B-Treaps: A Uniquely Represented Alternative to B-Trees

Daniel Golovin[⋆]

Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA
dgolovin@cs.cmu.edu

Abstract. We present the first uniquely represented data structure for an external memory model of computation, a B-tree analogue called a *B-treap*. Uniquely represented data structures represent each logical state with a unique machine state. Such data structures are *strongly history-independent*; they reveal no information about the historical sequence of operations that led to the current logical state. For example, a uniquely represented file-system would support the deletion of a file in a way that, in a strong information-theoretic sense, provably removes all evidence that the file ever existed. Like the B-tree, the B-treap has depth $O(\log_B n)$, uses linear space with high probability, where B is the block transfer size of the external memory, and supports efficient one-dimensional range queries.

1 Introduction

Most computer applications store a significant amount of information that is hidden from the application interface—sometimes intentionally but more often not. This information might consist of data left behind in memory or disk, but can also consist of much more subtle variations in the state of a structure due to previous actions or the ordering of the actions. To address the concern of releasing historical and potentially private information various notions of *history independence* have been derived along with data structures that support these notions [10, 12, 9, 6, 1]. Roughly, a data structure is history independent if someone with complete access to the memory layout of the data structure (henceforth called the "observer") can learn no more information than a legitimate user accessing the data structure via its standard interface (e.g., what is visible on screen). The most stringent form of history independence, *strong history independence*, requires that the behavior of the data structure under its standard interface along with a sequence of randomly generated bits, which are revealed to the observer, uniquely determine its memory representation. We say that such structures are *uniquely represented*.

Unique representation had been studied even earlier [15, 16, 2], in the context of theoretical investigations into the need for redundancy in efficient data structures. The results were mostly negative, however they were for various comparison-based and pointer machine-based models of computation, and did not hold for the RAM model.

There has been much recent progress on efficient uniquely represented data structures in the RAM model. Blelloch and Golovin [4] described a uniquely represented hash table

⋆ Computer Science Department, Carnegie Mellon University, Pittsburgh PA 15213. This work was supported in part by Microsoft Corporation through the Center for Computational Thinking at Carnegie Mellon and by NSF ITR grant CCR-0122581 (The Aladdin Center).

for the RAM model supporting insertion, deletion and queries in expected constant time, using linear space and only $O(\log n)$ random bits. They also provided a perfect hashing scheme that allows for $O(1)$ worst-case queries, and efficient uniquely represented data structures for ordered dictionaries and the order maintenance problem. Naor *et al.* [11] developed a second uniquely represented dynamic perfect hash table supporting deletions, based on cuckoo hashing. Finally, Blelloch *et al.* [5] developed efficient uniquely represented data structures for some common data structures in computational geometry. Several other results, as well as a more comprehensive discussion of uniquely represented data structures, may be found in the author's doctoral thesis [8].

Recent progress on uniquely represented data structures for the RAM opens up the possibility of full-fledged uniquely represented (and thus strongly history independent) systems. Consider a filesystem that supports a delete operation that provably leaves no trace that a file ever existed on the system, or a database that reveals nothing about the previous updates or queries to the system. Existing uniquely represented data structures allow us to build efficient versions of such systems on a RAM, however these systems are best modeled not in the RAM model of computation but in *external memory* (EM) models of computation [17] which account for the fact that modern computers have a memory hierarchy in which external memory (e.g., a disk drive) is several orders of magnitude slower than internal memory (e.g., DRAM).

To the best of my knowledge, there is no previous work on uniquely represented data structures in an EM model of computation. For background on the extensive work on conventional data structures in EM models, we refer the interested reader to the excellent book by Vitter [17]. Within this body of work, *extendible hash tables* and the *B-tree* and its variants (e.g., the B^+-tree and the B^*-tree) play a prominent role.

It is worth noting here that the extendible hashing construction of Fagin *et al.* [7] is almost uniquely represented, and in fact can be made uniquely represented with some minor modifications, the most significant of which is to use a uniquely represented hash table [4, 11] to layout blocks on disk. However in this paper we focus on uniquely represented B-tree analogs, which can support efficient one-dimensional range queries.

The B-tree was invented by Bayer and McCreight [3], to organize information on magnetic disk drives so as to minimize disk I/O operations. The salient features of the B-tree are that it stores $\Theta(B)$ keys in each node and each node (other than the root) has degree $\Theta(B)$, where B is a parameter of the tree. Thus the B-tree has height roughly $\log_B(n)$ and $\Theta(n/B)$ nodes. We will construct a uniquely represented tree that is analogous to a B-tree, based on the *treap* data structure [14]. Recall a treap is a binary search tree where each key also has an associated priority. In addition to the standard search tree constraint on keys, the keys must also be in heap order with respect to their priorities; each key must have priority less than its parent. We call the resulting data structure a *B-treap*, for "bushy-treap." It supports the following operations.

- $\mathrm{insert}(x)$: insert key x into the B-treap.
- $\mathrm{delete}(x)$: delete key x from the B-treap.
- $\mathrm{lookup}(x)$: determine if x is present in the B-treap, and if so, return a pointer to it.
- $\text{range-query}(x, y)$: return all keys between x and y in the B-treap.

It is easy to associate auxiliary data with the keys, though for simplicity of exposition we will assume there is no auxiliary data being stored. We will prove the following result.

Theorem 1. *There exists a uniquely represented B-treap that stores elements of a fixed size, such that if the B-treap contains n keys and $B = \Omega\left((\ln(n))^{1/(1-\epsilon)}\right)$ for some $\epsilon > 0$, then* lookup, insert, *and* delete *each touch at most $O(\frac{1}{\epsilon}\log_B(n))$ B-treap nodes in expectation, and* range-query *touches at most $O(\frac{1}{\epsilon}\log_B(n) + k/B)$ B-treap nodes in expectation where k is the size of the output. Furthermore, if $B = O(n^{\frac{1}{2}-\delta})$ for some $\delta > 0$, then with high probability the B-treap has depth $O(\frac{1}{\epsilon}\log_B(n))$ and requires only linear space to store.*

The External Memory Model. We use a variant of the parallel disk model of Vitter [17] with one processor and one disk, which measures performance in terms of disk I/Os. Internal memory is modeled as a 1-D array of data items, as in a RAM. External memory is modeled as a large 1-D array of *blocks* of data items. A block is a sequence of B data items, where B is a parameter called the *block transfer size*. The external memory can read (or write) a single block of data items to (or from) internal memory during a single I/O. Other parameters include the problem size, n, and the internal memory size m, both measured in units of data items. We will assume $m = \omega(B)$.

Uniquely Represented Memory Allocation. Uniquely represented hash tables [4, 11] can be used as the basis for a uniquely represented memory allocator. Intuitively, if the nodes of a pointer structure can be labeled with distinct hashable labels in a uniquely represented (i.e., strongly history independent) manner, and the pointer structure itself is uniquely represented in a pointer based model of computation, then these hash tables provide a way of mapping the pointer structure into a one dimensional memory array while preserving unique representation. Pointers are replaced by labels, and pointer dereferencing is replaced by hash table lookups. We will assume that the data items have distinct hashable labels. Similarly, we assume that the B-treap nodes are assigned distinct hashable labels in a uniquely represented manner. This can be achieved in any number of ways. For example, if the B-treap is the only object on disk, we may use the label of the minimum data item stored in the B-treap node.

Given these labels, we will hash B-treap nodes (one per block) into external memory. If we use distinct random bits for the hash table and everything else, this inflates number of the expected I/Os by a constant factor. (For B-treaps, uniquely represented dynamic perfect hash tables [4, 11] may be an attractive option, since in expectation most of the I/Os will involve reads, even in the case of insertions and deletions.) We thus focus on the problem of building a uniquely represented pointer structure with the desired properties.

Notation. For $n \in \mathbb{Z}$, let $[n]$ denote $\{1, 2, \ldots, n\}$. For a tree T, let $|T|$ be the number of nodes in T, and for a node $v \in T$, let T_v denote the subtree rooted at v. Finally, let $\mathrm{node}(k)$ denote the tree node with key k.

2 The Treap Partitioning Technique

The *treap partitioning* technique was introduced in [4], and is a crucial element in the design of B-treaps. It is a uniquely represented scheme that partitions a dynamic ordered set U into small contiguous subsets. It works as follows. Given a treap T and an element

$x \in T$, let its *weight* $|T_x|$ be the number of its descendants, including itself. For a parameter s, let

$$\mathcal{L}_s[T] := \{x \in T : |T_x| \geq s\} \cup \{\mathrm{root}(T)\}$$

be the *weight s partition leaders* of T[1]. We will often refer to these nodes simply as *leaders*. For every $x \in T$ let $\ell(x, T)$ be the least (deepest) ancestor of x in T that is a partition leader. Here, each node is considered an ancestor of itself. We will call a node x a *follower* of its leader $\ell(x, T)$. The weight s partition leaders partition the treap into the sets $\{\{y \in T : \ell(y, T) = x\} : x \in \mathcal{L}_s[T]\}$, each of which is a contiguous block of keys from T consisting of the followers of some leader. It is not hard to see that each set in the partition has at most $2s - 1$ elements.

We implement treap partitioning by storing the set U in a treap, where each node v has a key field storing an element of U that induces an ordering on treap nodes. Also, the treap priority for a node u is generated by hashing u.key, and the treap nodes are hashed into memory using their keys. Each node v additionally has a leader field which stores the representative of the set it is in. With some additional subtree size information, we can support finger insertions, finger deletions, and leader queries in expected constant time. For simplicity, however, we will describe a variant that supports finger insertions and deletions in expected $O(\log s)$ time. Each node v will have a size field, and we will maintain the following invariant on the contents of these fields: *For all v with $|T_v| < s$, v.size $= |T_v|$. Otherwise v.size $= \infty$.*

The treap partitioning scheme has several useful properties. We will make use of the following lemmata, which are proved in Section 5.3.4 of [8].

Lemma 1. *Fix any treap T. Inserting or deleting a node u can alter the assignment of nodes to their weight s leaders in T or the size fields for at most $2s$ other nodes in T, namely those within distance s of u in key-order.*

Lemma 2. *Let T be a treap of size n with priorities generated by an 11-wise independent hash function h from keys to $[r]$ for some $r \geq n^3$. Then $\mathbf{Pr}[|T_v| = k] = O(1/k^2)$ for any $1 \leq k < n$, $\mathbf{Pr}[|T_v| = n] = O(1/n)$, and for any $k \geq 1$, $\mathbf{Pr}[|T_v| \geq k] = O(1/k)$, so for any $s \geq 1$ each node is a weight s partition leader with probability $O(1/s)$.*

Lemma 3. *Let T be a random treap with relative priorities determined by a random permutation selected uniformly at random. Let n be the number of nodes in T and fix $s \leq n$. Then $|\mathcal{L}_s[T]| = O(n/s)$ with probability $1 - O\left(\exp\{-\frac{2n}{s(s+1)}\}\right)$.*

3 B-Treap Organization

Fix a parameter α, such that $2 < \alpha = \Theta(B)$. We will say the B-treap described below is of *order* α. For convenience, we will analyze the B-treap in terms of α rather than B. Our construction will store at most $2\alpha - 1$ keys in any given node. See Figure 1 for a depiction of a B-treap. Suppose we wish to store a set of keys U. We first describe how the B-treap is organized, and then discuss how to implement the operations.

First consider a uniquely represented treap T storing U [4]. We first describe the organization of the B-treap informally, and then give a formal description. We make use

[1] For technical reasons we include $\mathrm{root}(T)$ in $\mathcal{L}_s[T]$ ensuring that $\mathcal{L}_s[T]$ is nonempty.

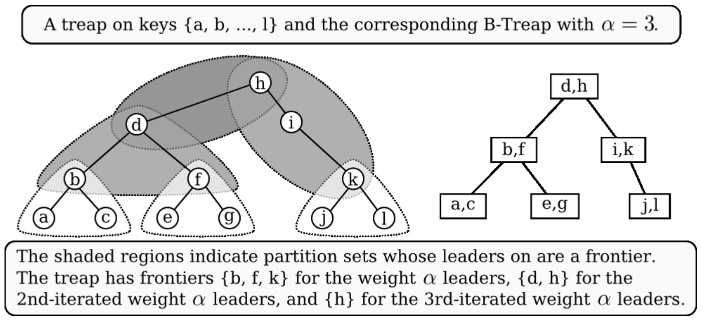
A treap on keys {a, b, ..., l} and the corresponding B-Treap with $\alpha = 3$.
h
d
i
b
f
k
a
c
e
g
j
l
d,h
b,f
i,k
a,c
e,g
j,l
The shaded regions indicate partition sets whose leaders on are a frontier.
The treap has frontiers {b, f, k} for the weight α leaders, {d, h} for the
2nd-iterated weight α leaders, and {h} for the 3rd-iterated weight α leaders.

Fig. 1. A depiction of a B-treap.

of a refinement of the treap partitioning scheme. Let $\mathcal{L}_\alpha[T]$ be the weight α leaders of T, and let $\ell(u)$ be the leader of u in $\mathcal{L}_\alpha[T]$. We will make use of the following definition.

Definition 1 (Frontier). *For a set of nodes $S \subseteq U$, let the* frontier of S, *denoted $\mathcal{F}[S]$, be the nodes in S that have at least one child not in S.*

We will store the followers in the partition sets whose leaders are in the frontier, one set of followers per node of the B-treap. That is, for each $v \in \mathcal{F}[\mathcal{L}_\alpha[T]]$, we create a node for the B-treap and store $\{u : \ell(u) = v\} \setminus \{v\}$ in it. These will be the leaves of the B-treap. We then remove all the followers (i.e., elements of $U \setminus \mathcal{L}_\alpha[T]$), and perform the same procedure on the remaining portion of T. A B-treap leaf storing key set $\{u : \ell(u) = v\} \setminus \{v\}$ has as its parent the node containing key v. We repeat the process until T has less than α nodes left, in which case all the remaining nodes are assigned to the root node of the B-treap.

To formally describe the B-treap's organization, we will need the following definitions. Fix the random bits of the RAM. Given a set of keys $S \subseteq U$, let $\mathrm{treap}(S)$ be the uniquely represented treap on key set S.

Definition 2 (Iterated Leaders of T). *The i^{th}–iterated weight α leaders of T, denoted $\mathcal{L}^i_\alpha[T]$, are defined inductively as follows.*

- *If $i = 0$, $\mathcal{L}^i_\alpha[T]$ is the set of all keys in T.*
- *If $i \geq 1$ and $|\mathcal{L}^{i-1}_\alpha[T]| > 1$, then $\mathcal{L}^i_\alpha[T] = \mathcal{L}_\alpha[\mathrm{treap}(\mathcal{L}^{i-1}_\alpha[T])]$.*
- *If $i \geq 1$ and $|\mathcal{L}^{i-1}_\alpha[T]| \leq 1$ then $\mathcal{L}^i_\alpha[T] = \emptyset$.*

Furthermore, let $\ell^i(u)$ be the deepest ancestor of u in T that is an i^{th}–iterated weight α leader of T.

Definition 3 (Rank). *The* rank *of a node v in T, denoted* $\mathrm{rank}(v)$, *is the maximum integer k such that $v \in \mathcal{L}^k_\alpha[T]$. The rank of a tree is the rank of its root.*

Let $k = \mathrm{rank}(T)$ be the rank of the root of T. We store the keys in $\mathcal{L}^{k-1}_\alpha[T]$ and the root of T at the root of the B-treap $\bar{T}$. For $i = k-1$ to 1 in decreasing order, for each node $v \in \mathcal{F}[\mathcal{L}^i_\alpha[T]]$, construct a node $\bar{v}$ in $\bar{T}$ with a key set consisting of the followers of v in the i^{th} level treap partition, excluding v. In this case we will say that $\bar{v}$ *corresponds* to v. Formally, the key set of $\bar{v}$ is $\{u \ : \ u \neq v, \ell^i(u) = v,$ and $u \in \mathcal{L}^{i-1}_\alpha[T]\}$. Finally, make $\bar{v}$ a child of the node in $\bar{T}$ corresponding to $\ell^{i+1}(v)$. Note that it is possible for a node v to be in two different frontiers, so that $v \in \mathcal{F}[\mathcal{L}_\alpha^i[T]]$ and $v \in \mathcal{F}[\mathcal{L}_\alpha^j[T]]$ for $i < j$. In this case, we create a B-treap node corresponding to each instance in which v is in some frontier. In other words, we create a B-treap node corresponding to (v, i) and another one corresponding to (v, j). In this case, the key set of the former is $\{u \ : \ u \neq v, \ell^i(u) = v, \text{ and } u \in \mathcal{L}_\alpha^{i-1}[T]\}$ and the key set of the latter is $\{u \ : \ u \neq v, \ell^j(u) = v, \text{ and } u \in \mathcal{L}_\alpha^{j-1}[T]\}$. In the future, we will simply refer to the B-treap node corresponding to v, since it can always be inferred from the context which copy is meant.

We have described above how to assign keys to B-treap nodes. In fact, the B-treap will not store merely a set of keys in each node, rather it will store the corresponding treap nodes, with left and right fields for the left and right child of the current node. Finally, if a treap node v stored in B-treap node $\bar{v}$ has a child u that is stored in a different B-treap node $\bar{u}$, we store the label of $\bar{u}$ with v.

By storing small regions of the treap in each B-treap nodes, and storing abstract pointers (in the form of labels) corresponding to treap edges that cross from one region to another, we can search the B-treap for a key by using the underlying treap that it stores. However, to dynamically maintain the B-treap's organization we also must implement several layers of treap partitioning. Specifically, we must dynamically maintain $\text{rank}(T)$ treap partitioning instances simultaneously on the same treap T, where the i^{th} instance stores the weight α partition of $\text{treap}(\mathcal{L}_\alpha^{i-1}[T])$. Consider the treap partitioning scheme of Section 2, particularly the size field invariant. For the iterated treap partitioning scheme, we can modify the size fields to store a pair of integers and modify the invariant as follows. (Below, for a treap T and set of nodes S, we define $T \cap S$ as though T were the set of the nodes it contains.)

The size field invariant (iterated version): For each treap node v,
$v.\mathsf{size} = (\text{rank}(v),\ |T_v \cap \mathcal{L}_\alpha^{\text{rank}(v)}[T]|)$.

The invariant describing what the leader fields should be set to must also be modified. In particular, each node v with $v.\mathsf{size} \in \{i\} \times \mathbb{N}$ should have its leader field set to $\ell^{i+1}(v)$, its deepest ancestor in $\mathcal{L}_\alpha^{i+1}[T]$. Then v will be stored in the B-treap node corresponding to $\text{node}(v.\mathsf{leader})$.

4 Implementing B-Treaps

Let $\bar{T}$ be a B-treap storing a treap T. We implement the operations as follows.

Lookup: Given an input key u, start at the root $\bar{r}$ of $\bar{T}$, find the root r of the treap T (by finding the highest priority node stored in $\bar{r}$) and proceed as in a regular treap lookup, jumping from one B-treap node to the next as necessary.

Insertion: To insert a key, create a new node u with that key and search for the leaf position $\text{leaf}(u)$ that u would occupy in the treap T, if it had the lowest priority of any node. Rotate u up to its proper position in T, updating the size fields appropriately during the rotations, so as to maintain the iterated size field invariant. This can be done in $O(1)$ time per rotation, and ensures the size fields of all descendants of u are correct.

Suppose u is a leader in some partition – that is, $\text{rank}(u) > 0$. Find the predecessor a and successor b of u in T. Proceed up the a-to-u and b-to-u paths looking for $\ell^i(a)$

and $\ell^i(b)$ for all $i \in \{1, 2, \ldots, \mathrm{rank}(u)\}$. (Recall $\ell^i(v)$ is the deepest ancestor of v in $\mathcal{L}^i_\alpha[T]$.) These are easy to find given the size fields. If a node v was in $\mathcal{F}[\mathcal{L}^i_\alpha[T]]$ and no longer is, then the corresponding B-treap node must be destroyed. Similarly, if a node v is now in $\mathcal{F}[\mathcal{L}^i_\alpha[T]]$ and was not previously, then a corresponding B-treap node must be created. Furthermore, whenever a node has its leader field changed, we must move it to the B-treap node corresponding to its new leader.

Let a'_i and b'_i be the children of $\ell^i(a)$ and $\ell^i(b)$ that are ancestors of a and b, respectively. For each $i \in 1, 2, \ldots \mathrm{rank}(u)$, update the leader fields of all of the descendants of a'_i in $\mathcal{L}^{i-1}_\alpha[T]$ to $\ell^i(a)$.key and move them to the B-treap node corresponding to $\ell^i(a)$. Do likewise for the the descendants of b'_i in $\mathcal{L}^{i-1}_\alpha[T]$, with $\ell^i(b)$ in place of $\ell^i(a)$. Make sure to destroy all B-treap nodes that become empty of treap nodes during this process.

That addresses the descendants of u with rank less than $\mathrm{rank}(u)$. The descendants of u with rank equal to $\mathrm{rank}(u)$ will have their fields set correctly unless inserting u causes some node to be "promoted" (i.e., its rank increases). We will deal with that possibility later, and will now focus on setting u's leader field correctly. To do so, we find $\ell^{\mathrm{rank}(u)+1}(u)$. As in the case with treap partitioning, inserting u may cause a promotion of some ancestor of u into $\mathcal{L}^{\mathrm{rank}(u)+1}_\alpha[T]$. We can determine this as in treap partitioning. Specifically, set the leader field of u equal to the leader field of its parent node. Find the child of $\mathrm{node}(u.\mathsf{leader})$ that is an ancestor of u, which we denote by u'. Then there is a promotion if and only if $u'.\mathsf{size} \in \mathbb{N} \times \{\alpha - 1\}$ and $u' \neq u$. If there is no promotion, then merely increment the second coordinate of $w.\mathsf{size}$ for each w on the path from u to u' (excluding u and including u'), and insert u into the B-treap node corresponding to $\mathrm{node}(u.\mathsf{leader})$.

If u' is promoted, create a new B-treap node $\bar{u'}$ corresponding to it. Traverse $T_{u'} \cap \mathcal{L}^{\mathrm{rank}(u)}_\alpha[T]$, move all nodes therein to $\bar{u'}$, and change their leader fields to u'.key. Note that this correctly updates the leader fields and placements of the descendants of u with rank equal to $\mathrm{rank}(u)$. Make sure to destroy any B-treap node that is emptied of treap nodes in the process. Next, increment the second coordinate of $w.\mathsf{size}$ for each w on the path from u to u' (excluding u and u'), and update $u'.\mathsf{size}$ appropriately by incrementing the first coordinate and setting the second coordinate to one. Additionally, increment of second coordinate of the size field for each node on the path from u''s parent to its parent's leader (i.e., $\ell^{\mathrm{rank}(u)+2}(u')$), excluding $\ell^{\mathrm{rank}(u)+2}(u')$.

Finally, we must consider the possibility that the promotion of u' into $\mathcal{L}^{\mathrm{rank}(u)+1}_\alpha[T]$ may cause cascading promotions, potentially all the way up to the root. However, the promotion of a node w into $\mathcal{L}^k_\alpha[T]$ is like an insertion of w into the weight α partition on $\mathcal{L}^k_\alpha[T]$ with the additional fact that w is a leaf of $\mathrm{treap}(\mathcal{L}^k_\alpha[T])$. Hence we can handle it as discussed above.

Deletion: Let u be the node to be deleted. First, we handle potentially cascading "demotions" (i.e., decreases in rank). We repeat the following steps until the demotion or deletion of the current node w does not cause a demotion of its leader. Initialize w to u.

(1) Proceed up the path P from w to $w_l := \mathrm{node}(w.\mathsf{leader})$, decrementing the second coordinate of $w'.\mathsf{size}$ for each $w' \in P \setminus \{w, w_l\}$. Use the size fields of the children of w_l to check if $|T_{w_l} \cap \mathcal{L}^{\mathrm{rank}(w_l)-1}_\alpha[T]| = \alpha$ before the deletion or demotion of w.

(2) If $|T_{w_l} \cap \mathcal{L}^{\mathrm{rank}(w_l)-1}_\alpha[T]| = \alpha$ then w_l will be demoted, in which case we create a new B-treap node corresponding to the parent of w_l (if it does not already exist),

traverse $T_{w_l} \cap \mathcal{L}_\alpha^{\mathrm{rank}(w)}[T]$, move all of the nodes therein to the newly created B-treap node, and set their leader fields to the key of the parent of w_l. Delete any B-treap node that becomes empty. Also, decrement the first coordinate of w_l.size (i.e., its rank) and set its second coordinate to $\alpha - 1$.

(3) Set $w = w_l$.

Next, rotate u down to a leaf position, updating the subtree size and rank information appropriately on each rotation (this can be done in constant time per rotation). Make sure to account for the fact that u will ultimately be deleted from the treap when updating these size fields.

If initially $\mathrm{rank}(u) = 0$, then just delete u. If initially $\mathrm{rank}(u) \geq 1$, maintain a list L of nodes x such that we rotated on edge $\{x, u\}$ when rotating u down to a leaf position. Delete u from the treap, but retain a temporary copy of its fields. Let $\mathrm{rank}(u)$ denote the initial rank of u. For $i \in \{1, 2, \ldots, \mathrm{rank}(u)\}$ in increasing order, find the deepest element v_i of $\mathcal{L}_\alpha^i[T]$ in L using the recently updated size fields. For each i, if there is no B-treap node corresponding to v_i then create one. For each node $x \in L$, in order of increasing priority, if $x \in \mathcal{L}_\alpha^{i-1}[T]$ is a descendant of v_i and has a child $x' \in \mathcal{L}_\alpha^{i-1}[T]$ with x'.leader $\neq v_i$.key, update the leader field of each node in $T_{x'} \cap \mathcal{L}_\alpha^{i-1}[T]$ to v_i.key. Move all such nodes to the B-treap node corresponding to v_i. Also set x.leader $= v_i$.key and move it to the B-treap node corresponding to v_i. Throughout the whole operation, make sure to destroy all B-treap nodes that are emptied of treap nodes. If the deletion of u did not cause any demotions, then for each ancestor x of $v_{\mathrm{rank}(u)}$ in L set x.leader $= u$.leader. Also, if x has a child y of with $\mathrm{rank}(y) < \mathrm{rank}(x)$ then for each such child y test if y.leader $\neq x$.key. If so, do a traversal of $T_y \cap \mathcal{L}_\alpha^{\mathrm{rank}(y)}[T]$, update the leader field of each node in that subtree to x.key, and move each such node to the B-treap node corresponding to x.

Range Query: To return all keys between x and y, simply lookup x, then proceed to simulate an in-order traversal of the underlying treap until reaching y.

5 The Analysis of B-Treaps

For simplicity of exposition we assume the nodes have relative priorities determined by a random permutation selected uniformly at random. To remove this assumption and prove Theorem 1, it suffices to use the hash family of Östlin and Pagh [13] to generate priorities by hashing keys to a sufficiently large set of integers (e.g., $[n^3]$).

Unique Representation. The data structure is uniquely represented assuming the operations maintain the size and leader field invariants and the proper B-treap organization. In this case, the B-treap's representation in external memory is a deterministic function of the treap it stores and the hash table used to map it onto external memory. Thus it is independent of the historical sequence of operations that led to the current logical state, and must be the same for all such sequences of operations.

Bounding the Space Usage. If the treap priorities are generated via a random permutation and $\alpha = O(n^{\frac{1}{2}-\epsilon})$, then the space usage is linear with overwhelming probability (roughly

$1 - \exp\left(n/\alpha^2\right)$). While it is unfortunate that the analysis depends on the branching factor, the assumption that $\alpha = O(n^{\frac{1}{2}-\epsilon})$ is not unreasonable, and I conjecture that this dependence can be removed. In any event, Proposition 1 bounds the space needed for any given B-treap node at $O(\alpha)$ data items, and Lemmas 4 and 5 together bound the number of B-treap nodes at $O(n/\alpha)$. If $B = \Theta(\alpha)$, this implies that a B-treap with n keys uses only $O(n/B)$ blocks of space with high probability, thus proving the space bounds of Theorem 1. The following proposition is straightforward to prove.

Proposition 1. *Each B-treap node $\bar{v}$ has at most $2\alpha - 1$ treap nodes stored in it.*

Lemma 4. *A B-treap $\bar{T}$ on $n \geq \alpha$ keys obtained from the iterated weight α partition of a random treap T has $O(n/\alpha + l)$ nodes, where l is the number of leaves in the B-treap.*

Proof. A *chain* C of T is a connected set of degree two nodes in T such that for all $u, v \in C$, u is an ancestor of v or v is an ancestor of u. Thus, each chain C can be written as $\mathrm{ancestors}(u) \cap \mathrm{descendants}(v)$ for some nodes $u, v \in C$, called the *endpoints* of C. It is not hard to see that any chain C of T has all of its nodes stored in at most $\left\lfloor \frac{|C|}{\alpha} \right\rfloor + 2$ nodes in the B-treap $\bar{T}$. If, for example, $|C| \geq \alpha/2$, then we may amortize the storage required for these $\left\lfloor \frac{|C|}{\alpha} \right\rfloor + 2$ nodes (each of which takes $O(\alpha)$ data items worth of space) against the $|C|$ treap nodes. We thus *mark* all B-treap nodes that store at least one treap node from any chain of T of length at least $\alpha/2$. As per our previous remarks, there are at most $O(n/\alpha)$ marked nodes.

Next, consider the unmarked nodes. We claim that the number of unmarked nodes is at most $2l$, where l is the number of leaves in the B-treap (either marked or unmarked). To prove this, we first prove that in the B-treap, each unmarked internal node $\bar{v}$ other than the root has at least two children. Since $\bar{v}$ is an internal node, each treap node u stored in it has $\mathrm{rank}(u) \geq 1$, so that $|T_u| \geq \alpha$. Suppose the treap nodes in $\bar{v}$ have rank k. Then each $u \in \mathcal{F}[\mathcal{L}_\alpha^k[T]]$ stored in $\bar{v}$ corresponds to a child of $\bar{v}$ in $\bar{T}$. However, if there were only one such node, then $\bar{v}$ must store a chain of length at least $\alpha - 1$, contradicting the fact that $\bar{v}$ is an unmarked, internal, non-root node. This allows us to bound the number of unmarked nodes by $2l$ via a standard argument using the well-known facts that in any undirected graph $G = (V, E)$, $\sum_{v \in V} \deg(v) = 2|E|$ and in a tree $|E| = |V| - 1$.

Lemma 5. *Let $\bar{T}$ be a B-treap on $n \geq \alpha$ keys obtained from the iterated weight α partition of a random treap T with relative priorities determined by a random permutation selected uniformly at random. Let l be the number of leaves of $\bar{T}$. Then $l = O(n/\alpha)$ with probability* $1 - O\left(\exp\{-\frac{2n}{\alpha(\alpha+1)}\}\right)$.

Proof. The number of leaves in $\bar{T}$ equals $|\mathcal{F}[\mathcal{L}_\alpha[T]]|$, which is bounded by $|\mathcal{L}_\alpha[T]|$. The result thus follows from Lemma 3.

Bounding the Depth. The main purpose of the B-tree is to reduce the depth of the search tree from, e.g., $2\log_2(n)$ (for red-black trees) or $\sim 1.44\log_2(n)$ (for AVL trees), to $\log_\alpha n$ for a suitable parameter α. Indeed, α is often n^ϵ for some constant ϵ, so that the tree height is roughly $1/\epsilon$. So it is reasonable to require any proper B-tree alternative to also have height $O(\log_\alpha(n))$. The B-treap does indeed have this property with high probability if α is sufficiently large. Proving this fact involves some inductive probabilistic conditioning. Due to space limitations, we only sketch the proof below, and omit the proof of Lemma 6. For the full proof, see Section 5.5.4 of [8].

Theorem 2. *Fix a random treap T on n nodes with relative priorities determined by a random permutation selected uniformly at random. Let $\bar{T}$ be the corresponding B-treap generated from the iterated weight α partitioning of T. If $\alpha = \Omega\left(\ln(n)^{1/(1-\epsilon)}\right)$ for some positive constant ϵ, then $\operatorname{rank}(T) = O\left(\frac{1}{\epsilon}\log_\alpha(n)\right)$ with high probability. Furthermore, since the B-treap depth is bounded by the rank of its root node, these bounds apply to the B-treap depth as well.*

Proof. Let $f(m,k) := \mathbf{Pr}[\text{A random treap } T \text{ on } m \text{ nodes has } \operatorname{rank}(T) > k]$. In the definition of f, we assume the treap T has priorities determined by a random permutation on the keys. We claim that for any m and k, $f(m,r) \le f(m+1,r)$, and then proceed by induction on the treap size. Specifically, we use the following induction hypothesis, for suitable constants c_1, c_2 and c_3.

$$f\left(\left(\frac{\alpha}{c_2\ln(n)}\right)^k,\ c_1k\right) \;\le\; \frac{(3(\alpha+1))^k}{n^{c_3}} \tag{5.1}$$

The basis $k=1$ is straightforward. For the induction step, consider a treap T on $m = \left(\frac{\alpha}{c_2\ln(n)}\right)^k$ nodes, with keys $X := \{1,2,\ldots,m\}$. Let Y be the α nodes of T with the highest priorities. Since the priorities are random, Y is will be distributed uniformly at random on $\{V : V \subset X, |V| = \alpha\}$. Let $Y = \{y_1, y_2, \ldots, y_\alpha\}$, with $y_1 < y_2 < \cdots < y_\alpha$, and let $y_0 := 0, y_{\alpha+1} := m+1$. Define $Z_i := \{z : y_i < z < y_{i+1}\}$ for all $0 \le i \le \alpha$. We claim that

$$\operatorname{rank}(T) \le \max_{0\le i\le\alpha}\left(\operatorname{rank}(\operatorname{treap}(Z_i))\right) + 2 \tag{5.2}$$

Let $\rho := \max_{0\le i\le\alpha}\left(\operatorname{rank}(\operatorname{treap}(Z_i))\right)$. Note that the only nodes in T with rank $\rho+1$ must be in Y itself, since all the nodes in any Z_i have rank at most ρ. Thus T contains at most α nodes of rank $\rho+1$, and hence the root of T can have rank at most $\rho+2$.

Next we obtain a high probability bound on $\max_{0\le i\le\alpha}\left(\operatorname{rank}(\operatorname{treap}(Z_i))\right)$ using Lemma 6 and the induction hypothesis. Let A_i be the event that $|Z_i| > \left(\frac{\alpha}{c_2\ln(n)}\right)^{k-1}$, and let $A := \cup_i A_i$. Use Lemma 6 below to bound $\mathbf{Pr}[A]$. Note that if we condition on all the Z_i's being sufficiently small (i.e., the event $\bar{A}$), then the priorities on the nodes of each Z_i are still random. Let B be the event that there exists an i such that $\operatorname{rank}(\operatorname{treap}(Z_i)) > c_1(k-1)$. Apply the induction hypothesis to bound $\mathbf{Pr}\left[B|\bar{A}\right]$. Equation (5.2) then implies that $f\left(\left(\frac{\alpha}{c_2\ln(n)}\right)^k,\ c_1(k-1)+2\right) \;\le\; \mathbf{Pr}[A\cup B] \le \mathbf{Pr}[A] + \mathbf{Pr}\left[B|\bar{A}\right]$. If $c_1 \ge 2$, then $c_1(k-1)+2 \le c_1k$, and thus we have a bound for $f\left(\left(\frac{\alpha}{c_2\ln(n)}\right)^k,\ c_1\cdot k\right)$. This completes the induction. Next, let $d(n) := \frac{\ln(n)}{\ln\left(\frac{3(\alpha+1)}{c_2\ln(n)}\right)}$ and use the fact that $f(m,r) \le f(m+1,r)$ to show $f(n,\ c_1\lceil d(n)\rceil) \le \frac{(3(\alpha+1))^{\lceil d(n)\rceil}}{n^{c_3}}$. The rest of the proof follows from suitable assumptions on the size of α, c_1, c_2 and c_3.

Lemma 6. *Fix $\alpha, m \in \mathbb{N}$ with $\alpha < m$. Let $X = \{1, 2, \dots, m\}$. Select a subset Y of X uniformly at random from $\{V : V \subset X, |V| = \alpha\}$. Let $Y = \{y_1, y_2, \dots, y_\alpha\}$, with $y_1 < y_2 < \cdots < y_\alpha$, let $y_0 := 0, y_{\alpha+1} := m + 1$, and let $\Delta = \max_i\{y_{i+1} - y_i - 1\}$, where i ranges from zero to α. Then for all c and n,* $\mathbf{Pr}\big[\Delta > c\frac{m+1}{\alpha}\ln(n)\big] \leq \frac{m}{n^c}$.

Bounding the Running Time. We measure the running time of the B-treap operations in terms of the number t of B-treap nodes they inspect or alter. This is in line with the EM model, since the number of I/Os is $O(t)$ in expectation.

By this measure, lookups clearly inspect only $\text{depth}(\bar{T})$ B-treap nodes. Inserting u may require inspecting up to $\text{depth}(\bar{T})$ nodes to find $\text{leaf}(u)$. After that, if P is the rotation path of u from $\text{leaf}(u)$ to its proper location in treap T, then the operation might modify every B-treap node containing treap nodes in P, as well as B-treap nodes corresponding to elements of $F := P \cap \big(\cup_{i\geq 1}\mathcal{F}[\mathcal{L}^i_\alpha[T]]\big)$. Moreover, it is not hard to see that these are the only B-treap nodes that need to be updated. The number of B-treap nodes storing treap nodes in P is bounded by $\text{depth}(\bar{T})$. Bounding $|F|$ is trickier. Note that $\mathbf{E}[|F|] = O(1)$, since $|F| \leq |P|$ is bounded by the number of rotations in P, and $\mathbf{E}[|P|] = O(1)$ in a random treap [14]. The same argument holds true for deletions.

For range queries between x and y, it is not hard to see that the inspected B-treap nodes consist of the set Q of nodes containing treap nodes in the root-to-x and root-to-y paths, as well as the set of B-treap nodes R containing keys strictly between x and y. Using the analysis for the size of a B-treap on n treap nodes, it is not too difficult to show that $|R| = O(k/\alpha + \text{depth}(\bar{T}))$ with high probability, where k is the number of treap nodes in the output. Clearly $|Q| \leq 2\,\text{depth}(\bar{T})$, so this implies only $O(k/\alpha + \text{depth}(\bar{T}))$ B-treap nodes are inspected during the range query. If the internal memory has size $m \geq B \cdot \text{depth}(\bar{T})$, and we store the $\text{depth}(\bar{T})$ previously accessed B-treap nodes, then we need lookup each node in external memory at most once. Otherwise, if $m = O(B)$, we can bound the number of B-treap node lookups by the number of edges in the subtree of the B-treap containing $Q \cap R$, which is also $O(k/\alpha + \text{depth}(\bar{T}))$ in expectation. We have now proven the following result.

Lemma 7. *In a B-treap $\bar{T}$, the lookup operation inspects* $\text{depth}(\bar{T})$ *nodes, updates inspect or modify* $\text{depth}(\bar{T}) + O(1)$ *nodes in expectation, and range queries inspect* $O(k/\alpha + \text{depth}(\bar{T}))$ *nodes in expectation, where k is the size of the output.*

Combining Lemma 7 with Theorem 2 and noting that $\alpha = \Theta(B)$ then proves the running time bounds of Theorem 1.

Additional Supported Operations. There are several additional properties of treaps that the B-treap can exploit. Based on the reasoning above, it follows that finger insertions and deletions touch $O(1)$ B-treap nodes in expectation. Similarly, predecessor and successor queries can be answered by inspecting $O(1)$ B-treap nodes in expectation. Finally, given a sorted list of elements, a B-treap on them can be constructed in expected linear time.

6 Empirical Observations

Simulations in which the treap priorities were provided by a psuedorandom number generator suggest the B-treap has small depth (e.g., empirically bounded by $1.5\log_\alpha(n)$

for $\alpha = 100$ and $n \leq 10^6$) and space utilization of roughly $1/3^{\text{rd}}$. Refer to Section 5.5.5 of [8] for more detail. If the $1/3^{\text{rd}}$ space utilization is judged unacceptably low, there are various ways to improve it, at the cost of increasing the average number of I/Os per operation. For example, for any fixed $k \in \mathbb{N}$ we may divide each block into k equally sized *block-parts*, and store a B-treap node containing q treap nodes in $\lceil kq/(2\alpha - 1) \rceil$ block-parts dynamically allocated to it. This ensures there is at most one block-part of unused space per B-treap node, rather than a whole block.

References

1. Umut A. Acar, Guy E. Blelloch, Robert Harper, Jorge L. Vittes, and Shan L. Maverick Woo. Dynamizing static algorithms, with applications to dynamic trees and history independence. In *15th annual ACM-SIAM Symposium on Discrete Algorithms*, pages 531–540, 2004.
2. Arne Andersson and Thomas Ottmann. New tight bounds on uniquely represented dictionaries. *SIAM Journal of Computing*, 24(5):1091–1103, 1995.
3. Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.
4. Guy E. Blelloch and Daniel Golovin. Strongly history-independent hashing with applications. In *48th Annual IEEE Symposium on Foundations of Computer Science*, pages 272–282, 2007.
5. Guy E. Blelloch, Daniel Golovin, and Virginia Vassilevska. Uniquely represented data structures for computational geometry. In *SWAT '08: Proceedings of the 11th Scandinavian Workshop on Algorithm Theory*, pages 17–28, Gothenburg, Sweden, July 2008. Springer.
6. Niv Buchbinder and Erez Petrank. Lower and upper bounds on obtaining history independence. *Information and Computation*, 204(2):291–337, 2006.
7. R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing—a fast access method for dynamic files. *ACM Trans. Database Syst.*, 4(3):315–344, 1979.
8. Daniel Golovin. *Uniquely Represented Data Structures with Applications to Privacy*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 2008. Tech. Report CMU-CS-08-135.
9. Jason D. Hartline, Edwin S. Hong, Alexander E. Mohr, William R. Pentney, and Emily Rocke. Characterizing history independent data structures. *Algorithmica*, 42(1):57–74, 2005.
10. Daniele Micciancio. Oblivious data structures: applications to cryptography. In *STOC '97: Proceedings of the twenty-ninth annual ACM Symposium on Theory of Computing*, pages 456–464, New York, NY, USA, 1997. ACM Press.
11. Moni Naor, Gil Segev, and Udi Wieder. History-independent cuckoo hashing. In L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, editors, *Proceedings of 35th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 5126 of *Lecture Notes in Computer Science*, pages 631–642. Springer, 2008.
12. Moni Naor and Vanessa Teague. Anti-persistence: history independent data structures. In *STOC '01: Proceedings of the thirty-third annual ACM Symposium on Theory of Computing*, pages 492–501, New York, NY, USA, 2001. ACM Press.
13. Anna Östlin and Rasmus Pagh. Uniform hashing in constant time and linear space. In *STOC '03: Proceedings of the thirty-fifth annual ACM Symposium on Theory of Computing*, pages 622–628, New York, NY, USA, 2003. ACM Press.
14. R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996.
15. Lawrence Snyder. On uniquely representable data structures. In *FOCS '77: IEEE Symposium on Foundations of Computer Science*, pages 142–146. IEEE, 1977.
16. Rajamani Sundar and Robert E. Tarjan. Unique binary search tree representations and equality-testing of sets and sequences. In *STOC '90: Proceedings of the twenty-second annual ACM Symposium on Theory of Computing*, pages 18–25, New York, NY, USA, 1990. ACM Press.
17. Jeffrey Scott Vitter. Algorithms and data structures for external memory. *Foundations and Trends in Theoretical Computer Science*, 2(4):305–474, 2006.