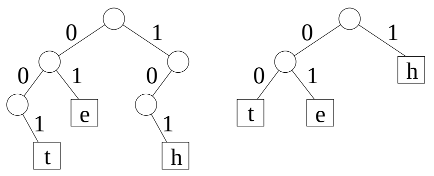

Figure 7: Letters correspond to leaves and codewords correspond to maximal paths. A left edge is read as ‘0’ and a right edge as ‘1’. The tree to the right is full and improves the code.

Huffman trees. Let w_i be the frequency of the letter c_i in the given text. It will be convenient to refer to w_i as the *weight* of c_i or of its external node. To get an efficient code, we choose short codewords for common letters. Suppose δ_i is the length of the codeword for c_i. Then the number of bits for encoding the entire text is

$$P = \sum_i w_i \cdot \delta_i.$$

Since δ_i is the depth of the leaf c_i, P is also known as the *weighted external path length* of the corresponding tree.

The *Huffman tree* for the c_i minimizes the weighted external path length. To construct this tree, we start with n nodes, one for each letter. At each stage of the algorithm, we greedily pick the two nodes with smallest weights and make them the children of a new node with weight equal to the sum of two weights. We repeat until only one node remains. The resulting tree for a collection of nine letters with displayed weights is shown in Figure 8. Ties that

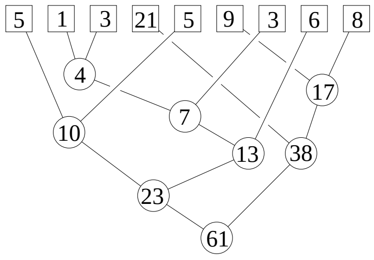

Figure 8: The numbers in the external nodes (squares) are the weights of the corresponding letters, and the ones in the internal nodes (circles) are the weights of these nodes. The Huffman tree is full by construction.

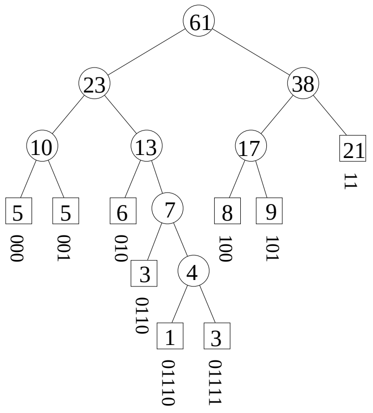

Figure 9: The weighted external path length is $15 + 15 + 18 + 12 + 5 + 15 + 24 + 27 + 42 = 173$.

arise during the algorithm are broken arbitrarily. We redraw the tree and order the children of a node as left and right child arbitrarily, as shown in Figure 9.

The algorithm works with a collection N of nodes which are the roots of the trees constructed so far. Initially, each leaf is a tree by itself. We denote the weight of a node by $w(\mu)$ and use a function EXTRACTMIN that returns the node with the smallest weight and, at the same time, removes this node from the collection.

```
Tree HUFFMAN
  loop $\mu = $ EXTRACTMIN$(N)$;
        if $N = \emptyset$ then return $\mu$ endif;
        $\nu = $ EXTRACTMIN$(N)$;
        create node $\kappa$ with children $\mu$ and $\nu$
            and weight $w(\kappa) = w(\mu) + w(\nu)$;
        add $\kappa$ to $N$
  forever.
```

Straightforward implementations use an array or a linked list and take time O(n) for each operation involving N. There are fewer than $2n$ extractions of the minimum and fewer than n additions, which implies that the total running time is O(n^2). We will see later that there are better ways to implement N leading to running time O($n \log n$).

An inequality. We prepare the proof that the Huffman tree indeed minimizes the weighted external path length. Let T be a full binary tree with weighted external path length $P(T)$. Let $\Lambda(T)$ be the set of leaves and let μ and ν be any two leaves with smallest weights. Then we can construct a new tree T' with

(1) set of leaves $\Lambda(T') = (\Lambda(T) - \{\mu, \nu\}) \dot\cup \{\kappa\}$,

(2) $w(\kappa) = w(\mu) + w(\nu)$,

(3) $P(T') \leq P(T) - w(\mu) - w(\nu)$, with equality if μ and ν are siblings.

We now argue that T' really exists. If μ and ν are siblings then we construct T' from T by removing μ and ν and declaring their parent, κ, as the new leaf. Then

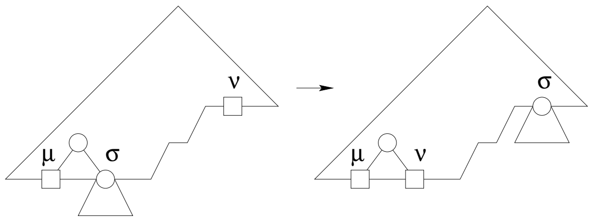

Figure 10: The increase in the depth of ν is compensated by the decrease in depth of the leaves in the subtree of σ.

$$\begin{aligned} P(T') &= P(T) - w(\mu)\delta - w(\nu)\delta + w(\kappa)(\delta - 1) \\ &= P(T) - w(\mu) - w(\nu), \end{aligned}$$

where $\delta = \delta(\mu) = \delta(\nu) = \delta(\kappa) + 1$ is the common depth of μ and ν. Otherwise, assume $\delta(\mu) \geq \delta(\nu)$ and let σ be the sibling of μ, which may or may not be a leaf. Exchange ν and σ. Since the length of the path from the root to σ is at least as long as the path to μ, the weighted external path length can only decrease; see Figure 10. Then do the same as in the other case.

Proof of optimality. The optimality of the Huffman tree can now be proved by induction.

HUFFMAN TREE THEOREM. Let T be the Huffman tree and X another tree with the same set of leaves and weights. Then $P(T) \leq P(X)$.

PROOF. If there are only two leaves then the claim is obvious. Otherwise, let μ and ν be the two leaves selected by the algorithm. Construct trees T' and X' with

$$\begin{aligned} P(T') &= P(T) - w(\mu) - w(\nu), \\ P(X') &\leq P(X) - w(\mu) - w(\nu). \end{aligned}$$

T' is the Huffman tree for $n-1$ leaves so we can use the inductive assumption and get $P(T') \leq P(X')$. It follows that

$$\begin{aligned} P(T) &= P(T') + w(\mu) + w(\nu) \\ &\leq P(X') + w(\mu) + w(\nu) \\ &\leq P(X). \end{aligned}$$

$\sqcup\!\!\!\!\sqcap$

Huffman codes are binary codes that correspond to Huffman trees as described. They are commonly used to compress text and other information. Although Huffman codes are optimal in the sense defined above, there are other codes that are also sensitive to the frequency of sequences of letters and this way outperform Huffman codes for general text.

Summary. The greedy algorithm for constructing Huffman trees works bottom-up by stepwise merging, rather than top-down by stepwise partitioning. If we run the greedy algorithm backwards, it becomes very similar to dynamic programming, except that it pursues only one of many possible partitions. Often this implies that it leads to suboptimal solutions. Nevertheless, there are problems that exhibit enough structure that the greedy algorithm succeeds in finding an optimum, and the scheduling and coding problems described above are two such examples.

First Homework Assignment

Write the solution to each problem on a single page. The deadline for handing in solutions is September 18.

Problem 1. (20 points). Consider two sums, $X = x_1 + x_2 + \ldots + x_n$ and $Y = y_1 + y_2 + \ldots + y_m$. Give an algorithm that finds indices i and j such that swapping x_i with y_j makes the two sums equal, that is, $X - x_i + y_j = Y - y_j + x_i$, if they exist. Analyze your algorithm. (You can use sorting as a subroutine. The amount of credit depends on the correctness of the analysis and the running time of your algorithm.)

Problem 2. ($20 = 10 + 10$ points). Consider distinct items $x_1, x_2, \ldots, x_n$ with positive weights $w_1, w_2, \ldots, w_n$ such that $\sum_{i=1}^{n} w_i = 1.0$. The *weighted median* is the item x_k that satisfies

$$\sum_{x_i < x_k} w_i < 0.5 \quad \text{and} \quad \sum_{x_j > x_k} w_j \leq 0.5.$$

(a) Show how to compute the weighted median of n items in worst-case time O($n \log n$) using sorting.

(b) Show how to compute the weighted median in worst-case time O(n) using a linear-time median algorithm.

Problem 3. ($20 = 6 + 14$ points). A game-board has n columns, each consisting of a top number, the cost of visiting the column, and a bottom number, the maximum number of columns you are allowed to jump to the right. The top number can be any positive integer, while the bottom number is either 1, 2, or 3. The objective is to travel from the first column off the board, to the right of the nth column. The cost of a game is the sum of the costs of the visited columns.

Assuming the board is represented in a two-dimensional array, $B[2, n]$, the following recursive procedure computes the cost of the cheapest game:

```
int CHEAPEST(int $i$)
  if $i > n$ then return 0 endif;
  $x = B[1, i] + \text{CHEAPEST}(i + 1)$;
  $y = B[1, i] + \text{CHEAPEST}(i + 2)$;
  $z = B[1, i] + \text{CHEAPEST}(i + 3)$;
  case $B[2, i] = 1$: return $x$;
       $B[2, i] = 2$: return $\min\{x, y\}$;
       $B[2, i] = 3$: return $\min\{x, y, z\}$
  endcase.
```

(a) Analyze the asymptotic running time of the procedure.

(b) Describe and analyze a more efficient algorithm for finding the cheapest game.

Problem 4. ($20 = 10 + 10$ points). Consider a set of n intervals $[a_i, b_i]$ that cover the unit interval, that is, $[0, 1]$ is contained in the union of the intervals.

(a) Describe an algorithm that computes a minimum subset of the intervals that also covers $[0, 1]$.

(b) Analyze the running time of your algorithm.

(For question (b) you get credit for the correctness of your analysis but also for the running time of your algorithm. In other words, a fast algorithm earns you more points than a slow algorithm.)

Problem 5. ($20 = 7 + 7 + 6$ points). Let $A[1..m]$ and $B[1..n]$ be two strings.

(a) Modify the dynamic programming algorithm for computing the edit distance between A and B for the case in which there are only two allowed operations, insertions and deletions of individual letters.

(b) A (not necessarily contiguous) *subsequence* of A is defined by the increasing sequence of its indices, $1 \leq i_1 < i_2 < \ldots < i_k \leq m$. Use dynamic programming to find the longest common subsequence of A and B and analyze its running time.

(c) What is the relationship between the edit distance defined in (a) and the longest common subsequence computed in (b)?

II SEARCHING

6 Binary Search Trees

One of the purposes of sorting is to facilitate fast searching. However, while a sorted sequence stored in a linear array is good for searching, it is expensive to add and delete items. Binary search trees give you the best of both worlds: fast search and fast update.

Definitions and terminology. We begin with a recursive definition of the most common type of tree used in algorithms. A *(rooted) binary tree* is either empty or a node (the *root*) with a binary tree as left subtree and binary tree as right subtree. We store items in the nodes of the tree. It is often convenient to say the items *are* the nodes. A binary tree is sorted if each item is between the smaller or equal items in the left subtree and the larger or equal items in the right subtree. For example, the tree illustrated in Figure 11 is sorted assuming the usual ordering of English characters. Terms for relations between family members such as *child*, *parent*, *sibling* are also used for nodes in a tree. Every node has one parent, except the root which has no parent. A *leaf* or *external node* is one without children; all other nodes are *internal*. A node ν is a *descendent* of μ if $\nu = \mu$ or ν is a descendent of a child of μ. Symmetrically, μ is an *ancestor* of ν if ν is a descendent of μ. The *subtree* of μ consists of all descendents of μ. An *edge* is a parent-child pair.

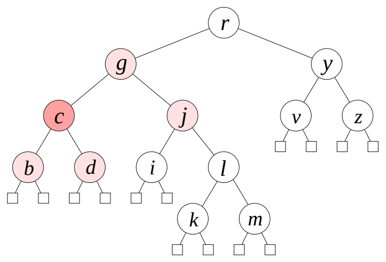

Figure 11: The parent, sibling and two children of the dark node are shaded. The internal nodes are drawn as circles while the leaves are drawn as squares.

The *size* of the tree is the number of nodes. A binary tree is *full* if every internal node has two children. Every full binary tree has one more leaf than internal node. To count its edges, we can either count 2 for each internal node or 1 for every node other than the root. Either way, the total number of edges is one less than the size of the tree. A *path* is a sequence of contiguous edges without repetitions. Usually we only consider paths that descend or paths that ascend. The *length* of a path is the number of edges. For every node μ, there is a unique path from the root to μ. The length of that path is the *depth* of μ. The *height* of the tree is the maximum depth of any node. The *path length* is the sum of depths over all nodes, and the *external path length* is the same sum restricted to the leaves in the tree.

Searching. A *binary search tree* is a sorted binary tree. We assume each node is a record storing an item and pointers to two children:

```
struct Node{item $info$; Node $*\ell$, $*r$};
typedef Node $*$ Tree.
```

Sometimes it is convenient to also store a pointer to the parent, but for now we will do without. We can search in a binary search tree by tracing a path starting at the root.

```
Node $*$ SEARCH(Tree $\varrho$, item $x$)
  case $\varrho$ = NULL: return NULL;
        $x < \varrho \to info$: return SEARCH($\varrho \to \ell, x$);
        $x = \varrho \to info$: return $\varrho$;
        $x > \varrho \to info$: return SEARCH($\varrho \to r, x$)
  endcase.
```

The running time depends on the length of the path, which is at most the height of the tree. Let n be the size. In the worst case the tree is a linked list and searching takes time $O(n)$. In the best case the tree is perfectly balanced and searching takes only time $O(\log n)$.

Insert. To add a new item is similarly straightforward: follow a path from the root to a leaf and replace that leaf by a new node storing the item. Figure 12 shows the tree obtained after adding w to the tree in Figure 11. The running time depends again on the length of the path. If the insertions come in a random order then the tree is usually

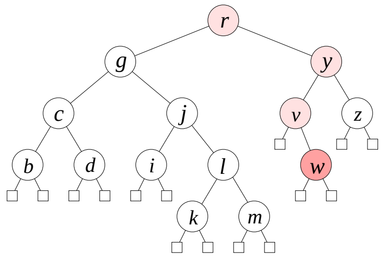

Figure 12: The shaded nodes indicate the path from the root we traverse when we insert w into the sorted tree.

close to being perfectly balanced. Indeed, the tree is the same as the one that arises in the analysis of quicksort. The expected number of comparisons for a (successful) search is one n-th of the expected running time of quicksort, which is roughly $2 \ln n$.

Delete. The main idea for deleting an item is the same as for inserting: follow the path from the root to the node ν that stores the item.

`Case 1.` ν has no internal node as a child. Remove ν.

`Case 2.` ν has one internal child. Make that child the child of the parent of ν.

`Case 3.` ν has two internal children. Find the rightmost internal node in the left subtree, remove it, and substitute it for ν, as shown in Figure 13.

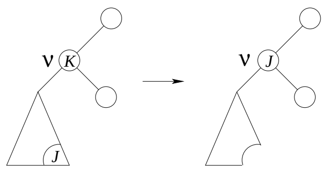

Figure 13: Store J in ν and delete the node that used to store J.

The analysis of the expected search time in a binary search tree constructed by a random sequence of insertions and deletions is considerably more challenging than if no deletions are present. Even the definition of a random sequence is ambiguous in this case.

Optimal binary search trees. Instead of hoping the incremental construction yields a shallow tree, we can construct the tree that minimizes the search time. We consider the common problem in which items have different probabilities to be the target of a search. For example, some words in the English dictionary are more commonly searched than others and are therefore assigned a higher probability. Let $a_1 < a_2 < \ldots < a_n$ be the items and p_i the corresponding probabilities. To simplify the discussion, we only consider successful searches and thus assume $\sum_{i=1}^{n} p_i = 1$. The expected number of comparisons for a successful search in a binary search tree T storing the n items is

$$\begin{aligned} 1 + C(T) &= \sum_{i=1}^{n} p_i \cdot (\delta_i + 1) \\ &= 1 + \sum_{i=1}^{n} p_i \cdot \delta_i, \end{aligned}$$

where δ_i is the depth of the node that stores a_i. $C(T)$ is the *weighted path length* or the *cost* of T. We study the problem of constructing a tree that minimizes the cost. To develop an example, let $n = 3$ and $p_1 = \frac{1}{2}$, $p_2 = \frac{1}{3}$, $p_3 = \frac{1}{6}$. Figure 14 shows the five binary trees with three nodes and states their costs. It can be shown that the number of different binary trees with n nodes is $\frac{1}{n+1}\binom{2n}{n}$, which is exponential in n. This is far too large to try all possibilities, so we need to look for a more efficient way to construct an optimum tree.

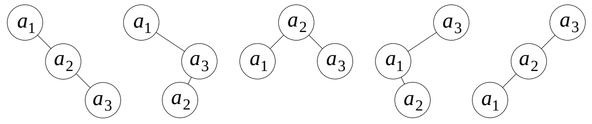

Figure 14: There are five different binary trees of three nodes. From left to right their costs are $\frac{2}{3}, \frac{5}{6}, \frac{2}{3}, \frac{7}{6}, \frac{4}{3}$. The first tree and the third tree are both optimal.

Dynamic programming. We write T_i^j for the optimum weighted binary search tree of $a_i, a_{i+1}, \ldots, a_j$, C_i^j for its cost, and $p_i^j = \sum_{k=i}^{j} p_k$ for the total probability of the items in T_i^j. Suppose we know that the optimum tree stores item a_k in its root. Then the left subtree is T_i^{k-1} and the right subtree is T_{k+1}^j. The cost of the optimum tree is therefore $C_i^j = C_i^{k-1} + C_{k+1}^j + p_i^j - p_k$. Since we do not know which item is in the root, we try all possibilities and find the minimum:

$$C_i^j = \min_{i \le k \le j} \{C_i^{k-1} + C_{k+1}^j + p_i^j - p_k\}.$$

This formula can be translated directly into a dynamic programming algorithm. We use three two-dimensional arrays, one for the sums of probabilities, p_i^j, one for the costs of optimum trees, C_i^j, and one for the indices of the items stored in their roots, R_i^j. We assume that the first array has already been computed. We initialize the other two arrays along the main diagonal and add one dummy diagonal for the cost.

```
for $k = 1$ to $n$ do
  $C[k, k-1] = C[k, k] = 0;\ R[k, k] = k$
endfor;
$C[n+1, n] = 0$.
```

We fill the rest of the two arrays one diagonal at a time.

```
for $\ell = 2$ to $n$ do
  for $i = 1$ to $n - \ell + 1$ do
    $j = i + \ell - 1;\ C[i, j] = \infty;$
    for $k = i$ to $j$ do
      $cost = C[i, k-1] + C[k+1, j]$
            $+ p[i, j] - p[k, k];$
      if $cost < C[i, j]$ then
        $C[i, j] = cost;\ R[i, j] = k$
      endif
    endfor
  endfor
endfor.
```

The main part of the algorithm consists of three nested loops each iterating through at most n values. The running time is therefore in $O(n^3)$.

Example. Table 1 shows the partial sums of probabilities for the data in the earlier example. Table 2 shows

$6p$	1	2	3
1	3	5	6
2		2	3
3			1

Table 1: Six times the partial sums of probabilities used by the dynamic programming algorithm.

the costs and the indices of the roots of the optimum trees computed for all contiguous subsequences. The optimum

$6C$	1	2	3
1	0	2	4
2		0	1
3			0

R	1	2	3
1	1	1	1
2		2	2
3			3

Table 2: Six times the costs and the roots of the optimum trees.

tree can be constructed from R as follows. The root stores the item with index $R[1, 3] = 1$. The left subtree is therefore empty and the right subtree stores a_2, a_3. The root of the optimum right subtree stores the item with index $R[2, 3] = 2$. Again the left subtree is empty and the right subtree consists of a single node storing a_3.

Improved running time. Notice that the array R in Table 2 is monotonic, both along rows and along columns. Indeed it is possible to prove $R_i^{j-1} \leq R_i^j$ in every row and $R_i^j \leq R_{i+1}^j$ in every column. We omit the proof and show how the two inequalities can be used to improve the dynamic programming algorithm. Instead of trying all roots from i through j we restrict the innermost for-loop to

```
for $k = R[i, j-1]$ to $R[i+1, j]$ do
```

The monotonicity property implies that this change does not alter the result of the algorithm. The running time of a single iteration of the outer for-loop is now

$$U_\ell(n) \quad = \quad \sum_{i=1}^{n-\ell+1} (R_{i+1}^j - R_i^{j-1} + 1).$$

Recall that $j = i + \ell - 1$ and note that most terms cancel, giving

$$\begin{aligned} U_\ell(n) &= R_{n-\ell+2}^n - R_1^{\ell-1} + (n - \ell + 1) \\ &\leq 2n. \end{aligned}$$

In words, each iteration of the outer for-loop takes only time $O(n)$, which implies that the entire algorithm takes only time $O(n^2)$.

7 Red-Black Trees

Binary search trees are an elegant implementation of the *dictionary* data type, which requires support for

```
item SEARCH (item),
void INSERT (item),
void DELETE (item),
```

and possible additional operations. Their main disadvantage is the worst case time $\Omega(n)$ for a single operation. The reasons are insertions and deletions that tend to get the tree unbalanced. It is possible to counteract this tendency with occasional local restructuring operations and to guarantee logarithmic time per operation.

2-3-4 trees. A special type of balanced tree is the 2-3-4 *tree*. Each internal node stores one, two, or three items and has two, three, or four children. Each leaf has the same depth. As shown in Figure 15, the items in the internal nodes separate the items stored in the subtrees and thus facilitate fast searching. In the smallest 2-3-4 tree of

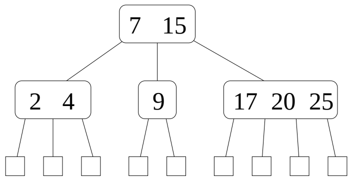

Figure 15: A 2-3-4 tree of height two. All items are stored in internal nodes.

height h, every internal node has exactly two children, so we have 2^h leaves and $2^h - 1$ internal nodes. In the largest 2-3-4 tree of height h, every internal node has four children, so we have 4^h leaves and $(4^h - 1)/3$ internal nodes. We can store a 2-3-4 tree in a binary tree by expanding a node with $i > 1$ items and $i + 1$ children into i nodes each with one item, as shown in Figure 16.

Red-black trees. Suppose we color each edge of a binary search tree either red or black. The color is conveniently stored in the lower node of the edge. Such a edge-colored tree is a *red-black tree* if

(1) there are no two consecutive red edges on any descending path and every maximal such path ends with a black edge;

(2) all maximal descending paths have the same number of black edges.

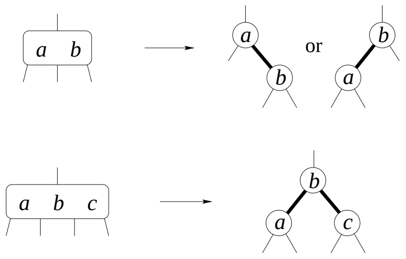

Figure 16: Transforming a 2-3-4 tree into a binary tree. Bold edges are called red and the others are called black.

The number of black edges on a maximal descending path is the *black height*, denoted as $bh(\varrho)$. When we transform a 2-3-4 tree into a binary tree as in Figure 16, we get a red-black tree. The result of transforming the tree in Figure 15

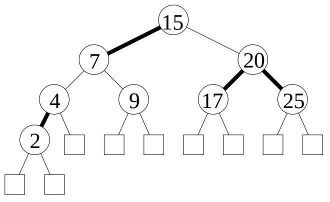

Figure 17: A red-black tree obtained from the 2-3-4 tree in Figure 15.

is shown in Figure 17.

HEIGHT LEMMA. A red-black tree with n internal nodes has height at most $2\log_2(n + 1)$.

PROOF. The number of leaves is $n + 1$. Contract each red edge to get a 2-3-4 tree with $n + 1$ leaves. Its height is $h \leq \log_2(n + 1)$. We have $bh(\varrho) = h$, and by Rule (1) the height of the red-black tree is at most $2bh(\varrho) \leq 2\log_2(n + 1)$. ◻

Rotations. Restructuring a red-black tree can be done with only one operation (and its symmetric version): a *rotation* that moves a subtree from one side to another, as shown in Figure 18. The ordered sequence of nodes in the left tree of Figure 18 is

$$\ldots, \text{order}(A), \nu, \text{order}(B), \mu, \text{order}(C), \ldots,$$

and this is also the ordered sequence of nodes in the right tree. In other words, a rotation maintains the ordering. Function ZIG below implements the right rotation:

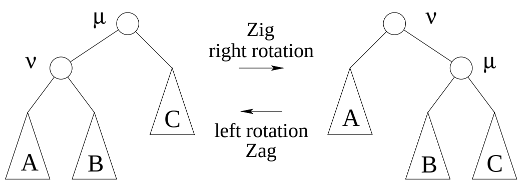

Figure 18: From left to right a right rotation and from right to left a left rotation.

```
Node $*$ ZIG(Node $*\mu$)
  assert $\mu \neq$ NULL and $\nu = \mu \rightarrow \ell \neq$ NULL;
  $\mu \rightarrow \ell = \nu \rightarrow r$; $\nu \rightarrow r = \mu$; return $\nu$.
```

Function ZAG is symmetric and performs a left rotation. Occasionally, it is necessary to perform two rotations in sequence, and it is convenient to combine them into a single operation referred to as a *double rotation*, as shown in Figure 19. We use a function ZIGZAG to implement a

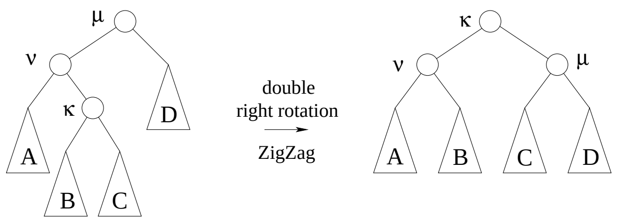

Figure 19: The double right rotation at μ is the concatenation of a single left rotation at ν and a single right rotation at μ.

double right rotation and the symmetric function ZAGZIG to implement a double left rotation.

```
Node $*$ ZIGZAG(Node $*\mu$)
  $\mu \rightarrow \ell = $ ZAG($\mu \rightarrow \ell$); return ZIG($\mu$).
```

The double right rotation is the composition of two single rotations: ZIGZAG(μ) = ZIG$(\mu) \circ$ ZAG(ν). Remember that the composition of functions is written from right to left, so the single left rotation of ν precedes the single right rotation of μ. Single rotations preserve the ordering of nodes and so do double rotations.

Insertion. Before studying the details of the restructuring algorithms for red-black trees, we look at the trees that arise in a short insertion sequence, as shown in Figure 20. After adding 10, 7, 13, 4, we have two red edges in sequence and repair this by promoting 10 (A). After adding 2, we repair the two red edges in sequence by a single rotation of 7 (B). After adding 5, we promote 4 (C), and after adding 6, we do a double rotation of 7 (D).

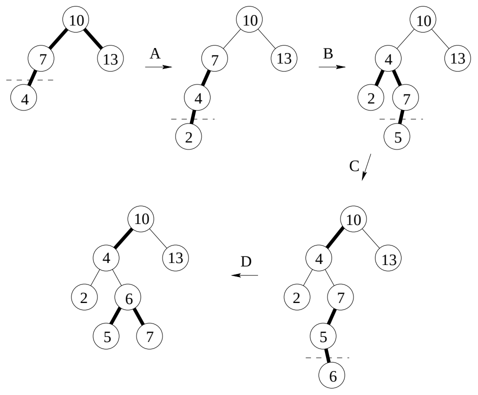

Figure 20: Sequence of red-black trees generated by inserting the items 10, 7, 13, 4, 2, 5, 6 in this sequence.

An item x is added by substituting a new internal node for a leaf at the appropriate position. To satisfy Rule (2) of the red-black tree definition, color the incoming edge of the new node red, as shown in Figure 21. Start the

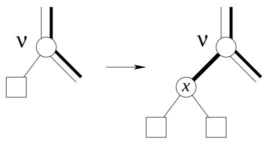

Figure 21: The incoming edge of a newly added node is always red.

adjustment of color and structure at the parent ν of the new node. We state the properties maintained by the insertion algorithm as invariants that apply to a node ν traced by the algorithm.

INVARIANT I. The only possible violation of the red-black tree properties is that of Rule (1) at the node ν, and if ν has a red incoming edge then it has exactly one red outgoing edge.

Observe that Invariant I holds right after adding x. We continue with the analysis of all the cases that may arise. The local adjustment operations depend on the neighborhood of ν.

`Case 1.` The incoming edge of ν is black. Done.

`Case 2.` The incoming edge of ν is red. Let μ be the parent of ν and assume ν is left child of μ.

`Case 2.1.` Both outgoing edges of μ are red, as in Figure 22. Promote μ. Let ν be the parent of μ and recurse.

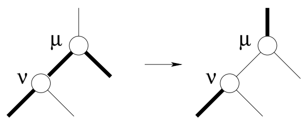

Figure 22: Promotion of μ. (The colors of the outgoing edges of ν may be the other way round).

`Case 2.2.` Only one outgoing edge of μ is red, namely the one from μ to ν.

`Case 2.2.1.` The left outgoing edge of ν is red, as in Figure 23 to the left. Right rotate μ. Done.

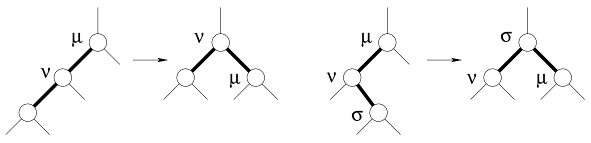

Figure 23: Right rotation of μ to the left and double right rotation of μ to the right.

`Case 2.2.2.` The right outgoing edge of ν is red, as in Figure 23 to the right. Double right rotate μ. Done.

Case 2 has a symmetric case where left and right are interchanged. An insertion may cause logarithmically many promotions but at most two rotations.

Deletion. First find the node π that is to be removed. If necessary, we substitute the inorder successor for π so we can assume that both children of π are leaves. If π is last in inorder we substitute symmetrically. Replace π by a leaf ν, as shown in Figure 24. If the incoming edge of π is red then change it to black. Otherwise, remember the incoming edge of ν as 'double-black', which counts as two black edges. Similar to insertions, it helps to understand the deletion algorithm in terms of a property it maintains.

INVARIANT D. The only possible violation of the red-black tree properties is a double-black incoming edge of ν.

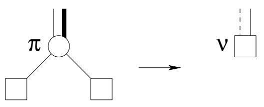

Figure 24: Deletion of node π. The dashed edge counts as two black edges when we compute the black depth.

Note that Invariant D holds right after we remove π. We now present the analysis of all the possible cases. The adjustment operation is chosen depending on the local neighborhood of ν.

`Case 1.` The incoming edge of ν is black. Done.

`Case 2.` The incoming edge of ν is double-black. Let μ be the parent and κ the sibling of ν. Assume ν is left child of μ and note that κ is internal.

`Case 2.1.` The edge from μ to κ is black.

`Case 2.1.1.` Both outgoing edges of κ are black, as in Figure 25. Demote μ. Recurse for $\nu = \mu$.

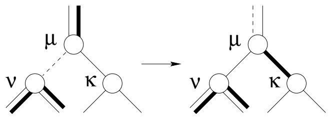

Figure 25: Demotion of μ.

`Case 2.1.2.` The right outgoing edge of κ is red, as in Figure 26 to the left. Change the color of that edge to black and left rotate μ. Done.

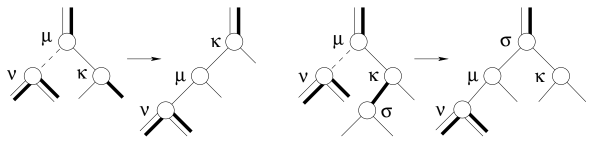

Figure 26: Left rotation of μ to the left and double left rotation of μ to the right.

`Case 2.1.3.` The right outgoing edge of κ is black, as in Figure 26 to the right. Change the color of the left outgoing edge to black and double left rotate μ. Done.

`Case 2.2.` The edge from μ to κ is red, as in Figure 27. Left rotate μ. Recurse for ν.

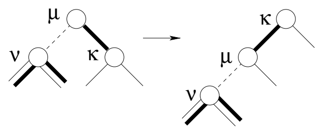

Figure 27: Left rotation of μ.

Case 2 has a symmetric case in which ν is the right child of μ. Case 2.2 seems problematic because it recurses without moving ν any closer to the root. However, the configuration excludes the possibility of Case 2.2 occurring again. If we enter Cases 2.1.2 or 2.1.3 then the termination is immediate. If we enter Case 2.1.1 then the termination follows because the incoming edge of μ is red. The deletion may cause logarithmically many demotions but at most three rotations.

Summary. The red-black tree is an implementation of the dictionary data type and supports the operations search, insert, delete in logarithmic time each. An insertion or deletion requires the equivalent of at most three single rotations. The red-black tree also supports finding the minimum, maximum and the inorder successor, predecessor of a given node in logarithmic time each.

8 Amortized Analysis

Amortization is an analysis technique that can influence the design of algorithms in a profound way. Later in this course, we will encounter data structures that owe their very existence to the insight gained in performance due to amortized analysis.

Binary counting. We illustrate the idea of amortization by analyzing the cost of counting in binary. Think of an integer as a linear array of bits, $n = \sum_{i \geq 0} A[i] \cdot 2^i$. The following loop keeps incrementing the integer stored in A.

```
loop $i = 0$;
  while $A[i] = 1$ do $A[i] = 0$; $i$++ endwhile;
  $A[i] = 1$.
forever.
```

We define the *cost* of counting as the total number of bit changes that are needed to increment the number one by one. What is the cost to count from 0 to n? Figure 28 shows that counting from 0 to 15 requires 26 bit changes. Since n takes only $1 + \lfloor \log_2 n \rfloor$ bits or positions in A,

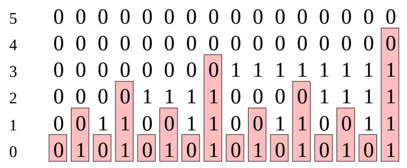

Figure 28: The numbers are written vertically from top to bottom. The boxed bits change when the number is incremented.

a single increment does at most $2 + \log_2 n$ steps. This implies that the cost of counting from 0 to n is at most $n \log_2 n + 2n$. Even though the upper bound of $2 + \log_2 n$ is almost tight for the worst single step, we can show that the total cost is much less than n times that. We do this with two slightly different amortization methods referred to as aggregation and accounting.

Aggregation. The aggregation method takes a global view of the problem. The pattern in Figure 28 suggests we define b_i equal to the number of 1s and t_i equal to the number of trailing 1s in the binary notation of i. Every other number has no trailing 1, every other number of the remaining ones has one trailing 1, etc. Assuming $n = 2^k - 1$, we therefore have exactly $j - 1$ trailing 1s for $2^{k-j} = (n+1)/2^j$ integers between 0 and $n - 1$. The total number of bit changes is therefore

$$T(n) \quad = \quad \sum_{i=0}^{n-1} (t_i + 1) \quad = \quad (n+1) \cdot \sum_{j=1}^{k} \frac{j}{2^j}.$$

We use index transformation to show that the sum on the right is less than 2:

$$\begin{aligned} \sum_{j \geq 1} \frac{j}{2^j} &= \sum_{j \geq 1} \frac{j-1}{2^{j-1}} \\ &= 2 \cdot \sum_{j \geq 1} \frac{j}{2^j} - \sum_{j \geq 1} \frac{1}{2^{j-1}} \\ &= 2. \end{aligned}$$

Hence the cost is $T(n) < 2(n+1)$. The *amortized cost* per operation is $\frac{T(n)}{n}$, which is about 2.

Accounting. The idea of the accounting method is to charge each operation what we think its amortized cost is. If the amortized cost exceeds the actual cost, then the surplus remains as a credit associated with the data structure. If the amortized cost is less than the actual cost, the accumulated credit is used to pay for the cost overflow. Define the amortized cost of a bit change $0 \to 1$ as \$2 and that of $1 \to 0$ as \$0. When we change 0 to 1 we pay \$1 for the actual expense and \$1 stays with the bit, which is now 1. This \$1 pays for the (later) cost of changing the 1 to 0. Each increment has amortized cost \$2, and together with the money in the system, this is enough to pay for all the bit changes. The cost is therefore at most $2n$.

We see how a little trick, like making the $0 \to 1$ changes pay for the $1 \to 0$ changes, leads to a very simple analysis that is even more accurate than the one obtained by aggregation.

Potential functions. We can further formalize the amortized analysis by using a potential function. The idea is similar to accounting, except there is no explicit credit saved anywhere. The accumulated credit is an expression of the well-being or potential of the data structure. Let c_i be the actual cost of the i-th operation and D_i the data structure after the i-th operation. Let $\Phi_i = \Phi(D_i)$ be the potential of D_i, which is some numerical value depending on the concrete application. Then we define $a_i = c_i + \Phi_i - \Phi_{i-1}$ as the *amortized cost* of the i-th operation. The sum of amortized costs of n operations is

$$\begin{aligned} \sum_{i=1}^{n} a_i &= \sum_{i=1}^{n} (c_i + \Phi_i - \Phi_{i-1}) \\ &= \sum_{i=1}^{n} c_i + \Phi_n - \Phi_0. \end{aligned}$$

We aim at choosing the potential such that $\Phi_0 = 0$ and $\Phi_n \geq 0$ because then we get $\sum a_i \geq \sum c_i$. In words, the sum of amortized costs covers the sum of actual costs. To apply the method to binary counting we define the potential equal to the number of 1s in the binary notation, $\Phi_i = b_i$. It follows that

$$\begin{aligned} \Phi_i - \Phi_{i-1} &= b_i - b_{i-1} \\ &= (b_{i-1} - t_{i-1} + 1) - b_{i-1} \\ &= 1 - t_{i-1}. \end{aligned}$$

The actual cost of the i-th operation is $c_i = 1 + t_{i-1}$, and the amortized cost is $a_i = c_i + \Phi_i - \Phi_{i-1} = 2$. We have $\Phi_0 = 0$ and $\Phi_n \geq 0$ as desired, and therefore $\sum c_i \leq \sum a_i = 2n$, which is consistent with the analysis of binary counting with the aggregation and the accounting methods.

2-3-4 trees. As a more complicated application of amortization we consider 2-3-4 trees and the cost of restructuring them under insertions and deletions. We have seen 2-3-4 trees earlier when we talked about red-black trees. A set of keys is stored in sorted order in the internal nodes of a 2-3-4 tree, which is characterized by the following rules:

(1) each internal node has $2 \leq d \leq 4$ children and stores $d - 1$ keys;

(2) all leaves have the same depth.

As for binary trees, being sorted means that the left-to-right order of the keys is sorted. The only meaningful definition of this ordering is the ordered sequence of the first subtree followed by the first key stored in the root followed by the ordered sequence of the second subtree followed by the second key, etc.

To insert a new key, we attach a new leaf and add the key to the parent ν of that leaf. All is fine unless ν overflows because it now has five children. If it does, we repair the violation of Rule (1) by climbing the tree one node at a time. We call an internal node *non-saturated* if it has fewer than four children.

`Case 1.` ν has five children and a non-saturated sibling to its left or right. Move one child from ν to that sibling, as in Figure 29.

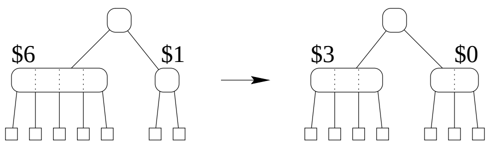

Figure 29: The overflowing node gives one child to a non-saturated sibling.

`Case 2.` ν has five children and no non-saturated sibling. Split ν into two nodes and recurse for the parent of ν, as in Figure 30. If ν has no parent then create a new root whose only children are the two nodes obtained from ν.

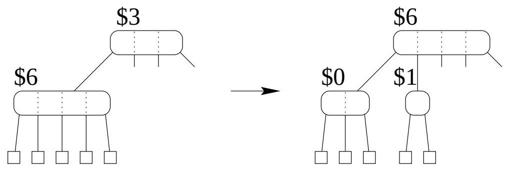

Figure 30: The overflowing node is split into two and the parent is treated recursively.

Deleting a key is done is a similar fashion, although there we have to battle with nodes ν that have too few children rather than too many. Let ν have only one child. We repair Rule (1) by adopting a child from a sibling or by merging ν with a sibling. In the latter case the parent of ν looses a child and needs to be visited recursively. The two operations are illustrated in Figures 31 and 32.

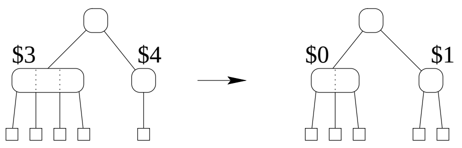

Figure 31: The underflowing node receives one child from a sibling.

Amortized analysis. The worst case for inserting a new key occurs when all internal nodes are saturated. The insertion then triggers logarithmically many splits. Symmetrically, the worst case for a deletion occurs when all

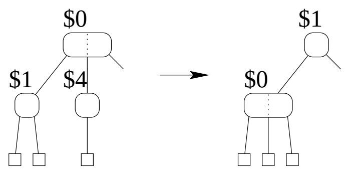

Figure 32: The underflowing node is merged with a sibling and the parent is treated recursively.

internal nodes have only two children. The deletion then triggers logarithmically many mergers. Nevertheless, we can show that in the amortized sense there are at most a constant number of split and merge operations per insertion and deletion.

We use the accounting method and store money in the internal nodes. The best internal nodes have three children because then they are flexible in both directions. They require no money, but all other nodes are given a positive amount to pay for future expenses caused by split and merge operations. Specifically, we store \$4, \$1, \$0, \$3, \$6 in each internal node with 1, 2, 3, 4, 5 children. As illustrated in Figures 29 and 31, an adoption moves money only from ν to its sibling. The operation keeps the total amount the same or decreases it, which is even better. As shown in Figure 30, a split frees up \$5 from ν and spends at most \$3 on the parent. The extra \$2 pay for the split operation. Similarly, a merger frees \$5 from the two affected nodes and spends at most \$3 on the parent. This is illustrated in Figure 32. An insertion makes an initial investment of at most \$3 to pay for creating a new leaf. Similarly, a deletion makes an initial investment of at most \$3 for destroying a leaf. If we charge $\$2$ for each split and each merge operation, the money in the system suffices to cover the expenses. This implies that for n insertions and deletions we get a total of at most $\frac{3n}{2}$ split and merge operations. In other words, the amortized number of split and merge operations is at most $\frac{3}{2}$.

Recall that there is a one-to-one correspondence between 2-3-4 tree and red-black trees. We can thus translate the above update procedure and get an algorithm for red-black trees with an amortized constant restructuring cost per insertion and deletion. We already proved that for red-black trees the number of rotations per insertion and deletion is at most a constant. The above argument implies that also the number of promotions and demotions is at most a constant, although in the amortized and not in the worst-case sense as for the rotations.

9 Splay Trees

Splay trees are similar to red-black trees except that they guarantee good shape (small height) only on the average. They are simpler to code than red-black trees and have the additional advantage of giving faster access to items that are more frequently searched. The reason for both is that splay trees are self-adjusting.

Self-adjusting binary search trees. Instead of explicitly maintaining the balance using additional information (such as the color of edges in the red-black tree), splay trees maintain balance implicitly through a self-adjusting mechanism. Good shape is a side-effect of the operations that are applied. These operations are applied while *splaying* a node, which means moving it up to the root of the tree, as illustrated in Figure 33. A detailed analysis will

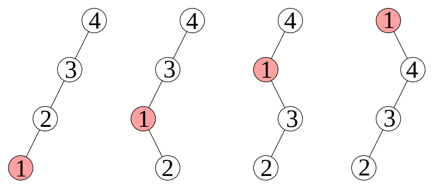

Figure 33: The node storing 1 is splayed using three single rotations.

reveal that single rotations do not imply good amortized performance but combinations of single rotations in pairs do. Aside from double rotations, we use *roller-coaster rotations* that compose two single left or two single right rotations, as shown in Figure 35. The sequence of the two single rotations is important, namely first the higher then the lower node. Recall that ZIG(κ) performs a single right rotation and returns the new root of the rotated subtree. The roller-coaster rotation to the right is then

```
Node $*$ ZIGZIG(Node $*\kappa$)
  return ZIG(ZIG($\kappa$)).
```

Function ZAGZAG is symmetric, exchanging left and right, and functions ZIGZAG and ZAGZIG are the two double rotations already used for red-black trees.

Splay. A splay operation finds an item and uses rotations to move the corresponding node up to the root position. Whenever possible, a double rotation or a roller-coaster rotation is used. We dispense with special cases and show Function SPLAY for the case the search item x is less than the item in the root.

```
if $x < \varrho \rightarrow$ info then $\mu = \varrho \rightarrow \ell$;
  if $x < \mu \rightarrow$ info then
    $\mu \rightarrow \ell = $ SPLAY$(\mu \rightarrow \ell, x)$;
    return ZIGZIG$(\varrho)$
  elseif $x > \mu \rightarrow$ info then
    $\mu \rightarrow r = $ SPLAY$(\mu \rightarrow r, x)$;
    return ZIGZAG$(\varrho)$
  else
    return ZIG$(\varrho)$
endif.
```

If x is stored in one of the children of ϱ then it is moved to the root by a single rotation. Otherwise, it is splayed recursively to the third level and moved to the root either by a double or a roller-coaster rotation. The number of rotation depends on the length of the path from ϱ to x. Specifically, if the path is i edges long then x is splayed in $\lfloor i/2 \rfloor$ double and roller-coaster rotations and zero or one single rotation. In the worst case, a single splay operation takes almost as many rotations as there are nodes in the tree. We will see shortly that the amortized number of rotations is at most logarithmic in the number of nodes.

Amortized cost. Recall that the amortized cost of an operation is the actual cost minus the cost for work put into improving the data structure. To analyze the cost, we use a potential function that measures the well-being of the data structure. We need definitions:

the *size* $s(\nu)$ is the number of descendents of node ν, including ν,

the *balance* $\beta(\nu)$ is twice the floor of the binary logarithm of the size, $\beta(\nu) = 2\lfloor \log_2 s(\nu) \rfloor$,

the *potential* Φ of a tree or a collection of trees is the sum of balances over all nodes, $\Phi = \sum \beta(\nu)$,

the *actual cost* c_i of the i-th splay operation is 1 plus the number of single rotations (counting a double or roller-coaster rotation as two single rotations).

the *amortized cost* a_i of the i-th splay operation is $a_i = c_i + \Phi_i - \Phi_{i-1}$.

We have $\Phi_0 = 0$ for the empty tree and $\Phi_i \geq 0$ in general. This implies that the total actual cost does not exceed the total amortized cost, $\sum c_i = \sum a_i - \Phi_n + \Phi_0 \leq \sum a_i$.

To get a feeling for the potential, we compute Φ for the two extreme cases. Note first that the integral of the natural logarithm is $\int \ln x = x \ln x - x$ and therefore $\int \log_2 x = x \log_2 x - x/\ln 2$. In the extreme unbalanced case, the balance of the i-th node from the bottom is $2\lfloor \log_2 i \rfloor$ and the potential is

$$\Phi = 2\sum_{i=1}^{n} \lfloor \log_2 i \rfloor = 2n \log_2 n - \mathrm{O}(n).$$

In the balanced case, we bound Φ from above by $2U(n)$, where $U(n) = 2U(\frac{n}{2}) + \log_2 n$. We prove that $U(n) < 2n$ for the case when $n = 2^k$. Consider the perfectly balanced tree with n leaves. The height of the tree is $k = \log_2 n$. We encode the term $\log_2 n$ of the recurrence relation by drawing the hook-like path from the root to the right child and then following left edges until we reach the leaf level. Each internal node encodes one of the recursively surfacing $\log$-terms by a hook-like path starting at that node. The paths are pairwise edge-disjoint, which implies that their total length is at most the number of edges in the tree, which is $2n - 2$.

Investment. The main part of the amortized time analysis is a detailed study of the three types of rotations: single, roller-coaster, and double. We write $\beta(\nu)$ for the balance of a node ν before the rotation and $\beta'(\nu)$ for the balance after the rotation. Let ν be the lowest node involved in the rotation. The goal is to prove that the amortized cost of a roller-coaster and a double rotation is at most $3[\beta'(\nu) - \beta(\nu)]$ each, and that of a single rotation is at most $1 + 3[\beta'(\nu) - \beta(\nu)]$. Summing these terms over the rotations of a splay operation gives a telescoping series in which all terms cancel except the first and the last. To this we add 1 for the at most one single rotation and another 1 for the constant cost in definition of actual cost.

INVESTMENT LEMMA. The amortized cost of splaying a node ν in a tree ϱ is at most $2 + 3[\beta(\varrho) - \beta(\nu)]$.

Before looking at the details of the three types of rotations, we prove that if two siblings have the same balance then their common parent has a larger balance. Because balances are even integers this means that the balance of the parent exceeds the balance of its children by at least 2.

BALANCE LEMMA. If μ has children ν, κ and $\beta(\nu) = \beta(\kappa) = \beta$ then $\beta(\mu) \geq \beta + 2$.

PROOF. By definition $\beta(\nu) = 2\lfloor \log_2 s(\nu) \rfloor$ and therefore $s(\nu) \geq 2^{\beta/2}$. We have $s(\mu) = 1 + s(\nu) + s(\kappa) \geq 2^{1+\beta/2}$, and therefore $\beta(\mu) \geq \beta + 2$. ◻

Single rotation. The amortized cost of a single rotation shown in Figure 34 is 1 for performing the rotation plus the change in the potential:

$$\begin{aligned} a &= 1 + \beta'(\nu) + \beta'(\mu) - \beta(\nu) - \beta(\mu) \\ &\leq 1 + 3[\beta'(\nu) - \beta(\nu)] \end{aligned}$$

because $\beta'(\mu) \leq \beta(\mu)$ and $\beta(\nu) \leq \beta'(\nu)$.

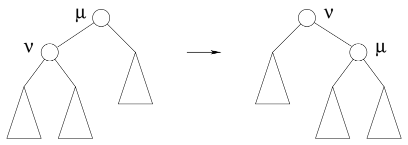

Figure 34: The size of μ decreases and that of ν increases from before to after the rotation.

Roller-coaster rotation. The amortized cost of a roller-coaster rotation shown in Figure 35 is

$$\begin{aligned} a &= 2 + \beta'(\nu) + \beta'(\mu) + \beta'(\kappa) \\ &\quad - \beta(\nu) - \beta(\mu) - \beta(\kappa) \\ &\leq 2 + 2[\beta'(\nu) - \beta(\nu)] \end{aligned}$$

because $\beta'(\kappa) \leq \beta(\kappa)$, $\beta'(\mu) \leq \beta'(\nu)$, and $\beta(\nu) \leq \beta(\mu)$. We distinguish two cases to prove that a is bounded from above by $3[\beta'(\nu) - \beta(\nu)]$. In both cases, the drop in the potential pays for the two single rotations.

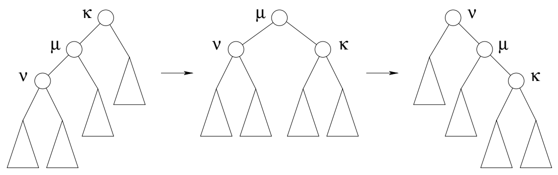

Figure 35: If in the middle tree the balance of ν is the same as the balance of μ then by the Balance Lemma the balance of κ is less than that common balance.

Case $\beta'(\nu) > \beta(\nu)$. The difference between the balance of ν before and after the roller-coaster rotation is at least 2. Hence $a \leq 3[\beta'(\nu) - \beta(\nu)]$.

Case $\beta'(\nu) = \beta(\nu) = \beta$. Then the balances of nodes ν and μ in the middle tree in Figure 35 are also equal to β. The Balance Lemma thus implies that the balance of κ in that middle tree is at most $\beta - 2$. But since the balance of κ after the roller-coaster rotation is the same as in the middle tree, we have $\beta'(\kappa) < \beta$. Hence $a \leq 0 = 3[\beta'(\nu) - \beta(\nu)]$.