A Simple Integer Successor-Delete Data Structure

Gerth Stølting Brodal ✉ ⌂ 🆔

Aarhus University, Department of Computer Science, Denmark

―――― **Abstract** ――――

We consider a simple decremental data structure for maintaining a set of integers, that supports initializing the set to $\{1, 2, \ldots, n\}$ followed by d deletions and s successor queries in arbitrary order in total $\mathcal{O}\big(n + d + s \cdot (1 + \log_{\max(2, s/n)} \min(s, n))\big)$ time. The data structure consists of a single array of n integers. A straightforward modification allows the data structure to also support p predecessor and r range queries, with a total output k, in total $\mathcal{O}\big(n + d + k + q \cdot (1 + \log_{\max(2, q/n)} \min(q, n))\big)$ time, where $q = s + p + r$. The data structure is essentially a special case of the classic union-find data structure with path compression but with unweighted linking (i.e., without linking by rank or size), that is known to achieve logarithmic amortized time bounds (Tarjan and van Leeuwen, 1984). In this paper we study the efficiency of this simple data structure, and compare it to other, theoretically superior, data structures.

1 Introduction

We consider a simple decremental data structure for maintaining a set of integers S, under the operations DELETE(i) and SUCC(i), where initially $S = \{1, 2, \ldots, n\}$. The delete operation DELETE(i) removes the value i from S (or leaves the set unchanged if $i \notin S$), and the successor query SUCC(i) returns the smallest value in S larger than or equal to i, i.e., $\text{succ}(S, i) = \min\{j \in S \mid j \geq i\}$. Figure 1 shows the full code for the operations and an example of the data structure for a set of values. The data structure consists of a single array $A[1..n]$ of n integers. We view $A[i]$ as a pointer from index i to index $A[i]$ in A: if $i \in S$ then $A[i] = i$; otherwise $i < A[i] \leq \text{succ}(S, i)$. Figure 2 shows a minimal sequence of operations resulting in non-nested pointers ($2 < 3 < A[2] < A[3]$). A simple modification of the data structure also supports the predecessor query PRED(i) that returns the largest value in S smaller than or equal to i, i.e., $\text{pred}(S, i) = \max\{j \in S \mid j \leq i\}$. The only modification is that for $i \in S$ we let $A[i] = \text{pred}(S, i - 1)$ instead of $A[i] = i$. See Figure 3.

The structure in Figure 1 is likely folklore in practice (but we failed to find a reference in the literature). The data structure is interesting due to its simplicity, low space usage (an array of n integers), and low constants in the running time, but there exist data structures in the literature with theoretically superior complexity. See Table 1.

1.1 Related work

To support successor queries on a statics set, the canonical solution is to store the values in sorted order in an array and perform successor queries using binary search in $\mathcal{O}(\log |S|)$ time. In the static case of integers, the successors can be stored explicitly in an array of size n, and successor queries can be looked up in constant time. If the set S is dynamic, the classic solution is to store the values in a balanced binary search tree, like an AVL-tree [1] or

```
Proc Init(n)                    Proc Delete(i)    Proc Succ(i)          Proc Succ(i)
  Create A[1..n]                  A[i] ← i + 1       if A[i] = i then       j ← i
  for i ← 1 to n do                                   return i             while j < A[j] do
    A[i] ← i                                         A[i] ← Succ(A[i])        j ← A[j]
                                                     return A[i]           while i < A[i] do
                                                                             i' ← A[i]
                                                                             A[i] ← j
                                                                             i ← i'
                                                                           return j
```
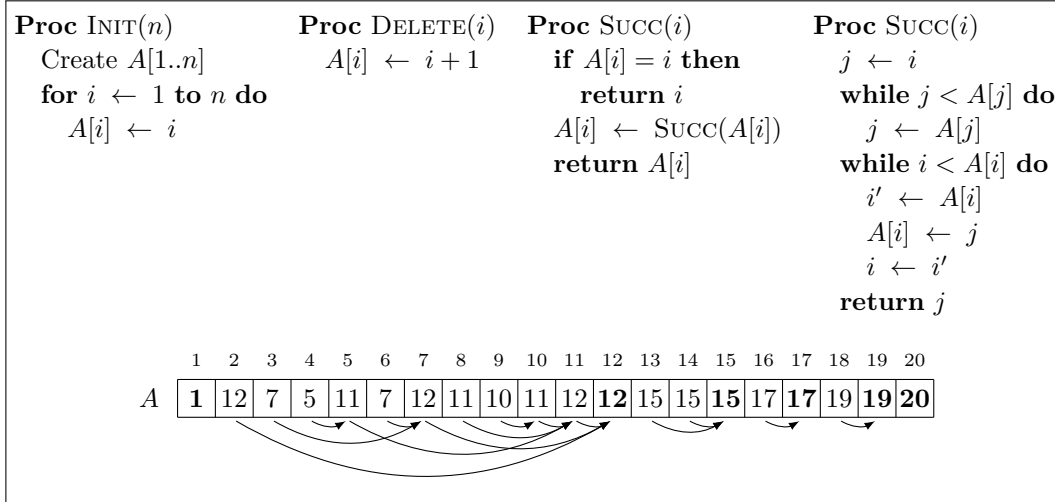

Figure 1 Succ-Delete implementation with two alternative implementations of Succ using path compression: (left) recursive and (right) iterative two-pass. (Bottom) a data structure for $S = \{1, 12, 15, 17, 19, 20\}$, where $A[i] = i$ for $i \in S$ and $i < A[i] \le \mathrm{succ}(S, i)$ for $i \notin S$. We view $A[i]$ as a "pointer" to another index of A, where $\mathrm{succ}(S, i) = \mathrm{succ}(S, A[i])$. The values of A depend on the sequence of operations performed. Note that pointers are not necessarily nested.
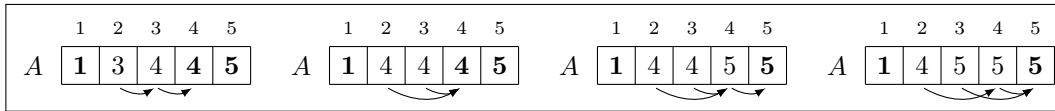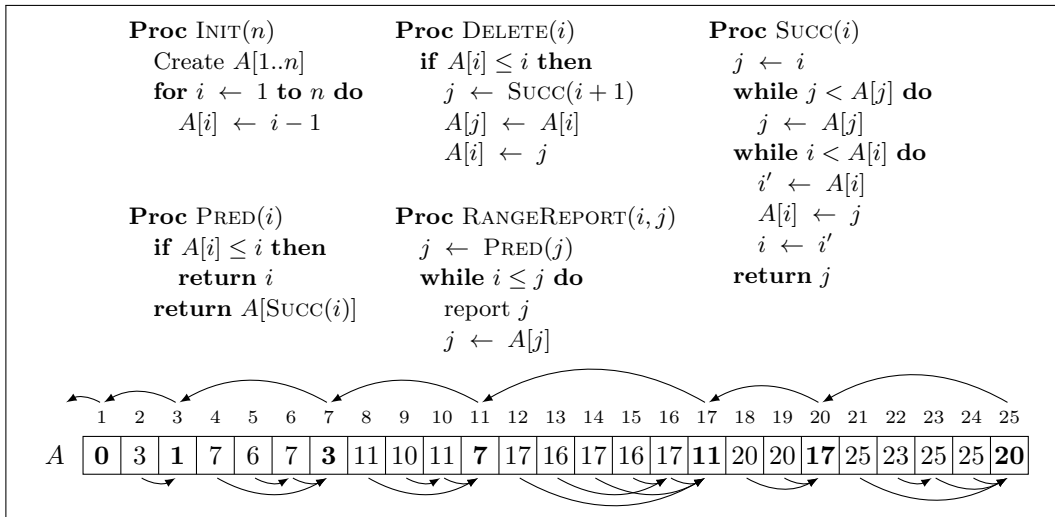

Figure 2 A minimal sequence of operations resulting in non-nested pointers: Init(5), Delete(2), Delete(3), Succ(2), Delete(4), and Succ(3). The figure shows the states after each of the last four operations.

```
Proc Init(n)              Proc Delete(i)             Proc Succ(i)
  Create A[1..n]            if A[i] ≤ i then            j ← i
  for i ← 1 to n do          j ← Succ(i + 1)           while j < A[j] do
    A[i] ← i − 1            A[j] ← A[i]                   j ← A[j]
                           A[i] ← j                    while i < A[i] do
                                                         i' ← A[i]
Proc Pred(i)              Proc RangeReport(i, j)        A[i] ← j
  if A[i] ≤ i then          j ← Pred(j)                 i ← i'
    return i               while i ≤ j do              return j
  return A[Succ(i)]          report j
                            j ← A[j]
```


Figure 3 Succ-Pred-Delete implementation and a data structure for $\{1, 3, 7, 11, 17, 20, 25\}$.

■ **Table 1** Various solutions to the successor-delete problem over the integers $\{1, 2, \ldots, n\}$. $\mathcal{O}_A$ denote amortized time bounds, $\alpha(m, n)$ is a very slowly growing inverse Ackermann like function.

Reference	Succ	Delete	Insert
Balanced binary search tree [1, 12]	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Augmented static binary tree	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
van Emde Boas trees [25, 26]	$\mathcal{O}(\log \log n)$	$\mathcal{O}(\log \log n)$	$\mathcal{O}(\log \log n)$
Succinct rank-select [14]	$\mathcal{O}\left(\frac{\log n}{\log \log n}\right)$	$\mathcal{O}\left(\frac{\log n}{\log \log n}\right)$	$\mathcal{O}\left(\frac{\log n}{\log \log n}\right)$
Succinct successor [19]	$\mathcal{O}(\log \log n)$	$\mathcal{O}_A\left(\frac{\log n}{\log \log n}\right)$	$\mathcal{O}_A\left(\frac{\log n}{\log \log n}\right)$
Union-find [21]	$\mathcal{O}_A(\alpha(m, n))$	$\mathcal{O}_A(\alpha(m, n))$	
Union-find, weighted linking only [6]	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	
Union-find, path compression only [23]	$\mathcal{O}_A(\log n)$	$\mathcal{O}_A(\log n)$	
Weighted quick-find (McIlroy and Morris)	$\mathcal{O}(1)$	$\mathcal{O}_A(\log n)$	
Interval union-find [9]	$\mathcal{O}(1)$	$\mathcal{O}_A(1)$	
This paper	$\mathcal{O}_A(\log n)$	$\mathcal{O}_A(1)$	

a red-black tree [12], supporting insertions, deletions, and successor and predecessor queries in $\mathcal{O}(\log |S|)$ time. Binary search trees are inherently node oriented, where each of the $|S|$ nodes stores at least a value and two pointers to its two children, and possibly some bits of balance information (splay trees [20] do not store any balance information in the nodes, but the time bounds are amortized). *Implicit* binary search trees [7, 8, 15] reduce the space usage to a single array of size $|S|$ storing a permutation of S (all additional information is encoded in the permutation of the values). In this paper we even allow the data structures to use space $\mathcal{O}(n)$, where n can be much larger than $|S|$ after many deletions. Another binary tree solution is an augmented static binary tree with leaves left-to-right representing $1, \ldots, n$, and where each node stores the maximum non-deleted node in its subtree. Such a tree can be stored in an array of size $\mathcal{O}(n)$ (where the children of node i are nodes $2i$ and $2i + 1$, like in the binary heaps by Williams [28]) and supports updates and successor queries in $\mathcal{O}(\log n)$ time.

Exploiting that values are integers, van Emde Boas, Kaas and Zijlstra [25, 26] presented a dynamic data structure supporting insertions and deletions, and predecessor and successor queries in $\mathcal{O}(\log \log n)$ time and $\mathcal{O}(n)$ space. Lower bound trade-offs between query time and space usage (in the cell-probe model) were studied by Beame and Fich [3] and Pătraşcu and Thorup [16]. Succinct rank-select data structures, i.e., data structures using space $\log_2 \binom{n}{|S|} + o(n)$ bits close to the information theoretic lower bound, can also be used to answer successor queries, see, e.g., the recent dynamic rank-select structure by Li, Liang,Yu, and Zhou [14]. Note that dynamic rank-select data structures have a lower bound of $\Omega(\log n / \log \log n)$ time, see Pătraşcu and Thorup [17] — a higher lower bound than for predecessor and successor queries. Pibiri and Venturini [19] presented a data structure supporting insertions and deletions in $\mathcal{O}\left(\frac{\log n}{\log \log n}\right)$ amortized time, and successor and predecessor queries in $\mathcal{O}(\log \log n)$ time, using $\log_2 \binom{n}{|S|} + O(|S|)$ bits of space. Dinklage, Fischer and Herlez [5] did a comprehensive experimental study of dynamic integer predecessor data structures supporting both insertions and deletions.

In this paper we only consider the decremental version of the problem, where values can only be deleted and never be inserted again. We allow the deletion of an already deleted value. This problem is in the literature known as the *interval union-find* problem: Consecutive deleted integers form intervals with their (non-deleted) successor. Deleting an integer i unions the two intervals of deleted integers to the left and right of i. Italiano and Raman [13,

Chapter 5.3] give a survey of the results for the interval union-find problem and list some applications. By exploiting the power of the RAM model, a data structure by Gabow and Tarjan [9] for the interval union-find problem supports successor queries in constant time and deletions in amortized constant time (see [22] for an introduction to amortized time analysis): Partition the integers $\{1, \ldots, n\}$ into *microsets* of $b = \lceil \log n \rceil$ consecutive values. For each microset store the values in S as a bitvector of length b in a single word, and handle queries on a microset either by tabulation or dedicated RAM instructions (for finding the least or most significant set bit in a word[1]). A separate interval-union data structure is maintained for the *macroset* set $\bar{S} \subseteq \{1, \ldots, \lceil n/b \rceil\}$, where $i \in \bar{S}$ if and only if the i-th microset is non-empty, i.e., $S \cap [1 + (i-1) \cdot b, i \cdot b] \neq \emptyset$. The *macroset* structure consists of an array of size $\lceil n/b \rceil$ storing pointers to the nearest non-empty successor microset structure. The pointers are updated using the "relabel the smaller half" method when a microset becomes empty. While deletions and queries to microsets and queries to the macroset take constant time, deletions to the macroset take amortized $\mathcal{O}(\log n)$ time, but only happens when all b values in a microset has been deleted, i.e., the overall time for DELETE becomes amortized $\mathcal{O}\big(1 + (\log n)/b\big) = \mathcal{O}(1)$. The macroset data structure is in the literature known as the *weighted quick-find* data structure, and is attributed to McIlroy and Morris by Aho, Hopcroft and Ullman [2].

The interval union-find problem is a special case of the *union-find* problem, where an initial set of n singleton sets are maintained under the union of two sets containing two values i and j, UNION(i, j), and the query FIND(i) that returns a representative value for the set containing i. Galler and Fischer [10] introduced the rooted tree representation for the union-find problem, where each set is represented by a rooted tree, the root is the representative of the set, and each node stores a pointer to the parent node. A FIND(i) operation traverses the path from node i to the root, and *compresses* the path so that all visited nodes are shortcut to have the root as their parent, and UNION(i, j) performs FIND(i) and FIND(j) and links the two roots, making the smaller tree (with respect to size or "rank") a child of the root of the larger tree. Path compression and weighted linked can be combined or applied separately. Only applying linking by size gives $\mathcal{O}(\log n)$ time UNION and FIND operations (Fischer [6]), whereas naive linking with path compression gives amortized $\mathcal{O}(\log n)$ time UNION and FIND operations (Tarjan and van Leeuwen [23, Theorem 4]). Tarjan [21] showed that combining path compression with weighted linking implies $m \geq n$ FIND operations take $\mathcal{O}(m \cdot \alpha(m, n))$ time, where $\alpha(m, n)$ is a very slowly growing inverse Ackermann like function. Tarjan and van Leeuwen [23] analyzed various variations of path compression for the union-find problem. They proved that *path halving* achieves asymptotically identical time $\Theta\big(m \log_{1+m/n} n\big)$ as path compression, for $m \geq n$ FIND operations. Path halving was introduced by van Leeuwen and van der Weide [24, 27]. Union-find structures replacing weighted linking by randomized linking were considered by Goel, Khanna, Larkin and Tarjan [11], avoiding the space usage for weights or ranks. See Italiano and Raman [13, Chapter 4.2] for a survey of the results for the union-find problem. An experimental evaluation of 55 union-find data structures was done by Patwary, Balir and Manne [18] in the context of computing the connected components of various graphs, where a variant of Rem's union-find data structure (described by Dijkstra [4, Chapter 23]) turned out to achieve the best performance.

Any union-find structure can be combined with the microset-macroset idea to solve the interval union-find problem. For the union-find structure using linking by rank and path compression, we obtain an interval union-find structure supporting deletions and successor

---

[1] `https://en.wikipedia.org/wiki/Find_first_set`

queries in amortized $\mathcal{O}(1)$ time: d deletions cause at most $\mathcal{O}(d/\log n)$ unions in the macro-structure, creating $O(d/\log n)$ links. Since a link changing parent during a path compression is linked to a higher ranked node, a link can at most be changed $\mathcal{O}(\log n)$ times, and the total number of link updates during path compressions is $\mathcal{O}(d/\log n \cdot \log n) = \mathcal{O}(d)$. The resulting structure uses space $\mathcal{O}(n)$ bits.

The data structure we consider in Figure 1, is essentially the same as the union-find data structure by Rem, both consisting of an array of self-loops and forward pointers, except that we only consider the restricted case of the interval union-find problem, allowing us to simplify the operations. We apply path compression without weighted linking, where the representative of a set is the successor of all values in the set, and where $\textsc{Delete}(i)$ is $\textsc{Union}(i, i+1)$, that links i below $i+1$ without performing $\textsc{Find}(i+1)$, i.e., without performing a path compression and where $i+1$ is not necessarily a root.

1.2 Results

Heavily inspired by the results of Tarjan and van Leeuwen [23], we obtain the following results. For the successor-delete data structure in Figure 1 we obtain the following result.

▶ **Theorem 1.** *A sequence of* $\textsc{Init}(n)$ *followed by* d $\textsc{Delete}$ *and* s $\textsc{Succ}$ *operations in arbitrary order takes* $\mathcal{O}\big(n + d + s \cdot \big(1 + \log_{\max(2, s/n)} \min(s, n)\big)\big)$ *time.*

Note that for a sublinear number of queries, $s \leq n$, the time is $\mathcal{O}(n + d + s \cdot \log s)$, e.g., for $s = \mathcal{O}(n/\log n)$ the time is $\mathcal{O}(n+d)$. For a superlinear number of queries, where $s \geq n^{1+\varepsilon}$ for a constant $\varepsilon > 0$, the time is $\mathcal{O}\big(n + d + s \cdot \big(1 + \log_{n^{1+\varepsilon}/n} n\big)\big) = \mathcal{O}\big(n + d + s \cdot (1 + 1/\varepsilon)\big)$.

Our second result is that the data structure also supports the range reporting query $\textsc{RangeReport}(i, j)$ in Figure 4, that reports $S \cap [i, j]$ in sorted order. Provided we never delete a value already deleted or we use the $\textsc{Delete}(i)$ operation in Figure 4, that only modifies $A[i]$ if i has not yet been deleted, we obtain the following result.

▶ **Theorem 2.** *A sequence of* $\textsc{Init}(n)$, *followed by* d $\textsc{Delete}$, s $\textsc{Succ}$, *and* r $\textsc{RangeReport}$ *operations in arbitrary order, where the total output of the range reporting queries is* k, *takes* $\mathcal{O}\big(n + d + k + q \cdot \big(1 + \log_{\max(2, q/n)} \min(q, n)\big)\big)$ *time, where* $q = s + r$.

Finally, a slight modification of our data structure also supports predecessor queries, see Figure 3 in Section 4, and obtains the following result.

▶ **Theorem 3.** *A sequence of* $\textsc{Init}(n)$, *followed by* d $\textsc{Delete}$, s $\textsc{Succ}$, p $\textsc{Pred}$ *operations, and* r $\textsc{RangeReport}$ *operations in arbitrary order, where the total output of the range reporting queries is* k, *takes* $\mathcal{O}\big(n + d + k + q \cdot \big(1 + \log_{\max(2, q/n)} \min(q, n)\big)\big)$ *time, where* $q = s + p + r$.

It should be noted that maintaining two copies of the successor-delete data structure of Theorem 2, one for each direction for reporting successor and predecessor queries, respectively, actually achieves Theorem 3. The advantage of the data structure in Section 4 is that the space usage is only a single array of n integers, and deletions only need to update a single array instead of two arrays.

In Section 6 we present our experimental results of comparing our data structure to the weighted quick-find data structure and the two-pass weighted union-find data structure with path compression, with and without using microsets.

Proc Delete(i)	**Proc** Succ(i)	**Proc** RangeReport(i, j)
if $A[i] = i$ **then**	**while** $A[i] > i$ **do**	**while** $i \leq j$ **do**
$\quad A[i] \leftarrow i + 1$	$\quad A[i] \leftarrow A[A[i]]$	$\quad i \leftarrow$ Succ(i)
	$\quad i \leftarrow A[i]$	$\quad$ **if** $i \leq j$ **then**
	return i	$\quad\quad$ report i
		$\quad\quad i \leftarrow i + 1$

🟨 **Figure 4** Implementation of Delete that ensures $A[i]$ is increasing over time, a one-pass implementation of Succ using path halving, and range reporting query RangeReport(i, j), where $1 \leq i \leq j \leq n$.

1.3 Preliminaries

Throughout this paper we consider maintaining a set $S \subseteq \{1, 2, \ldots, n\}$. We let $\mathrm{succ}(S, i) = \min\{j \in S \mid j \geq i\}$ and $\mathrm{pred}(S, i) = \max\{j \in S \mid j \leq i\}$ denote the successor and predecessor of i in S, respectively. We assume that the values 1 and n are never deleted, such that the $\mathrm{succ}(S, i)$ and $\mathrm{pred}(S, i)$ are always defined, for $1 \leq i \leq n$. (In our implementations we actually consider the integers $\{0, 1, \ldots, n, n+1\}$, where 0 and $n+1$ are never deleted, such that intuitively 0 acts as $-\infty$ and $n+1$ as $+\infty$.)

2 Successor-delete

Our basic successor-delete data structure is shown in Figure 1. It stores a set S, where $\{1, n\} \subseteq S \subseteq \{1, \ldots, n\}$. It consists of a single array $A[1..n]$. For $1 \leq i \leq n$, it is an invariant that $i < A[i] \leq \mathrm{succ}(S, i)$ if $i \notin S$ and $A[i] = i$ if $i \in S$. We view $A[i]$ as a pointer from index i to index $A[i]$ in A, such that all elements with j as their successor in S form a rooted tree with j as the root. See the pointers below the array A in Figure 1, where 12 is the root of the tree containing $\{2, 3, \ldots, 12\}$.

To compute $\mathrm{succ}(S, i)$, we can repeatedly let $i \leftarrow A[i]$ until $i = A[i]$. In the worst case this will take $\mathcal{O}(\mathrm{succ}(S, i) - i + 1)$ time. To achieve a better amortized performance, we apply *path compression* such that for each accessed i we let it point directly to $\mathrm{succ}(S, i)$. Figure 1 shows two implementations of Succ(i) with two different path compressions: recursively or through two iterative passes: first follow the path from i to find the root $\mathrm{succ}(S, i)$, and in a second traversal of the path sets all pointers to point to $\mathrm{succ}(S, i)$. Applying the recursive compression or the two-pass path compression results in the same data structure, but the two-pass path compression does not require a recursion stack. Slightly less aggressive than path compression is *path halving*, where every second node i on the path is shortcut to its grandparent, i.e., $A[i] \leftarrow A[A[i]]$. Path halving has the benefit that it can be implemented in a single traversal of the path from i to $\mathrm{succ}(S, i)$, see Figure 4. The analysis in Section 5 only considers path compression.

A Delete(i) operation could set $A[i]$ to $\mathrm{succ}(S, i+1)$. This would require a call to Succ($i+1$) to update $A[i]$. We apply the simpler idea of just setting $A[i] \leftarrow i+1$, avoiding the call to Succ during Delete that would be the natural approach if we implemented the successor data structure using a union-find data structure. If $i \notin S$ when performing Delete(i), we in the code in Figure 1 naively reset $A[i] \leftarrow i+1$, to avoid the check if $A[i] = i$. If one added this check, see Delete in Figure 4, we would only update $A[i]$ if $i \in S$. This would ensure that $A[i]$ over time only increases. Note that either way of implementing Delete can cause pointers not to be nested, as shown in Figure 2.

3 Range reporting

Another operation one could consider supported is the RANGEREPORT(i, j) operation that reports the values in $S \cap [i, j]$ in sorted order, where $1 \leq i \leq j \leq n$. If all elements in S were linked in increasing order, RANGEREPORT(i, j) could trivially be supported by performing SUCC(i) to find the first element to report (provided it is smaller than or equal to j), and then traverse and report the elements of the linked list until the first element larger than j is encountered. The time would be the time for a SUCC query and then linear in the output. The data structure in Figure 1 can support range reporting queries without such additional links: The first element to report can be found by performing $i \leftarrow$ SUCC(i), the next element in S can the found by performing $i \leftarrow$ SUCC$(i + 1)$, and so on, until i is larger than j. See code in Figure 4. Even that this approach applies multiple SUCC queries, we in Section 5 show that the amortized time for RANGEREPORT is still the time for a single SUCC query plus linear in the output, provided we are slightly careful about deletions.

4 Successor-predecessor-delete structure

If both successor and predecessor queries should be supported, one could maintain two separate data structures, using one array for each query direction. A deletion would then require performing a deletion on two data structures. A more space efficient way is to use the data structure in Figure 1, but where we for each $i \in S$ store $A[i] = \mathrm{pred}(S, i - 1) < i$, except for $A[1] = 0$. It follows that all values in S are linked in decreasing order in A. For $i \notin S$, we define and maintain $i < A[i] \leq \mathrm{succ}(S, i)$ exactly as before. Figure 3 shows the resulting data structure, where SUCC is implemented using two-pass path compression. Since DELETE(i) for an $i \in S$ needs to remove i from the linked list of values in S, we need to compute $\mathrm{succ}(S, i + 1)$ using SUCC$(i + 1)$, to update $A[\mathrm{succ}(S, i + 1)] \leftarrow A[i]$. We can shortcut $A[i]$ to point directly to $\mathrm{succ}(S, i + 1)$, now it is anyway computed. To support PRED(i), observe that $\mathrm{pred}(S, i)$ is i, when $i \in S$, i.e., $A[i] \leq i$. Otherwise, $\mathrm{pred}(S, i) = A[\mathrm{succ}(i)]$ for $i \notin S$. Since we have a decreasing linked list of all elements in S, RANGEREPORT(i, j) can be implemented by first finding the largest value in $S \cap [i, j]$ by performing PRED(j), and then traversing and reporting the values on the linked list in decreasing order until a value smaller than i is encountered. The drawback of this solution is that for each DELETE(i) operation a successor query is required to update $A[$SUCC$(i + 1)]$. Elements are reported in decreasing order but can be changed to be increasing order by reversing the output (or mirroring the whole data structure), if this is desired.

5 Analysis

In this section we analyze the theoretical performance of the proposed data structures, i.e., we give the proofs of Theorems 1–3. The analysis follows the same lines of reasoning as used by Tarjan and van Leeuwen for the analysis of union-find data structures using naive linking [23].

5.1 Proof of Theorem 1

Consider the operations in Figure 1, and assume we perform INIT(n) followed by d DELETE and s SUCC operations in an arbitrary order. The INIT operation together with the d DELETE operations clearly take $\mathcal{O}(n + d)$ time. What remains is to analyze the time spent on the s SUCC queries.

We assume that we never delete a value that is already deleted (or we use the DELETE operation in Figure 4). We later show that this assumption is not necessary. Under this assumption, $A[i] = i$ until i is deleted, where $A[i]$ is set to $i + 1$. Subsequently, $A[i]$ is only changed by path compressions increasing $A[i]$, i.e., over time $A[i]$ is non-decreasing.

For a successor query SUCC(i), we denote the index succ(S, i) the *root* of the query. Let R be the set of all root indices for the s SUCC queries during the sequence of operations. Clearly $|R| \leq \min(s, n)$. Consider the indegree of an index i over time. While $i - 1$ is not deleted, the indegree of i is zero. When $i - 1$ is deleted, $A[i - 1]$ is set to point to i, and i gets indegree one. If i is never the root of a query, $i \notin R$, i will never get another in-pointer, since path compressions only change pointers to point to the root of a query, and when $A[i - 1]$ is increased (because of a path compression), i gets indegree zero and the indegree of i remains zero for the remainder of the sequence of operations.

A SUCC(i) query accesses a sequence of indices $i_0 < i_1 < \cdots < i_p$ in A, where $i_0 = i$, $i_{j+1} = A[i_j]$ for $0 \leq j < p$, and $A[i_p] = i_p = \text{succ}(S, i)$. For the query we count separately the accesses to three indices: the index i_0 where the query starts, and the last two indices i_{p-1} and i_p, where the pointers do not change. For the *intermediate* nodes $i_1, \ldots, i_{p-2}$, we know they have indegree at least one before the query (since they are on the query path) and the in-pointer is updated (since we use path compression). If $i_j \notin R$ $(1 \leq j \leq p - 2)$, then by the above discussion i_j had indegree one before the query (a pointer from $i_j - 1$) and indegree zero after the query, and no further query can have i_j as an intermediate index (but we could visit i_j because of a SUCC(i_j) query, where it would play the role of "i_0"). It follows that the total number of accesses by all SUCC queries to intermediate nodes not in R is at most n. It follows that the total number of nodes accessed by all queries is at most $3s + n$ plus the number of accesses to intermediate nodes in R.

To bound the remaining number of accesses to intermediate nodes in R we introduce the notion of *rank* (like in [23]). For $i \in R$ with $i < A[i]$, we define the *rank* of i to be $r_i = \lfloor \log_\Delta w_i \rfloor$, where $\Delta = \max(2, \lceil s/n \rceil)$ and $w_i = |R \cap [i, A[i])| \geq 1$, i.e., the rank of i is the base Δ logarithm of the number of indices between i and $A[i]$ in the array (excluding $A[i]$) that eventually become a root during the sequence of operations. If $i \in R$ has rank r, then $\Delta^r \leq w_i < \Delta^{r+1}$. Ranks are upper bounded by $\lfloor \log_\Delta |R| \rfloor$. We refine each rank r into $\Delta - 1$ *levels*, $1 \leq \ell < \Delta$, where an index i of rank r has *level* ℓ if $\ell \cdot \Delta^r \leq w_i < (\ell + 1) \cdot \Delta^r$. Consider a path compression that shortcuts an index $i \in R$ over another index $j \in R$, i.e., $i \leq A[i] \leq j \leq A[j]$ before the shortcut is made and the new value of $A[i]$ is at least the old value of $A[j]$. If $r_i < r_j$ then the rank of i increases by at least one (to rank $\geq r_j$), but if $r_i = r_j$, then r_i does not necessarily increase, but the level of index i increases by at least one. It follows that for each SUCC query and rank r at most one accessed index of rank r in R does not change neither rank nor level (the rightmost rank r index accessed). Over the sequence of operations, an index in R can at most increase rank $\lfloor \log_\Delta |R| \rfloor$ times and for each of the $1 + \lfloor \log_\Delta |R| \rfloor$ ranks it can increase level at most $\Delta - 2$ times. It follows that the total number of accesses to nodes for all SUCC queries is at most

$$3s + n + s \cdot (1 + \lfloor \log_\Delta |R| \rfloor) + |R| \cdot \left(\lfloor \log_\Delta |R| \rfloor + (\Delta - 2)(1 + \lfloor \log_\Delta |R| \rfloor) \right) .$$

The total time for the sequence of operations becomes $\mathcal{O}\left(n + d + s + (s + |R|\Delta) \log_\Delta |R| \right)$. Using $\Delta = \max(2, \lceil s/n \rceil)$ and $|R| \leq \min(s, n)$, the total time for INIT(n), d DELETE and s SUCC queries is $\mathcal{O}\left(n + d + s \cdot (1 + \log_{\max(2, s/n)} \min(s, n)) \right)$, as stated in Theorem 1.

To avoid the assumption that we never delete a value that is already deleted, observe that performing DELETE(i) when $A[i] > i$, resets $A[i] \leftarrow i + 1$. If $i \in R$, then this causes w_i to be reset to one, i.e., the rank of i is reset to zero and the level of i to one. But notice that the

first path compression accessing i will increase $A[i]$ to at least the value before the deletion, i.e., the lost (rank, level) potential is regained by the first access to $A[i]$. So, in the analysis above we only need to account for one additional access to i for each DELETE(i) operation. If $i+1 \notin R$, then we above used that $i+1$ would never get indegree one again, if $A[i] > i+1$, and argued that we only would visit i once as an intermediate. Setting $A[i] \leftarrow i+1$ will cause $i+1$ once again to gain indegree one, allowing one additional access to $A[i]$ as an intermidate node. We charge these two additional accesses to the deletion. The remaining analysis is unchanged, and Theorem 1 holds without the assumption that we never delete a value that is already deleted.

5.2 Proof of Theorem 2

In Figure 4 we show how to implement RANGEREPORT(i, j) using repeated calls to SUCC. If k is the size of the total output by r RANGEREPORT operations, these in total perform at most $r + k$ calls to SUCC. We can replace s in Theorem 1 with $s + r + k$ to obtain a bound on the total time for a sequence of operations also containing r RANGEREPORT queries. This bound is not linear in the output size k though.

To give a better bound, we denote a subset of the links *outer links*. If i_1 and i_2 are two consecutive elements in S, i.e., $i_2 = \text{succ}(S, i_1 + 1)$, then the links on the path from $i_1 + 1$ to i_2 are the outer links. Observe, that RANGEREPORT after the first SUCC query has been performed only accesses outer links, which are shortcut, such that the outer path afterwards between i_1 and i_2 consists of a pointer directly from $i_1 + 1$ to i_2, i.e., all traversed links between i_1 and i_2 become non-outer links except for one. I.e., accessing the outer links can be charged to the output size plus the number of outer links becoming non-outer links, and the initial SUCC(i) query.

An outer link from i is created when i is deleted. Provided that we never set $A[i] \leftarrow i+1$ except for the initial deletion of i, a link that has become non-outer will never become outer again. It follows that we charge each DELETE the creation of an outer link, and the total time becomes the bound given by Theorem 1 plus $\mathcal{O}(k)$ for the output size k, and replacing s with $s + r$ in the bound, to cover the initial SUCC query performed by each RANGEREPORT. This gives the bound stated in Theorem 2.

5.3 Proof of Theorem 3

The construction in Figure 3 is essentially the same as the structure in Figure 1 with respect to forward pointers for deleted indices. To maintain the linked predecessor chain, a DELETE(i) operation for $i \in S$ needs to perform a SUCC($i + 1$) operation to update $A[\text{succ}(S, i + 1)] \leftarrow A[i]$, before setting $A[i] \leftarrow i + 1$. The work is $\mathcal{O}(1)$ in addition to the cost for the call to SUCC($i + 1$).

Similar to the argument for range-reporting queries in Theorem 2, the links followed by SUCC($i + 1$) are outer links, and we can apply the same accounting as in Theorem 2 that all outer links except one become non-outer links, and can be charged for the traversal. The link from i to $i + 1$ becomes a new outer link.

Each of the p PRED and r RANGEREPORT queries essentially perform one SUCC query, plus constant work and work linear in the output, respectively. The total work becomes the cost of $s + p + r$ SUCC queries, plus the output size k, plus the cost of the INIT and DELETE operations. The bound stated in Theorem 3 follows from Theorem 2.

6   Experimental evaluation

Variants of the proposed successor-delete data structure and some data structures from the literature were implemented in C, storing A using 64 bit integers. The experiments were run on an HP EliteBook 640 G10 with an Intel Core i7-1365U processor, running Windows 11 (23H2), using GCC 14.2.0 (MSYS2), and compiled with optimization level -O3. The source code is available on GitHub[2].

Each data point includes the time for INIT(n), and all DELETE and SUCC operations performed. The time is the average over at least five runs, but sufficiently many runs to get a total running time of at least one second. Each data point is the minimum observed repeating the evaluation three times. We tested the correctness of all algorithms evaluated by checking if they returned the same answers to all tested successor queries (this test is not included in the measured running times).
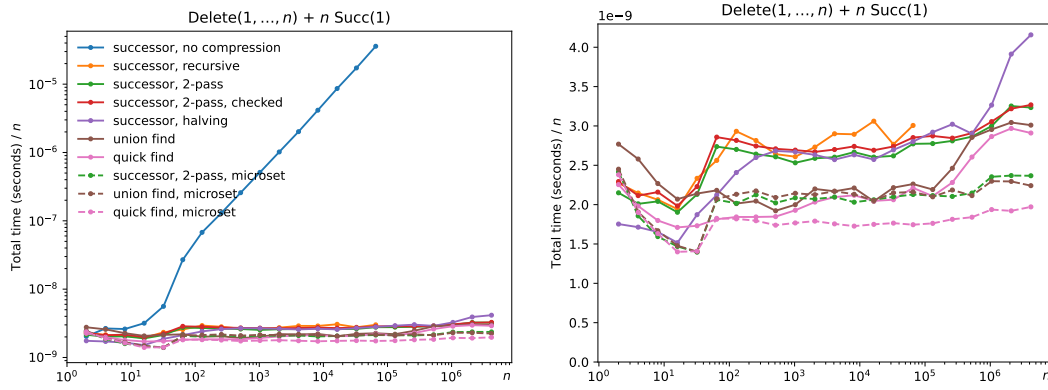
We implemented five versions of the proposed successor-delete data structure: 1) with no path compression, 2) using recursive path compression, 3) a two-pass path compression, 4) a two-pass path compression and checked deletions, and 5) path halving. For $1)-3)$ and $5)$, deletions are implemented as in Figure 1, where $A[i]$ is always set to $i + 1$ when i is deleted, whereas 4) only updates $A[i]$ if i is still in the set, i.e., $A[i] = i$ (Figure 4). Furthermore, we implemented 6) the weighted quick-find data structure and 7) the union-find data structure with weighted linking and two-pass path compression. Whereas $1)-5)$ only store the array A of integers, both 6) and 7) store an array with three integers per node (parent pointer, weight, and the maximum in the set, i.e., the successor of all elements in the set). Finally, we applied the microset-macroset idea from [9] to 8) the weighted quick-find structure, 9) the union-find structure with path compression and linking by weight, and 10) the successor-delete data structure with two-pass path compression. For $8)-10)$ a microset consists of 64 bits stored in a single `long long`, and we use the GCC builtin function `__builtin_ctzll` to compute the successor inside a microset[3].

That path compression is crucial to our data structure should be obvious: If we delete all elements without performing any path compression, all indices form one long chain of length n, and SUCC(1) will take linear time. Figure 5(left) shows that the total time is quadratic for INIT(n) followed by deleting all elements and performing SUCC(1) n times (since the time divided by n is linear with slope 1 in a log-log plot). Figure 5(right) shows the same data without the variant with no path compression. All algorithms have a theoretical running time of $\mathcal{O}(n)$ for this sequence of operations, which is confirmed by all measured running times divide by n being about constant. For $n > 2^{16}$, the recursive path compression failed because of a stack overflow on Windows, and we only show the results for $n \le 2^{16}$ for the recursive path compression variant for these sequences of operations.

As noted after the statement of Theorem 1, the worst-case behavior of our algorithm per successor query is when the number of successor queries s is about n. A sequence of $\Theta(n)$ operations forcing the successor-delete data structure with path compression to do $\Theta(n \log n)$ work is to delete the elements in increasing order, and after DELETE(i) pick a worst-case SUCC(j) query, i.e., a query with longest path from j to $i + 1$. The upper bound follows by Theorem 1, whereas the matching lower bound for this sequence follows by a result of Fischer [6, Theorem 1], who considered this sequence to prove an $\Omega(n \log n)$ lower bound for the union-find problem with unweighted linking and path compression. Whereas our

---

[2] https://github.com/gsbrodal/sea25
[3] https://gcc.gnu.org/onlinedocs/gcc/Bit-Operation-Builtins.html

Figure 5 Running time for deleting $1, \ldots, n$ in increasing order, followed by n calls to Succ(1). (left) with a variant with no path compression; (right) without the no path compression variant, where all data structures require total time $\Theta(n)$ for this input.
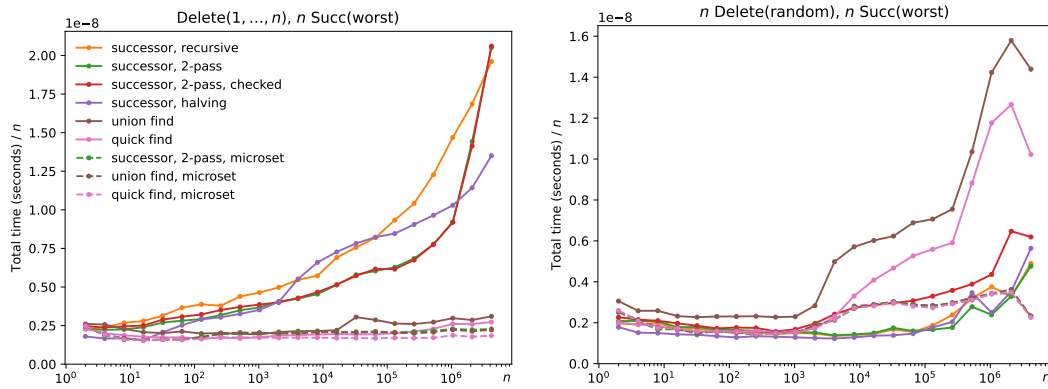
Figure 6 (left) Running time for deleting $1, \ldots, n$ in increasing order, and n times Succ(1) with path compression. Since only the first query to a successor-delete data structure with path compression will perform a path compression, the theoretical running time for these is $\mathcal{O}(n)$. (right) Running time for deleting $1, \ldots, n$ in increasing order, and after each Delete(i) performing a Succ(j) query for some j with longest path from j to $i+1$ when using the successor-delete data structure with path compression.

successor-delete data structure has a total running time of $\Theta(n \log n)$ for this input, both the weighted quick-find and the union-find data structures will only need $\mathcal{O}(n)$ time, since all unions link a set of size i with a set of size one, i.e., both the union-find and the weighted quick-find data structures will be a star containing all the deleted elements. Figure 6(left) shows the running time for this sequence of operations. The results show that the running times of the successor-delete data structures are in the same ballpark (but asymptotic slower than) the theoretically more efficient data structures.

Figure 6(right) shows results for n times deleting a random integer from $\{1, \ldots, n\}$ (possibly already deleted elements), and after each deletion performing a Succ(j) query for some j with longest path from j. The successor-delete data structures on this input benefit from that for a long time deletions are sparse in the input, and there are many small rooted trees with short leaf-to-root paths. Concerning the variant where deletions check if an element is already deleted, it appears from Figure 6(right) that for random deletions in inputs of size

larger than 10^4 this generates a non-negligible slowdown ("successor, 2-pass, checked" vs. "successor, 2-pass"). One could speculate that this is due to branch mispredictions and/or cache misses.

From Figure 6 it seems that combining the successor-delete data structure with the microset-macroset idea ("successor, 2-pass, microset") makes its performance on the same level as the union-find and weighted quick-find data structures combined with microsets ("union find, microset" and "quick find, microset", respectively) on the two types of sequences of operations we consider.

Figures 7 and 8 consider the same setup as in Figure 6, except that the number of queries is varied from $\frac{1}{8}n$ to $8n$ and are performed uniformly interleaved with the deletions. For incremental deletion sequences (Figure 7), it appears that the gap in performance between the successor-delete data structure and the theoretically more efficient data structures is maximized when there is one query per deletion. For random deletion sequences (Figure 8), the gap between the performance of using deletions with and without checking if an element has already been deleted appears consistent independently of the number of queries.

7   Conclusion

We have considered a simple data structure for supporting successor queries in a set of integers in the range $\{1, 2 \ldots, n\}$, and an implementation shows that the data structure has low constants in practice, although there exist data structures with lower asymptotic worst-case behavior. The experiments show that the running time of the proposed algorithm is in the same ballpark as weighted quick-find and union-find with path-compression and linking by weight. The benefit of the proposed algorithm is that it only needs to store a single array of integers, opposed to three arrays for the two other structures. The code for the proposed data structure operations is very short, but this is also the case for the union-find and weighted quick-find data structures, so all data structures have a low constant in their running time. An open problem is to analyze the theoretical performance of the data structure when using path halving.

References

1   Georgy M. Adelson-Velsky and Evgenii M. Landis. An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences (in Russian)*, 146:263–266, 1962.
2   Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
3   Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, 65(1):38–72, 2002. `doi:10.1006/JCSS.2002.1822`.
4   Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
5   Patrick Dinklage, Johannes Fischer, and Alexander Herlez. Engineering predecessor data structures for dynamic integer sets. In David Coudert and Emanuele Natale, editors, *19th International Symposium on Experimental Algorithms, SEA 2021, June 7-9, 2021, Nice, France*, volume 190 of *LIPIcs*, pages 7:1–7:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPICS.SEA.2021.7`.
6   Michael J. Fischer. Efficiency of equivalence algorithms. In Raymond E. Miller and James W. Thatcher, editors, *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, The IBM Research Symposia Series, pages 153–167. Plenum Press, New York, 1972. `doi:10.1007/978-1-4684-2001-2_14`.

7   Gianni Franceschini and Roberto Grossi. Optimal worst-case operations for implicit cache-oblivious search trees. In Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Michiel H. M. Smid, editors, *Algorithms and Data Structures, 8th International Workshop, WADS 2003, Ottawa, Ontario, Canada, July 30 - August 1, 2003, Proceedings*, volume 2748 of *Lecture Notes in Computer Science*, pages 114–126. Springer, 2003. `doi:10.1007/978-3-540-45078-8\_11`.

8   Gianni Franceschini and Roberto Grossi. Optimal implicit dictionaries over unbounded universes. *Theory of Computing Systems*, 39(2):321–345, 2006. `doi:10.1007/S00224-005-1167-9`.

9   Harold N. Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985. `doi:10.1016/0022-0000(85)90014-5`.

10   Bernard A. Galler and Michael J. Fischer. An improved equivalence algorithm. *Communications of the ACM*, 7(5):301–303, 1964. `doi:10.1145/364099.364331`.

11   Ashish Goel, Sanjeev Khanna, Daniel H. Larkin, and Robert Endre Tarjan. Disjoint set union with randomized linking. In Chandra Chekuri, editor, *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 1005–1017. SIAM, 2014. `doi:10.1137/1.9781611973402.75`.

12   Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978*, pages 8–21. IEEE Computer Society, 1978. `doi:10.1109/SFCS.1978.3`.

13   Giuseppe F. Italiano and Rajeev Raman. Topics in data structures. In Mikhail J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, Chapman & Hall/CRC Applied Algorithms and Data Structures series. CRC Press, 1999. `doi:10.1201/9781420049503-C6`.

14   Tianxiao Li, Jingxun Liang, Huacheng Yu, and Renfei Zhou. Dynamic "Succincter". In *64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023, Santa Cruz, CA, USA, November 6-9, 2023*, pages 1715–1733. IEEE, 2023. `doi:10.1109/FOCS57990.2023.00104`.

15   J. Ian Munro. An implicit data structure supporting insertion, deletion, and search in $O(\log^2 n)$ time. *Journal of Computer and System Sciences*, 33(1):66–74, 1986. `doi:10.1016/0022-0000(86)90043-7`.

16   Mihai Pătraşcu and Mikkel Thorup. Time-space trade-offs for predecessor search. In Jon M. Kleinberg, editor, *Proceedings of the 38th Annual ACM Symposium on Theory of Computing, Seattle, WA, USA, May 21-23, 2006*, pages 232–240. ACM, 2006. `doi:10.1145/1132516.1132551`.

17   Mihai Pătraşcu and Mikkel Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 166–175. IEEE Computer Society, 2014. `doi:10.1109/FOCS.2014.26`.

18   Md. Mostofa Ali Patwary, Jean R. S. Blair, and Fredrik Manne. Experiments on union-find algorithms for the disjoint-set data structure. In Paola Festa, editor, *Experimental Algorithms, 9th International Symposium, SEA 2010, Ischia Island, Naples, Italy, May 20-22, 2010. Proceedings*, volume 6049 of *Lecture Notes in Computer Science*, pages 411–423. Springer, 2010. `doi:10.1007/978-3-642-13193-6_35`.

19   Giulio Ermanno Pibiri and Rossano Venturini. Dynamic elias-fano representation. In Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter, editors, *28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017, July 4-6, 2017, Warsaw, Poland*, volume 78 of *LIPIcs*, pages 30:1–30:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. `doi:10.4230/LIPICS.CPM.2017.30`.

20   Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985. `doi:10.1145/3828.3835`.

21   Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975. `doi:10.1145/321879.321884`.

22   Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985. `doi:10.1137/0606031`.

23   Robert Endre Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–281, 1984. `doi:10.1145/62.2160`.

24   Theodorus Petrus van der Weide. *Datastructures: An Axiomatic Approach and the Use of Binomial Trees in Developing and Analyzing Algorithms*. PhD thesis, Mathematisch Centrum, Amsterdam, 1980. URL: `https://www.cs.ru.nl/Th.P.vanderWeide/docs/1980-Weide-AxAppr.pdf`.

25   Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977. `doi:10.1016/0020-0190(77)90031-X`.

26   Peter van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977. `doi:10.1007/BF01683268`.

27   Jan van Leeuwen and Theo van der Weide. Alternative path compression techniques. Technical Report RUU-CS-77-3, Department of Computer Science, University of Utrecht, The Neterlands, 1977. URL: `https://dspace.library.uu.nl/handle/1874/15885`.

28   J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.
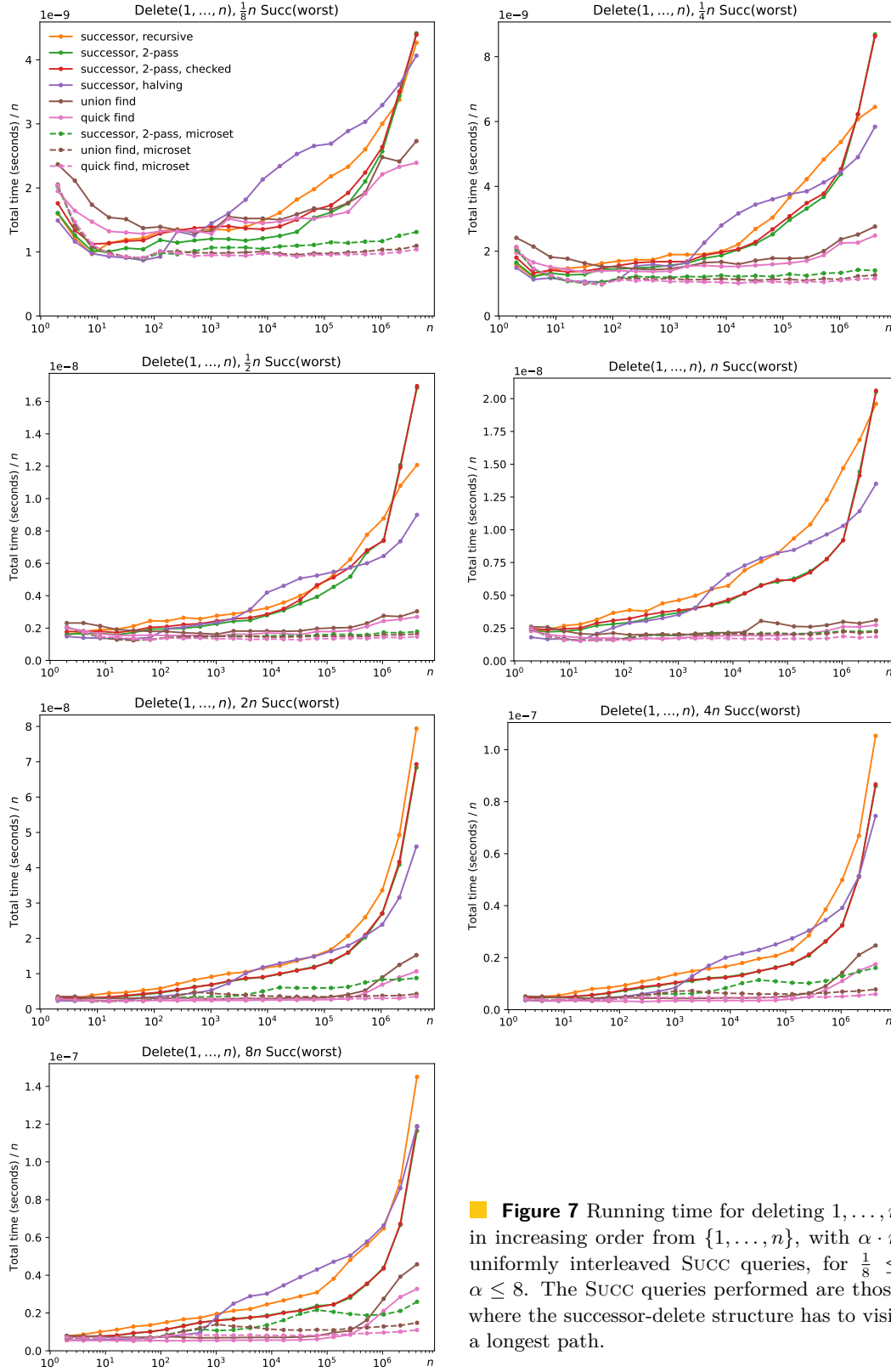
Figure 7 Running time for deleting $1, \ldots, n$ in increasing order from $\{1, \ldots, n\}$, with $\alpha \cdot n$ uniformly interleaved Succ queries, for $\frac{1}{8} \leq \alpha \leq 8$. The Succ queries performed are those where the successor-delete structure has to visit a longest path.
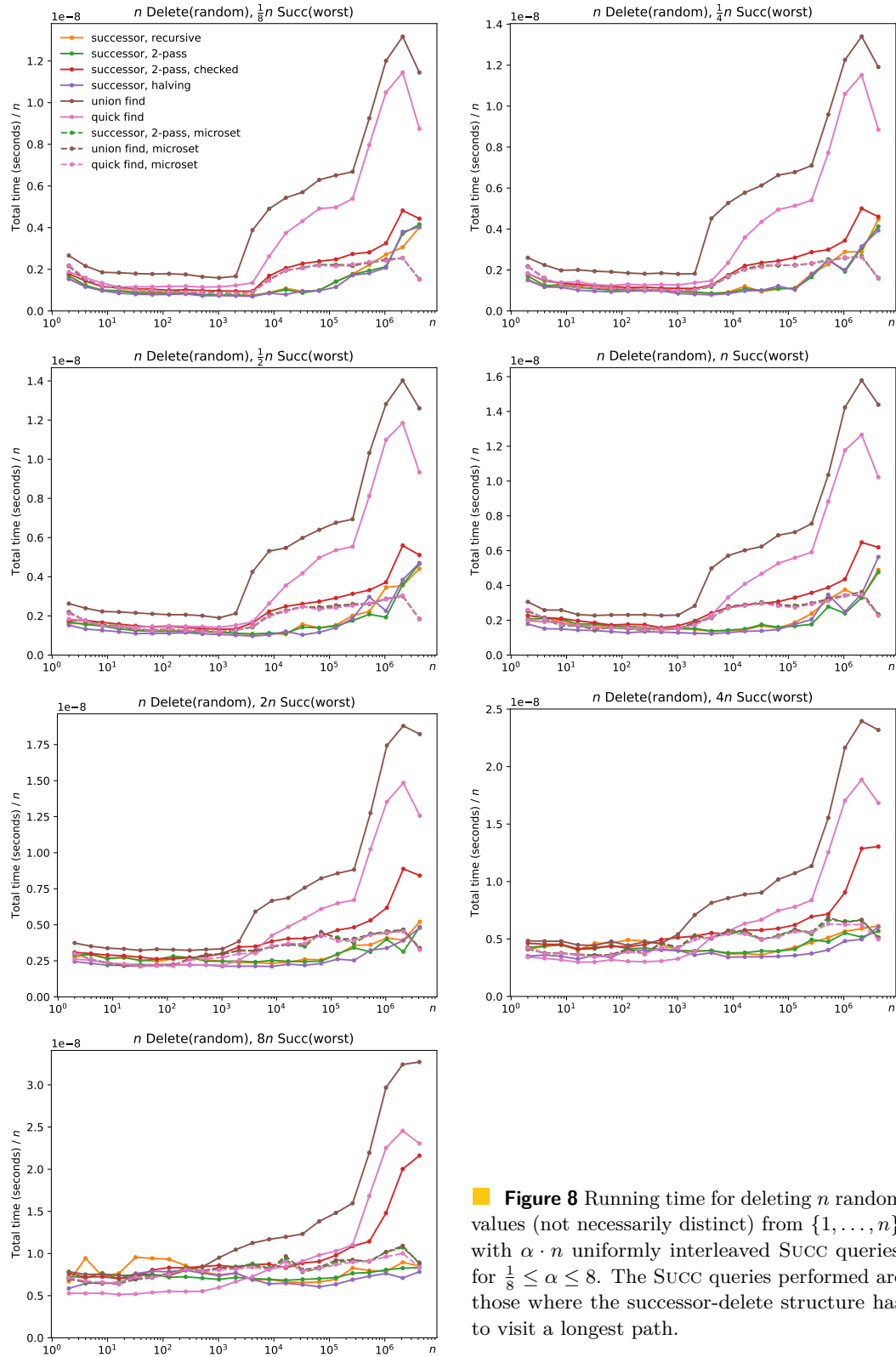
Figure 8 Running time for deleting n random values (not necessarily distinct) from $\{1, \ldots, n\}$, with $\alpha \cdot n$ uniformly interleaved Succ queries, for $\frac{1}{8} \leq \alpha \leq 8$. The Succ queries performed are those where the successor-delete structure has to visit a longest path.