The combinatorics of some abstract data types

M D Atkinson and D Tulley

School of Mathematical and Computational Sciences,
University of St Andrews, North Haugh,
St Andrews, Fife, KY16 9SS, UK

Abstract

Abstract data types (*ADTs*) may be regarded as abstract machines and then a program for an ADT is any sequence of operations allowed by its specification. The effect of such programs on container ADTs is captured by the relationship between each input sequence and the set of possible output sequences that can result from it. This relationship is studied principally in the case of dictionaries, stacks and priority queues and a distinction is drawn between ADTs of unbounded and bounded capacity.

1 Introduction

Abstract machines have a long and honourable history in Computer Science. Turing machines, push-down automata, and finite-state machines are three well known types; they have been used to study general purpose computers, compilers, and string manipulation although their importance goes well beyond these three applications. The central theoretical issue for these (and other) machines is the characterisation of the languages associated with them. The aim of this paper is to study some abstract data types in the same spirit.

Abstract data types (ADTs) have a relatively short and honourable history. They are central to the point of view adopted in object-oriented programming (which is setting the direction of programming in the 1990s) and which now permeates all large software projects. Although abstract data typing was initially adopted primarily for its use as a software design tool it has always been recognised that each data type had a rigorous mathematical definition.

Each ADT is characterised by the set of operations that can be performed on it. Therefore an ADT can be regarded as an abstract machine whose instruction set is the set of operations it supports. Some of these operations may supply input or output while others may examine or change the state of the ADT. The precise specification of these instruction sets has been studied in considerable depth by algebraic means (see [HEO92a, HEO92b] for a survey). However, the classical abstract machines are studied at a much deeper level; their behaviour in response to *arbitrary* sequences of instructions (i.e. programs) is studied and this behaviour is captured by the idea of the language recognised by the machine. As yet, such a study has hardly begun for ADTs although, as indicated below, there is a very natural extension of the language notion to ADTs.

There is an infinite number of data types and it seems to be infeasible to give a general theory of their associated languages which has deep implications for all of them. However, in practice, only a small number of ADTs recur frequently in software and algorithm design (stacks, queues, arrays, dictionaries etc) and it is perhaps more profitable to study only those which have demonstrable software utility.

This paper will concentrate on *container* ADTs: those for which Insert and Delete operations are defined. Such ADTs act as data transformers, outputting their input data in a permuted order. If, except for housekeeping operations, Insert and Delete are the only operations supported by the data type then the functional behaviour of the ADT is essentially defined by the possible ways in which it can permute the data. A sequence of Insert and Delete operations constitutes a program for the ADT when it is regarded as a machine. For such an ADT it is not sensible to define its associated language to be the set of input sequences that lead to an accepting state, since that discards so much essential information about the output. Instead, we propose that the associated language should be defined to be the set of (input, output) pairs of sequences that can arise from the execution of an ADT program (indeed, even for Turing machines, this definition is attractive since it avoids fudges about how the input is to be encoded). As we shall see, there are a number of questions about the language associated with an ADT whose formulation and solution require combinatorial machinery. We shall list some of these questions and then go on to discuss their solutions for some particular data types.

The different container data types are generally distinguished from each other by the type of Delete operation that they support. Table 1 shows four common container data types and the properties of their Delete operation.

Let A be any container data type. We shall consider only programs for A which begin and end with A in the empty state; this represents the normal way that a container data type would be used. Such a program then consists of a sequence of Insert and Delete operations in which every initial segment contains at least as many Inserts as Deletes (this condition

Table 1. Some ADTs and their Delete operations

Name of ADT	Delete operation
Queue	Delete the item that has been in the queue the longest
Stack	Delete the item that was placed in the stack most recently
Dictionary	Delete any item
Priority queue	Delete the smallest item

is to ensure that a Delete operation is never executed when A is empty) and having equal numbers of Inserts and Deletes (to ensure that that the final state of A is empty). Let σ be any sequence of length n and let P be any program with n Inserts and n Deletes. The execution of P with σ as the input sequence results in the members of σ being inserted into A in order of occurrence in σ; the Delete operations generate a sequence τ which we call the output of P. A pair (σ, τ) which is related by a program P in this way is called *allowable* (or A-allowable when the ADT cannot be deduced from context). The set of allowable pairs is denoted by $L(A)$ and is called the language associated with the data type A. Basic combinatorial questions about $L(A)$ include:

1. How many A-allowable pairs with each component of length n are there?

2. Is there a characterisation of the A-allowable pairs that enables them to be recognised quickly?

3. Is there an efficient algorithm that, given an input sequence σ, can determine how many A-allowable pairs (σ, τ) there are? And, dually,

4. Is there an efficient algorithm that, given an output sequence τ, can determine how many A-allowable pairs (σ, τ) there are?

In practice, container ADTs generally have a bounded size, either enforced by their implementation or the physical limits of the hardware, so it makes sense to consider the above questions when no more than k elements can be stored at any time in the ADT. We therefore introduce the idea of *k-allowability* by defining the language $L_k(A)$ of a *k-bounded* ADT A to be the set of allowable pairs (σ, τ) for which there is a program P which can transform σ into τ without requiring more than k elements to be stored at any one time.

There are other variations we can introduce as well. We have not yet stipulated what form the input sequence takes; it could be taken as a sequence of distinct elements (which we can assume to be the elements $1, 2, \ldots n$ in some order, without loss of generality), as a word over the binary alphabet, or as a reordering of an arbitrary multiset $S = \{1^{a_1}, 2^{a_2}, \ldots, r^{a_r}\}$.

This leads to a large number of questions to be studied and we shall present at least partial solutions to many of them in this paper. In section 2 we present results about dictionaries concentrating mainly on the case when the input sequence is a word over the binary alphabet. Then, in section 3 we consider stacks and show that, in the binary case, they behave like dictionaries. Section 4 contains results for queues and deques (*double ended queues*) and finally in section 5 we study priority queues and double ended priority queues.

Most of the results are related to questions 1,3 and 4 in the above list but there has been some progress on question 2. This is mostly, but not exclusively, based on the idea of *avoided patterns* as used by Pratt ([Pra73]) and Knuth ([Knu73a, 2.2.1; Q 5]). A pattern of length m is a permutation $\rho = (\rho_1, \rho_2, \ldots \rho_m)$ of $1, 2, \ldots, m$, and a sequence $\sigma = (\sigma_1, \sigma_2, \ldots \sigma_n)$ is said to *contain* the pattern if there is a subsequence σ' of σ such that $|\sigma'| = m$ and $\rho_i < \rho_j$ if and only if $\sigma'_i < \sigma'_j$. As an example, the sequence $5, 4, 2, 3, 1$ contains the pattern $3, 1, 2$ because it contains the subsequence $5, 2, 3$, but on the other hand $5, 4, 3, 2, 1$ does not contain the pattern $1, 2, 3$. If ρ does not occur within σ we shall say that σ *avoids* ρ.

2 Dictionaries

Dictionaries are the class of abstract data types with the most general delete operation. They allow the removal of any element which is currently stored in the dictionary. Their behaviour was studied in [ALT] where they are referred to as *buffers* and, in the case of a bounded capacity, as *bounded buffers.* When the input sequence is a permutation of distinct elements, the order of the elements has no effect on the number of outputs possible and so instead of studying the number of allowable pairs it is only necessary to consider how many output sequences are possible from the input sequence $1, 2, \ldots, n$. Then the number of allowable pairs is $n!$ times the number of allowable output sequences from $1, 2, \ldots, n$. In the unbounded case there is not any great challenge since it is clear that the input sequence can be permuted into any output sequence by merely inserting the entire input sequence and then deleting the elements in the required order. Thus $L(Dictionary) = \{(\sigma, \tau) | \tau \text{ is a rearrangement of } \sigma\}$ and for this reason the unbounded dictionary is the most permutationally powerful abstract data type. The bounded case is also relatively simple for we have in [ALT]

Lemma 1. *For a bounded dictionary of capacity k, each input sequence of n distinct elements gives $k^{n-k}k!$ allowable output sequences.*

Lemma 2. *The allowable output sequences of a bounded dictionary with input sequence* $1, 2, \ldots, n$ *are precisely the sequences that avoid all patterns of length* $k+1$ *which begin with their maximal element.*

In the case when the input sequence is a binary sequence the results are a little more complex.

Lemma 3. *For a bounded dictionary of size* k *the number* x_n *of binary allowable pairs of length* n *satisfies the recurrence*

$$\begin{aligned} x_{k+n} &= \sum_{i=1}^{r}(-1)^{i+1}x_{k+n-i}a_{i,k} \text{ with } r = \lceil \tfrac{k}{2} \rceil \qquad (2.1)\\ a_{0,k} &= 1 \\ a_{i,k} &= 2\binom{k-i}{i-1} + \binom{k-i}{i} \end{aligned}$$

PROOF: Let $w_n^{(i)}$ be the number of allowable pairs of length n of the form $(0^i\alpha, \beta)$, then obviously $x_n = w_n^{(0)}$, and

$$\begin{aligned} w_{n+1}^{(0)} &= 2w_{n+1}^{(1)} \text{ for } n \geq 0 \\ w_{n+1}^{(i)} &= w_n^{(i-1)} + \sum_{j=i}^{k-1} w_n^{(j)} \text{ for } n \geq i > 0 \end{aligned}$$

The first of these two equations holds because the number of pairs of the form $(0\alpha, \beta)$ is the same as the number of pairs of the form $(1\alpha, \beta)$ and all allowable pairs have one of these two forms.

For the second notice that all pairs $(0^i\alpha, \beta)$, of length $n+1$, either have the form $(0^i\alpha, 0\beta')$ or $(0^i\alpha, 1\beta')$. There are $w_n^{(i-1)}$ pairs of the first form because the first 0 is input and immediately output, then there are $w_n^{(i-1)}$ ways to complete the pair. The second can be split into several further forms, $(0^j1\gamma, 1\beta')$ for $i \leq j < k$. For each of these the dictionary must insert all j 0's and the 1 and then immediately output the 1. There are then $w_n^{(j)}$ ways to complete the pair and thus there are $\sum_{j=i}^{k-1} w_n^{(j)}$ allowable pairs of the second form.

We can represent this recurrence more succinctly using matrix notation. Let

$$w_n = \begin{pmatrix} w_n^{(1)} \\ \vdots \\ w_n^{(k-1)} \end{pmatrix}, \; A_k = \begin{pmatrix} 3 & 1 & \cdots & 1 & 1 \\ 1 & 1 & \cdots & 1 & 1 \\ 0 & 1 & \cdots & 1 & 1 \\ \vdots & \ddots & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & 1 & 1 \end{pmatrix}$$

then the recurrence equation is

$$w_{n+1} = A_k w_n \tag{2.2}$$

Hence, for any constants $d_0, d_1, \dots d_t$, $\sum_{i=0}^{t} d_i w_{n+i} = \sum_{i=0}^{t} d_i A_k^i w_n$ and if we choose the constants so that $\sum_{i=0}^{t} d_i \lambda^i$ is the characteristic polynomial of A then we shall have $\sum_{i=0}^{t} d_i w_{n+i} = 0$. We can derive a recurrence equation for the characteristic polynomial, $u_k(\lambda) = det(A_k - \lambda I)$, of A_k by subtracting the $(k-1)^{th}$ column of the determinant from the k^{th} column and expanding it by the k^{th} column. It is then easily seen that

$$u_k(\lambda) = -\lambda(u_{k-1}(\lambda) + u_{k-2}(\lambda)) \text{ for all } k \geq 3$$

The initial cases

$$\begin{aligned} u_0(\lambda) &= 1 \\ u_1(\lambda) &= 3 - \lambda \\ u_2(\lambda) &= \lambda^2 - 4\lambda + 2 \end{aligned}$$

are calculated directly.

From the recurrence it follows easily by induction on k that there exists polynomials y_{2r} and y_{2r+1} each of degree $r+1$ for which

$$u_{2r}(\lambda) = \lambda^{r-1} y_{2r}(\lambda) \tag{2.3}$$
$$u_{2r+1}(\lambda) = \lambda^{r} y_{2r+1}(\lambda) \tag{2.4}$$

and that the polynomials satisfy

$$y_{2r}(\lambda) = -\lambda y_{2r-1}(\lambda) - y_{2r-2}(\lambda) \tag{2.5}$$
$$y_{2r+1}(\lambda) = -y_{2r}(\lambda) - y_{2r-1}(\lambda) \tag{2.6}$$

Now a standard inductive proof using binomial coefficient identities proves

$$y_{k-1}(\lambda) = (-1)^{k-1} \sum_{i=0}^{r} \lambda^{r-i} (-1)^i a_{i,k} \text{ where} \tag{2.7}$$
$$r = \left\lceil \tfrac{k}{2} \right\rceil, a_{0,k} = 1, a_{i,k} = 2\binom{k-i}{i-1} + \binom{k-i}{i}$$

Combining (2.3) and (2.4) gives

$$\begin{aligned} u_{k-1}(\lambda) &= \lambda^{\left\lfloor \frac{k-2}{2} \right\rfloor} y_{k-1}(\lambda) \\ &= (-1)^{k-1} \sum_{i=0}^{r} \lambda^{r-i+\left\lfloor \frac{k-2}{2} \right\rfloor} (-1)^i a_{i,k} \end{aligned}$$

Therefore the sequence (w_n) satisfies

$$\sum_{i=0}^{r} w_{r+\lfloor\frac{k-2}{2}\rfloor+n-i}(-1)^{i+1}a_{i,k} = 0 \text{ where } r = \left\lceil\frac{k}{2}\right\rceil$$

Since $r + \left\lfloor\frac{k-2}{2}\right\rfloor \leq k$, $w_n^{(1)}$ is an element of the vector w_n and $x_n = w_n^{(0)} = 2w_n^{(1)}$, we have

$$x_{k+n} = \sum_{i=1}^{r} x_{k+n-i}(-1)^{i+1}a_{i,k}$$

□

The recurrence of this lemma shows how x_n can be computed once initial values $x_0, x_1, \ldots x_{k-1}$ are known. However, for $t \leq k-1$ (indeed $t \leq k$) the number of binary allowable pairs of length t is unconstrained by the dictionary size k. Therefore, if $t \leq k$, it is easily seen that

$$x_t = \sum_{i=0}^{t} \binom{t}{i}^2 = \binom{2t}{t}$$

For example, with $k = 3$, the recurrence becomes

$$x_{n+3} = 4x_{n+2} - 2x_{n+1}$$

and the values of x_n are

n	0	1	2	3	4	5	6
x_n	1	2	6	20	68	112	312

3 Stacks

Unbounded stacks were studied extensively in the 1970's and a number of significant connections with other combinatorial objects were found. Because the permutational power of the stack comes from its structure and not from the relative values of the elements it is processing, the allowable pairs are closely related to the valid programs of a stack. This leads to several correspondences; for example, the number of valid programs of length $2n$ which a stack can execute, and thus, given a fixed input sequence, the number of allowable output sequences of length n, is in a one to one correspondence with the number of balanced bracket sequences of length $2n$. There

Table 2. α_k for small values of k

Stack size	α_k	Numerical Estimate
1	1	1
2	2	2
3	$\frac{3+\sqrt(5)}{2}$	2.61
4	3	3
5	largest root of x^3-5x^2+6x-1	3.247

are then similar correspondences with trees ([Knu73a, 2.3.4.4],[Rot75]), triangulations of polygons ([CLR92, pp320–324]), Young Tableaux ([Knu73b, pp63–64]), lattice paths ([Moh79]) and ballot sequences ([Knu73b, p531], [RV78]). These connections are of great interest in the design of efficient algorithms (see [Knu73b, Pra73, Tar72]) and all point to the stack's fundamental role in giving precise expression to informal concepts such as "nesting", "structured decomposition" and "hierarchy".

It is known from the many correspondences above that, given a fixed input sequence of distinct elements, there are $c_n = \binom{2n}{n}/(n+1)$ (the n^{th} Catalan number) possible output sequences. Therefore the number of allowable pairs of length n is $n!c_n$. It is also known that, if the input sequence is $1, 2, \ldots n$, the allowable output sequences of an unbounded stack are those sequences which avoid the pattern $3, 1, 2$([Knu73a]).

In the bounded case we can apply the techniques of [ALT] which show that the number of output sequences for a fixed input sequence of distinct elements rises exponentially in the length of the input sequence. The base α_k of the exponent depends on the stack size k and some values of α_k are given in Table 2. Because of the asymptotic behaviour of c_n, $\alpha_k \to 4$ as $k \to \infty$. This case was also studied in [dBKR72], where it was posed as the problem of finding the average height of planted plane trees. This corresponds to the average capacity require to generate a randomly chosen output sequence. The result is that on average

$$k = \sqrt{\pi n} - \frac{1}{2} + O\left(\frac{1}{\sqrt{n}} \log n\right)$$

stack locations are required.

When the input sequence is a binary sequence the stack has exactly the same behaviour as a dictionary for both the bounded and unbounded cases, as the following shows.

Lemma 4. *The allowable output sequences of a stack on a binary input are precisely those of a dictionary of the same capacity on a binary input.*

PROOF: Suppose $(\sigma, \tau) \in L_k(\mathit{Dictionary})$, then there is a program of insert and delete operations which transform σ into τ. Among all such programs we choose one, C say, in which all the insert operations are delayed as long as possible; in C an insert operation only occurs if it is not possible to produce any more of the output sequences τ from the elements stored in the dictionary. So, when C is executed, the only point at which both a 0 and a 1 are stored in the dictionary is when there are one or more 0's stored already, a 1 was inserted by the last operation in C which has been executed and the 1 will be deleted by the next operation in C (and the same situation with 1 and 0 interchanged).

A stack can execute the same sequence, C, and will produce the same output sequence τ from the input sequence σ because, for every delete operation, the element which the dictionary would output is either the last one inserted (in which case the stack can also output it), or there are only elements of a single value stored in the dictionary/stack. Therefore $(\sigma, \tau) \in L_K(\mathit{Stack})$ and so $L_k(\mathit{Dictionary}) \subseteq L_K(\mathit{Stack})$.

The other inclusion is trivial and so we have $L_k(\mathit{Dictionary}) = L_K(\mathit{Stack})$ $\square$

It follows from this that the number of allowable pairs of a stack with binary input sequences satisfies the recurrence given for the bounded dictionary in section 2.

When the input sequence is allowed to have duplicated elements little is know about the behaviour, but some progress has been made in [ALW] in the case that equal symbols occur adjacently.

4 Queues and deques

The analysis of queues is entirely trivial. In both the bounded and unbounded cases, the only possible output sequence is the input sequence, so there are $n!$ allowable pairs of length n. It is interesting to note that with such a permutationally weak data type even the transition from bounded to unbounded has no effect on its power. In all the other data types considered here it is a significant transition.

Double ended queues (*deques*) are queues which allow two insert and two delete operations. It is possible to insert elements at both ends of the queue and it is possible to delete elements from both ends. There are also variants of the deque; an *input restricted deque* has only one input operation (it can only insert elements at one end of the queue) and an *output restricted deque* has only one delete operation (it can only remove elements from one end of the queue). The third possible variant, where we have only one input operation and one output operation, can have two forms. If we allow insertion and deletion at the same end of the queue we have a stack, otherwise we have an ordinary queue. Both of these possibilities have been considered previously so we shall ignore them here. Deques and their two variants were studied in [Pra73], where most of the results are presented as excluded pattern conditions.

The number of cases to study is reduced a little because the properties of input restricted deques and output restricted deques are symmetric. This is because, a pair (σ, τ) is allowable (or k-allowable) for an input restricted deque if and only if the pair (τ^R, σ^R) is allowable (or k-allowable) for an output restricted deque (σ^R denotes the reversal of the sequence σ). Knuth [Knu73b, p534; Q 13] states that the generating function for the number of allowable output sequences for an output restricted deque, with some input sequence σ, is

$$G(z) = \frac{1}{2}(1 + z - \sqrt{1 - 6z + z^2})$$

Pratt then shows that there is a 2-1 correspondence between these outputs and honest trees with n leaves (an honest tree is a general tree with no nodes of out degree 1). He then goes on to prove that, on input $1, 2, \ldots n$, the allowable outputs are precisely those which avoid the patterns $4, 2, 3, 1$ and $4, 1, 3, 2$. The corresponding patterns for an input restricted deque are $4, 2, 3, 1$ and $4, 2, 1, 3$. He then shows that the allowable outputs for a deque with input $1, 2, \ldots n$ are those sequences which avoid the infinite set of patterns shown in table 3.

The bounded versions of the above deque variants can be handled by the techniques of [ALT]. For example, for a deque of size 3 we can show there are 2.3^{n-2} allowable outputs for any fixed input. Similarly for a deque of size 4 we can show that the number of allowable outputs for a fixed input sequence is 6.4^{n-3}. For size 5 the number of allowable outputs grows exponentially with base α, where α is the largest root of $x^3 - 7x^2 + 10x + 2$ (approximately 4.855).

5 Priority queues

There has been a great deal of work in recent years on the combinatorial properties of priority queues, both bounded and unbounded, operating on input sequences formed from distinct elements, the binary alphabet and multisets.

It is convenient to define

$$\begin{aligned} s(\tau) &= |\{\sigma | (\sigma, \tau) \in L(\text{Priority Queue})\}| \\ t(\sigma) &= |\{\tau | (\sigma, \tau) \in L(\text{Priority Queue})\}| \\ s_k(\tau) &= |\{\sigma | (\sigma, \tau) \in L_k(\text{Priority Queue})\}| \\ t_k(\sigma) &= |\{\tau | (\sigma, \tau) \in L_k(\text{Priority Queue})\}| \end{aligned}$$

Table 3. The excluded patterns which characterise a deque

$5, 2, 3, 4, 1$
$5, 2, 7, 4, 1, 6, 3$
$5, 2, 7, 4, 9, 6, 3, 8, 1$
$5, 2, 7, 4, 9, 6, 11, 8, 1, 10, 3$
$5, 2, 7, 4, 9, 6, 11, 8, 13, 10, 3, 12, 1$
etc.
and those obtained by exchanging 1 and 2
and/or the last two elements in each pattern

In [AT93] it is shown that, when the input is formed from n distinct elements, there are $(n+1)^{n-1}$ allowable pairs of length n for an unbounded priority queue. For this case algorithms are presented in [AB] which calculate $s(\tau)$ in $\mathrm{O}(n)$ time and $t(\sigma)$ in $\mathrm{O}(n^4)$ time. The transitive closure of the allowability relation is also found.

When the inputs are restricted to binary sequences it was shown in [Atk93] that there are c_{n+1} allowable pairs of length n. An $\mathrm{O}(n^2)$ algorithm is then presented which calculates $s(\tau)$. It is also shown that (σ, τ) is allowable if and only if (τ^R, σ^R) is allowable and this gives an $O(n^2)$ algorithm to compute $t(\sigma)$.

Some very recent work in [ALW] gives the only result we know of in the case when the input is a rearrangement of a multiset $S = \{1^{a_1}, 2^{a_2}, \ldots, r^{a_r}\}$. Here it is shown that there are

$$\frac{1}{n+1} \prod \binom{n+1}{a_i}$$

allowable pairs.

All the results above are for unbounded priority queues. The study of the bounded case was begun in [AT]. For input sequences of distinct elements the only progress made has been for the priority queue of size 2. In this case, if x_n is the number of allowable pairs then

$$\sum x_n \frac{t^n}{n!} = \frac{1}{1 + \log(1-t)}$$

and from this it can be deduced that $x_n/n!$ is asymptotic to $(e/(e-1))^n$ as $n \to \infty$. It was also shown how to compute $s_2(\tau)$ and $t_2(\sigma)$ in time $O(n^2)$.

When the input is a binary sequence more general results are known. For example, $s_k(\tau)$ and $t_k(\sigma)$ can be computed in time $O(n^2)$. It was also shown in [AT] that there is a $1-1$ correspondence between k-allowable pairs of length n and ordered forests of height no more than $k+2$ on $n+2$ nodes.

The study of double ended priority queues has been rather less productive. This data type has two kinds of delete operation: Delete-Minimum and Delete-Maximum (denoted by "d" and "D" respectively). Many experimental results and conjectures are reported in [Thi93]. Linton has given a characterisation of allowable pairs in terms of avoided pairs of patterns extending the idea of pattern avoidance in section 1. The only other results we know of bear on questions 3 and 4 of the introduction.

Let $\mathcal{D}$ be any sequence of n Delete-Minimum and Delete-Maximum operations and let $\pi(\mathcal{D})$ be the permutation of $1, 2, \ldots, n$ defined by

$$\mathcal{D}(I)_i = \begin{cases} n-j & \text{if } \mathcal{D}_i = D \\ i-1-j & \text{if } \mathcal{D}_i = d \end{cases}$$

where j is the number of D's among $\mathcal{D}_1 \ldots \mathcal{D}_{i-1}$
Also define the complementary permutation $\bar{\pi}(\mathcal{D})$ by $\bar{\pi}(\mathcal{D})_i = n+1-\pi(\mathcal{D})_i$. Then we have

Lemma 5. *Consider the set of double ended priority queue programs in which the sequence of delete operations is a fixed sequence $\mathcal{D}$, and let $A(\mathcal{D})$ be the set of all (σ, τ) related by such programs. Further, let $s_{\mathcal{D}}(\tau) = |\{\sigma|(\sigma,\tau) \in A(\mathcal{D})\}|$ and $t_{\mathcal{D}}(\sigma) = |\{\tau|(\sigma,\tau) \in A(\mathcal{D})\}|$. Then*

1. *$s_{\mathcal{D}}(\tau)$ is maximal when $\tau = \pi(\mathcal{D})$*
2. *$s_{\mathcal{D}}(\tau)$ is minimal when $\tau = \bar{\pi}(\mathcal{D})$*
3. *$t_{\mathcal{D}}(\sigma)$ is minimal when $\sigma = \pi(\mathcal{D})$*

6 Conclusion

We have presented a unified framework in which the (input,output) relation for various container data types can be studied. The properties of the relation for stacks, queues, dictionaries and priority queues are fairly well understood.

The main unsolved problems requiring further research include

1. A treatment of deques and double ended priority queues as complete as that for stacks, queues, dictionaries and unbounded priority queues.
2. A better understanding of bounded priority queues on non binary inputs.

3. An extension of the theory to networks of container data types (as proposed by Tarjan for stacks and queues in [Tar72]). Note that networks of bounded dictionaries, queues, deques and stacks can, in principle, be handled by the techniques of [ALT].

Bibliography

[AB] M D Atkinson and R Beals. Priority queues and permutations. To appear SIAM J. Comput.

[ALT] M D Atkinson, M J Livesey, and D Tulley. Networks of bounded buffers. Submitted to SIAM J. Comput.

[ALW] M D Atkinson, S A Linton, and L A Walker. Priority queues and multi-sets. In preparation.

[AT] M D Atkinson and D Tulley. Bounded capacity priority queues. Submitted to Theoretical Computer Science.

[AT93] M D Atkinson and M Thiyagarajah. The permutational power of a priority queue. *BIT*, 33:2–6, 1993.

[Atk93] M D Atkinson. Transforming binary sequences with priority queues. *Order*, 10:31–36, 1993.

[AW93] M D Atkinson and L A Walker. Enumerating k-way trees. *Information Processing Letters*, 48:73–75, 1993.

[CLR92] T H Cormen, C E Leiserson, and R L Rivest. *Introduction to Algorithms*. McGraw-Hill, Cambridge, Mass., 1992.

[dBKR72] N G de Bruijn, D E Knuth, and S O Rice. The average height of planted plane trees. In R C Read, editor, *Graph Theory and Computing*, pages 15–22. Academic Press, 1972.

[HEO92a] I Classen H Ehrig, B Mahr and F Orejas. Introduction to algebraic specification. part 1:formal methods for software development. *Computer Journal*, 35:451–459, 1992.

[HEO92b] I Classen H Ehrig, B Mahr and F Orejas. Introduction to algebraic specification. part 2:from classical view to foundations of systems specifications. *Computer Journal*, 35:460–467, 1992.

[Knu73a] D E Knuth. *Fundamental Algorithms, The Art of Computer Programming.* Addison-Wesley, Reading, Mass, 1973.

[Knu73b] D E Knuth. *Sorting and Searching, The Art of Computer Programming.* Addison-Wesley, Reading, Mass, 1973.

[Moh79] S G Mohanty. *Lattice path counting and applications.* Academic Press, New York - London, 1979.

[Pra73] V R Pratt. Computing permutations with double-ended queues, parallel stacks and parallel queues. *5th ACM Symposium on Theory of Computing*, pages 268–277, 1973.

[Rot75] D Roton. On a correspondencebetween binary trees and a certain type of permutation. *IPL*, 4:58–61, 1975.

[RV78] D Roton and Y L Varol. Generation of binary trees from ballot sequences. *JACM*, 25:396–404, 1978.

[Tar72] R E Tarjan. Sorting using networks of queues and stacks. *JACM*, 19:341–346, 1972.

[Thi93] M Thiyagarajah. Permutational power of priority queues. Master's thesis, School of Computer Science, Carleton University, 1993.