

# Chapter 12

## Saving Space

Most of the data structures presented in this text have fast update and query times (often  $O(\log n)$ ) and fast construction times (often  $O(n \log n)$ ). This means that, in practice, the size of these data structures is not limited by running time, but rather by the amount of available memory.<sup>1</sup> In this chapter, we consider some of the techniques used to keep memory requirements as small as possible.

Until now, we have described the space of data structures in terms of the number of data elements and pieces of bookkeeping information. For example, when we say that a data structure has size  $O(n)$ , we mean that it stores  $O(n)$  data items as well as  $O(n)$  pointers, integers, floating point numbers, and so on. To give a more precise discussion of space requirements, we need to refine this model.

Let  $N$  be a global upper bound on  $n$ , the number of data items stored in any data structure at any point in time. For example,  $N$  might be the size of the main memory in the computer. Define a *memory unit* as  $\lceil \log N \rceil$  bits, so that a single memory unit can store a pointer to any memory location, or an integer in the range  $\{0, \dots, 2^N - 1\}$ . Our data structures store opaque data items, where each data item occupies  $w$  memory units. Our data structures can *allocate* a block of memory of any size  $k \geq 1$ , which is an array of  $k$  memory units as well as  $O(1)$  additional memory units of bookkeeping information. A data structure can *deallocate* a previously allocated block by giving a pointer to the block. In addition to allocated blocks, the data structure can have  $O(1)$  memory units of static information that does not need to be allocated. At any point in time, the *memory usage* of a data structure  $D$ , which has currently allocated (but not yet deallocated) blocks of sizes  $N_1, \dots, N_m$  is

$$M(D) = O(m + 1) + \sum_{i=1}^m N_i . \quad (12.1)$$

We say that the *overhead* of a data structure  $D$  that is currently storing  $n$  data items is  $M(D) - nw$ . That is, the overhead is any memory usage beyond the minimum required to store  $n$  data items. The memory overhead of data structure  $D$  is  $f(n)$  if, at any time  $i$  that the data structure stores

---

<sup>1</sup> Actually, in practice it will appear as if the data structure becomes incredibly slow once its size gets too large. This is due to the fact that modern operating systems will attempt to simulate a large internal (RAM) memory by using slow external (disk) memory.

$n_i$  elements,  $M(D) - n_i w \leq f(n_i)$ . In this chapter, we study data structures that attempt to keep the overhead as small as possible while still being fast. Before we begin describing data structures, we first give a lower-bound on how low we can expect to keep the overhead.

Suppose we have a data structure  $D$  that supports the insert operation. That is, individual data items can be added to  $D$ . The following informal argument shows that  $D$  must have memory overhead  $M(D) = \Omega(\sqrt{n})$ . Consider a sequence of  $n$  insert operations beginning with an initially empty structure. Suppose  $D$  uses  $k$  memory blocks, the largest of which has size  $sw$ . In order to store all  $n$  elements, it must be that  $s \times k \geq n$ , so at least one of  $s$  or  $k$  is at least  $\sqrt{n}$ . If  $k$  is greater than  $\sqrt{n}$ , then we are done, since each memory block has a constant overhead, so the overhead is  $\Omega(\sqrt{n})$ . On the other hand, if  $s \geq \sqrt{n}$ , then there is some block  $B$  of size at least  $w\sqrt{n}$ . Immediately after  $B$  was allocated, it did not contain any data elements, so *at that point in time*, the memory overhead was  $\Omega(w\sqrt{n})$ . Therefore  $D$  will, at some point during a sequence of  $n$  insert operation, have a memory overhead of  $\Omega(\sqrt{n})$ , so the memory overhead of  $D$  is  $\Omega(\sqrt{n})$ .

## 12.1 Resizeable Arrays

Arrays are a fundamental data structure that have myriad applications because (a) they allow constant time access to any element through its index and (b) they are the most compact method of storing a set of homogeneous elements. In most programming languages, the size of an array is determined at the time it is allocated and this size can not be increased or decreased later.

An extension of arrays that allows for dynamic resizing has become known as vectors. A *vector*  $V = \langle V[0], \dots, V[n-1] \rangle$  supports the operations of *reading* or *writing*  $V[i]$  for any  $i \in \{0, \dots, n-1\}$  and *growing*  $V$  (incrementing  $n$ ) or *shrinking*  $V$  (decrementing  $n$ ). All these operations take constant (amortized) time.

The classic method of implementing a vector is as follows: We maintain an array  $A$  of size  $n' \geq n$ , where  $A[i]$  stores the value of  $V[i]$ . When the size of  $n' = n$  and the user increments  $n$ , a new array  $A'$  is allocated of size  $2n'$ , the  $n$  elements of  $A$  are copied into the first  $n$  elements of  $A'$ , and then  $A$  is discarded. To avoid a situation where  $A$  becomes much bigger than necessary, any time  $n < n'/3$ , the size of  $A$  is halved by allocating an array  $A'$  of size  $n/2$ , copying the  $n$  elements of  $A$  into  $A'$  and discarding  $A$ . Notice that, between any two resizing operations,  $\Omega(n)$  growing or shrinking operations have occurred so the amortized cost of growing and shrinking is  $O(1)$ . The cost of reading and writing is  $O(1)$  since the vector item  $V[i]$  is simply stored in  $A[i]$ .

The problem with the classic solution is that it is wasteful. Immediately after allocating  $A'$ , half of the array  $A'$  is unused. In fact, during the growing process,  $A$  and  $A'$  are using  $3n$  memory cells to store only  $n$  elements. In some applications, this can pose a severe limitation on the amount of data that can be processed. Of course, rather than doubling or halving the size of  $A$  during each reallocation, we could multiply or divide the size of  $A$  by  $(1 + \epsilon)$  for any  $\epsilon > 0$ , so that the wasted space is only  $\epsilon n$ . Nevertheless, there will always be the problem that, during the reallocation process, a working space of size  $(2 + \epsilon)n$  is required.

Next we present a vector implementation in which there is never more than  $O(\sqrt{n})$  wasted space. The vector  $V$  is represented as a collection of blocks  $A_1, \dots, A_k$ . The block  $A_i$  is an array of length  $i$ , so the total amount of available space is  $\sum_{i=1}^k i = k(k+1)/2$ . In particular, to represent a

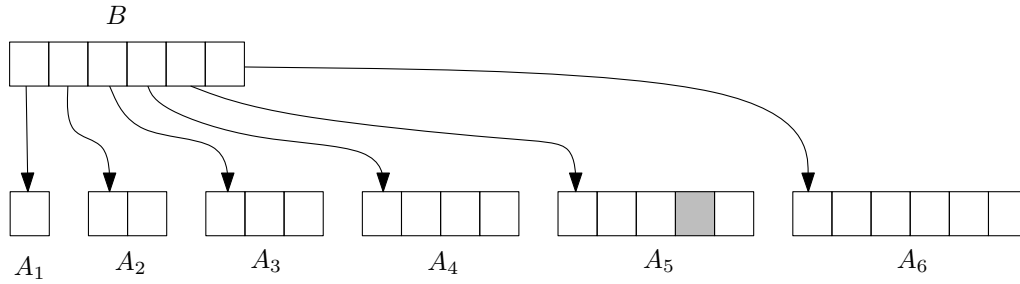


Figure 12.1: A vector  $V$  represented as a sequence of blocks. The element  $V[13]$  is shown in gray ( $i' = \lceil \sqrt{2i + 1/4} - 1/2 \rceil = 5$ ,  $i'' = 13 - (i'(i' - 1))/2 = 3$ ).

vector of length  $n$  we only need  $k = \lceil \sqrt{2n + 1/4} - 1/2 \rceil = O(\sqrt{n})$  blocks, the largest of which has size  $k = O(\sqrt{n})$ . To keep track of the blocks, we use another vector  $B$  of size  $k$  where  $B[i]$  is a pointer to block  $A_i$ . The vector  $B$  is maintained using the classic implementation of vectors described above.

In this implementation, growing or shrinking  $V$  is straightforward. To grow  $V$ , we check if  $n + 1 > k(k + 1)/2$  and, if so, increment  $k$  and add a new block. To shrink  $V$ , we check if  $k - 2 > \lceil \sqrt{2n + 1/4} - 1/2 \rceil$  and, if so, we free the block  $A_k$  and decrement  $k$ . These rules guarantee that the number of grow and shrink operations that occur between the allocation/freeing of two blocks is  $\Omega(k)$  and the number of empty blocks is never more than 2. Thus, the grow and shrink operations take constant amortized time and the amount of space not dedicated to data is  $O(k) = O(\sqrt{n}(w + 1))$ .

See Figure 12.1. To access  $V[i]$ , we compute  $i' = \lceil \sqrt{2i + 1/4} - 1/2 \rceil$ . Then  $B[i']$  is a pointer to the block  $A_{i'}$  containing  $V[i]$ , and  $V[i]$  is at index  $i'' = i - (i'(i' - 1))/2$  within  $A_{i'}$ . Thus,  $V[i]$  can be read or written in constant time provided that we can compute  $i'$  efficiently.

Unfortunately, computing  $i'$  involves the computation of  $\sqrt{2i + 1/4}$ , which is may be irrational. Luckily, we can rewrite  $i'$  as  $i' = \lceil \lceil \sqrt{8i + 1} \rceil + 1 \rceil / 2$ . Still, most computer architectures do not provide for constant-time square root computations, even if we only want the integer part of the square root. Therefore, we show how these operations can be implemented by table lookup into a table of size  $O(\sqrt{i})$ .

Let  $x$  be an integer with  $2^{r-1} \leq x < 2^r$ , so that  $x$  is represented using  $r$  bits. Computing  $\lceil \sqrt{x} \rceil$  is too hard, so instead, we will use a lookup table to compute  $\lceil \sqrt{x'} \rceil$ , where

$$x' = \lfloor x/2^{\lfloor r/2 \rfloor} \rfloor \cdot 2^{\lfloor r/2 \rfloor} .$$

That is,  $x'$  is obtained by setting the  $\lfloor r/2 \rfloor$  lower-order bits of  $x$  to 0. Notice that there are only  $2^{\lfloor r/2 \rfloor}$  possible different values of  $x'$ , so these can be stored in an array  $T_r$  of size  $2^{\lfloor r/2 \rfloor}$ , where

$$T_r \left[ \lfloor x'/2^{\lfloor r/2 \rfloor} \rfloor \right] = \lceil \sqrt{x'} \rceil .$$

All that remains is to show how we can determine  $\lceil \sqrt{x} \rceil$  given  $\lceil \sqrt{x'} \rceil$ . To do this, we will show that these two quantities differ by at most 1. To see this, first observe that  $x' \geq x - \sqrt{x}$  and consider that, for any  $x \geq 1$ ,

$$\sqrt{x} - \sqrt{x - \sqrt{x}} \leq 1 .$$

(This is easily proven by squaring both sides of the inequality  $\sqrt{x} - 1 \leq \sqrt{x - \sqrt{x}}$ .) Therefore,

$$\sqrt{x} - \sqrt{x'} \leq 1$$

so that  $\lceil \sqrt{x} \rceil$  is either  $\lceil \sqrt{x'} \rceil$  or  $\lceil \sqrt{x'} \rceil + 1$ . That is, we can compute  $\lceil \sqrt{x} \rceil$  from  $\lceil \sqrt{x'} \rceil$  using the conditional assignment

$$\lceil \sqrt{x} \rceil = \begin{cases} \lceil \sqrt{x'} \rceil + 1 & \text{if } (\lceil \sqrt{x'} \rceil + 1)^2 \leq x \\ \lceil \sqrt{x'} \rceil & \text{otherwise.} \end{cases}$$

At this point, it looks like the problem is completely solved. However, we have created another problem to overcome; knowing which table  $T_r$  to look in to compute  $\lceil \sqrt{x} \rceil$  involves computing  $r = \lfloor \log x \rfloor + 1$ . Note that  $r$  is simply the index of the most significant 1 bit in  $x$ . Many computers have a hardware instruction for computing this index. For those computers that don't, or those programming languages that don't offer access to this instruction, we can use another lookup table. For any number  $x$  that can be represented using  $b$  bits, we use a table  $L$  of size  $2^{\lceil b/2 \rceil}$ . The table stores the value  $L[i] = \lfloor \log i \rfloor$ , for all  $i \in \{0, \dots, 2^{\lceil b/2 \rceil}\}$ . We can then compute  $\lfloor \log x \rfloor$  in constant time using the formula

$$\lfloor \log x \rfloor = \begin{cases} L[x] & \text{if } x < 2^{\lceil b/2 \rceil} \\ \lceil b/2 \rceil + L[\lfloor x/2^{\lceil b/2 \rceil} \rfloor] & \text{otherwise.} \end{cases}$$

Since the point of this entire exercise was to implement vectors without too much wasted space, we now analyze the space requirements of the tables  $T_1 \dots, T_b$  and  $L$ . These tables must be able to handle all query values  $x \leq n$ . That is, they must handle  $b$ -bit integers for  $b = \lceil \log n \rceil$ , so the table  $L$  has size  $O(2^{b/2}) = O(\sqrt{n})$ . The tables  $T_r$ , for  $r = 1, \dots, b$  have total size

$$\sum_{i=1}^b O(2^{i/2}) = O(2^{b/2}) = O(\sqrt{n}).$$

thus, the tables only contribute an additional  $O(\sqrt{n})$  to the space requirements of the vector. As the value of  $n$  increases or decreases, new tables  $T_r$  can be created and deleted as necessary and the size of  $L$  can be increased or decreased as necessary. We leave it to the reader to verify that this does not increase the amortized cost of growing and shrinking the vector  $V$ .

**Theorem 26.** *There exists a vector data structure supporting, read, write, grow, and shrink operations in  $O(1)$  amortized time per operation and whose size, when storing  $n$  elements of size  $w$  is  $nw + O(\sqrt{n(w+1)})$ .*

## 12.2 Putting a Dictionary on a Diet

A dictionary is one of the most commonly used data structures, and this book contains several implementations of dictionaries including treaps, skiplists, splay trees, scapegoat trees, and power trees. In all of these implementations, each data item is stored in a *node*, which is a dynamically allocated block, that also contains one or more pointers. Therefore, the memory overhead associated with each of these implementations is at least  $cn$  for some constant  $c$ , and usually  $c \geq 3$ . If data items are simple (such as integers in the range  $\{0, \dots, N\}$ ) so that  $w = 1$  then this means that most of the memory usage is actually memory overhead. If the amount of available memory is  $N$ , this implies that the data structure is limited to storing  $N/(c+1)$  items so, obviously, we would like to make  $c$  as small as possible.

In this section, we discuss a practical trick for reducing the storage overhead associated with dictionaries. The basic idea behind this trick is to not store individual elements in the dictionary, but rather to store blocks of  $b \pm 1$  consecutive elements called *chunks*. Let  $D$  be a dictionary data structure that has costs of  $S(n)$ ,  $I(n)$  and  $D(n)$  for searching, inserting, and deleting, respectively, in a set of size  $n$ , and assume that  $D$  has a memory overhead of at most  $cn$  when storing  $n$  elements.

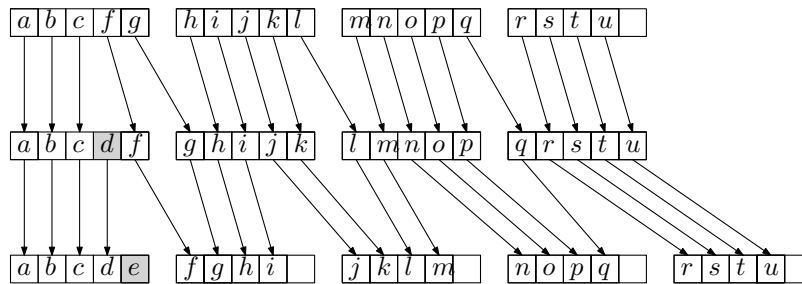


Figure 12.2: Two insertions with a chunk size of  $b = 4$ . When inserting  $d$ , the elements are shifted among existing chunks. When inserting  $e$ , a new chunk is created.

We will use  $D$  to store *chunks*, where a chunk consists of an array of size  $b + 1$ , a counter that indicates whether the chunk contains  $b - 1$ ,  $b$ , or  $b + 1$  elements, and two pointers to the previous and next chunk. Notice that the size of a chunk is therefore  $(b + 1)w + O(1)$ . Furthermore, since each chunk contains at least  $b - 1$  elements, the total memory overhead for storing  $n$  elements is  $O(n(1 + w)/(b - 1))$ . By increasing  $b$ , we can make the overhead as small as we like.

The elements stored in a chunk are kept in sorted order and previous and next pointers are used to link all chunks into a doubly-linked list that keeps all elements in order. Note that a data element  $x$  can be compared to a chunk  $B$  in the following way: To test if  $x < B$ , we compare  $x$  to the first element of  $B$ . To test if  $x > B$  we compare  $x$  to the last element of  $B$ . Otherwise, we use the convention that  $x = B$ . In order to never have a chunk with fewer than  $b - 1$  elements, our dictionary will always store  $b$  dummy elements,  $\lfloor b/2 \rfloor$  of which have the value  $-\infty$  and  $\lceil b/2 \rceil$  of which have the value  $+\infty$ .

To search for an element  $x$  in  $D$ , we perform a search in  $D$  for  $x$  in order to find a chunk  $B$  such that  $x = B$ . The use of the dummy elements guarantees that such a chunk exists. We then perform binary search in the chunk  $B$  to find  $x$ . This takes  $O(S(n/b) + \log b)$  time.

Refer to Figure 12.2. To insert an element  $x$  into  $D$  we first find the chunk  $B$  such that  $x = B$ . Next, we examine up to  $b$  chunks in the neighbourhood around  $B$  using the next and previous pointers. If any of these have fewer than  $b + 1$  elements then by shifting elements between these chunks we can make room for  $x$  without creating any new chunks. Otherwise, we have found  $b$  consecutive chunks, each of which contains  $b + 1$  elements. By shifting elements amongst these chunks, we can convert them into  $b + 1$  chunks, each of which contains  $b$  elements, and insert  $x$  into the appropriate chunk. At this point, we have an extra chunk that needs to be inserted into  $D$ , and we do so. The shifting of elements among the  $b$  chunks takes  $O(wb^2)$  time. Therefore the cost of insertion, if we ignore the cost of finding  $B$ , is  $O(S(n/b) + I(n/b) + wb^2)$ .

To delete an element  $x$  from  $D$ , we find the chunk  $B$  that contains  $x$ . We then search the surrounding  $B$  chunks to see if any of them contains more than  $b - 1$  elements. If so, by shifting elements we can remove  $x$  from  $B$  and still maintain the invariant that all chunks contain at least  $b - 1$  elements, and not have to delete any chunks from  $D$ .

Otherwise, we have found  $b$  consecutive chunks, each of which contains  $b - 1$  elements. By shifting elements among these chunks we can convert them into  $b - 1$  chunks, each containing  $b$  elements, and one empty chunk. We then remove  $x$  from the chunk that contains it and delete the empty chunk from  $D$ . Again, the manipulations on the  $b$  chunks can easily be done in  $O(wb^2)$  time so the cost of

deletion, ignoring the cost of finding  $B$ , is  $O(S(n/b) + D(n/b) + wb^2)$ .

Notice that most of the time we can expect that insertion or deletion can be done only by manipulating chunks, without affecting the data structure  $D$ . We can make this intuition precise by using amortized analysis. Define the potential  $\Phi(B)$  of a chunk  $B$  as

$$\Phi(B) = \begin{cases} 0 & \text{if } B \text{ contains } b \text{ elements} \\ \frac{1}{b}I(n/b) & \text{if } B \text{ } b+1 \text{ elements} \\ \frac{1}{b}D(n/b) & \text{if } B \text{ } b-1 \text{ elements} \end{cases}$$

We define the potential of  $D$  as the sum of the potentials of its chunks.

We only analyze the amortized cost of insertion, since the analysis of deletion is exactly the same. During an insertion there are two cases to consider. If no new chunk is created during the insertion, then the real cost of the insertion is  $O(wb^2)$  and the potential increases by at most  $\frac{1}{b}I(n/b)$ , for an amortized cost of  $O(S(n/b) + \frac{1}{b}I(n/b) + wb^2)$ . If a new chunk is created, then the real cost of the insertion is  $O(S(n/b) + I(n/b) + wb^2)$ . However, in this case, the potential decreases by  $(1 - \frac{1}{b})I(n/b)$  since  $b$  chunks go from having potential  $\frac{1}{b}I(n/b)$  to having potential 0. Therefore, in this case, the amortized cost of insertion is also

$$O\left(S(n/b) + I(n/b) + wb^2 - \left(1 - \frac{1}{b}\right)I(n/b)\right) = O\left(S(n/b) + \frac{1}{b}I(n/b) + wb^2\right).$$

Thus, in either case, the amortized cost of insertion is at most  $O(S(n/b) + \frac{1}{b}I(n/b) + wb^2)$ .

**Theorem 27.** *Given a comparison-based dictionary data structure  $D$  with search cost  $S(n)$ , insertion cost  $I(n)$  and deletion cost  $D(n)$  and that has memory overhead  $cn$ , there exists a comparison-based dictionary data structure  $D'$  with*

1. *memory overhead*  $M(D') = O(n(w + c + 1)/b)$ ,
2. *search cost*  $S'(n) = O(S(n/b) + \log b)$ ,
3. *amortized insertion cost*  $I'(n) = O(S(n/b) + \frac{1}{b}I(n/b) + wb^2)$ , and
4. *amortized deletion cost*  $D'(n) = O(S(n/b) + \frac{1}{b}D(n/b) + wb^2)$

Note that the chunk size  $b$  need not be constant. An interesting choice occurs when the running times of operations on  $D$  are  $O(\log n)$  and we take  $b = \sqrt{\log n}$ . In this case, we get a dictionary  $D'$  with  $O(\log n)$  time for all operations and with a sublinear memory overhead of  $O(n(w + 1)/\sqrt{\log n})$ .

### 12.3 Suboptimal Implicit Dictionaries

Any textbook on data structures will contain the two classics: the priority queue and the dictionary. The classic implementation of a priority queue that contains a maximum of  $n$  elements uses a binary heap whose elements are stored in an array of size  $n$ . By using a simple calculation to determine parent/child relationships, this implementation entirely avoids the need for pointers while still supporting the classic priority queue operations in  $O(\log n)$  time. If, instead of an array, we use the space-efficient vector

implementation of Section 12.1, then we obtain a priority queue data structure with no upper bound on its size, that supports all operations in  $O(\log n)$  time, and whose memory overhead is  $O(\sqrt{n}(w+1))$  where  $n$  is the current size of the priority queue.

The obvious question now is whether or not a dictionary can be implemented as efficiently. The previous section showed how a dictionary data structure can be made to have a memory overhead of  $O(n(w+1)/\sqrt{\log n})$  while still maintaining logarithmic time per operation. In this section we show that, if all elements stored in the dictionary are distinct then the space overhead can be reduced to  $O(\sqrt{n}(w+1))$  while still maintaining fast (but not  $O(\log n)$ ) search time. More specifically, we will show how to implement a dictionary that support insertion, deletion, and searching in  $O(\log^2 n)$  and that has a memory overhead of  $O(\sqrt{n}(w+1))$ .

The structure depends on an encoding trick that hides information in chunks, and this trick requires that all the keys in the dictionary be distinct. Ignore for the time-being that the number,  $n$ , of elements in the dictionary changes over time and choose the chunk size  $b = \lceil c \log n \rceil$  for some constant  $c$ . In the previous implementation of dictionaries, the elements within a chunk are stored in sorted order. By relaxing this just a little, we can encode  $\lfloor (b-1)/2 \rfloor$  bits of information within any chunk. Consider the elements at indices  $2i$  and  $2i+1$  within a chunk. By storing these elements in order or out of order we can encode a 0 or a 1, respectively. Doing this for each  $i \in \{0, \dots, \lfloor (b-1)/2 \rfloor - 1\}$  allows us to encode  $\Theta(c \log n)$  bits but still keeps the chunk “sorted-enough” that we can still access the element that should be at any position  $j$  within the chunk by examining the elements at positions  $2\lfloor j/2 \rfloor$  and  $2\lfloor j/2 \rfloor + 1$ . Furthermore, each encoded bit is easily read and/or modified in constant time by looking at two elements of the chunk.

What should we do with these extra bits that we can encode? The answer is that we should use them to encode pointers and any other data associated with a node. Recall that the chunks are linked into a doubly-linked list and are also part of some dictionary. Assume that the dictionary is implemented as some kind of balanced binary search tree in which each node maintains pointers to its two children and its parent. Therefore, each chunk has  $O(1)$  pointers and auxiliary data<sup>2</sup> associated with it. Rather than store these pointers and auxiliary data explicitly, we can encode them within the chunk, provided that  $c$  is large enough.

What remains is to show how the chunks are stored and managed. The chunks are stored in three vectors  $V_{-1}$ ,  $V_0$  and  $V_{+1}$ , where  $V_i$ ,  $i \in \{-1, 0, +1\}$ , stores all chunks that contain  $b+i$  data items. To perform a search for a key  $k$  in this implicit dictionary, we start at the root chunk  $B$  and compare  $k$  to  $B$ . If  $k \neq B$  then the search proceeds in one of  $B$ 's children and we determine the location of this child by decoding the bits of  $B$ . Thus, one search step in this structure takes  $O(b)$  time, for a total search time of  $O(b \log n)$ .

During an update (insert or delete) operation, up to  $b+1$  chunks are modified, which may require them to move between vectors. To move a chunk  $B$  from  $V_i$  to  $V_j$  we swap  $B$  with the last chunk  $B'$  in  $V_i$ , grow  $V_j$ , append  $B$  to  $V_j$ , and shrink  $V_i$ . During this process, it is important that we update all elements that point to  $B$  and  $B'$  so that they point to the new locations of  $B$  and  $B'$ . This is easy to do since each chunk keeps track of its two children, its parent, and its neighbours in the doubly-linked list of chunks. So, moving a chunk requires updating of  $O(1)$  other chunks and can therefore be done in  $O(b)$  time. Since we may do this for at most  $b+1$  chunks, the manipulations of chunks that take place during an update can be done in  $O(b^2)$  time.

---

<sup>2</sup>For example, if  $D$  is a red-black tree then the auxiliary data consists of a single bit.

By choosing  $b = c \log n$  for sufficiently large constant  $c$ , we get  $O(\log^2 n)$  search time and  $O((w+1) \log^2 n)$  update time. Thus far, we have glossed over the fact that the value of  $n$  (and therefore  $b$ ) is changing over time. The usual trick to deal with this type of problem is to use some value  $n' = \Theta(n)$  and rebuild the entire structure whenever  $n'$  differs from  $n$  by more than a constant factor. Unfortunately, this standard solution is problematic since, unless we are extremely careful, it requires us to manipulate the elements *in place* in order to avoid having both the new and old structure in memory at the same time.

Luckily, there is an elegant solution to this problem. Partition the structure into  $t = O(\log \log n)$  substructures,  $D_1, \dots, D_t$ , where  $D_i$ , for each  $i \in \{1, \dots, t-1\}$ , contains exactly  $2^{2^i}$  elements and  $D_t$  contains the remaining elements. The data structures  $D_1, \dots, D_t$  are stored in vectors  $V_{i,-1}$ ,  $V_{i,0}$  and  $V_{i,+1}$ . When necessary, we increase or decrease the value of  $t$ , thereby creating or deleting 3 vectors, as  $n$  increases or decreases. Another vector, of length  $3t$  is used to maintain pointers to  $V_{i,j}$ , for each  $i \in \{1, \dots, t\}$  and  $j \in \{-1, 0, +1\}$ .

Notice that, since  $D_i$  contains at most  $2^{2^i}$  elements and these are packed into a vector of length  $2^{2^i}$ , the pointers in  $D_i$  can be replaced with indices of length  $\log 2^{2^i} = 2^i$ . Therefore, the data structure  $D_i$  can use a chunk size of  $b_i = c 2^i$  for a sufficiently large constant  $c$ . In this way, the chunk size is always large enough to encode any necessary pointers, and is always bounded by  $O(\log n)$  where  $n$  is the number of items currently stored in the structure.

To search for a key  $x$  we search in each of  $D_1, \dots, D_t$ . To insert a key  $x$  we insert  $x$  into  $D_t$ . To delete a key  $x$  we delete it from the structure  $D_i$  that contains it and then delete any element  $y$  from  $D_t$  and insert  $y$  into  $D_i$ . In this way, the cost of inserting and deleting is bounded by  $O(\log^2 n)$  and the cost of searching is bounded by

$$\sum_{i=1}^t \log^2 2^{2^i} = \sum_{i=1}^t 2^{2^i} = O(2^{2 \log \log n}) = O(\log^2 n) .$$

In terms of memory overhead, a vector of length  $O(\log \log n)$  is used to store pointers to each  $D_{i,j}$ , and each  $D_i$ , for  $i < t$ , has a memory overhead of  $O(\sqrt{2^{2^i}})$ , for a total memory overhead of

$$O(\log \log n) + \sum_{i=1}^{t-1} O(\sqrt{2^{2^i}}(w+1)) + O(\sqrt{n}(w+1)) = O(\sqrt{n}(w+1))$$

**Theorem 28.** *There exists a comparison-based dictionary data structure that supports insertion, deletion, and searching in  $O(\log^2 n)$  amortized time per operation and has a memory overhead of  $O(\sqrt{n}(w+1))$  when storing  $n$  data items each of size  $w$ .*

## 12.4 Discussion and References

Succinct and implicit data structures started out with the practically-motivated problem of minimizing wasted memory in implementations of data structures, but has become a highly technical field. Many of the succinct and implicit data structures presented in the literature are mainly of theoretical interest because of both high implementation complexity and hidden constants in the running times. This chapter attempts to be pragmatic by illustrating the commonly-used techniques that give efficient data structures that are not the theoretically best possible.

The usual definition of an implicit data structure is one that uses a working memory of size  $O(w + 1)$  as well as an array  $A$  of size  $nw$  that grows and shrinks as  $n$  increases or decreases. The assumption about  $A$  growing and shrinking sidesteps the  $\Omega(\sqrt{n})$  lower bound on the memory overhead of data structures that support insertion. It also makes the implementation of an implicit dictionary much more challenging since it eliminates the  $O(\sqrt{n})$  wiggle-room provided by the lower bound.

Some references need to be added here.