Treesets: Efficient Finite Sets with Unique Representations

Grant Jurgensen

Kestrel Institute

November 16, 2026



Outline



1. Motivation
2. Hash-Ordered Treaps
3. Implementation and Verification
4. The Library
5. Iterators
6. Evaluation
7. Conclusion

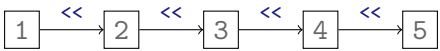
Motivation



Unique Representations



- Reasoning with `equal` is far more convenient than reasoning with weaker equivalence relations.
- So we want *unique representations*.
- `osets` (Davis, 2004): a set is a list strictly ordered by `<<`.
  - Set equality is just `equal`.
  - But `in`, `insert`, and `delete` take linear time.



Uniqueness vs. Balance



- Binary search trees (BSTs) are the obvious refinement of ordered lists.
- But common self-balancing BSTs (e.g. red-black, AVL, etc.) are *history-dependent*: their shape depends on the order of updates.
- Sundar & Tarjan (1994): keeping a unique BST at logarithmic height requires a linear number of rotations per update, in the worst case.
- **Treaps**: unique BSTs with *expected* logarithmic node depth.
  - The lower bound still holds, but the worst case becomes highly improbable.

Hash-Ordered Treaps



Hash-Ordered Treaps



- A treap (tree + heap) is a BST under one order and a max-heap under another.
- BST order: `<<`. Heap order: `heap<`, comparing hashes (ties broken by `<<`).

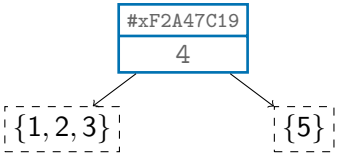
`#x2E90FB43`	`#xB6D3018E`	`#x51C7A06B`	`#xF2A47C19`	`#x7C4A22D5`
1	2	3	4	5

The set $\{1, \ldots, 5\}$ with illustrative hashes. `4` has the largest hash, so it must be the root.

Hash-Ordered Treaps



- A treap (tree + heap) is a BST under one order and a max-heap under another.
- BST order: <<. Heap order: `heap<`, comparing hashes (ties broken by <<).

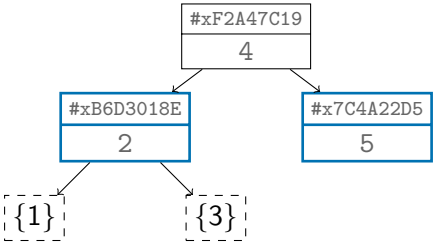


Elements << 4 go left; the rest go right.

Hash-Ordered Treaps



- A treap (tree + heap) is a BST under one order and a max-heap under another.
- BST order: `<<`. Heap order: `heap<`, comparing hashes (ties broken by `<<`).

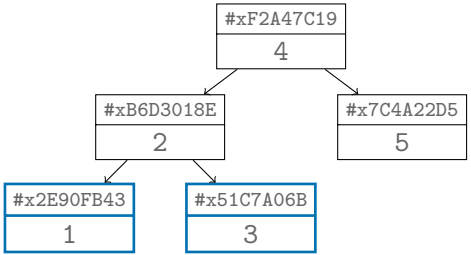


Recurse on each side: `2` has the largest hash of $\{1, 2, 3\}$, and `5` is alone.

Hash-Ordered Treaps



- A treap (tree + heap) is a BST under one order and a max-heap under another.
- BST order: `<<`. Heap order: `heap<`, comparing hashes (ties broken by `<<`).



Recurse once more. Inserting in any order gives this tree.

Implementation and Verification



The Hash Function



- The Jenkins one-at-a-time hash (`jenkins-bytes`) over a byte list.
- `to-bytes` serializes any ACL2 object: a tag byte per type, length-prefixed strings, LEB128 naturals, zigzag integers.
- In execution, the two are fused into a single walk.
- `to-bytes` is prefix-free, hence injective. Collisions can only come from Jenkins (or bad atoms).
- No proof depends on Jenkins; any 32-bit hash could be swapped in.

```
(define jenkins (x)
  :returns (hash (unsigned-byte-p 32 hash))
  (mbe :logic (jenkins-bytes (to-bytes x))
       :exec (jenkins-finalize
              (jenkins-acc x 0))))
```

```
(defrule to-bytes-injective
  (implies (and (no-bad-atoms-p x)
                (no-bad-atoms-p y))
           (equal (equal (to-bytes x)
                         (to-bytes y))
                  (equal x y))))
```

Trees and Invariants



- A node holds a tree element (a value and its cached hash), plus left and right children. Leaves are `nil`.
  - Standard recognizers, constructors, accessors, and fixers: `treep`, `tree-node`, `tree->left`, `tree-element->val`, `tree-fix`, etc.

```
(define bstp ((tree treep))
  (or (tree-empty-p tree)
      (mbe :logic (and (bstp (tree->left tree))
                       (bstp (tree->right tree))
                       (<<-all-l (tree->left tree) (tree-element->val (tree->head tree)))
                       (<<-all-r (tree-element->val (tree->head tree)) (tree->right tree)))
           :exec (mv-let (bstp min max) (fast-bstp-nonempty tree)
                   (declare (ignore min max))
                   bstp))))
```

- `heapp` has the same shape, but both children must be entirely `heap<` the head.

Treap Operations



- The core set operations (membership, insertion, union, subset, etc.) are implemented and verified on treaps.
- For example, union: the head that is larger under `heap<` becomes the root. Split the other treap around it, and union the halves with the corresponding children.

```
(define tree-union ((x treep) (y treep))
  :returns (tree treep)
  (cond ((tree-empty-p x) (tree-fix y))
        ((tree-empty-p y) (tree-fix x))
        ;; `heap<` on the elements' values, reusing their cached hashes
        ((heap<-with-tree-element (tree->head x) (tree->head y))
         (mv-let (in left right)
                 (tree-split (tree-element->val (tree->head y)) x)
           (declare (ignore in))
           (tree-node (tree->head y)
                      (tree-union left (tree->left y))
                      (tree-union right (tree->right y)))))
        (t ...))) ;; Symmetric, with x and y swapped
```

Verifying Union



```
(defrule bstp-of-tree-union-when-bstp
  (implies (and (bstp x) (bstp y))
           (bstp (tree-union x y))))

(defrule heapp-of-tree-union-when-heapp
  (implies (and (heapp x) (heapp y))
           (heapp (tree-union x y))))

(defrule tree-in-of-tree-union
  (equal (tree-in a (tree-union x y))
         (or (tree-in a x)
             (tree-in a y))))
```

- For every treap operation, we prove that both invariants are preserved, and characterize membership in the result.
- Each proof is by the function's own induction. Only the two mirror-image split cases are interesting:
  - `bstp`: `tree-split`'s halves lie strictly below and above the splitter.
  - `heapp`: we always pick the larger head.
  - Membership: `tree-split` partitions the treap.

Unique Representation



- Uniqueness is equivalent to subset antisymmetry for well-formed treaps:

```
(defrule tree-subset-p-antisymmetry-weak
  (implies (and (treep x) (treep y)
                (bstp x) (bstp y)
                (heapp x) (heapp y)
                (tree-subset-p x y)
                (tree-subset-p y x))
           (equal x y)))
```

- By simultaneous induction on `x` and `y`.
  - **Heads are equal**: the larger head would also be in the other treap, exceeding its head and contradicting `heapp`.
  - **Children are equal**: with equal heads, `bstp` makes the left children mutual subsets, and likewise the right. Apply the induction hypothesis.
- The library builds its extensionality rule on this.

The Library



The Public Interface



- The tree is never exposed: executable counterparts are disabled and type prescriptions suppressed. Users reason only about the set primitives.
- All `osets` primitives, except an efficient `tail`.
- No left/right subset accessors: those subsets would be arbitrary. Traverse with iterators instead.
- `min` is slower than `set::head`, but `max` is faster.

Operation	`osets`	`treeset`
`in`	$O(kn)$	$O(k \log(n))$
`insert`	$O(kn)$	$O(k \log(n))$
`delete`	$O(kn)$	$O(k \log(n))$
`min`	$O(1)$	$O(\log(n))$
`max`	$O(n)^{\dagger}$	$O(\log(n))$
`union`, etc.	$O(k(n+m))$	$O(km \log(n/m+1))$
`cardinality`	$O(n)$	$O(n)$

k: element size. n, m: set sizes, $m \leq n$.
`treeset` bounds are expected.
†`osets` has no `max`; this is what it would cost.

Extensionality



- Set equality reduces to membership of an arbitrary element, via one rewrite rule:

```
(defruled extensionality
  (implies (and (setp x) (setp y))
           (equal (equal x y)
                  (let ((arbitrary-elem (ext-equal-witness x y)))
                    (equal (in arbitrary-elem x)
                           (in arbitrary-elem y))))))
```

```
(define-sk ext-equal (x y)
  (forall (elem)
    (equal (in elem x)
           (in elem y)))
  :skolem-name ext-equal-witness)
```

- `ext-equal-witness` picks an element on which `x` and `y` disagree, if any, so it serves as the arbitrary element.
- `osets` gets this from double containment plus pick-a-point; no subset statements are needed here.

Isomorphism to osets



- to-oset and from-oset are inverses (up to the fixers).
- Rules like to-oset-of-diff push the conversions through each operation. These are collected in to-oset-theory and from-oset-theory.
- So theorems are trivial to port from osets.

```
(defrule to-oset-of-diff
  (equal (to-oset (diff x y))
         (set::difference (to-oset x)
                          (to-oset y))))
```

```
(defrule cardinality-of-diff
  (equal (cardinality (diff x y))
         (- (cardinality x)
            (cardinality (intersect x y))))
  :enable to-oset-theory
  :disable from-oset-theory)
```

Treemaps



- Nodes hold key-value pairs; the invariants constrain only the keys.
- `lookup` returns just the value (or a default), not an `assoc`-style pair.
- Normalize on the key set: a key is bound when it is in `(keys map)`, which reuses the `treeset` theory.
- As with `treeset`s: an isomorphism to `omaps`, and extensionality.

```
(define-sk ext-equal (x y)
  (forall (key default)
    (equal (lookup key x :default default)
           (lookup key y :default default)))
  :skolem-name ext-equal-witness)
```

- Extensionality must quantify over the default too. Otherwise an unbound key looks like a key bound to the default.

Iterators

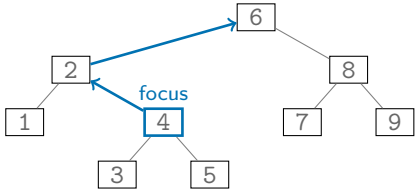


Traversing a Treeset

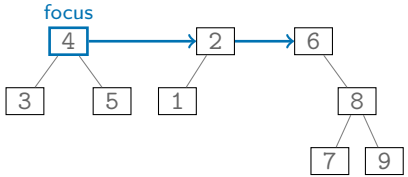


- `osets` are traversed with `set::head` and `set::tail`. For `treeset`s, the alternatives fall short:
  - Tree accessors (head, left, right) leak the representation, and users would have to rebuild an in-order traversal.
  - Repeatedly taking the `min` and deleting it is $O(kn \log(n))$, not $O(n)$.
  - Converting with `to-oset` is linear, but pays for every element, even after an early exit.
- **Goal**: in-order traversal in $O(n)$, paying only for the elements visited, without breaking the abstraction.
- **Solution**: bidirectional iterators, based on Huet's zippers (1997).

Zippers



overlaid on the tree



pulled apart

- A zipper is a focused subtree plus the path back up to the root.
- Pulled apart, the focus is the effective root, and each frame of the path holds a fragment of the original tree.
- Moving the focus is one step: down pushes a frame onto the path; up pops one and plugs the focus back in.

Iterators



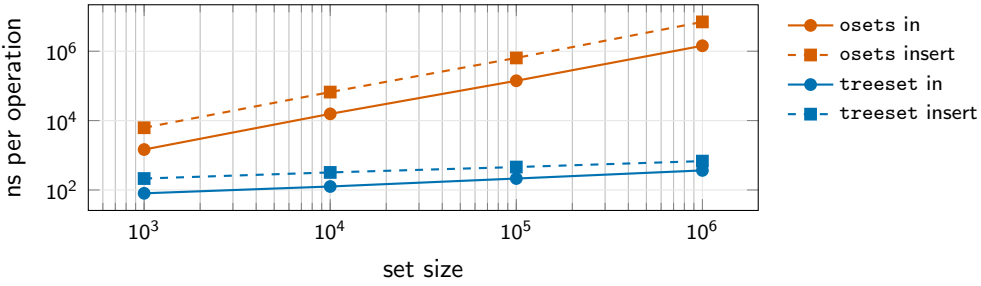
- An iterator is a zipper focused on an element, or a marker before the first or after the last element.
- `next` and `prev` take $O(1)$ amortized time. Counts of lefts and rights in the path detect the ends in constant time.
- Iterators are unique: equal exactly when they mark the same position in the same set.

`iter-min`, `iter-max`	start at the first/last element	`from-iter`	the whole set
`next`, `prev`	step forward/backward	`before`, `after`	the elements on either side
`value`	the current element	`has-valuep`	at an element?
`nexts`, `prevs`	elements remaining each way	`before-firstp`	before the first element?
		`after-lastp`	after the last element?

Evaluation



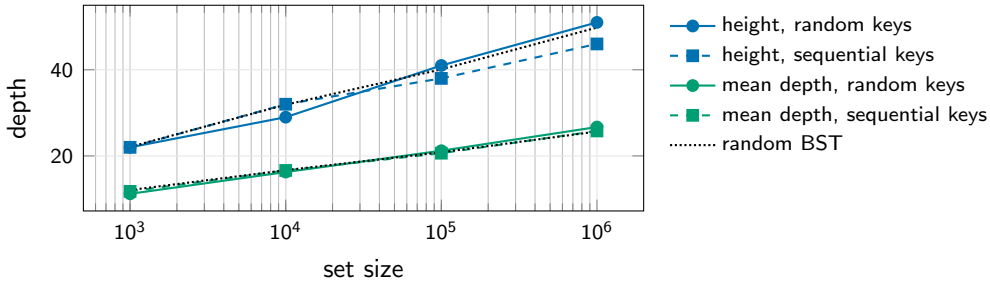
Performance



- Complexity isn't verified; we benchmark `in` (node depth) and `insert` (adds hashing).
- At 10^6 elements: `in` 366 ns vs. 1.44 ms (∼3,900×); `insert` 679 ns vs. 7.0 ms (∼10,000×).

Random unsigned `fixnum`s, half of the queries in the set; medians of 30 rounds; SBCL 2.5.8 on an i7-12700K.

Do Our Hashes Behave Like Random Priorities?



- Height and mean node depth track true random BSTs, even with sequential keys.
- Hashing is cheap: 11.2% of `insert` time (10,000 random `fixnum`s), and `insert` is the only operation that hashes.

Conclusion



Conclusion



- `treeset`: ordered finite sets with a unique representation and expected logarithmic search and update.
  - Verified core operations, and an isomorphism to `osets`.
  - Iterators for efficient traversal, without breaking the abstraction.
  - `treemap`: the same, for finite maps.
- Future work: more iterator operations (e.g. an `in` that returns an iterator), multisets as `treemap`s to positive numbers, and graphs as `treemap`s from nodes to `treeset`s of neighbors.
- In the community books: `kestrel/data/treeset` and `kestrel/data/treemap`.
- Thank you to the reviewers for their insightful comments.

Questions?

Backup: Balance



- Balance depends on how correlated the two orders are.
  - If larger elements always had larger hashes, the treap would skew entirely left.
- This is why the heap order compares hashes: they look random with respect to $<<$, but are deterministic.
- Assuming hashes behave like random priorities, each node has expected depth $O(\log(n))$ (Seidel & Aragon, 1996).
- Balance is likely, not guaranteed: one could search out elements whose hashes collide, or whose hash order agrees with $<<$.

Backup: Complexity



Operation	`osets`	`treeset`	Operation	`osets`	`treeset`
`setp`	$O(kn)$	$O(kn)$	`subset`	$O(k(n+m))$	$O(km\log(n/m+1))$
`emptyp`	$O(1)$	$O(1)$	`union`	$O(k(n+m))$	$O(km\log(n/m+1))$
`in`	$O(kn)$	$O(k\log(n))$	`intersect`	$O(k(n+m))$	$O(km\log(n/m+1))$
`insert`	$O(kn)$	$O(k\log(n))$	`diff`	$O(k(n+m))$	$O(km\log(n/m+1))$
`delete`	$O(kn)$	$O(k\log(n))$	`cardinality`	$O(n)$	$O(n)$
`min`	$O(1)$	$O(\log(n))$	`to-oset`	—	$O(n)$
`max`	$O(n)^\dagger$	$O(\log(n))$	`from-oset`	—	$O(kn)$

- k is the size of elements; n and m are set sizes, with $m \leq n$. `treeset` bounds are expected.
- The `osets` `min` is `set::head`. †`osets` has no `max`; this is what it would cost.
- `union`, `intersect`, and `diff` follow Blelloch & Reid-Miller (1998); `subset` follows the same recursion as `diff` without building a result.

Backup: Serialization



Type	Tag	Byte sequence
`cons`	`#x70`	The serialization of the `car` followed by the serialization of the `cdr`.
`symbol`	`#x71`	The untagged serialization of the package name string followed by the untagged serialization of the symbol name string.
`string`	`#x72`	The LEB128-encoded string length followed by the bytes corresponding to each character, in order.
`character`	`#x73`	The single byte obtained from `char-code`.
`integer`	`#x74`	The LEB128-encoded natural obtained by zigzag-encoding the integer.
`rational`	`#x75`	The untagged serialization of the numerator followed by the LEB128-encoded denominator.
`complex`	`#x76`	The untagged serialization of the real part followed by the untagged serialization of the imaginary part.
`bad-atom`	`#x77`	No bytes beyond the tag byte.

LEB128: 7 bits per byte, high bit set if more follow. Zigzag maps `0`, `-1`, `1`, `-2`, ... to `0`, `1`, `2`, `3`, ...

Backup: Antisymmetry Lemmas



```
(defrule tree->head-when-tree-subset-p-tree-subset-p
  (implies (and (tree-subset-p x y)
                (tree-subset-p y x)
                (heapp x) (heapp y))
           (equal (tree-element->val (tree->head x))
                  (tree-element->val (tree->head y)))))

(defrule tree-subset-p-of-children-when-same-head
  (implies (and (tree-subset-p x y)
                (bstp x) (bstp y)
                (equal (tree-element->val (tree->head x))
                       (tree-element->val (tree->head y))))
           (and (tree-subset-p (tree->left x) (tree->left y))
                (tree-subset-p (tree->right x) (tree->right y)))))
```

- The induction scheme, `tree-bi-induct`, recurs on the left children of both trees, and on the right children of both.

Backup: Iterator Operations



`iterp`	$O(kn)$	`before-firstp`	$O(1)$
`iter-min`	$O(\log(n))$	`after-lastp`	$O(1)$
`iter-max`	$O(\log(n))$	`has-valuep`	$O(1)$
`value`	$O(1)$	`before`	$O(\log(n))$
`next`	$O(1)$	`after`	$O(\log(n))$
`prev`	$O(1)$	`nexts`	$O(n)$
`from-iter`	$O(\log(n))$	`prevs`	$O(n)$

k: element size. n: set size. Bounds are expected. `next` and `prev` are amortized over single-threaded use; a single step may take $O(\log(n))$.

```
(defrule from-iter-of-next
  (equal (from-iter (next iter))
         (from-iter iter)))

(defrule before-of-next
  (implies (has-valuep iter)
           (equal (before (next iter))
                  (insert (value iter)
                          (before iter)))))
```

Symmetric theorems hold for `prev`.

Backup: Short-Circuiting Search



- Check whether any element is even (under `nfix`), stopping at the first one.

```
(define iter-any-evenp ((iter iterp))
  (and (has-valuep iter)
       (or (evenp (nfix (value iter)))
           (iter-any-evenp (next iter))))
  :measure (nexts iter))

(define any-evenp ((set setp))
  (mbe :logic (and (not (emptyp set))
                   (or (evenp (nfix (min set)))
                       (any-evenp (delete (min set) set))))
       :exec (iter-any-evenp (iter-min set)))
  :measure (cardinality set)
  :verify-guards nil ;; Verified below
  ///
  ...)
```

Backup: Connecting the Two



- `any-evenp`'s logic uses `min` and `delete`, but it executes with the iterator. Guard verification needs them to agree:

```
(define any-evenp ((set setp))
  ...
  ///
  (defrule iter-any-evenp-becomes-any-evenp
    (equal (iter-any-evenp iter)
           (and (has-valuep iter)
                (or (evenp (nfix (value iter)))
                    (any-evenp (after iter)))))
    :induct t
    :enable (iter-any-evenp any-evenp))

  (verify-guards any-evenp))
```

- The induction relies on:

```
(defrule after-of-next-when-has-valuep
  (implies (has-valuep iter)
           (equal (after (next iter))
                  (delete (min (after iter))
                          (after iter)))))
```

Backup: Treemap Operations



Operation	Complexity	Operation	Complexity
`emptyp`	$O(1)$	`mapp`	$O(kn)$
`in`	$O(k \log(n))$	`size`	$O(n)$
`lookup`	$O(k \log(n))$	`keys`	$O(n)$
`update`	$O(k \log(n))$	`values`	$O(kn \log(n))$
`delete`	$O(k \log(n))$	`submap`	$O(km \log(n/m + 1))$
`min`	$O(\log(n))$	`update*`	$O(km \log(n/m + 1))$
`max`	$O(\log(n))$	`restrict`	$O(km \log(n/m + 1))$
		`to-omap`	$O(n)$
		`from-omap`	$O(kn)$

k: key size. n, m: map sizes, $m \leq n$. Bounds are expected.

Backup: Related Work



- Unique representations in ACL2: `osets` led to `omaps` and `obags` (multisets). `misc/records` orders keys by `<<` and binds no key to `nil` (Kaufmann & Sumners, 2002).
- Set theories over unordered lists: Bevier's set equivalence, `std/lists`, and Moore (2001), who adds tags to distinguish sets from other lists.
- Treaps are verified in Isabelle/HOL, including uniqueness (Eberl, Haslbeck & Nipkow, 2020), without deletion.
- Blelloch & Golovin (2007): a treap ordered by a perfect hash is strongly history-independent.
- Appel & Leroy (2023): verified, unique binary tries, with keys that are positive integers; other key types need an injective encoding.