Putting your data structure on a diet

Hervé Brönnimann[1] Jyrki Katajainen[2,*] Pat Morin[3]

[1] *Department of Computer and Information Science, Polytechnic University*
Six Metrotech, Brooklyn NY 11201, USA
[2] *Department of Computing, University of Copenhagen*
Universitetsparken 1, 2100 Copenhagen East, Denmark
[3] *School of Computer Science, Carleton University*
1125 Colonel By Drive, Ottawa, Ontario, Canada K1S 5B6

Abstract. Consider a data structure $\mathcal{D}$ that stores a dynamic collection of elements. Assume that $\mathcal{D}$ uses a linear number of words in addition to the elements stored. In this paper several data-structural transformations are described that can be used to transform $\mathcal{D}$ into another data structure $\mathcal{D}'$ that supports the same operations as $\mathcal{D}$, has considerably smaller memory overhead than $\mathcal{D}$, and performs the supported operations by a small constant factor or a small additive term slower than $\mathcal{D}$, depending on the data structure and operation in question. The compaction technique has been successfully applied for linked lists, dictionaries, and priority queues.

Keywords. Data structures, lists, dictionaries, priority queues, space efficiency

1. Introduction

In this paper we consider the space efficiency of data structures that can be used for maintaining a dynamic collection of elements. Earlier research in this area has concentrated on the space efficiency of some specific data structures or on the development of implicit data structures. Our focus is on data-structural transformations that can be used to transform a data structure $\mathcal{D}$ into another data structure $\mathcal{D}'$ that has the same—or augmented—functionality as $\mathcal{D}$, has about the same efficiency as $\mathcal{D}$, but uses significantly less space than $\mathcal{D}$. In particular, we consider a compaction technique that can be applied with minor variations to several different data structures.

One particular aspect of concrete implementations of element containers that has received much attention is the issue of *memory overhead*, which is the amount of storage used by a data structure beyond what is actually required to store the elements manipulated. The starting point for our research was the known implicit data structures where the goal is to reduce the memory overhead to $O(1)$ words or $O(\lg n)$ bits, where n denotes the number

*Partially supported by the Danish Natural Science Research Council under contract 272-05-0272 (project "Generic programming—algorithms and tools").

CPH STL Report 2007-1, February 2007. This paper was presented at the 6th STL Workshop held in Copenhagen on October 30th, 2006.

of elements stored. A classical example is the binary heap of Williams [41], which is a priority queue supporting methods *find-min*, *insert*, and *extract-min*. Another example is the searchable heap of Franceschini and Grossi [20], which is an ordered dictionary supporting methods *find-member*, *find-predecessor*, *find-successor*, *insert*, and *erase*. A searchable heap is a complicated data structure and because of the bit encoding techniques used it can only store a set, not a multiset.

One should observe that implicit data structures are designed on the assumption that there is an infinite array available to be used for storing the elements. In other words, it is assumed that the whole memory is used for the data structure alone and no other processes are run simultaneously. To relax this assumption, a resizable array could be used instead. It is known that any realization of a resizable array requires at least $\Omega(\sqrt{n})$ extra space for pointers and/or elements [6], and realizations exist that only require $O(\sqrt{n})$ extra space [6, 31]. In [6], it was shown that a space-efficient resizable array can perform well under a realistic model of dynamic memory allocation. In particular, the waste of memory due to internal fragmentation is $O(\sqrt{n})$ even though external fragmentation can be large [34].

We call a data structure *elementary* if it only allows key-based access to elements. In particular, all implicit data structures are elementary. An important requirement often imposed by modern libraries (e.g. C++ standard library [3]) is to provide location-based access to elements, as well as to provide iterators to step through a set of elements. In general, for efficiency reasons it might be advantageous to allow application programs to maintain references, pointers, or iterators to elements stored inside a data structure. For the sake of correctness, it is important to keep these external references valid at all times. We call this issue *referential integrity*. For non-elementary priority queues, location-based access is essential since without it methods *erase* or *decrease* cannot be provided. For dictionaries, a finger indicates a location, so to support finger search location-based access must be possible. In the widely-used textbook by Cormen et al. [12, Section 6.5], *handles* are proposed as a general solution to guarantee referential integrity, but this technique alone increases the memory overhead of the data structure in question by $2n$ pointers.

For fundamental data structures, like dictionaries and priority queues studied in this paper, many implementations are available, each having own advantages and disadvantages. In earlier research, the issues considered include the repertoire of methods supported, complexity of implementation, constants involved in running times, worst-case versus average-case versus amortized performance guarantees, memory overhead, and so on. Many implementations are also special because they provide certain *properties*, not provided by implicit data structures. Examples of such properties include the working-set property [26, 39], the queueish property [27], the unified property [26, 39], the ability to *insert* and *erase* in constant time [19, 25, 32], and the (static or dynamic) finger property [2, 9, 11, 24, 25]. These special properties are a large part of the reason that so many implementations of

fundamental data structures exist. Many algorithmic applications require data structures that have one or more of these special properties to ensure the bounds on their running times or storage requirements. Thus, it is not always possible to substitute one implementation for another.

Several dictionary implementations, including variants of height-balanced search trees [8], self-adjusting search trees [1, 22, 39], and randomized search trees [2, 38], have been proposed that reduce the extra storage to two pointers per element. However, when a dictionary has to support location-based access to elements, additional pointers (like parent pointers) are often needed. By using the child-sibling representation of binary trees (see, e.g. [40, Section 4.1]) and by packing the bits describing the type and colour of nodes in pointers (as proposed, for example, in [7]), red-black trees supporting location-based access are obtained that only use $2n + O(1)$ words of memory in addition to the n elements stored.

In contrast with fully-implicit binary heaps, many priority-queue implementations rely on pointers. Actually, pointer-based implementations are necessary to provide referential integrity, and to support *erase* and *decrease*. Of the known implementations supporting *find-min*, *insert*, and *decrease* in constant time in the worst case, relaxed weak queues [16] require $3n+O(\lg n)$ extra words and other structures [4, 14, 18, 29, 30] more. If *decrease* is allowed to take logarithmic time, the memory overhead can be reduced to $2n + O(\lg n)$ words as pointed out, for example, in [16, 13]. We note that the implicit priority queue structure of [35], which is used to study the complexity of implicit priority queues that support *decrease* in $O(1)$ time, addresses the issue of referential integrity by forcing the calling application to maintain a mapping between elements and their location within the structure. Thus, although their priority queue only uses a constant number of words of overhead, any application that wishes to use *decrease* is responsible for keeping track of the locations of the elements to be accessed.

In this paper we elaborate upon a technique that can be used to make a pointer-based data structure that uses cn words of extra storage, for a positive constant c, into an equivalent data structure that uses either εn or $(1+\varepsilon)n$ words of extra storage, for any $\varepsilon > 0$ and sufficiently large $n > n(\varepsilon)$. By applying this technique, any application program can employ the data structure (and its properties) without worrying about the memory overhead. Under reasonable assumptions, the data-structural transformations increase the running times of the operations on the data structure only by a small constant factor independent of ε and/or an additive term of $O(1/\varepsilon)$.

In the case of elementary dictionaries, we can even make the amount of extra space used as small as $O(n/\lg n)$ such that searches and updates are supported in $O(\lg n)$ time. More precisely, the data structure requires exactly n locations for elements and at most $O(n/\lg n)$ locations for pointers and integers. In addition, the whole dictionary can occupy a contiguous segment of memory. Making a reasonable assumption that each of the pointers and integers used takes $O(\lg n)$ bits, the memory overhead of our elementary dictionary becomes $O(n)$ if measured in bits. The same bound—with a

smaller constant factor—is achieved by relying on a succinct representation of binary trees proposed in [37]. When such a representation is applied to a height-balanced search tree like a red-black tree, a data structure is obtained which supports searches in logarithmic time and updates in poly-logarithmic time. When updates are frequent, our approach is clearly preferable.

The compaction technique used by us is simple, and similar to the idea used in degree-balanced search trees [25] where each node contains at least a elements and at most b elements for appropriately chosen a and b. That is, instead of operating on elements themselves, we operate on groups of elements that we call *chunks*. When the compaction technique is applied to different data structures, there are minor variations in the scheme depending on how chunks are implemented (as an array or a singly-linked list), how elements are ordered within chunks (in sorted or in unsorted or in some other order), and how the chunks are related to each other.

The technique itself is not new; it has been explicitly used, for example, in the context of AVL-trees by Munro [36] and in the context of run-relaxed heaps by Driscoll et al. [14]. In the latter paper, referential integrity was not an issue because the elements in the structure are placed in a memory location at the time of insertion and are never moved again. In particular, the space allocated to these elements is never released until the priority queue is destroyed. While this is sufficient for some applications (e.g. in an implementation of Dijkstra's algorithm for solving the single-source shortest-paths problem) it is obviously undesirable in a general-purpose priority queue. An idea similar to ours has also been applied by to general binary search trees by Jung and Sahni [28], but their application of the idea does not guarantee an arbitrarily small memory overhead and may increase the running times of operations on binary search trees by more than a constant factor.

Our purpose in this paper is to show that, carefully implemented, this technique is widely applicable and general. Also, preliminary tests suggest that the technique seems to be of practical value so our plan is to verify this and perform more rigorous experiments.

The remainder of the paper is organized as follows. In Section 2, the data structures studied are defined to fix the notation used throughout the paper. In Section 3, we show how the compaction technique can be used for reducing the memory overhead of any elementary dictionary. More specifically, we extend an old result of Munro [36] by proving that any elementary dictionary can be converted into an equivalent data structure that requires at most $O(n/\lg n)$ extra space without affecting the asymptotic running times of algorithms for searching, inserting, and erasing. In Section 4, we describe two more applications of the technique that transform any linear-size dictionary providing iterator support into a more space-efficient data structure. In Section 5 we prove similar results for priority queues. Finally, in Section 6 we mention some other applications of the compaction technique and conclude with a few final remarks.

2. Data structures studied

When defining the data structures considered, we have used the interfaces specified in the C++ standard [3, Clause 23] as a model, but we use our own conventions when naming the methods. The main reason for using generic method names is brevity and our aim at using a unified notation independent of whether a container stores a set or a multiset.

2.1 Elementary dictionaries

An *ordered dictionary*, or simply a *dictionary*, is a container that stores a collection of elements drawn from an ordered universe and supports insertion and erasure of elements, and search of elements with respect to a given element. We define the methods in a generic form, in which they can take parameters, $\propto$ and r, specified at compile time and parameters, $\mathcal{S}$ and x, specified at run time. Parameter $\propto \in \{\diamond, <, >, =\}$ is a binary predicate used for specifying an interval of interest in $\mathcal{S}$. Binary predicate $\diamond$ is special since $x \diamond y$ is defined to be true for any two elements x and y. We only allow r to take two values 1 and -1, meaning the first and the last element on an interval, respectively.

In its elementary form, a realization of a dictionary should provide (a subset of) the following methods:

search $\langle \propto, r \rangle (\mathcal{S}, x)$. Return a reference to the rth element among all elements v in container $\mathcal{S}$ for which predicate $v \propto x$ is true. If no such element exists, return a reference to a special *null* element.

insert$(\mathcal{S}, x)$. Insert a copy of element x into container $\mathcal{S}$.

erase $\langle \propto, r \rangle (\mathcal{S}, x)$. Remove the rth element among all elements v in container $\mathcal{S}$ for which predicate $v \propto x$ is true. If no such element exists, do nothing.

For these methods argument x can be partially constructed; i.e. if an element is a pair of a key and some satellite data, only the key needs to be specified before a search.

The generic form of *search* is an archetype for 8 different methods depending on how the compile-time parameters are set. If $\mathcal{S}$ stores a set, generic parameter r would be superfluous. Colloquially, invocations of various forms of *search* are called *searches*, and invocations of *insert* and *erase* *updates*. Conventional names for methods *search* $\langle =, 1 \rangle$, *search* $\langle <, -1 \rangle$, *search* $\langle >, 1 \rangle$, *search* $\langle \diamond, 1 \rangle$, *search* $\langle \diamond, -1 \rangle$, *erase* $\langle \diamond, 1 \rangle$, and *erase* $\langle \diamond, -1 \rangle$ would be *find-member*, *find-predecessor*, *find-successor*, *find-min*, *find-max*, *extract-min*, and *extract-max*, respectively. In addition to the above-mentioned methods, there should be methods for constructing and destroying elements and sets, and methods for examining the cardinality of sets, but these are algorithmically less interesting and therefore omitted in our discussion.

Assuming that at a given point in time an elementary dictionary contains n elements, an *implicit dictionary* stores the elements in the first n locations of an infinite array and uses at most $O(1)$ additional words for book-keeping

purposes. Because implicit dictionaries can be complicated [20], our goal is to develop a *semi-implicit dictionary*, as discussed by Munro [36], which stores the data elements compactly at the beginning of the infinite array and only uses $o(n)$ words of extra memory for book-keeping purposes. The main motivation for considering semi-implicit dictionaries is to provide a space-efficient dictionary data structure that can store multisets, which is not known to be possible for implicit dictionaries. Of course, in a practical implementation a resizable array would be used instead of an infinite array.

2.2 Dictionaries providing iterator support

In modern libraries, a dictionary data structure is supposed to support both key-based and location-based access to elements. A location can be specified by a reference, a pointer, or an object specially designed for this purpose—like a locator or an iterator to be defined next.

A *locator* is a mechanism for maintaining the association between an element and its location in a data structure [23, Section 6.4]. A locator follows its element even if the element changes its location inside the data structure. In terms of the C++ programming language, a locator is an object that provides default constructor, copy constructor, copy assignment `=`, unary operator * (prefix), and binary operators `==` and `!=`. That is, for locator p, $*p$ denotes the element stored at the specified location. An *iterator* is a generalization of a locator that captures the concepts *location* and *iteration* in a container of elements. In addition to the operations supported for a locator, a *bidirectional iterator* [3, Clause 24] provides unary operators `++` and `--` (both prefix and postfix). That is, for iterator p, $++p$ returns an iterator pointing to the element following the current element according to some specific order, and an error occurs if p points to the past-the-end element of the corresponding container.

For an ordered dictionary providing location-based access to elements by means of iterators, the methods *search*, *insert*, and *erase* are overloaded as follows:

search $\langle \propto, r \rangle (\mathcal{S}, p, x)$. Start the search from the element pointed to by iterator p and return an iterator to the rth element among all elements v in container $\mathcal{S}$ for which predicate $v \propto x$ is true. If no such element exists, return an iterator to the past-the-end element.

insert$(\mathcal{S}, p, x)$. Insert a copy of element x *immediately before* the element pointed to by iterator p in container $\mathcal{S}$, and return an iterator pointing to the newly inserted element.

erase$(\mathcal{S}, p)$. Remove the element pointed to by iterator p from container $\mathcal{S}$.

To iterate over a collection of elements, it is also possible to access some special locations:

begin$(\mathcal{S})$. Return an iterator pointing to the first element in container $\mathcal{S}$.

end$(\mathcal{S})$. Return an iterator pointing to the past-the-end element for container $\mathcal{S}$.

Since *begin*($\mathcal{S}$) points to the minimum element of $\mathcal{S}$ and --*end*($\mathcal{S}$) to the maximum element (if any), methods *find-min* and *find-max* become superfluous.

An ordered dictionary stores its elements in sorted order. Therefore, it is natural to augment a dictionary (as discussed, for example, in [12, Section 14.1]) to support *order-statistic queries* returning the ith element stored in a dictionary, and *rank queries* returning the position (i.e. index) of a given element in a dictionary. An order-statistic query can be seen as a generalization of generic routine *search* $\langle \propto, r \rangle$ where the parameter r is specified at run time and is allowed to take arbitrary integer values. Instead of providing separate methods to carry out these queries, one could simply upgrade the iterators to provide random access. A *random-access iterator* [3, Clause 24] is as a bidirectional iterator, but iterator additions, iterator subtractions, and iterator comparisons are also allowed. That is, for integer i and iterators p and q, $p + i$, $p - i$, $p - q$, and $p < q$ would be valid expressions. For example, $p + i$ advances i positions from the element pointed to by p and returns an iterator to that element, provided that the past-the-end element is not passed.

2.3 Elementary priority queues

In its elementary form, a *priority queue* stores a collection of elements and supports operations *find-min*, *insert*, and *extract-min*. We want to point out that we do not consider elementary priority queues since a fully-implicit priority queue exists that can support *find-min* and *insert* in $O(1)$ worst-case time, and *extract-min* in $O(\lg n)$ worst-case time [10], n being the number of elements stored. As mentioned in Section 1, in a realistic model of computation, the amount of extra space used by such a data structure is only $O(\sqrt{n})$. In [10], it was even shown that there exists a fully-implicit double-ended priority queue, i.e. a priority queue that supports *find-max* and *extract-max* as well.

2.4 Priority queues providing locator support

In a general form, a *priority queue* is an element container that provides methods *find-min* and location-based *erase* as well as (a subset of) the following methods:

insert($\mathcal{Q}, x$). Insert a copy of element x into container $\mathcal{Q}$ and return a locator pointing to the newly inserted element.

borrow($\mathcal{Q}$). Remove an *unspecified* element from container $\mathcal{Q}$ and return a locator pointing to that element. If $\mathcal{Q}$ is empty, return a locator pointing to the past-the-end element.

decrease($\mathcal{S}, p, x$). Replace the element pointed to by locator p with element x, which is assumed to be no greater than the original element pointed to by p.

Of these methods, *borrow* is non-standard, but according to our earlier research [15, 17] it is a valuable tool, for instance, in data-structural transformations. Our point to include it in the repertoire of priority-queue operations is to show that our compaction scheme can handle it easily: If the original priority queue provides it, the compact version will do the same at about equal efficiency.

As dictionaries, priority queues could also provide iterator support. However, in a priority queue there is no natural order of storing the elements. In a basic form, an iteration starting from *begin* and ending at *end* may just be guaranteed to visit all elements. Iterator operations could be realized in the same way as for dictionaries, so we just concentrate on priority queues that support location-based access to elements.

3. Compact elementary dictionaries

In this section we describe transformations that can be applied to elementary ordered dictionaries to make them more space-efficient. Our approach is to partition the data into chunks and store these chunks in the ordered dictionary under transformation (skiplist, 2-3 tree, red-black tree, etc). To avoid confusion, we will refer the data structure we are using as "the data structure" ($\mathcal{D}$) and the data structure we are describing as "the dictionary" ($\mathcal{D}'$). Similarly, we will use the terms chunks and elements to refer to groups of elements and individual elements, respectively.

We study three different schemes based on this idea. All schemes have the following in common: The data structure is used for storing chunks and the elements of each chunk are stored in an array. Each chunk has a *header* containing a constant amount of data, the elements stored within each chunk are kept in sorted order, and the chunks themselves are stored in a doubly-linked list (the pointers for which are stored in the header). At all times we maintain the *ordering invariant* that every element in chunk A is less than or equal to every element in chunk B if A appears before B in this list. Thus, a traversal of this *chunk list* is sufficient to output all elements in sorted order.

The chunks are stored in the data structure in the same order they appear in the list. Since chunks will be reallocated frequently, the data structure only stores pointers to the chunks. That is, each node in the data structure stores a pointer to a header and that header stores a pointer back to this node. In this way, when a chunk is reallocated, only these pointers need to be updated. To insert a new chunk into the data structure or to erase an existing chunk from the data structure, we rely on location-based access. To search an element in the data structure, we can compare an element to a chunk as follows. To test if element x is less than chunk A, we test if x is less than the first element of A. To test if x is greater than A, we test if x is greater than the last element of A. If x is neither greater than nor less than A, then x is equal to A.

With this representation, it is easy to see that *search* can be implemented by performing at most one search in the data structure followed by one (linear or binary) search in a chunk. The special searches *find-min* and *find-max* are even easier; we simply return the first element of the first chunk or the last element of the last chunk, respectively. Our implementations differ in how they implement *insert* and *erase*.

Even if the compaction technique can be applied for many different kinds of data structures, we get our best results for data structures that fulfil the following *regularity requirements*:

Referential integrity: External references must be kept valid at all times. This is essential because of the pointers from headers to nodes.

Location-based access: Location-based *insert* and *erase* must be supported, but this assumption is not strictly necessary if the elements to be stored are known to be distinct.

Side-effect freeness: Let x, y, and z be three consecutive items stored in the data structure. It should be possible to replace y with y' without informing the data structure, as long as $x \leq y' \leq z$. This is essential because of reallocation of chunks.

Little redundancy: Each item is stored only once at the place pointed to by its locator (i.e. the pointer from a header).

Freely movable nodes: Every node knows who points to it so that, when the node is moved, the pointers at the neighbouring nodes can be corrected accordingly.

Constant in-degree: There must be at most a constant number of pointers pointing to a node so that nodes can be moved in constant time.

For example, a carefully implemented red-black tree fulfils these requirements.

3.1 Fixed-sized chunks

Let b be a positive integer, $b \geq 2$. The simplest realization of the above ideas is to store the chunks as fixed-size arrays, each of size $b+1$. With the exception of at most one chunk, each chunk will store either $b-1$, b, or $b+1$ elements.

To insert element x into the data structure, we first search for x in the data structure. The result of this is some chunk A that is either the first chunk, the last chunk, or a chunk in which x is greater than the first element and less than the last element. Next, by traversing the chunk list we examine up to b consecutive chunks including A.

If any of these b consecutive chunks contains fewer than $b+1$ elements, then we can reassign the elements to chunks so as to place x into some chunk and restore our ordering invariant. Otherwise, each of the b chunks contains $b+1$ elements. In this case we create a new chunk and reassign these $b(b+1)$ elements across the $b+1$ chunks so that each chunk contains exactly b elements. Because of these element movements the data structure

must guarantee side-effect freeness. Finally, we place x in the appropriate chunk and insert the newly created chunk into the data structure.

Erasing an element is similar to insertion. To erase element x, we first search for the chunk A containing x. Then, in the b consecutive chunks including A either: (1) there is some chunk containing b or $b+1$ elements, in which case we remove x and redistribute elements so as to maintain the ordering invariant, or (2) all b chunks contain exactly $b-1$ elements, in which case we erase one chunk from the data structure and redistribute the $b(b-1)-1$ elements among the $b-1$ remaining chunks so that each chunk stores b elements, except one, that contains $b-1$ elements.

It is clear that both insertion and erasure require one search in the data structure, $O(b^2)$ work, and at most one insertion or erasure, respectively, in the data structure. Indeed a slightly more careful amortized analysis shows that a sequence of n *insert*/*erase* operations on the dictionary only requires $O(n/b)$ *insert*/*erase* operations on the data structure.

Theorem 1. *Given an ordered dictionary that for a collection of n elements requires $S(n)$, $I(n)$, and $E(n)$ time to search, insert, and erase (respectively) and that has a memory overhead of cn words for a positive constant c, we can construct an equivalent dictionary that requires*

1. *$O(S(n/b)+\lg b)$, $O(S(n/b)+I(n/b)+b^2)$, and $O(S(n/b)+E(n/b)+b^2)$ worst-case time to search, insert, and erase (respectively),*
2. *$O(S(n/b)+\lg b)$, $O(S(n/b)+\frac{1}{b}I(n/b)+b^2)$, and $O(S(n/b)+\frac{1}{b}E(n/b)+b^2)$ amortized time to search, insert, and erase (respectively), and*
3. *$O(n/b)$ extra space.*

One issue that several researchers have been concerned with is that of fragmentation. *Internal fragmentation* is the space allocated that is not used. *External fragmentation* is the space that cannot be used because the allocation of memory segments was disadvantageous. Using the above scheme, in combination with simple techniques for maintaining a pool of buffers for the chunks, the entire dictionary described above can be made to live in one resizable array that has space for $n+O(n/b)$ elements and $O(n/b)$ pointers/integers. In this case, the internal and external fragmentation is no worse than the fragmentation required in an implementation of a resizable array.

3.2 Variable-sized chunks

One drawback of the scheme described in the previous subsection is the $O(b^2)$ term in the running time for *insert* and *erase*. In this subsection we describe an alternative that avoids this problem but requires more extensive memory management and, consequently, for which it is more difficult to avoid fragmentation. In this scheme, all but one chunk contains somewhere between $b/2$ and $2b$ elements, inclusive. Throughout this section we assume b is an even integer, $b \geq 2$, so that $b/2$ is an integer. However, unlike the previous scheme, each chunk is allocated to the precise size required to store the number of elements it currently contains.

An insertion of an element x into the data structure proceeds as follows: The chunk A that should contain x is located. If A contains fewer than $2b$ elements, then A is resized to make room for x and x is added to A. Otherwise, if A already contains $2b$ elements, A is split into two chunks, each containing b elements, one of the new chunks is inserted into the data structure, and x is added to the appropriate chunk.

The *erase* procedure is similar to the *insert* procedure. If the chunk A containing x contains more than $b/2$ elements, then A is resized and x is removed. Otherwise, we check one of the neighbouring chunks, B, of A. If B contains more than $b/2$ elements, an element is removed from B and used to replace x in A and then B is resized. Otherwise, each of A and B contains exactly $b/2$ elements, in which case we merge them into a single chunk containing b elements and erase one of the chunks from the data structure.

Again, it is clear that both *insert* and *erase* can be done using at most one search, $O(b)$ time to operate on the at most two chunks involved, and at most one insertion and erasure. Again, a simple amortization argument shows that a sequence of n *insert* and *erase* operations on the dictionary results in only $O(n/b)$ *insert* and *erase* operations in the data structure.

Theorem 2. *Given an ordered dictionary that for a collection of n elements requires $S(n)$, $I(n)$, and $E(n)$ time to search, insert, and erase (respectively) and that has a memory overhead of cn words for a positive constant c, we can construct an equivalent dictionary that requires*

1. *$O(S(n/b)+\lg b)$, $O(S(n/b)+I(n/b)+b)$, and $O(S(n/b)+E(n/b)+b)$ worst-case time to search, insert, and erase (respectively),*
2. *$O(S(n/b)+\lg b)$, $O(S(n/b)+\frac{1}{b}I(n/b)+b)$, and $O(S(n/b)+\frac{1}{b}E(n/b)+b)$ amortized time to search, insert, and erase (respectively), and*
3. *$O(n/b)$ extra space.*

Knowing that element-based operations on an ordered dictionary take at least logarithmic time in the worst case, it is tempting to choose $b = \lg n$. The problem with this choice is that n, the number of elements stored, is varying whereas b must be kept fixed. To avoid this problem, we use Frederickson's partitioning scheme [21] and partition the data structure into $O(\lg \lg n)$ separate *portions* of doubly-exponentially increasing sizes 2^{2^1}, 2^{2^2}, 2^{2^3}, and so on, except that the last portion can be smaller. Inside the ith portion we can use a fixed chunk size $b_i = 2^i$. Insertions are done in the last portion, erasures are performed by borrowing an element from the last portion, and searches visit all structures but the overall cost is dominated by the cost incurred by the last portion.

Technically, to handle the doubly-exponential grow of the portion sizes, we assume that functions expressing running times are *reasonably growing*, i.e. for any non-negative integers k and n, $n \leq 2^{2^k}$, and function $T(n)$, $T(n) \leq T(2^{2^k})$ and $2 \cdot T(2^{2^k}) \leq T(2^{2^{k+1}})$. In particular, if the data structure supports searches in $O(\lg n)$ time, all $O(\lg \lg n)$ searches still take $O(\lg n)$ time in total. Under this assumption the performance of the dictionary

operations can be summarized as follows.

Theorem 3. *Given an ordered dictionary that for a collection of n elements requires $S(n)$, $I(n)$, and $E(n)$ time to search, insert, and erase (respectively) and that has a memory overhead of cn words for a positive constant c, we can construct an equivalent dictionary that requires*

1. *$O(S(n/\lg n)+\lg\lg n)$, $O(S(n/\lg n)+I(n/\lg n)+\lg n)$, and $O(S(n/\lg n)+E(n/\lg n)+I(n/\lg n)+\lg n)$ worst-case time to search, insert, and erase (respectively),*
2. *$O(S(n/\lg n)+\lg\lg n)$, $O(S(n/\lg n)+\frac{1}{\lg n}I(n/\lg n)+\lg n)$, and $O(S(n/\lg n)+\frac{1}{\lg n}E(n/\lg n)+\frac{1}{\lg n}I(n/\lg n)+\lg n)$ amortized time to search, insert, and erase (respectively), and*
3. *$O(n/\lg n)$ extra space.*

As an example, we can apply Theorem 3 to any worst-case efficient balanced search tree such as a red-black tree to obtain the following result:

Corollary 1. *There exists an ordered dictionary that supports search, insert, and erase in $O(\lg n)$ worst-case time, find-min and find-max in $O(1)$ worst-case time, and has a memory overhead of $O(n/\lg n)$ words.*

3.3 Semi-implicit dictionaries

One problem with the foregoing construction relying on variable-sized chunks is excess memory fragmentation. This is in big contrast with the result proved by Munro [36, Theorem 1] that a semi-implicit dictionary can be maintained in a contiguous segment of memory requiring at most $n + b^2$ locations for elements and at most $b + O(n/b)$ locations for additional information. So even if chunks have a size varying between 1 and b, internal fragmentation can be avoided almost completely. In this subsection we improve Munro's result in two ways. First, we show that internal fragmentation can be avoided altogether. In particular, the chunks can be kept in a contiguous memory segment of size n without wasting any space between the chunks. Second, we show that by applying Frederickson's partitioning scheme it is possible to reduce the memory overhead to $O(n/\lg n)$ extra words without any loss in the asymptotic efficiency of dictionary operations.

More formally, our goal is to prove the following theorem.

Theorem 4. *Let n denote the number of elements stored in an ordered dictionary $\mathcal{D}$ prior to each operation. Assume that $\mathcal{D}$ supports search, insert, and erase in $S(n)$, $I(n)$, and $E(n)$ time (respectively), requires $O(n)$ space, and fulfils all of our regularity requirements. It is possible to transform $\mathcal{D}$ into a semi-implicit ordered dictionary $\mathcal{D}'$ that supports search, insert, and erase in $O(S(n/\lg n)+\lg\lg n)$, $O(I(n/\lg n)+\lg n)$, and $O(S(n/\lg n)+E(n/\lg n)+I(n/\lg n)+\lg n)$ time (respectively). The dictionary $\mathcal{D}'$ requires exactly n locations for elements and at most $O(n/\lg n)$ locations for pointers and integers. Furthermore, the whole dictionary $\mathcal{D}'$ can occupy a contiguous segment of memory.*

We prove this result by applying a subset of techniques used by Franceschini and Grossi [20] in the construction of an implicit dictionary. Interestingly, the semi-implicit data structure has two advantages compared to the implicit data structure even if the latter achieves better space utilization. First, the semi-implicit dictionary uses a fewer number of pointers, thus being simpler. Second, the semi-implicit dictionary can be used to store a multiset since no encoding of pointers (using pairs of distinct elements) is necessary.

It turns out that memory management will be more complex if we allow merges and splits of chunks. To make memory management easier, we change the representation of the chunks such that, instead of using a single array for storing the elements, we maintain two arrays. Also, we make sure that the size of a chunk is always increased by one or decreased by one. We call the first of the two arrays the *trunk array* and the second the *overflow array*. A trunk array stores exactly b elements and an overflow array, if it exists, at least one element and at most b elements. The header of each chunk stores the sizes and start positions of both arrays. And as earlier, the data structure is used for storing pointers to the headers.

The key tool used by us is the compactor-zone technique used, for example, in [20]. All arrays of the same size are kept in the same *zone* which is a contiguous memory segment, and the b zones are stored compactly after each other (some of which may be empty): the first zone holds arrays of size one, the second zone holds arrays of size two, and so on. To be able to keep the zones without any gaps, we maintain an invariant that in each zone there can be at most one array that is split into two *pieces*. If there is a split array in a zone, the two pieces are kept at their respective ends of that zone. A separate array of size b, called the *zone table*, is used to recall the beginning and end of each zone, as well as to maintain detailed information about the split array in each zone, if there is any.

Let us analyse the memory overhead of this construction. The data structure is assumed to be of linear size so the total space used by it is $O(n/b)$. Because each node of the data structure is assumed to have a constant in-degree and because each chunk header also has a pointer back to the corresponding node in the data structure, nodes can be freely moved in memory. After each move, all neighbouring objects are just informed about this move and their pointers are updated accordingly. Chunk headers are also constant-size objects and they know the objects who point to it. That is, chuck headers can also be moved freely in memory in constant time. The total amount of space used by chunk headers is $O(n/b)$ words and that used by the zone table $O(b)$ words. If the zone table is maintained as a circular array, it can be smoothly rotated whenever necessary. So this component can be moved in memory, too. By storing the n elements held in the compactor zones first and thereafter the freely movable objects, the whole dictionary can be stored in a contiguous segment of memory, as claimed.

To insert a new element x, we search for the proper node using the data structure and the proper position of x inside the chunk corresponding to the found node. After putting x into its proper place (to retain the sorted order

of elements) and moving the following elements one position to the right, the overflow array becomes one element too short. To handle this overflow, that particular overflow array (if there is any) is swapped with the last complete array of the zone which involves a block swap and some pointer updates in the zone table. Then this array is interchanged with the second piece of the split array (if any). Of course, proper tests should be carried out to see whether the current overflow array is the split array in its zone. This special case is handled easily by swapping the split array with the last full array of the zone; if there is only one array that is split, the array is made complete by interchanging the two pieces. So now the overflow array, if it was of size i, occupies the last i locations of the ith zone. Thereafter, the higher zones are rotated one element to the right. In this process the zone boundaries must be updated in the zone table. Also, the information about the split arrays are to be updated. Now a hole has been created for the element that overflowed and the overflow array is updated to include this element. Finally, this array is made part of the $(i+1)$st zone. Since at the beginning of this zone there can be a piece of a split array, the new array and this piece are interchanged so that this piece locates at the beginning of the zone. Again the zone table is updated accordingly.

Erasure is symmetric to insertion; in a sense everything is just done backwards. Searching involves searching in the data structure and searching inside the relevant chunk. Even if a chunk is divided into two pieces, binary search can still be used.

Let $S(n)$, $I(n)$, and $E(n)$ be the running time of *search*, *insert*, and *erase* in the data structure, respectively. Since the size of a chunk is at most b, since there are at most $O(n/b)$ chunks, and since there are at most b zones, the running time of *insert* is $S(n/b) + I(n/b) + O(b)$. Similarly, the running time of *erase* is $S(n/b) + E(n/b) + O(b)$. Due to binary search, the running time of *search* is $S(n/b) + O(\lg b)$.

Now we can use Frederickson's partitioning and maintain $O(\lg \lg n)$ portions of doubly-exponentially increasing sizes. Since the size of the portions is fixed, except that of the last portion, the elements stored in these portions can be stored contiguously, portion by portion in increasing order. The moving parts of each portion can be stored after the elements. Because of a change in one portion due to *insert* or *erase*, it may be necessary to rotate the following portions. However, since there are at most $O(\lg \lg n)$ portions, this rotation of memory areas will only increase the overall running time of dictionary operations by an additive term of $O(\lg \lg n)$. This completes the proof of Theorem 4.

4. Compact dictionaries providing iterator support

The data structural transformations described in the previous sections can be used to reduce the memory overhead of dictionary data structures. However, they break down when the interface to those data structures allows

access to elements using locators or iterators. The reason for this is that, as updates are performed, individual elements migrate between chunks. Thus, without some special mechanism the iterators pointing to these elements will become invalid.

In this section we show two ways in which the dictionaries can be extended to allow iterator access. The first method is preferable when the total number of elements referenced by iterators is small and when unused iterators are cleaned up. The second method is preferable when there are many iterators in use simultaneously.

4.1 Lists of element handles

In this section we present a method of maintaining iterators where the memory overhead at any given time is proportional to the total number of iterators in use at that time. This method augments either of the two schemes described in the previous section in the following way: The header of each chunk is augmented so that it maintains a pointer to an ordered dictionary with iterator support that allows constant time insertion and erasure.[1] This dictionary, which we call the *element handle list* for the chunk stores element handles. Each *element handle* maintains a reference count, a pointer to the chunk that contains the element referenced, and the index within the chunk of the element referenced. The iterators themselves are pointers to element handles.

When a new iterator for an element x is created, the element handle list for the chunk containing x is searched. If there is already an element handle for x, its reference count is increased by one and a pointer to this handle is used as the iterator. Otherwise, a new element handle for x is created, added to the list, and initialized with a reference count of 1. Since an element handle list stores at most b element handles, this process takes $O(\lg b)$ time.

When an iterator for an element x is destroyed, the reference count of the element handle is decreased by one. If the reference count drops to zero, then the element handle is erased from the list and the memory associated with it is freed. This process takes $O(1)$ time.

When the algorithms of the previous section operate on chunks by either growing them, splitting them, merging or moving elements between them, the element handles and element handle lists associated with them must be updated appropriately to reflect the changes. However, these changes only require changing the contents and/or pointers of the element handles and therefore, once they are updated, the iterators remain valid. This updating of element handles is easily done without increasing the ($O(b)$) running time of each of the above operations.

It is clear that the iterators defined this way offer constant time access to the corresponding elements. They can be created in $O(\lg b)$ time and destroyed in $O(1)$ time. Furthermore, because the element handle lists support

[1] In practice, this is probably overkill and a sorted doubly-linked list would suffice, but this will increase the search time in Theorem 5 to $O(S(n/b) + b)$.

iteration and constant time insertion and erasure, iterators can be increased and decreased in $O(1)$ time. If k is the total number of elements in the dictionary referenced by iterators at any point in time, then the memory overhead of the dictionary, at that point in time, is $O(n/b + k)$. Note that the $O(k)$ term is in some sense necessary since, if a program using the dictionary maintains k active iterators, then these must be stored somewhere within the program and thus there is already $\Omega(k)$ space used within the program to keep track of these iterators. Applying this augmentation to the data structure of Theorem 2 we obtain the following result:

Theorem 5. *Given an ordered dictionary with iterator support that for a collection of n elements requires $S(n)$, $I(n)$, and $E(n)$ time to search, insert, and erase (respectively) and that has a memory overhead of cn words for a positive constant c, we can construct an equivalent dictionary that requires*

1. *$O(S(n/b) + \lg b)$, $O(I(n/b) + b)$, and $O(E(n/b) + b)$ worst-case time to search, insert, and erase (respectively),*
2. *$O(S(n/b) + \lg b)$, $O(\frac{1}{b}I(n/b) + b)$, and $O(\frac{1}{b}E(n/b) + b)$ amortized time to search, insert, and erase (respectively), and*
3. *$O(k + n/b)$ extra space where k is the number of elements currently referenced by iterators.*

4.2 Storing chunks as linked lists

An alternative method of implementing ordered dictionaries providing iterator support is to apply any of the schemes used in Section 3 but, instead of using a sorted array for each chunk, to use a sorted singly-linked list. In this case, a iterator pointing to element x is implemented simply as a pointer to the list node containing x. All the other operations remain the same except that a search within a chunk must now be done by linear search as opposed to binary search.

To save space, we reuse the pointer at the last list node to point to the header of the corresponding chunk. Because of this extension, each list node must be augmented with a bit indicating the type of the pointer stored; that is, whether the pointer is to the next list node or to the chunk header. Iterator operations *begin*, *end*, and operator `++` are trivial to implement provided that the underlying data structure provides iterator support. Operator `--` is also simple, though more expensive, since we have to access the chunk header to get to the predecessor, or to the last element of a chunk if the given iterator points to the first element of a chunk. To guarantee that iterator operations take $O(1)$ time we must fix the size of the chunks to a constant.

Compared to the data structure of Theorem 2, the memory overhead is increased by one pointer plus one bit per element. In normal circumstances, the last two bits of a pointer are unused because of word alignment, so an indicator bit and a pointer can be packed into one word. That is, under the assumption that this bit packing is possible, the support of bidirectional iterators introduces an additional memory overhead of one word per element.

The efficiency of the construction based on singly-linked lists is summarized as follows.

Theorem 6. *Given an ordered dictionary providing iterator support that for a collection of n elements requires $S(n)$, $I(n)$, and $E(n)$ time to search, insert, and erase (respectively) and that has a memory overhead of cn for a positive constant c, we can construct an equivalent dictionary that requires*

1. *$O(S(n/b) + b)$, $O(I(n/b) + b)$, and $O(E(n/b) + b)$ worst-case time to search, insert, and erase (respectively),*
2. *$O(S(n/b) + b)$, $O(\frac{1}{b}I(n/b) + b)$, and $O(\frac{1}{b}E(n/b) + b)$ amortized time to search, insert, and erase (respectively), and*
3. *$n + O(n/b)$ words of additional storage under the assumption that a bit and a pointer can be packed into one word.*

As a concrete example of the usage of Theorem 6, let us apply it to the ordered dictionary presented in [5]. By making b large enough so that $\Theta(\frac{1}{b}) = \varepsilon$, we get the following corollary.

Corollary 2. *There exists an ordered dictionary that supports location-based updates in $O(1/\varepsilon)$ worst-case time, finger searches in $O(\lg d + 1/\varepsilon)$ worst-case time, d being the distance to the target, and requires at most $(1 + \varepsilon)n$ words of extra storage, for any $\varepsilon > 0$ and sufficiently large $n > n(\varepsilon)$.*

4.3 Augmentation with random-access iterators

The standard way of augmenting a binary search tree, like a red-black tree, to support order-statistic queries and rank queries is to store the size of underlying subtree at each node [12, Section 14.1]. Actually, such an augmentation for a single data structure can be used to provide the same facilities for any dictionary. When the support for order-statistic queries and rank queries is available, random-access iterators can be provided quite easily. Next we show that a compact dictionary produced by our data-structural transformations can be augmented to support random-access iterators with a negligible memory overhead.

We maintain a leaf-oriented balanced tree above the chunks such that in the tree there is one leaf per chunk. Each leaf should store the size of the corresponding chunk and each internal node should store the sum of the sizes stored at its children. Such a tree can be easily made to support location-based insertions and erasures of leaves, and increments and decrements of the size of a leaf such that the running time of these operations is logarithmic on the number of leaves stored. The construction is very similar to that provided for red-black trees in [12, Section 14.1]. When such a tree is available, the total sum of sizes of the leaves prior to a given node can computed by traversing the tree in a bottom-up manner starting from the given node. Similarly, the tree can be used for finding the chunk containing the ith element by traversing the tree in a top-down manner starting from the root.

Assume that iterator p and integer j are given and that we want to compute $p + j$. First, the rank of the element pointed to by p is calculated. Let

this be i. Second, integer assignment $k \leftarrow i + j$ is executed. Third, the kth smallest element is retrieved and an iterator pointing to that element is returned. In the first step, the rank in the linked list is determined and then the tree above the chunks is used to determine the number of elements prior to the given chunk. Similarly, in the third step, the tree is first traversed down and after that the list corresponding to the retrieved chunk is consulted.

Iterator subtraction and iterator difference can be handled in a similar manner. Iterator comparisons can be carried out by determining the rank of the given nodes, and thereafter comparing the ranks.

The above discussion is summed up in the following theorem.

Theorem 7. *A compact dictionary storing n elements and using chunks of size b can be augmented to provide random-access iterators such that the running time of random-access-iterator operations is $O(\lg n + b)$. As a consequence of this augmentation, the running time of updates is increased by an additive term of $O(\lg n)$ and the memory overhead by $O(n/b)$ words.*

5. Compact priority queues providing locator support

Any ordered dictionary can be used as a priority queue as well. However, two methods, *insert* and *decrease*, have turned out to be special since there exist priority queues that support both of them in constant time. When our previous transformations are applied to such a priority queue, the resulting data structure is not necessarily optimal with respect to these operations. Therefore, in this section we describe a transformation that is suitable for priority queues.

We again use the idea of partitioning the elements into chunks. However, this time the chunks are not sorted, but the minimum element within each chunk is stored as the first element of the chunk. The minimum (first) element within each chunk is referred to as the *representative* of the chunk. Furthermore, the chunks are not linked into a list, and there is no special relationship between the elements of one chunk and the elements of another chunk. One special chunk, denoted by B and called the *buffer*, is treated differently. All chunks except B are stored in a priority queue where the key of a chunk is its representative. As before we refer to the data structure we are transforming as the "the data structure" and the data structure we are describing as "the priority queue".

Either of the two methods described in Section 4 could be used to provide support for locators (as well as for iterators). That is, we can either store each chunk as an array and create a locator handle list for each chunk, or we can store each chunk as a singly-linked list. In either case, locators for an element can be created in constant time if the element is the first or last element of its chunk.

Next we consider how the various priority-queue operations can be realized.

find-min($\mathcal{Q}$). The minimum is stored either in buffer B or the minimum representative in the data structure. A locator for the minimum of the two elements can be returned. Since both the relevant elements are the first elements in their respective chunks, this operation can be accomplished in $O(1)$ worst-case time.

insert($\mathcal{Q}, x$). Insert x into buffer B. If x is smaller than the current minimum of B, store it as the first element; otherwise, store it somewhere convenient. If the size of B is equal to $2b$, insert B into the data structure and create a new empty buffer. In addition to a single invocation of *insert* for the data structure, only a constant amount of work is done (under the assumption that chunks know their size).

borrow($\mathcal{Q}$). If B is non-empty, extract a convenient element and return (an locator pointing to) it. Otherwise, B is empty, extract one chunk from the data structure and make that a new buffer B. If, after this, B is still empty, return an locator to the past-the-end element and stop. Otherwise, take a convenient element from B and return (an locator pointing to) that element. Again one invocation of *borrow* for the data structure may be necessary, but otherwise only a constant amount of work is done.

decrease($\mathcal{Q}, p, x$). Access the header of the chunk to which p points. Thereafter, carry out the element replacement and make the replaced element the representative of the chunk if it is smaller than the previous representative. If the chunk is not buffer B and if the representative of the chunk changes, invoke *decrease* on the data structure. The work done in addition to one invocation of *decrease* depends on the method being used. If arrays are being used to store chunks, then the element handle contains a pointer to the header of the chunk so finding the header takes $O(1)$ time. If the chunks are stored as lists, it may be necessary to traverse the entire list to find the header of the chunk and adjust the appropriate pointers, so the time is $O(b)$.

erase($\mathcal{Q}, p$). Access the header of the chunk to which p points. If the size of the chunk becomes smaller than b or if the representative of the chunk is removed, remove the chunk from the data structure. Then merge the removed chunk with buffer B to form a new buffer. If the size of B is larger than $2b$, cut off a piece of size $2b$ from B, make the rest a buffer and insert the chunk cut off into the data structure. To sum up, one invocation of *erase* and *insert* for the data structure may be necessary, but the other work done is proportional to b.

The discussion is summed up in the following two theorems, that come from applying each of the two methods in Section 4, respectively. Let n denote the number of elements stored prior to each operation. We use the notation that, for the underlying data structure, *find-min*, *insert*, *borrow*, *decrease*, and *erase* require $F(n)$, $I(n)$, $B(n)$, $D(n)$, and $E(n)$ time, respectively. To get the claimed result, we should just make b large enough to get the amount of extra storage used down from $O(n)$ to εn.

Theorem 8. *A priority queue that uses $O(n)$ words of extra storage can be transformed into an equivalent priority queue that, when at most k elements are referenced by iterators, uses $O(k) + \varepsilon n$ words of extra storage, for an*

arbitrary $\varepsilon > 0$ and sufficiently large $n > n(\varepsilon)$, and performs find-min, insert, borrow, decrease, and erase in $F(\varepsilon n)+O(1)$, $I(\varepsilon n)+O(1)$, $B(\varepsilon n)+O(1)$, $D(\varepsilon n)+O(1)$, $E(\varepsilon n)+I(\varepsilon n)+O(1/\varepsilon)$ time, respectively.

Theorem 9. *A priority queue that uses $O(n)$ words of extra storage can be transformed into an equivalent priority queue that uses $(1+\varepsilon)n$ words of extra storage, for an arbitrary $\varepsilon > 0$ and sufficiently large $n > n(\varepsilon)$, and performs find-min, insert, borrow, decrease, and erase in $F(\varepsilon n)+O(1)$, $I(\varepsilon n)+O(1)$, $B(\varepsilon n)+O(1)$, $D(\varepsilon n)+O(1/\varepsilon)$, $E(\varepsilon n)+I(\varepsilon n)+O(1/\varepsilon)$ time, respectively.*

To illustrate the power of Theorem 9, let us apply it to the priority queue described in [18]. As an outcome we get the following result:

Corollary 3. *There exists a priority queue that supports find-min and insert in $O(1)$ worst-case time, decrease in $O(1/\varepsilon)$ worst-case time, erase in $O(\lg n+1/\varepsilon)$ worst-case time including at most $\lg n+3\lg\lg n+O(1/\varepsilon)$ element comparisons, and requires at most $(1+\varepsilon)n$ words of extra storage, for any $\varepsilon > 0$ and sufficiently large $n > n(\varepsilon)$.*

6. Discussion

We showed that many data structures of size $O(n)$ can be put on a diet, so that the memory overhead is reduced to $O(n/\lg n)$, εn, or $(1+\varepsilon)n$ words and elements for any $\varepsilon > 0$ and sufficiently large $n > n(\varepsilon)$. This diet does not change the running times of the operations on the data structures except by a small constant factor independent of ε and/or an additive term of $O(1/\varepsilon)$.

We considered ordered dictionaries and single-ended priority queues, but the compaction technique can be used for slimming down other data structures as well. Other applications, where the technique is known to work, include positional sequences (doubly-linked lists), unordered dictionaries (e.g. hash tables relying on linear hashing [33]), and double-ended priority queues. The details follow closely the guidelines given in this paper, so we leave them to the interested reader.

The practical value of the compaction technique is that users of data structures need not worry about the memory overhead associated with the data structure they choose. If it becomes a problem, they can simply apply the technique and the problem is solved. The theoretical value of the compaction technique is that designers of data structures need no longer make unnecessarily complicated algorithms and representations for the sake of reducing the memory overhead. They can design data structures that have fast running times and useful properties, and put less emphasis on the memory overhead of the data structures.

Acknowledgements

We thank Nicolai Esbensen and Andrew Turpin for helpful discussions.

References

[1] A. Andersson, General balanced trees, *Journal of Algorithms* **30**, 1 (1999), 1–28.

[2] C. R. Aragon and R. Seidel, Randomized search trees, *Algorithmica* **16**, 4 (1996), 464–497.

[3] British Standards Institute, *The C++ Standard: Incorporating Technical Corrigendum 1, BS ISO/IEC 14882:2003*, John Wiley and Sons, Ltd. (2003).

[4] G. S. Brodal, Worst-case efficient priority queues, *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, ACM/SIAM (1996), 52–58.

[5] G. S. Brodal, G. Lagogiannis, C. Makris, A. Tsakalidis, and K. Tsichlas, Optimal finger search trees in the pointer machine, *Journal of Computer and System Sciences* **67**, 2 (2003), 381–418.

[6] A. Brodnik, S. Carlsson, E. D. Demaine, J. I. Munro, and R. Sedgewick, Resizable arrays in optimal time and space, *Proceedings of the 6th International Workshop on Algorithms and Data Structures*, *Lecture Notes in Computer Science* **1663**, Springer-Verlag (1999), 37–48.

[7] H. Brönnimann and J. Katajainen, Efficiency of various forms of red-black trees, CPH STL Report 2006-2, Department of Computing, University of Copenhagen (2006).

[8] M. R. Brown, Addendum to "A storage scheme for height-balanced trees", *Information Processing Letters* **8**, 3 (1979), 154–156.

[9] M. R. Brown and R. E. Tarjan, The design and analysis of a data structure for representing sorted lists, *SIAM Journal on Computing* **9**, 3 (1980), 594–614.

[10] S. Carlsson, J. I. Munro, and P. V. Poblete, An implicit binomial queue with constant insertion time, *Proceedings of the 1st Scandinavian Workshop on Algorithm Theory*, *Lecture Notes in Computer Science* **318**, Springer-Verlag (1988), 1–13.

[11] R. Cole, On the dynamic finger conjecture for splay trees part II: The proof, Technical Report TR1995-701, Courant Institute, New York University (1995).

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd Edition, The MIT Press (2001).

[13] Y. Ding and M. A. Weiss, The relaxed min-max heap: A mergeable double-ended priority queue, *Acta Informatica* **20**, 2 (1993), 215–231.

[14] J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan, Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation, *Communications of the ACM* **31**, 11 (1988), 1343–1354.

[15] A. Elmasry, C. Jensen, and J. Katajainen, A framework for speeding up priority-queue operations, CPH STL Report 2004-3, Department of Computing, University of Copenhagen (2004). Available at `http://cphstl.dk`.

[16] A. Elmasry, C. Jensen, and J. Katajainen, Relaxed weak queues: An alternative to run-relaxed heaps, CPH STL Report 2005-2, Department of Computing, University of Copenhagen (2005).

[17] A. Elmasry, C. Jensen, and J. Katajainen, Two new methods for transforming priority queues into double-ended priority queues, CPH STL Report 2006-9 (2006). Available at `http://cphstl.dk`.

[18] A. Elmasry, C. Jensen, and J. Katajainen, Two-tier relaxed heaps, *Proceedings of the 17th International Symposium on Algorithms and Computation*, *Lecture Notes in Computer Science* **4288**, Springer-Verlag (2006), 308–317.

[19] R. Fleischer, A simple balanced search tree with $O(1)$ worst-case update time, *International Journal of Foundations of Computer Science* **7**, 2 (1996), 137–150.

[20] G. Franceschini and R. Grossi, Optimal worst-case operations for implicit cache-oblivious search trees, *Proceedings of the 8th International Workshop on Algorithms and Data Structures*, *Lecture Notes in Computer Science* **2748**, Springer-Verlag (2003), 114–126.

[21] G. N. Frederickson, Implicit data structures for the dictionary problem, *Journal of the ACM* **30**, 1 (1983), 80–94.

[22] I. Galperin and R. L. Rivest, Scapegoat trees, *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, ACM/SIAM (1993), 165–174.

[23] M. T. Goodrich and R. Tamassia, *Data Structures and Algorithms in Java*, John Wiley & Sons, Inc. (1998).
[24] L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts, A new representation for linear lists, *Proceedings of the 9th Annual ACM Symposium on the Theory of Computing*, ACM (1977), 49–60.
[25] S. Huddleston and K. Mehlhorn, A new data structure for representing sorted lists, *Acta Informatica* **17**, 2 (1982), 157–184.
[26] J. Iacono, Alternatives to splay trees with $O(\log n)$ worst-case access times, *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, ACM/SIAM (2001), 516–522.
[27] J. Iacono and S. Langerman, Queaps, *Algorithmica* **42**, 1 (2005), 49–56.
[28] H. Jung and S. Sahni, Supernode binary search tree, *International Journal on Foundations of Computer Science* **14**, 3 (2003), 465–490.
[29] H. Kaplan, N. Shafrir, , and R. E. Tarjan, Meldable heaps and Boolean union-find, *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, ACM (2002), 573–582.
[30] H. Kaplan and R. E. Tarjan, New heap data structures, Technical Report TR-597-99, Department of Computer Science, Princeton University (1999).
[31] J. Katajainen and B. B. Mortensen, Experiences with the design and implementation of space-efficient deques, *Proceedings of the 5th Workshop on Algorithm Engineering*, *Lecture Notes in Computer Science* **2141**, Springer-Verlag (2001), 39–50.
[32] C. Levcopoulos and M. H. Overmars, A balanced search tree with $O(1)$ worst-case update time, *Acta Informatica* **26**, 3 (1988), 269–277.
[33] W. Litwin, Linear hashing: A new tool for file and table addressing, *Proceedings of the 6th International Conference on Very Large Data Bases*, IEEE Computer Society (1980), 212–223.
[34] E. L. Lloyd and M. C. Loui, On the worst case performance of buddy systems, *Acta Informatica* **22**, 4 (1985), 451–473.
[35] C. W. Mortensen and S. Pettie, The complexity of implicit and space-efficient priority queues, *Proceedings of the 9th Workshop on Algorithms and Data Structures*, *Lecture Notes in Computer Science* **3608**, Springer-Verlag (2005), 49–60.
[36] J. I. Munro, An implicit data structure supporting insertion, deletion, and search in $O(\log^2 n)$ time, *Journal of Computer and System Sciences* **33**, 1 (1986), 66–74.
[37] J. I. Munro, V. Raman, and A. J. Storm, Representing dynamic binary trees succinctly, *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, ACM/SIAM (2001), 529–536.
[38] W. Pugh, Skip lists: A probabilistic alternative to balanced trees, *Communications of the ACM* **33**, 6 (1990), 668–676.
[39] D. D. Sleator and R. E. Tarjan, Amortized efficiency of list update and paging rules, *Communications of the ACM* **28**, 2 (1985), 202–208.
[40] R. E. Tarjan, *Data Structures and Network Algorithms*, SIAM (1983).
[41] J. W. J. Williams, Algorithm 232: Heapsort, *Communications of the ACM* **7**, 6 (1964), 347–348.