Department of Informatics, University of Zürich

MSc Basismodul

The Adaptive Radix Tree

Rafael Kallis

Matrikelnummer: 14-708-887

Email: `rk@rafaelkallis.com`

September 18, 2018

supervised by Prof. Dr. Michael Böhlen and Kevin Wellenzohn

University of Zurich UZH

Department of Informatics

1 Introduction

The goal of this project is to study and implement the Adaptive Radix Tree (ART), as proposed by Leis et al. [2]. ART, which is a trie based data structure, achieves its performance, and space efficiency, by compressing the tree both vertically, i.e., if a node has no siblings it is merged with its parent, and horizontally, i.e., uses an array which grows as the number of children increases. Vertical compression reduces the tree height and horizontal compression decreases a node's size.

In Section 3 we describe how ART is constructed by applying vertical and horizontal compression to a trie. Next, we describe the point query procedure, as well as key deletion in Section 4. Finally, a benchmark of ART, a red-black tree and a hashtable is presented in Section 5.

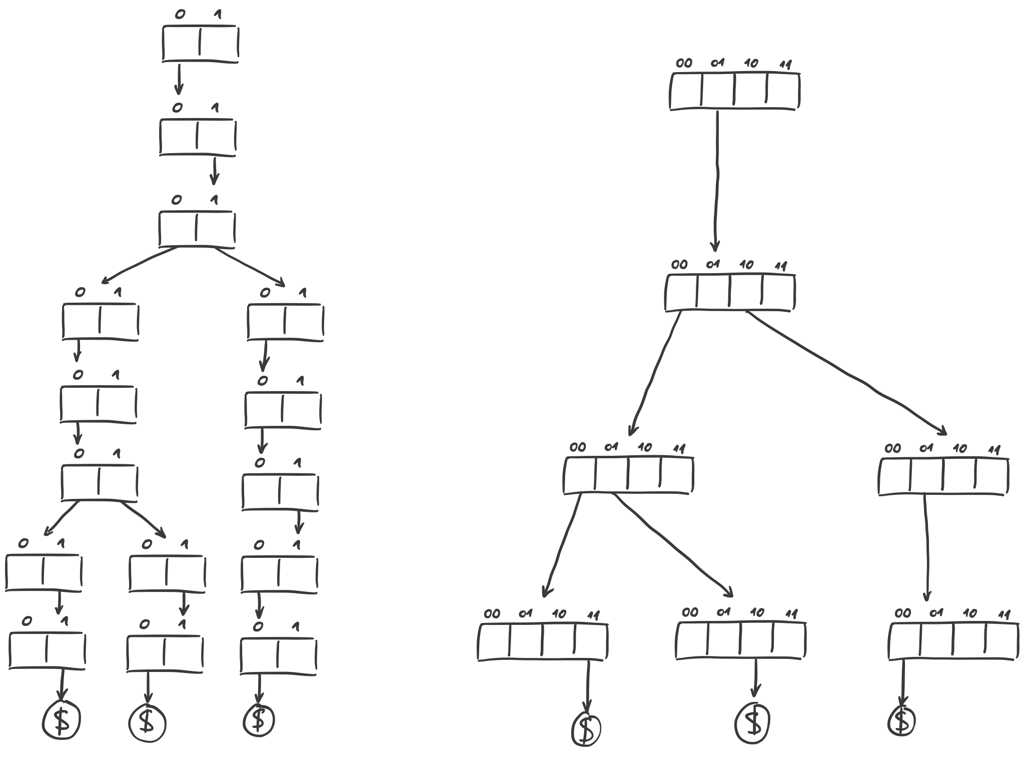
2 Background - Tries

A trie [1] is a hierarchical data structure which stores key-value pairs. Tries can answer both point and range queries efficiently since keys are stored in lexicographic order. Unlike a comparison-based search tree, a trie does not store keys in nodes. Rather, the digital representation of a search key is split into partial keys used to index the nodes. When constructing a trie from a set of keys, all insertion orders result in the same tree. Tries have no notion of balance and therefore do not require rebalancing operations.

Keys are split into partial keys of s bits each, where s is called *span*. Inner nodes have 2^s child pointers (possibly `null`), one for each possible s-bit sequence. During tree traversal, we descend down to the child node identified by the d-th s-bit partial key of the search key, where d is the depth of the current node. Using an array of 2^s pointers, this lookup can be done without any additional comparison.

Figures 1a and 1b depict tries storing the 8-bit keys “01000011”, “01000110” and “01100100” with $s \in \{1, 2\}$. Nodes with `$` are terminator nodes and are used to indicate the end of a key. Span s is critical for the performance of the trie because s determines the height of the trie. When a trie's height increases, performance deteriorates as more nodes have to be traversed during a search. We observe that by increasing the span, we decrease the tree height. A trie storing k bit keys has $\lceil \frac{k}{s} \rceil$ levels of nodes. As a consequence, point queries, insertions and deletions have $O(k)$ complexity because a trie cannot have a height greater than the length of keys it stores.

Span s also determines the space consumption of the tree. A node with span s requires 2^s pointers. An apparent trade off exists between tree height versus space efficiency that depends on s. Increasing s yields a decrease in the tree's height, resulting in faster lookups because less nodes are traversed. On the other hand, by increasing s, nodes require to store more child pointers. Every increment in s halves the height of the tree but doubles the amount of child pointers of each node. Generally, increasing s yields an increase to the space consumption of a trie.



(a) $s = 1$ (b) $s = 2$

Figure 1: Tries with span $s \in \{1, 2\}$ storing keys "01000011", "01000110" and "01100100".

3 Adaptive Radix Tree (ART)

The *Adaptive Radix Tree* (ART) is a space efficient trie, which achieves its low memory consumption using *vertical* and *horizontal* compression. Using vertical compression, ART reduces the tree height significantly by merging parent nodes with child nodes under certain circumstances. Horizontal compression reduces the amount of space required by each node depending on the number of child nodes.

3.1 Vertical (Prefix) Compression

When storing long keys, chains of nodes start to form where each node only has a single child. (e.g. the trie in Figure 1a stores "01100100", we see a chain of five nodes). As a consequence, space is wasted as many nodes with little actual content are kept and traversals are slowed down because many nodes are traversed. Space wasting is further amplified with sparse datasets or a small span.

Morrison introduced *Patricia* [3]. Patricia is a space-optimized trie in which each node with no siblings is merged with its parent, i.e., inner nodes are only created if they are required to distinguish at least two leaf nodes. Doing so, chains caused by long keys are eliminated, which make tries space-inefficient. Morrison's Patricia tree is a bitwise trie, i.e., has a span $s = 1$, but the technique can be applied to tries with any span, although it becomes less effective as span s increases. We refer to this technique as *vertical compression*.

Leis et al. mention three approaches to deal with vertical compression:

- Pessimistic: We store an additional variable, called `prefix`, inside each node. This variable stores the concatenation of partial keys of descendants that were eliminated because they had no siblings. During lookup, `prefix` is compared to the search key before proceding to the next child. If a mismatch occurs, then the key is not contained and the search terminates returning `null`. Figure 2 depicts two tries, one with and one without vertical compression using the pessimistic approach. We observe that nodes with no siblings, color coded red, are eliminated and their partial key is appended to the parent's prefix (highlighted in gray).

- Optimistic: Only the number of compressed nodes is stored. Lookups skip this number of partial keys without comparing them to the search key. Full keys are required to be stored on leaf nodes. When a lookup arrives to a leaf node, its key must be compared to the search key. In case of a mismatch, the search key is not contained and the search is terminated returning `null`.

- Hybrid: We store the number of compressed nodes and a static, fixed-size array in order to store a part of the compressed path. If the number of compressed nodes exceeds the size of the array, the optimistic strategy is used instead.

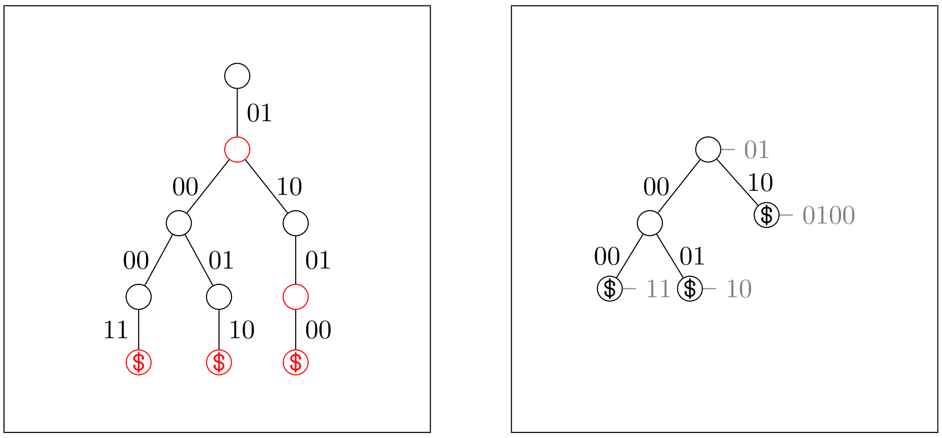


Figure 2: Tries with span $s = 2$ storing keys "01000011", "01000110" and "01100100". The trie on the right incorporates (pessimistic) vertical compression. Red nodes indicate nodes which get eliminated under vertical compression. Gray strings represent the value of the prefix variable.

3.2 Horizontal Compression (Adaptive Nodes)

With large values of span s, an excessive amount of space is sacrificed to achieve a smaller tree height. Larger nodes means space is allocated for pointers which keep references to child nodes, even if they are unused.

In order to reduce space needed to keep such references, Leis et al. propose *Adaptive Nodes* [2], which make use of dynamic data structures instead of static arrays for child node bookkeeping. Doing so, we allocate less space when the number of children is small and add more space if required, i.e., more children are added. We refer to this technique as *horizontal compression.* Leis et al. fix the span $s = 8$, i.e., partial keys are 1 byte long and therefore each node can have up to $2^8 = 256$ children.

When applying horizontal compression, a node is in one of four configurations, depending on the number of child nodes. Each of the four configurations is optimized for a different amount of children. When keys are inserted/deleted, the nodes are adapted accordingly. The most compact configuration is called `Node4` which can carry up to four children. In the same manner, we also have `Node16`, `Node48` and `Node256`. All nodes have a header which stores the node type, the number of children and the prefix variable, which contains the compressed path (c.f. Section 3.1).

Type	Children	Space (bytes)
`Node4`	2-4	$h + 4 + 4 \cdot 8 = h + 36$
`Node16`	5-16	$h + 16 + 16 \cdot 8 = h + 144$
`Node48`	17-48	$h + 256 + 48 \cdot 8 = h + 640$
`Node256`	49-256	$h + 256 \cdot 8 = h + 2048$

Table 1: Space consumption for each inner node type. h is equal to the size of the header.

We now describe the structure of each of the four configurations. Table 1 shows the space consumption for each inner node type. Note that h is equal to the header's size. Figure 3 illustrates the state of a node for each node type, when storing the partial keys 65 ("01000001"), 82 ("01010010"), 84 ("01010100") and pointers to their corresponding child nodes α, β, γ. Note that $\varnothing$ represents a `null` pointer.

A node of type `Node4` contains two 4-element arrays, one called "partial keys" and one called "children". The "partial keys" array holds partial keys of the child nodes. The "children" array, holds pointers to the child nodes. Partial keys and pointers are stored at corresponding positions in their respective arrays and the partial keys are sorted.

A node of type `Node16` is structured similarly to `Node4`, the only difference being the lengths of the two static arrays, which are 16 each.

An instance of `Node48` contains a 256-element array named "indexes" (256 bytes) and a 48-element array called "children" ($48 \cdot 8$ bytes). Partial keys are stored implicitly in "indexes", i.e., can be indexed with partial key bytes directly. As the name suggests, "indexes" stores the index of a child node inside the "children" array. 48 is a special value in the "indexes" array and is used to signal that the partial key does not exist.

Finally, a node of type `Node256` contains an array of 256 pointers which can be indexed with partial key bytes directly. Child nodes can be found with a single lookup.

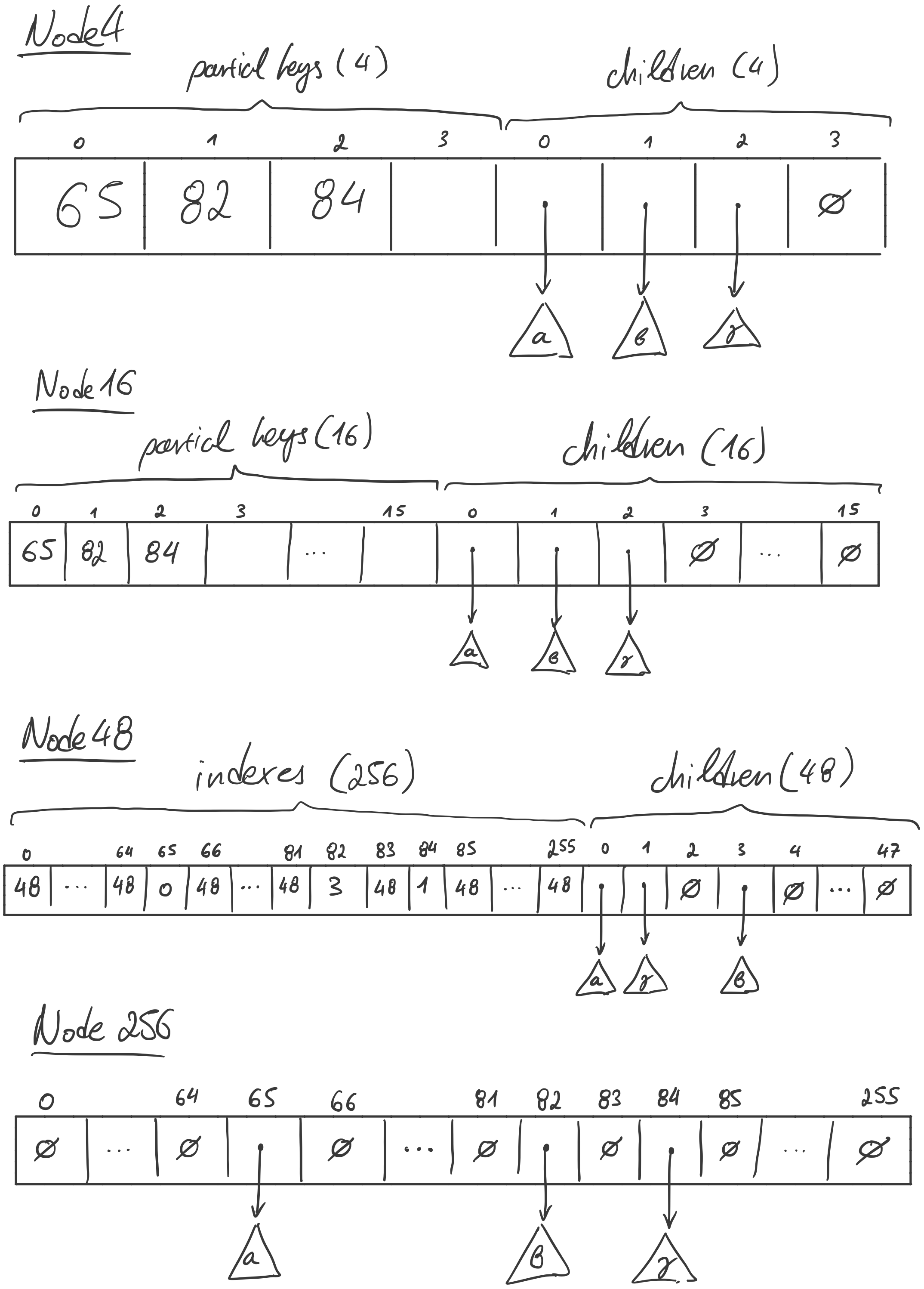


Figure 3: When horizontal compression is applied, a node is in one of four configurations, namely `Node4`, `Node16`, `Node48` and `Node256`. Each of the four configurations is optimized for a different number of child nodes. We store the partial keys 65, 82, 84 and their corresponding child nodes α, β, γ in an instance of each node type. Note that $\varnothing$ represents a `null` pointer.

4 Algorithms

We will now describe how two fundamental operations, namely point query and deletion are implemented in ART. Range queries and insertions are similar and are omitted for brevity.

Note that our implementation uses *single-value leaves* (c.f. Leis et al.), i.e., values are stored using an additional leaf node type, conveniently called `Node0`, which stores one value. `Node0` is equivalent to a terminator node mentioned in Section 2. Additionally, we utilize pessimistic vertical compression (c.f. Section 3.1), i.e., each inner node stores the entire compressed path inside the `prefix` variable using a variable length partial key array. We present our implementations below.

4.1 Point Query

The code fragment in Figure 4 shows the implementation of a point query on ART in C++. Method `get` accepts a key and its length as an argument and returns a pointer to the value associated with the given key, or `null` if the key is not found.

In lines 2-3 we declare and initialize a pointer, `cur`, which references the current node during tree traversal, `child` which references `cur`'s child and `depth`, which holds the depth of the current node. We now enter a loop in which we check if `cur` references `null` during the beginning of each iteration, and if so, the search key is not contained and we return `null`.

In lines 5-8 we check if a prefix mismatch occurs. This step is required because of vertical compression (c.f. Section 3.1). If a prefix mismatch is discovered, `null` is returned. Method `check_prefix` is a member of the `node` class which determines the number of matching bytes between `cur`'s prefix and `key` w.r.t. the current depth, e.g., given a node with prefix "`abbd`", a search key "`aaabbbccc`" and depth 2, then `check_prefix` returns 3, since the 4-th byte of the prefix ('`d`'), does not match the (depth + 4)-th byte of the search key ('`b`').

Lines 9-12 check for an exact match of the key at the current node. If so, we return the value of the current node. Finally, we traverse to the next child node. Variable `depth` is adjusted according to the number of nodes merged due to vertical compression. We lookup the next child node, which is assigned to `cur`. If no such child exists, the search key does not exist and `null` is returned.

```
1  template <class T> T * art<T>::get(const char *key, const int key_len) const {
2    node<T> *cur = root_, **child = nullptr;
3    int depth = 0;
4    while (cur != nullptr) {
5      if (cur->prefix_len_ != cur->check_prefix(key + depth, key_len - depth))
6        /* prefix mismatch */
7        return nullptr;
8      if (cur->prefix_len_ == key_len - depth)
9        /* exact match */
10       return cur->value_;
11     child = cur->find_child(key[depth + cur->prefix_len_]);
12     cur = child != nullptr ? *child : nullptr;
13     depth += cur->prefix_len_ + 1;
14   }
15   return nullptr;
16 }
```

Figure 4: Point query implemented in C++.

4.2 Deletion

Figure 5 presents our implementation of key deletion on ART in C++. During deletion, the leaf node identified by the search key is removed from an inner node, which is shrunk if necessary. If the leaf to remove only has one sibling, vertical compression is applied. We assume all leaf nodes are of type `Node0`.

In lines 3-5, we declare and initialize two pointers, `cur` and `par` which reference the current and parent node during tree traversal. Variable `cur_partial_key` holds the partial key which indexes the current node in the parent's child lookup table. Variable `depth` holds the depth of the current node. We loop until `cur` references either the right leaf node or `null`. In the latter case, the is not contained and `null` is returned.

Line 12 checks if `cur` is a leaf node, i.e. the node that contains the search key has been found. If that is not the case, we continue descending down (lines 57-60). Given `cur` is a leaf, one of three cases is possible depending on `cur`'s siblings:

- If `cur` has *no siblings* (lines 18-24), `cur` must be the root. We delete `cur` and set the root to `null`.

- If `cur` has *one sibling* (lines 24-48), vertical compression is applied, effectively removing both `cur` and its parent `par`. During this process, `par`'s prefix, as well as the sibling's partial key, are prepended to the sibling's prefix.

  Lines 26-30 search for `cur`'s sibling. Next, lines 34-39 assign the concatenation of: 1) `par`'s prefix, 2) the sibling's partial key and 3) the sibling's old prefix, to the sibling's new prefix.

  Lines 40-45 perform cleanup operations, freeing memory previously allocated by the sibling's old prefix, `cur`, `cur`'s prefix, `par` and `par`'s prefix.

  Finally, `sibling` replaces the parent (line 46).

- If `cur` has *more than one sibling* (lines 48-54), we remove `cur` from its parent `par` and shrink `par` if needed (c.f. Section 3.2).

Finally, the value associated with the search key is returned (in case the method callee has to free resources).

```cpp
template <class T> T *art<T>::del(const char *key, const int key_len) {
  if (root_ == nullptr) { return nullptr; }
  node<T> **par = nullptr, **cur = &root_;
  char cur_partial_key;
  int depth = 0;

  while (cur != nullptr) {
    if ((**cur).prefix_len_ != (**cur).check_prefix(key + depth, key_len - depth)) {
      /* prefix mismatch */
      return nullptr;
    }
    if (key_len == depth + (**cur).prefix_len_) {
      /* exact match */
      T *value = (**cur).value_;
      (**cur).value_ = nullptr;
      int n_siblings = par != nullptr ? (**par).n_children() - 1 : 0;

      if (n_siblings == 0) {
        /* cur is root */
        if ((**cur).prefix_ != nullptr) { delete[](**cur).prefix_; }
        delete (*cur);
        *cur = nullptr;

      } else if (n_siblings == 1) {
        /* find sibling and apply vertical compression */
        char sibling_partial_key = (**par).next_partial_key(0);
        if (sibling_partial_key == cur_partial_key) {
          sibling_partial_key = (**par).next_partial_key(cur_partial_key + 1);
        }
        node<T> *sibling = *(**par).find_child(sibling_partial_key);
        char *old_prefix = sibling->prefix_;
        int old_prefix_len = sibling->prefix_len_;
        /* compute new prefix of sibling */
        sibling->prefix_ = new char[(**par).prefix_len_ + 1 + old_prefix_len];
        sibling->prefix_len_ = (**par).prefix_len_ + 1 + old_prefix_len;
        std::memcpy(sibling->prefix_, (**par).prefix_, (**par).prefix_len_);
        sibling->prefix_[(**par).prefix_len_] = sibling_partial_key;
        std::memcpy(sibling->prefix_ + (**par).prefix_len_ + 1, old_prefix,
               old_prefix_len);
        if (old_prefix != nullptr) { delete[] old_prefix; }
        /* remove cur and par */
        if ((**cur).prefix_ != nullptr) { delete[](**cur).prefix_; }
        delete (*cur);
        if ((**par).prefix_ != nullptr) { delete[](**par).prefix_; }
        delete (*par);
        *par = sibling;

      } else if (n_siblings > 1) {
        /* remove cur */
        if ((**cur).prefix_ != nullptr) { delete[](**cur).prefix_; }
        delete (*cur);
        (**par).del_child(cur_partial_key);
        if ((**par).is_underfull()) { *par = (**par).shrink(); }
      }
      return value;
    }
    cur_partial_key = key[depth + (**cur).prefix_len_];
    depth += (**cur).prefix_len_ + 1;
    par = cur;
    cur = (**cur).find_child(cur_partial_key);
  }
  return nullptr;
}
```

Figure 5: Key deletion implemented in C++.

5 Benchmarks

In this section we compare the performance of ART against the two textbook data structures red-black tree (RBT) and chained hash table (HT). We chose RBT to compete against ART because, like ART, RBT is a hierarchical data structure with sorted keys, and supports answering both point and range queries efficiently. HT does answer point queries faster but lacks support for efficient range queries. We use C++ standard library containers `std::map` and `std::unordered_map`[1] which are implementations of RBT and HT, respectively.

Let k be the length of the keys and n be the size of the dataset, then ART, RBT and HT have an average case complexity of $O(k)$, $O(\log n)$ and $O(1)$,[2] respectively, for point queries, insertion and deletion. ART's and RBT's big-O complexities are incomparable, as one depends on the length of the keys and the other on the size of the dataset n. HT, with an average case complexity of $O(1)$, should outperform ART and RBT.

We first conduct a series of microbenchmarks in order to assess point query, insertion and deletion performance and then conduct an experiment in order to measure how often vertical compression is applied on ART.

There are three datasets, i.e., set of keys, used for the benchmarks: a sparse dataset, a dense dataset and a mixed dataset, called the Dell dataset [4], with sparse and dense regions. A dataset is dense (sparse) if many (few) keys share common prefixes.

The dataset of size $n = 16$ million is constructed by drawing n uniformly distributed 64-bit integers. The dense dataset of size n is constructed by picking 64-bit integers from 0 to $n - 1$ (leading bits are filled with zeroes).

5.1 Microbenchmarks

During the lookup microbenchmark, we execute point queries on an instance of each of the three data structures (ART, RBT, HT) containing the same $n = 16$ million keys. The keys are sparse (i.e., uniformly distributed), 64-bit integers. Each key is queried exactly once. The insertion microbenchmark inserts the same sixteen million sparse 64-bit keys used previously. Finally, the deletion microbenchmark, structured similar to the lookup microbenchmark, attempts to delete every stored key from the data structure.

Figures 6a to 6c show the results of the three microbenchmarks. ART is positioned in between its competitors in all three microbenchmarks and shows a robust performance. ART's performance depends on key length k, but HT does not (constant), which explains why ART is slower than HT. Although ART's and RBT's big-O complexities are incomparable as the one depends on the key length k and the other on the size of the dataset n, ART seems to outperform RBT in practice for point queries, insertion and deletion.

We also observe that ART performs half as fast compared to HT during the lookup microbenchmark, but is as fast as HT during insertion. We believe this happens because during insertion, HT occasionally requires a table resize and full rehash.

[1] https://en.cppreference.com/w/cpp/container
[2] amortized $O(1)$ during insertion and deletion.

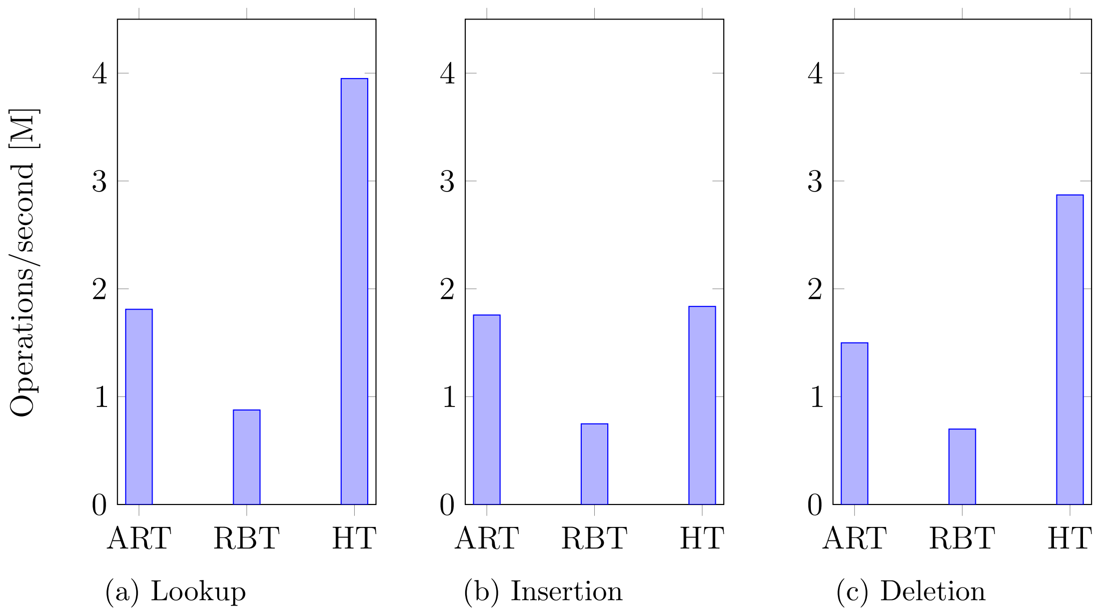


Figure 6: Lookup, insertion and deletion performance of ART, red-black trees (RBT) and hashtables (HT) over the sparse dataset.

5.2 Vertical Compression

Vertical compression is an expensive operation, as it involves structural index modifications when applied. Such structural index modifications can be difficult to implement efficiently for concurrency control because synchronization of concurrent transactions is required during such operations [4]. Our last experiment measures how often vertical compression is applied when we store keys from the sparse, dense and Dell dataset. As mentioned in Section 3.1, when deleting a node with a single sibling, the sibling is compressed, i.e., merged with its parent.

Analogously, when inserting a key, if a leaf is encountered or the search key differs from the compressed path, *expansion* is applied, i.e., a new inner node is created above the current node and the compressed paths are adjusted accordingly. We report only the number of vertical compressions during deletion, but found that the number of compressions during deletion equals the number of expansions during insertion.

The number of compressions depends on span s. If $s = 1$, every node must have exactly one sibling, and therefore every deletion requires a compression, unless the root is deleted. ART has a span of $s = 8$ and we therefore do not expect a compression for every deletion.

Results should vary depending if keys are sparse or dense. Dense keys yield a higher node fanout which causes fewer compressions. We therefore expect that having dense keys implies a smaller number of compressions.

Figures 7a to 7c depict number of vertical compressions over number of transactions for the sparse, Dell and dense dataset. We observe a linear relation between the two variables. As expected, dense keys require less compressions compared to sparse keys.

That is because with the dense dataset, ART nodes have a higher fanout. If a node has many children, more deletions are required until compression is applied. We also observe that the Dell dataset lies between the sparse and dense dataset in terms of compressions per operation but tends towards the dense dataset.

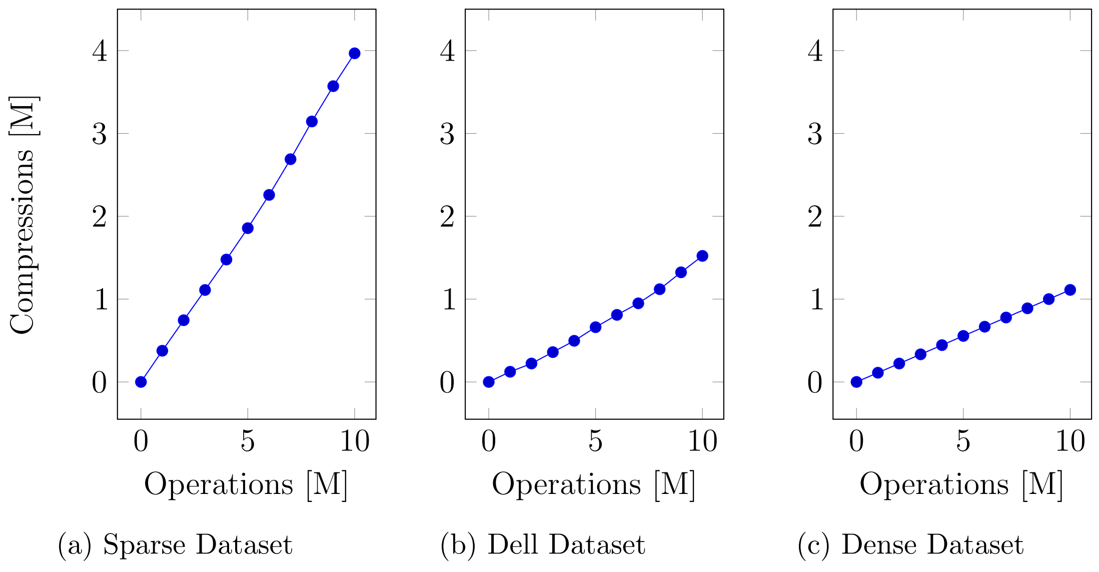


Figure 7: Number of compressions over number of operations.

6 Future Work

In this project we implemented all basic functionalities of ART, but left potential for performance improvements.

We plan to optimize ART's sequential transactional throughput with the help of extensive profiling (e.g., call history, instructions executed, etc.). Performance can also be improved by utilizing Single Instruction Multiple Data (SIMD) instructions as mentioned in [2]. A memory profile analysis would be interesting for the purpose of comparing ART's space utilization against red-black trees and hashtables w.r.t. sparse and dense datasets.

Leis et al. use hybrid vertical compression in their implementation, i.e., only store the 8 first bytes of the compressed path in a static array and the number of compressed nodes. We use pessimistic vertical compression, which stores the entire compressed path. Their approach leverages on-CPU caches better compared to the pessimistic approach, but keys must be stored at leaves and one additional comparison is required when encountering a leaf node. We want to know how strong we benefit from hybrid vertical compression and how much it impacts memory consumption.

Finally, our implementation dictates single threaded usage, but we intend to incorporate concurrency control in order to further increase the transactional throughput of ART when multiple cores can be utilized.

References

[1] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[2] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 38–49. IEEE, 2013.

[3] D. R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM (JACM)*, 15(4):514–534, 1968.

[4] K. Wellenzohn, M. Böhlen, S. Helmer, M. Reutegger, and S. Sakr. Workload-aware contention-management in indexes for hierarchical data. To be published.