1 Ideas for Introduction and Motivation

- Cite splay trees, Iacono, and Bose-Douïeb-Langerman SODA 2008.
- Working set is important because real applications exhibit significant locality of reference; it implies static optimality; the best general-purpose compression algorithms (bzip2) is based on an encoding with the working-set property [what's the running-time?].
- Working-set with leading constant C gives:
  - Static optimality with leading constant C
  - Compression algorithm with cost $m(CH + o(H))$ that runs in time $O(mH)$.
- Unfortunately, splay trees and Iacono's structure are impractical because of the large constants. For example, when $n = 10^6$, binary search takes 20 comparisons. Splay trees have the leading constant 4, so they only beat binary search when the working set number is less than 32. Out of the million elements, there are only 32 that can be accessed faster using a splay tree than using binary search. This only counts the cost of comparisons, and doesn't even account for the rotations done during a splay operation.
- Iacono's trick of grouping into groups with working set numbers that are increasing doubly-exponentially has been used repeatedly and is the basis of X, Y, Z, ... (Even Bose-Douïeb-Langerman splits the skiplist into layers defined this way.) A new idea is needed.

  Our structure is a significant departure from this method. It uses a sequence of interlinked structures $D_0, \ldots, D_k$ whose sizes increase sub-exponentially.

2 The Structure

We store a totally-ordered set, S, of size n in a sequence of sorted lists $D_0, \ldots, D_k$. For non-negative integers, i, let

$$n_i = \begin{cases} 1 & \text{if } i = 0 \\ 2^i / f(i) & \text{otherwise} \end{cases}$$

for some function $f(i)$ to be defined later. These lists have the following properties:

1. If $w(x) \le n_i$, then $x \in D_i$, for all $x \in S$ and all $i \in \{0, \ldots, k\}$.
2. $D_i \subseteq D_{i+1}$, for all $i \in \{0, \ldots, k-1\}$.
3. A list node that contains a value x in D_i contains a pointer to the list node that contains x in D_{i+1}.
4. For each $i \in \{0, \ldots, k\}$, if x and y are two consecutive values in D_i, then at least one of x and y appears in D_{i-1}.

5. $|D_0| \in O(1)$.

In addition, the data structure stores a sequence of queues, $Q_0,\ldots,Q_k$, where Q_i stores exactly the elements $x \in S$ such that $w(x) \le n_i$. Within Q_i, elements are ordered by working-set number so that the jth element in Q_i has working set number j. Cross pointers are used to link the elements in Q_i with the corresponding elements in D_i. (Or, more efficiently, each *node* contains four pointers: two for the doubly-linked list D_i, one to link the node to the corresponding node in D_{i+1}, and one to allow the node to take part in the queue, Q_i.)

Lemma 1. *For any one-to-one assignment of working set numbers $w : S \to \{1,\ldots,n\}$, a data structure satisfying Properties 1–5 exists.*

Proof. Properties 1 and 2 are satisfied by including, in each D_i, the n_i elements with working-set number at most n_i. Properties 3 and 4 can be satisfied by adding at most half the elements from D_k into D_{k-1}, then adding at most half the element from D_{k-1} into D_{k-2}, and so on. We call this last step in the construction, the *cascading step*.

All that remains is to show that Property 5 is satisfied. In particular, we need to show that the cascading step does not increase the size of D_0 by more than a constant. From the definition of the cascading phase, we have:

$$\begin{aligned}|D_0| &\le n_0 + \sum_{i=1}^{k} n_i/2^i \\ &= 1 + \sum_{i=1}^{k} 1/f(i) \\ &< 1 + \sum_{i=1}^{\infty} 1/f(i) \\ &= 1 + O(1)\ . \qquad \square\end{aligned}$$

provided that $f(i) \in \Omega(i(logi)^{1+\epsilon})$ for some constant $\epsilon > 0$ (for example $f(i) = i^2$, works well.

A search for a value, x, in this structure proceeds as follows:

```
1:  $i \leftarrow 0$
2:  locate the successor, $x'$, of $x$ in $D_0$
3:  **while** $x' \neq x$ **do**
4:      $i \leftarrow i + 1$
5:      locate $x'$ in $D_i$
6:      **if** $x \le$ predecessor of $x'$ in $D_i$ **then**
7:          $x' \leftarrow$ predecessor of $x'$ in $D_i$
8:      {now $x'$ is the successor of $x$ in $D_i$}
9:      add $x$ to $D_j$ for each $j \in \{0,\ldots,i-1\}$
10:     if $|D_0| > 10$, then rebalance
```

Notes:

1. The choice of n_i ensures that x is found by the time the algorithm reaches level
$$i(x) = (1 + o(1))\log w(x) = \log w(x) + o(\log w(x))$$

2. In the ternary comparison model, the whole algorithm does $\log w(x) + o(\log w(x))$ comparisons

3. In the binary comparison model, we can skip the equality comparison in the condition of the while loop except when i is a perfect square. This only increases the number of comparison by $O(\sqrt{\log w(x)})$.

4. The search operation eventually leads to a violation of Property 5. We still have to show that Property 5 can be maintained through judicious use of partial rebuilding.

   We can get this to work if we take $f(i) = 1/(2 - \epsilon/2)^i$, so that $n_i = (2 - \epsilon)^i$. When we rebuild we start the rebuild at the first level in which $|D_i| \le (2 - \epsilon/2)^i$. Unfortunately, this choice of n_i only yields a query algorithm that does
$$\log_{2-\epsilon} w(x) \le (1 + \epsilon/e)\log w(x) + O(\log\log w(x))$$
   comparisons.

   A potential way around this is to take $f(i) = i^2$, rebuild at the first level i where $|D_i| \le 2^i$, but have special levels that only contain 1/3 of the elements below them. When we rebuild at level i, level $i - 1$ becomes a special level. The trick then is to ensure that among the first i levels, the number of special levels is $o(i)$. This approach works provided that we can answer the following combinatorial question in the affirmative:

   Question: For an increasing function $g(i)$, let $B(k,g)$ denote the set of all bitstrings, $b_0,\dots,b_k$, such that the i-prefix $b_0,\dots,b_i$ contains at most $i/g(i)$ one bits, for all $i \in \{1,\dots,k\}$. Does there exists a function $g(i) \in o(i) \cap \omega(1)$ such that $|B(k,g)| \ge 2^{i-o(i)}$?

   Update: The answer is no, even if the requirement is weakened to $b_0,\dots,b_k$ contains at most $k/g(k)$ one bits. To see this, recall that, for $x \le k/2$, the number of bitstrings of length k that have at most x one bits is
$$\sum_{i=0}^{x}\binom{k}{i} \le \binom{k}{x}\left(\frac{k-x+1}{k-2x+1}\right)$$
   [`http://goo.gl/GOX1p`]. Also
$$\binom{k}{x} \le \left(\frac{ke}{x}\right)^x \quad .$$

Substituting $x = k/g(k)$, we get

$$\begin{aligned}
\sum_{i=0}^{k/g(k)} \binom{k}{i} &\leq \binom{k}{k/g(k)}(1+o(1)) \\
&\leq \left(\frac{ke}{k/g(k)}\right)^{k/g(k)} (1+o(1)) \\
&= (e \cdot g(k))^{k/g(k)}(1+o(1)) \\
&= e^{k/g(k)+k\ln g(k)/g(k)}(1+o(1)) \\
&= e^{o(k)}(1+o(1)) \\
&\notin 2^{k-o(k)} \quad .
\end{aligned}$$

5. **Next idea:** micro-trees so that rebuilding a level of size 2^i can be done in $O(2^i/f(n))$ time for some slow-growing function $f(n)$ like $f(n) = c\log n$.