Department of Informatics, University of Zürich

MSc Basismodul

The Adaptive Radix Tree

Rafael Kallis

Matrikelnummer: 14-708-887

Email: rk@rafaelkallis.com

September 1, 2018

supervised by Prof. Dr. Michael Böhlen and Kevin Wellenzohn

University of Zurich[UZH]
Department of Informatics

1 Introduction

Main-Memory Databases increasingly become a viable option for many applications due to the considerably faster access times of volatile memory in comparison to secondary storage.

Leis et al. [2] propose the Adaptive Radix Tree (ART), an in-memory data structure which efficiently stores and retrieves data. As we will see later, ART achieves its performance, and space efficiency, by compressing the tree both vertically and horizontally.

The goal of this project is to study and implement ART, as proposed by [2]. 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 Preliminaries

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. A node's key can be reconstructed from its path. When constructing a trie from a set of keys, all insertion orders result in the same tree. Tries do not require rebalancing operations and therefore have no notion of balance, unlike comparison based search trees.

Keys are split into chunks of s bits, 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 propagate down to the child node identified by the d-th s-bit chunk of the 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.

Figure 1 depicts tries storing the 8-bit keys "01000011", "01000110" and "01100100" with $s = 1, 2$. Span s is critical for the performance of the trie because s determines the height of the trie. 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 inner nodes. As a consequence, point queries, insertions and deletions have $O(k)$ complexity.

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.

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.

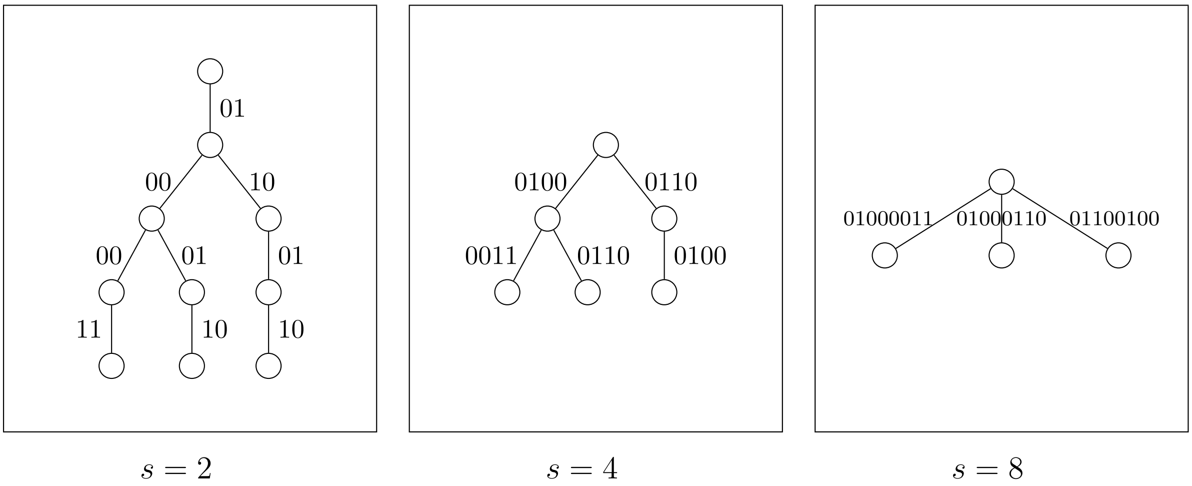


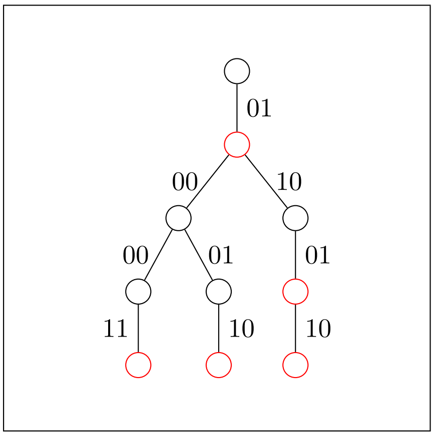
Figure 1: Tries with span $s = 2, 4, 8$ storing keys "01000011", "01000110" and "01100100".

3.1 Vertical (Prefix) Compression

When storing long keys, chains of nodes start to form where each node only has a single child. As a consequence, a lot of 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. Although Morrison's Patricia tree is a bitwise trie, i.e. has a span $s = 1$, the technique can be applied to tries with any span. We refer to this technique as *vertical compression*.

Vertical compression is implemented by storing 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. Figure 2 depicts two tries, one with and one without vertical compression. We observe that nodes with no siblings, color coded red, are eliminated and their partial key is appended to the parent's prefix.



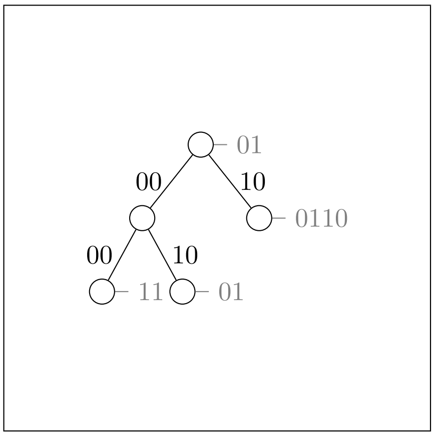


Figure 2: Tries with span $s = 2$ storing keys "01000011", "01000110" and "01100100". The trie on the right incorporates vertical compression. Red nodes indicate nodes which get eliminated under vertical compression. Gray strings represent the value of the prefix variable.

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. Space is allocated for pointers which keep references to child nodes, even if they are unused.

In order to reduce the 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 a minimal amount of 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).

We now describe the structure of each of the four configurations. Figure 3 shows the space consumption for each inner node type. Note that h is equal to the header's size. Figure 4 illustrates the state of a node for each node type, when storing the partial keys 65, 82, 84 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 which identify children

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	$h + 256 \cdot 8 = h + 2048$

Figure 3: Space consumption for each inner node type. h is equal to the size of the header.

of that node. The "children" array, holds pointers to the child nodes. Partial keys and pointers are stored at corresponding positions 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" and a 48-element array called "children". 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. This node can be compared to virtual memory, since the address space (256 addresses) is wider than the actual available storage space (48 slots).

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.

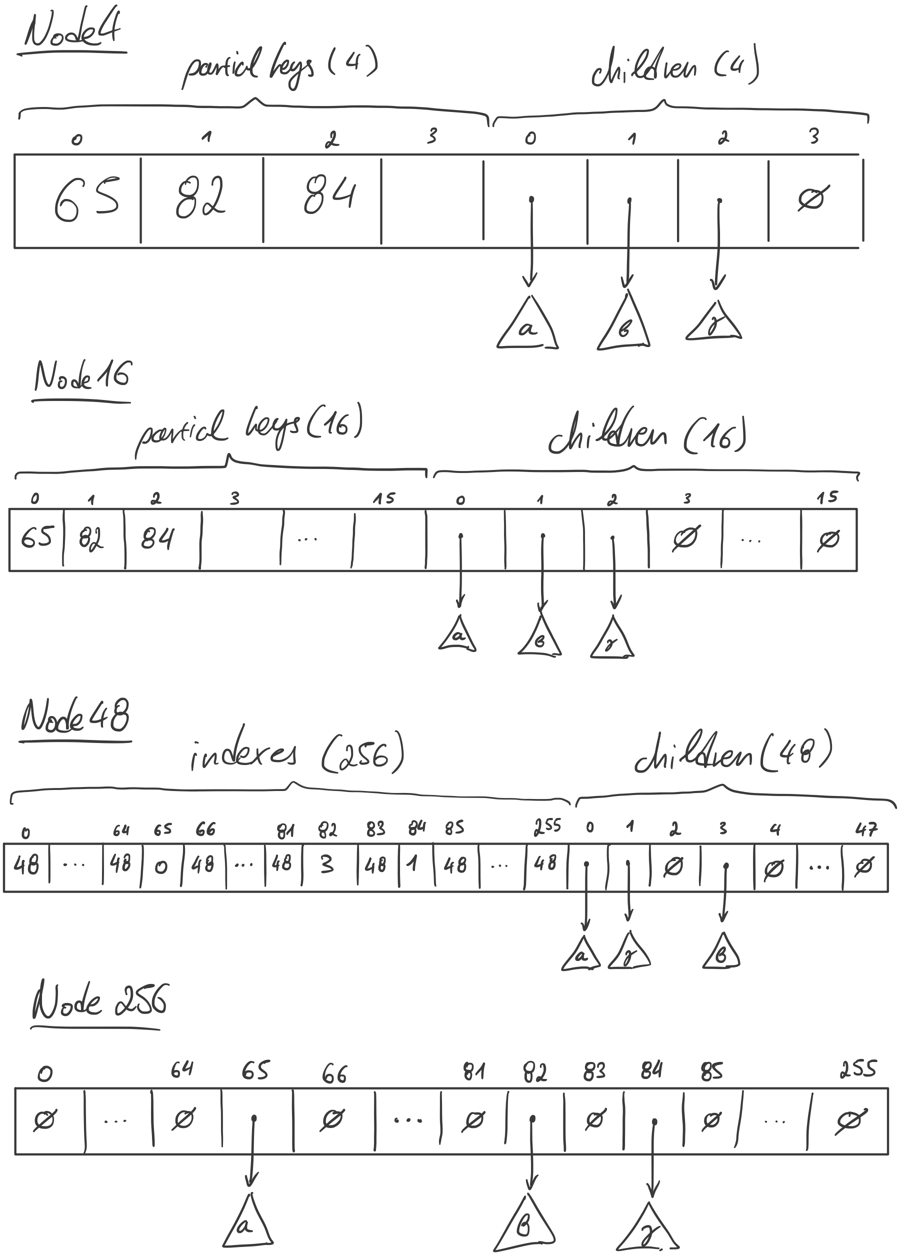


Figure 4: When horizontal compression is applied, a node is in one of four configurations, namely `Node4`, `Node16`, `Node48` and `Node256`. Each of the four configurations is optimized for a different number of child nodes. We store the partial keys 65, 82, 84 and their corresponding child nodes α, 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. The algorithms mentioned below are based on the implementations presented by Leis et al. [2]. Note that our implementation uses *single-value leaves* (c.f. Leis et al. [2]), i.e. values are stored using an additional leaf node type, conveniently called `Node0`, which stores one value. Additionally, we utilize *pessimistic* vertical compression, i.e. each inner node stores the entire compressed path inside the prefix variable using a variable length partial key vector. During traversal, prefix is compared to the search key. We present our implementations below.

4.1 Point Query

The code fragment shows the implementation of a point query on ART in C++. Method `get` accepts a key 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 7-8 we declare and initialize a pointer, `cur`, which references the current node during tree traversal, and an integer, `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, we return `null`.

In lines 13-19 we check if a prefix mismatch occurs. This step is required because of vertical compression (c.f. Section 3.1). 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. If a prefix mismatch is discovered, `null` is returned.

Lines 20-24 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. `depth` is increased to account for the nodes merged due to vertical compression. We lookup the next child node, which is assigned to `cur`.

```cpp
template <class T> class art {
public:
  /**
   * Finds the value associated with the given key.
   */
  T *get(const key_type &key) const {
    node<T> *cur = root_;
    int depth = 0;
    while (cur != nullptr) {
      const int prefix_len =
        cur->get_prefix().length();
      const bool is_prefix_match =
        cur->check_prefix(key, depth) == prefix_len;
      if (!is_prefix_match) {
        return nullptr;
      }
      const bool is_exact_match =
        prefix_len == key.length() - depth;
      if (is_exact_match) {
        return cur->get_value();
      }
      depth += prefix_len;
      node<T> **next = cur->find_child(key[depth]);
      cur = next != nullptr ? *next : nullptr;
      ++depth;
    }
    return nullptr;
  }

  /* ... */
private:
  node<T> *root_ = nullptr;
}
```

Figure 5: Point query implemented in C++.

4.2 Deletion

Figure 6 presents our implementation of key deletion on ART in C++. During deletion, the leaf node is removed from an inner node, which is shrunk if necessary. If the leaf to remove only has one sibling, vertical compression is applied to the parent, effectively deleting it.

In lines 7-11, 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. Variables `depth` and `key_len` hold the depth of the current node and length of the key, respectively.

Next, we loop until `cur` is a `null` pointer. During each iteration we define the following variables:

- `prefix` compressed path of the current node.
- `prefix_len` length of the compressed path, i.e. the number of nodes vertically compressed.
- `is_prefix_match` evaluates to true iff `prefix` is a prefix of the search key, starting from position `depth` (e.g. if `prefix` = "aa" and `key` = "aaa", then `is_prefix_match` $\iff$ `depth` $> 0 \wedge$ `depth` < 2).
- `is_exact_match` evaluates to true iff `prefix` matches the search key, starting from position `depth`. (e.g. if `prefix` = "aa" and `key` = "aaa", then `is_exact_match` $\iff$ `depth` $= 1$).

If there is a prefix mismatch, i.e. `is_prefix_match` evaluates to false, the search key is not found and a `null` pointer is returned. If there is a exact match i.e. `is_exact_match` evaluates to true, we found the node to delete, otherwise we continue traversing down the tree.

After the node to delete has been found, we first extract its associated value and determine its number of siblings. It must have at least one sibling, unless the node to delete is the root node.

If the node to delete has exactly one sibling (lines 26-40), we replace the parent node with the sibling, append to the sibling's prefix the parent's prefix and the sibling's partial key and then delete the `cur` and `par`. We also delete `par` because of vertical compression.

If the node to delete has more than one sibling (lines 40-42), then no additional action is required besides deleting `cur`.

Lines 44-46 check if the parent node is undefull, i.e. requires shrinking (c.f. Section 3.2).

Finally, the value associated with the deleted node is returned in case the procedure callee has to free resources.

5 Benchmarks

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.

```cpp
template <class T> class art {
public:
  /**
   * Deletes the the given key and returns its associated value.
   */
  T *del(const key_type &key) {
    node<T> **cur = &root_;
    partial_key_type cur_partial_key = 0;
    node<T> **par = nullptr;
    int depth = 0;
    const int key_len = key.length();

    while (*cur != nullptr) {
      const key_type prefix = (*cur)->get_prefix();
      const int prefix_len = prefix.length();
      const bool is_prefix_match = prefix_len == (*cur)->check_prefix(key, depth);
      if (!is_prefix_match) {
        return nullptr;
      }
      const bool is_exact_match = key_len - (depth + prefix_len) == 0;

      if (is_exact_match) {
        T *value = (*cur)->get_value();
        const int n_siblings = par != nullptr ? (*par)->get_n_children() - 1 : 0;

        if (n_siblings == 1) {
          /* find sibling and compress */
          partial_key_type 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);
          const key_type new_sibling_prefix = (*par)->get_prefix() +
                                              key_type(1, sibling_partial_key) +
                                              sibling->get_prefix();
          sibling->set_prefix(new_sibling_prefix);
          delete (*par);
          *par = sibling;

        } else if (n_siblings > 1) {
          (*par)->del_child(cur_partial_key);
        }

        if (par != nullptr && (*par)->is_underfull()) {
          *par = (*par)->shrink();
        }
        delete(*cur);
        *cur = nullptr;
        return value;
      }
      cur_partial_key = key[depth + prefix_len];
      par = cur;
      cur = (*cur)->find_child(cur_partial_key);
      depth += prefix_len + 1;
    }
    return nullptr;
  }

  /* ... */
private:
  node<T> *root_ = nullptr;
}
```

Figure 6: Key deletion implemented in C++.