van Emde Boas trees.

Consider the datatype *ordered dictionary.* In addition to the usual dictionary operations (insert, delete, lookup) it supports operations for finding the immediately larger entry and possibly also the immediately smaller entry compared to a given element (findnext, findprevious). Of course, the extra operations require keys to be taken from an ordered domain.

In a standard binary search tree implementation of an ordered dictionary some operations take time $O(\log n)$. van Emde Boas observed that if keys are restricted to the set $\{1, 2, 3, ..., n\}$ then all operations can be done in time $O(\log \log n)$ and space $O(n)$ using a special data structure that has become known under the name of *van Emde Boas trees* [3, 2].

This note describes Emde Boas trees with an application, namely the implementation of Union-Split-Find on intervals. Other applications are given in the problem set.

We start by considering a restricted datatype defining only three operations:

- Name: Tree(n)
- State: T is subset of $\{1, 2, 3, \ldots, n\}$.
- Operations:

  `insert`(x) Add x to T.

  `delete`(x) Remove x from T.

  `findsucc`(x) Return the smallest element in T that is $\geq x$.

Consider possible implementations. If the set is represented as a bit array, `insert` and `delete` take time $O(1)$, but `findsucc` is very inefficient; in the worst case, it may take time $O(n)$. Using red-black trees allows a time bound of $O(\log n)$ for all operations.

Something better is possible, if we exploit that the underlying key universe is the set $\{1, 2, 3, \ldots, n\}$. We can make a recursive representation, where Tree(n) distribute elements over $\sqrt{n}$ distinct versions of Tree($\sqrt{n}$). The latter trees are named bottom[i] for $i = 0, 1, 2, \ldots, \sqrt{n}-1$. An element $i = a\sqrt{n}+b$ is represented by letting bottom[a] have the entry b.

Note that the translation from i to a, b (and conversely) is simple and expedient, since a and b is the most significant half of bits in the binary representation of i, respectively the least significant half of bits ($\sqrt{n}$ is rounded to a power of 2).

Tree(n) uses an additional version of Tree($\sqrt{n}$) called top. top contains a if and only if bottom[a] is nonempty. It is now possible to implement an operation of Tree(n) in constant time plus the time for calling a single operation in Tree($\sqrt{n}$) (implying $O(\log \log n)$ time per operation).

This method may be viewed as binary search on the path from the root to a leaf in a balanced binary tree. Initially, we spend constant time deciding whether to call a top operation (upper half of path) or a bottom operation (lower half of path). This is followed by the recursive call.

In this way, `findsucc` can be done in time $O(\log \log n)$. It requires more sophistication to obtain the same bound for `delete` and `insert`, but a partial implementation is sketched in datastructure 1.

Data structure 1 Tree(n) in version 1

```
State: $T$ is a subset of $\{1, 2, 3, ..., n\}$ represented by
  size: integer (number of elements in $T$)
  max,min: integer (largest and smallest element of $T$)
  top: Tree($\sqrt{n}$)
  bottom: array [1..$\sqrt{n}$] of Tree($\sqrt{n}$)
Operation findsucc($i$) $\{i = a\sqrt{n} + b\}$
  if bottom[$a$].max $\geq b$ $\rightarrow$ $j$ := $a\sqrt{n}$+ bottom[$a$].findsucc($b$)
  [] else $\rightarrow$ $c$ := top.findsucc($a + 1$); $j$ := $c\sqrt{n}$+ bottom[$c$].min
  fi
  return $j$;
Operation insert($i$) $\{i = a\sqrt{n} + b\}$
  if bottom[$a$].size $= 0$ $\rightarrow$ top.insert($a$) fi
  bottom[$a$].insert($b$);
  update size,max,min
Operation delete($i$) $\{i = a\sqrt{n} + b\}$
  bottom[$a$].delete($b$);
  if bottom[$a$].size $= 0$ $\rightarrow$ top.delete($a$) fi
  update size,max,min
```

Note that max,min may be updated in time $O(1)$ when using updated values of max,min from top and bottom.

In the specific case when we insert the first element in a bottom tree or delete the last element from a bottom tree, we need in addition to insert, respectively remove, an element from the top tree. This makes a worst case recurrence for time usage of $T(n) = 2T(\sqrt{n})+O(1)$ with the solution $T(n) = O(\log n)$. To obtain a better result, we must modify the implementation to allow insertion of the first element into an empty tree in constant time and to allow deletion of the last element from a tree in constant time, i.e. we should avoid recursive calls. It is possible to modify the implementation in the suggested manner and still make it work. In every nonempty tree we keep one element (say the minimum) out of the recursive structure. The details are shown in datastructure 2, and it leads to the recurrence $T(n) = T(\sqrt{n})+O(1)$ that has the solution $T(n) = O(\log\log n)$.

Data structure 2 Tree(n) in version 2

```
State: Identical to version 1
Operation findsucc($i$)
  identical to findsucc from version 1 with the following statement added
  just before the return statement:
  if $i \leq$ min $< j$ → $j$ := min fi
Operation insert($i$)
  if size $= 0$ → update size,max,min
  [] size $> 0$ →
    if min $> i$ → swap(min,$i$) fi
    insert($i$) from version 1
  fi
Operation delete($i$)
  if size $\leq 1$ → update size,max,min
  [] size $\geq 2$ →
    if min $= i$ → $i$ := min := top.min$\cdot\sqrt{n}$+bottom[top.min].min fi
    delete($i$) from version 1
  fi
```

Note that we might also have kept max out of the recursive structure by a straightforward generalisation of data structure 2.

The implementation is described recursively, but for small n the set should be represented directly in a bitstring or a red-black tree.

In practice we start with a welldefined tree structure that is either empty or completely full. Either possibility can be handled in time $O(1)$ using *lazy initialization.*

We have restricted the description to the operations `insert`, `delete` and `findsucc`, but in great detail. These operations are fundamental, and it is easy to implement additional operations such as `deletemin` being equal to `delete(findsucc(1))` and `decreasekey` that may be implemented by combining `delete` and `insert`.

This ordered dictionary has several applications (see the problem set). We describe one of them here, namely Union-Find-Split on intervals.

Union-Split-Find on intervals.

The usual union-find datastructure is very efficient for representing a partitioning of a set into classes with operations for uniting two classes and for getting the name of the class containing some given element. If one wants an extra operation for splitting a class into two classes, it is necessary to define more precisely what such a split operation should do. A class with k elements may be split in $2^k - 2$ different ways. For a survey of different union-split-find problems and known solutions, see [1].

We present here a very efficient solution (time $O(\log \log n)$ per operation) for a variant of union-split-find on intervals, ie. we consider the following datatype:

Union-Split-Find:

- Name: Interval(n)
- State: T is a partitioning of $\{1, 2, 3, \ldots, n\}$ into intervals.
- Operations:

  `find`(x) returns the name of the interval containing x.

  `union`(x): Unites the interval containing x with the immediately following interval.

  `split`(x): The interval I containing x is split into two intervals $I \cap [1; x]$ and $I \cap [x + 1; n]$.

If an interval is represented by its rightmost element, then the partitioning $[1; x_1], [x_1 + 1; x_2], \ldots, [x_k + 1; n]$ can be represented uniquely by the set $S = \{x_1, x_2, ..., x_k, n\}$. Operation `find`($x$) must return the smallest element in S that is greater than (or equal to) x, while operations `union` and `split` removes (respectively inserts) an element from (into) S. Implementation of union-split-find on intervals using van Emde Boas trees is straightforward using this strategy.

Note that the representation is reversible: We may implement an ordered dictionary by means of a union-split-find data structure. The two datastructures are in fact equivalent.

References

[1] Zvi Galil and Giuseppe F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys (CSUR)* **23**(3) (1991), 319–344.

[2] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inform. Process. Lett.* **6**(3) (1977), 80–82.

[3] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Systems Theory* **10**(2) (1976/77), 99–127. Sixteenth Annual Symposium on Foundations of Computer Science (Berkeley, Calif., 1975), selected papers.