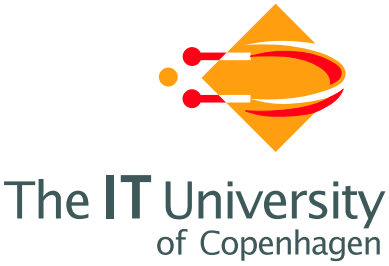

Dynamic Nested Brackets

Stephen Alstrup
Thore Husfeldt
Theis Rauhe

IT University Technical Report Series **TR-2001-9**

ISSN 1600–6100 **November 2001**

Copyright © 2001, **Stephen Alstrup**
Thore Husfeldt
Theis Rauhe

IT University of Copenhagen
All rights reserved.

Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.

ISSN 1600–6100

ISBN 87-7949-012-3

Copies may be obtained by contacting:

IT University of Copenhagen
Glentevej 67
DK-2400 Copenhagen NV
Denmark

Telephone: **+45 38 16 88 88**
Telefax: **+45 38 16 88 99**
Web **`www.it-c.dk`**

Dynamic Nested Brackets [1]

Stephen Alstrup
IT University of Copenhagen
Glentevej 65-67, DK-2400, Denmark IT-C
E-mail : stephen@it-c.dk

and

Thore Husfeldt
Computer Science, Lund University
Box 118, S-221 00, Sweden
E-mail : thore@cs.lth.se

and

Theis Rauhe
IT University of Copenhagen
Glentevej 65-67, DK-2400, Denmark
E-mail : theis@it-c.dk

Version: 2 November 2001

We consider the problem of maintaining a string of brackets like $(\,)(\,(\,)\,)$ of length n under a single operation *reverse*(i). The operation *reverse*(i) changes the ith letter from '(' to ')' or vice versa, and returns 'yes' if and only if the updated string is balanced. We give lower and upper bounds showing that the complexity of *reverse*(i) is $\Theta(\log n/\log\log n)$.

1. INTRODUCTION

A string of brackets like $(\,)(\,(\,)\,)$ is *properly nested* or *balanced*, while $(\,($ and $(\,))(\,)$ are not. Deciding which is which is a classical computational problem that appears in many introductory textbooks on data structures. In sequential computation, it illustrates the power of a *stack* (the related formal language, the Dyck language, requires a push-down automaton), and in parallel computation, it captures the concept of *counting* (the problem is AC_0-complete for TC_0).

In this paper we characterise the complexity of the natural *dynamic* variant of the problem, where the string is subject to changes. Consider '()()' for example. Reversing the second bracket will destroy balance: '(((', and subsequently reversing either the second or third bracket will re-establish it, but in different ways: '()()' or '(())'.

This problem is encountered by many modern editors for programming languages, which contain incremental parsers that perform an on-line syntax check whenever the user changes the text, including a check for properly nested brackets.

To be precise we consider the problem of maintaining a string of brackets of length n under a single operation *reverse*(i) that changes the ith letter from '(' to ')' or vice versa, and returns 'yes' if and only if the updated string is balanced. Our model of computation is a unit-cost RAM [1] with word size $\log n$. We show that the complexity of '*reverse*' is $\Theta(\log n/\log\log n)$:

THEOREM 1.1. *There is a data structure supporting* reverse *in worst case time* $O(\log n/\log\log n)$. *The data structure uses linear preprocessing time and linear space. Moreover, this is optimal in the sense that every data*

[1]Part of this work was done while the first author visited BRICS and Lund University, and while the last author visited the Fields Institute of Toronto and working at BRICS. This work was partially supported by the ESPRIT Long Term Research Programme of the EU, project number 20244 (ALCOM-IT). The second author was partially supported by a grant from the Swedish TFR. The first and third author was partially supported by a grant from the Danish SNF. Some of the results in this paper were claimed in [2] but not proved there because of lack of space.

structure (no matter its space and preprocessing complexity) needs time $\Omega(\log n/\log\log n)$ *to support* reverse *in the worst case.*

The upper bound relies on a new data structure that may have independent interest and is described in Sec. 1.1. The lower bound is proved by a reduction to the marked ancestor problem [2].

Discussion and related work. The best previous results for our problem are an $O(\log n)$ upper bound and an $\Omega(\log\log n/\log\log\log n)$ lower bound, both from [3]. That reference also considers strings like '([ ]( ))' with more than one type of bracket. Lower bounds of size $\Omega(\log n/\log\log n)$ for computationally harder problems about nested brackets (interval queries, finding matching brackets, more than one type of bracket) are given in [3, 7, 8]. These results are entailed by Thm. 1. The dynamic nested bracket problem was studied in a different model of dynamic computation (dynamic first order logic) by Immerman and Patnaik [9]. The dynamic cell probe complexity of regular languages was studied in [4].

It was pointed out in [3] that the *two-sided* version of the problem, where both $(\,)$ and $)($ are considered balanced, can be solved in constant time per operation. This problem is essentially a counting problem – the string balances if and only if it contains the same number of opening and closing brackets. In many models of computation, the complexities of the one- and two-sided problems are the same. The present paper shows that for dynamic computation, the one-sided problem is much harder, and since the underlying counting problem can be solved in constant time, the complexity rests entirely on the global nesting structure that must be maintained under local changes.

1.1. Suffix-Change Priority Queues

A suffix-change priority queues (s-queue for short) supports the following operations. Let s be a sequence of integers, $s = s_1, \ldots, s_m$ with $s_i = -n \,..\, n$.

init$(\hat{s}_1, \ldots, \hat{s}_m)$: set $s_i = \hat{s}_i$, where $\hat{s}_i \in -n \,..\, n$,

value(i): return s_i,

min: return $\min_i s_i$,

change(i, c): let $s_i = s_i + c$ provided $s_i + c \in -n \,..\, n$, where $i \in 1 \,..\, m$, $c \in -r \,..\, r$,

suffix-change(i, c): let $s_j = s_j + c$ for all $j > i$ provided $s_j + c \in -n \,..\, n$, where $i \in 1 \,..\, m-1$, $c \in -r \,..\, r$,

With the exception of *suffix-change*, these are the operations of a priority queue (with *delete*). Observe that *change* can be implemented by two applications to *suffix-change*, and is included only for convenience. The *value* operation provided by our structure is not needed for the present application, and is provided for completeness.

LEMMA 1. *Let* $0 < \epsilon < 1$. *If* $r \le n^{\log^{-\epsilon} n}$ *and* $m \le \frac{1}{3}\log^{\epsilon} n$ *then an s-queue can be implemented on a RAM with word size* $\log n$ *such that* init *takes* $O(m)$ *time and the other s-queue operations take constant time. The data structure can be initialised in* $O(n)$ *time and take* $O(n)$ *space.*

Discussion and related work. The interesting part of the result is that there is no restriction on the range of the values s_i other than that they fit into a constant number of machine words. Instead, we restrict the range r of increments. Had we instead restricted the range of values s_i by for example $r \in 0 \,..\, r-1$ then the entire sequence would fit into $m \log r < \log n$ bits, and the result of every operation could be tabulated in linear space beforehand for constant update time.

In [5] a data structure for a small set of integers was given, which supports standard search and priority queues in constant time per operation, and was used to construct the first linear time minimum spanning tree algorithm. Similarly, our data structure given in Section 1.1 works on a small set of integers to support priority queue operations in constant time per operation. In addition we show how to update in constant time a subset of the stored integers.

2. THE UPPER BOUND

Let x_i denote the ith letter of the bracket string x and represent '(' by $+1$ and ')' by -1. Construct a balanced tree T with n leaves whose ith leaf corresponds to x_i. Each internal node has exactly b ordered children (b will be fixed later); the ancestors of the nth leaf may have fewer than b children.

For every non-root node v we let $p(v)$ denote its parent and $i(v)$ denote its index among its siblings, so that v is the $i(v)$th child of $p(v)$. Let $c(v,i)$ denote the ith child of v and let $l(v)$ and $r(v)$ be the indices of its leftmost and rightmost leaf descendants, respectively.

At each internal node we maintain to values,

$$\begin{aligned}\operatorname{sum}(v) &= x_{l(v)} + \cdots + x_{r(v)}\\ \operatorname{minprefix}(v) &= \min_{k\in l(v)..r(v)} x_{l(v)} + \cdots + x_k.\end{aligned}$$

We also define $\operatorname{sum}(l) = \operatorname{minprefix}(l) = x(l)$ for every leaf l.

Observe that at the root r we have $\operatorname{sum}(r) = x_1 + \cdots + x_n$ and $\operatorname{minprefix}(r) = \min_{k\in 1..n} x_1 + \cdots + x_k$, and that x is balanced if and only if both these values are 0. We will show how to maintain $\operatorname{sum}(v)$ and $\operatorname{minprefix}(v)$ for each node v in T.

Observe that when a leaf is changed we can easily update $\operatorname{sum}(v)$ for all its ancestors in time equal to the height of the tree. The difficult part is to update minprefix.

To this end we maintain at each internal node a sequence of b values that contain information about the minimal prefix sums of its children: Let v be an internal node and $w = c(v,i)$ be the ith child of v. Define

$$\begin{aligned}s_i(v) &= \operatorname{minprefix}(w) + \sum_{j\in 1..i-1} \operatorname{sum}(c(v,j))\\ &= \min_{k\in l(w)..r(w)} x_{l(v)} + \cdots + x_k.\end{aligned}$$

Thus $\operatorname{minprefix}(v) = \min_{i\in 1..b} s_i(v)$. To store and update these values we maintain an s-queue over $s_1(v),\ldots,s_b(v)$ at every internal node v. The space and initialisation time used for all these queues is linear in the number of children in the tree, in total $\mathrm{O}(n)$.

After an update to leaf l, if v is on the path from l to the root then some of the values $s_i(v)$ have to be updated (no other nodes contain values that depend on l). Assume the path passes trough v's ith child w. Then the change to $s_i(v)$ is the same as the change to $\operatorname{minprefix}(w)$. Then change to $s_j(v)$ for $j > i$ is the same as the change to $x(l)$. The values $s_j(v)$ for $j < i$ remain unchanged. This gives rise to the next lemma.

LEMMA 2. *Every* reverse *operation can be implemented with* $\mathrm{O}(\log n/\log b)$ *primitive operations and operations on s-queues with* b *elements each. The value of* c *in the* change *and* suffix-change *operations is in the range* $-2..2$.

Proof. The procedure for updating the tree is described in Fig. 1. Inspection of the figure shows that at each level in the tree we use a constant number of primitive operations and s-queue operations. The height of a balanced tree with degree b is $\mathrm{O}(\log n/\log b)$. For the last part of the lemma we observe that $\delta \in -2\,..\,2$ invariantly: when the values of the s-queue at $p(v)$ are changed with *change* or *suffix-change*, the minimum changes by at most 2, so the subsequent assignment to δ will preserve the invariant. ∎

3. CONSTANT TIME DATA STRUCTURE FOR S-QUEUES

Let $\mu = \min_i \hat{s}_i$. We will maintain three tables of m values each,

T_1: We set $T_1[i] = \hat{s}_i$ at initialisation.

T_2: At initialisation, we set $T_2[i] = rm + \min\{s_i - \mu, 2rm\}$, and after *suffix-change*(i,c) we set

$$T_2[j] = \min\{T_2[j] + c, 3rm\}, \quad \text{for all } j > i. \tag{1}$$

T_3: We maintain $s_i = T_1[i] + T_3[i] - rm$ for all i after initialisation and after every *suffix-change*.

```
Procedure reverse(l)
    x(l) ← −x(l)
    minprefix(l) ← x(l)
    sum(l) ← x(l)
    δ ← 2x(l)
    v ← l
    repeat
        p(v).change(i(v), δ)
        p(v).suffix-change(i(v), 2x(l))
        v ← p(v)
        δ ← v.min − minprefix(v)
        minprefix(v) ← v.min
        sum(v) ← v.sum() + 2x(l)
    until v is the root
    if minprefix(v) = 0 and sum(v) = 0
        then return 'yes'
        else return 'no'
```

FIG. 1 Implementation of reverse using an s-queue. We write v.*change* for the *change* operation of the s-queue stored at v, and similarly for the other operations.

Observe that after the initialisation, we have

$$\min_i T_2[i] = rm, \tag{2}$$

and at any time during the first m updates, we ensure

$$T_2[i], T_3[i] \in 0 \,..\, 3rm \quad (i \in 1 \,..\, m). \tag{3}$$

LEMMA 3. *At any time during the first m updates,*

$$\min_i s_i = \min_i T_2[i] + \mu - rm. \tag{4}$$

Proof. An index j for which $T_2[j] \neq s_j - \mu - rm$ is called *incorrect*. To establish the lemma it suffices to show that when index j becomes incorrect after an update or initialisation then

$$s_j \geq \min_i s_i \tag{5}$$

and

$$T_2[j] \geq \min_i T_2[i] \tag{6}$$

for the remainder of the first m updates. First note that since each of the m updates changes the value of s_i by at most r, we have $\min_i s_i \leq \mu + rm$ and $\min_i T_2[i] \leq 2rm$ from (2).

Assume that j becomes incorrect at initialisation; this only happens if $s_j \geq 2rm + \mu$. Especially, $s_j \geq rm + \mu \geq \min_i s_i$, establishing (5). Also, in that case we will have initialised $T_2[j] = 3rm$, so after at most m updates we have $T_2[j] \geq 2rm \geq \min_i T_2[i]$, establishing (6).

Now assume that j was initialised correctly and consider the first update that makes j incorrect. Let s_j and $T_2[j]$ denote the values prior to the update. Since the index becomes incorrect we must have $T_2[j] + c \geq 3rm$, so by prior correctness we establish $s_j \geq 2rm + \mu$, which we already analysed. ∎

We can now sketch our data structure. We use T_3 to answer *value*-queries and T_2 to answer *min*-queries during the first m updates. We show below how to perform all updates to T_2 and T_3, as well as the minimisation query to T_2, in constant time. After m updates we re-initialise the structure in $O(m)$ time, which yields constant amortised time per operation. The work can be distributed over the updates to achieve a constant time worst-case bound.

3.1. Updating the tables.

We use standard tabulation techniques [6] to inspect and update T_2 and T_3 in constant time. Below, we give a detailed description of how to update T_2 according to (1). The remaining table operations are to look up or change $T_3[i]$ and to compute $\min_i T_2[i]$; these operations can be handled in constant time by similar tabulations.

Let $w(T)$ be the representation of table T, and assume that a single $\log n$ bits machine word can store both $w(T)$, and index i ($i \in 1\,..\,m$), and a value c ($c \in -r\,..\,r$). We pre-compute another table M with at most n entries, such that $M[w(T_2)]$ contains the representation of the table resulting from the update (1). Thus the update can be performed in constant time by replacing T_2 with $M[w(T_2)]$.

We conclude that a single machine word must be able to contain a table of m elements, each of which is in the range given by (3), together with the representation of $i \in 1\,..\,m$, and the representation of $c \in -r\,..\,r$. In total, this requires

$$m\lceil\log(3rm+1)\rceil + \lceil\log m\rceil + \lceil\log(2r+1)\rceil$$

bits which is at most $\log n$ for $m = \frac{1}{3}\log^{\epsilon} n$, $r \leq n^{\log^{-\epsilon} n}$, and n sufficiently large.

It remains to show that the table M can be constructed in linear time. For each table T_2, for each index i, and for each value c we need to perform the update (1) for up to m entries. The total number of entries we need to update is

$$(3rm+1)^m m^2(2r+1) = O(n).$$

Each of these updates consists of comparison, assignment, or addition of a block of bits in a word, which can be handled with word-level comparisons, assignment, and addition using additional pre-computed tables. For concreteness, assume that the values $T_2[1],\ldots,T_2[m]$ in T_2 are stored as the number $\sum_{i=1}^{m} T_2[i]2^{(i-1)\lceil\log(3rm+1)\rceil}$. For example, the value of $T_2[j]$ needed to perform the comparison can be looked up in another pre-computed table, and the addition of c to $T_2[j]$ can be performed by adding $c2^{(j-1)\lceil\log(3rm+1)\rceil}$ to $w(T_2)$, the exponents needed for such computations can also be prepared advance.

3.2. Worst case bounds.

To achieve constant worst case time bounds we re-build the data structure in the background. After $\frac{1}{2}m$ updates, at time t, we start preparing T_2 and T_3 during the remaining $\frac{1}{2}m$ updates. When the new tables are finished, they will be somewhat outdated in that they do not reflect any of the updates since time t. To this end we introduce another table, U, which contains the correct values of T_2, T_3 corresponding to the recent updates. The entries in U are indexed by the two outdated tables, and the $\frac{1}{2}m$ recent updates. Again, it can be verified that U can constructed in linear time and space for n sufficiently large.

4. THE LOWER BOUND

Let T be a rooted tree with n nodes, each of which can be in two states: *marked* or *unmarked*. The nodes on the unique path from v to the root are denoted $\pi(v)$, which includes v and the root. The *marked ancestor problem* is to maintain a data structure with the following operations:

mark(v): mark node v,

unmark(v): remove the mark from node v,

exists(v): return 'yes' if and only if $\pi(v)$ contains any marked node.

From [2] we have that the following lower bound in the cell probe model with word size $\log n$:

THEOREM 4.1 ([2]). *The marked ancestor problem requires* $\Omega(\log n/\log\log n)$ *worst case time per operation.*

To prove the lower bound stated in Thm. 1.1 we show that each *marked ancestor* operation can be supported by a constant number of reverse operations.

The tree T with n nodes is represented by a balanced string s of length $4n$. To initialise the structure we perform a depth first search in T. Let c be a counter initialised to 0. Each time we visit node v for the first or last time we

increment c by 2, and assign the values $\mathrm{first}(v) = c$ and $\mathrm{last}(v) = c$ respectively. A node v corresponds to four letters in s at positions $\mathrm{first}(v) - 1, \mathrm{first}(v), \mathrm{last}(v) - 1$ and $\mathrm{last}(v)$ defined as follows. Let

$$\begin{aligned} x &= s(1) \cdots s(\mathrm{first}(v) - 2) \\ y &= s(\mathrm{first}(v) + 1) \cdots s(\mathrm{last}(v) - 2) \\ z &= s(\mathrm{last}(v) + 1) \cdots s(4n) \end{aligned}$$

If v is marked then we let $s = x((y))z$, otherwise $s = x()y()z$. By virtue of the depth first search, the string s balances. To maintain the correspondence we only need to perform 2 reversals for every *mark* and *unmark* operation. Next we show how to support *exists*(v) using 4 reversals. Assume that v is unmarked (the other case is easy). First, perform $\mathrm{reverse}(\mathrm{first}(v) - 1)$ and $\mathrm{reverse}(\mathrm{last}(v))$. We claim that the last reversal returns 'yes' if and only if v has a marked ancestor. Finally, perform $\mathrm{reverse}(\mathrm{first}(v) - 1)$ and $\mathrm{reverse}(\mathrm{last}(v))$ once more to re-establish the correspondence.

To see that this approach works consider *exists*(v) on an unmarked node v. We have $s = x(\,)y(\,)z$, which we updated to $s' = x))y((z$ with the first two reversals. Note that y is a balanced string corresponding to the proper subtrees of v and that xz is a balanced string corresponding to the tree T without the subtree rooted at v. A node w that is not a proper ancestor to v will be represented by brackets in x or y, but not both. A proper ancestor w to v is represented with '((' in x and '))' in z if it is marked; otherwise it is represented with '()' in both x and z. Thus if v has no marked ancestors, both x and z will balance but s' will not. On the other hand, if v has a marked ancestor, the string s' will have the form

$$\cdots((\cdots))y((\cdots))\cdots$$

and balance.

REFERENCES

[1] A. Aho, J. Hopcroft, and J. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.

[2] S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. In *Proc 39th Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 534–543, 1998.

[3] G. S. Frandsen, T. Husfeldt, P. B. Miltersen, T. Rauhe, and S. Skyum. Dynamic algorithms for the dyck languages. In *Proc. 4th Workshop on Algorithms and Data Structures (WADS)*, pages 98–108, 1995.

[4] G. S. Frandsen, P. B. Miltersen, and S. Skyum. Dynamic word problems. *Journal of the ACM*, 1998.

[5] M. Fredman and D. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47:424–436, 1993.

[6] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and Systems Sciences*, 30(2):209–221, 1985.

[7] T. Husfeldt and T. Rauhe. Hardness result for dynamic problems by extensions of Fredman and Saks chronogram method. In *Proc 25th International Colloquium on Automata, Languages, and Programming (ICALP)*, Lecture Notes in Computer Science. Springer Verlag, Berlin, 1998.

[8] T. Husfeldt, T. Rauhe, and S. Skyum. Lower bounds for dynamic transitive closure, planar point location, and parentheses matching. *Nordic Journal of Computing*, 3(4):323–336, 1996.

[9] S. Patnaik and N. Immerman. Dyn-FO: A parallel, dynamic complexity class. volume 13 of *Proceedings of the acm sigact sigmod sigart symposium on principles of database systems 1994*, pages 210–221, 1994.