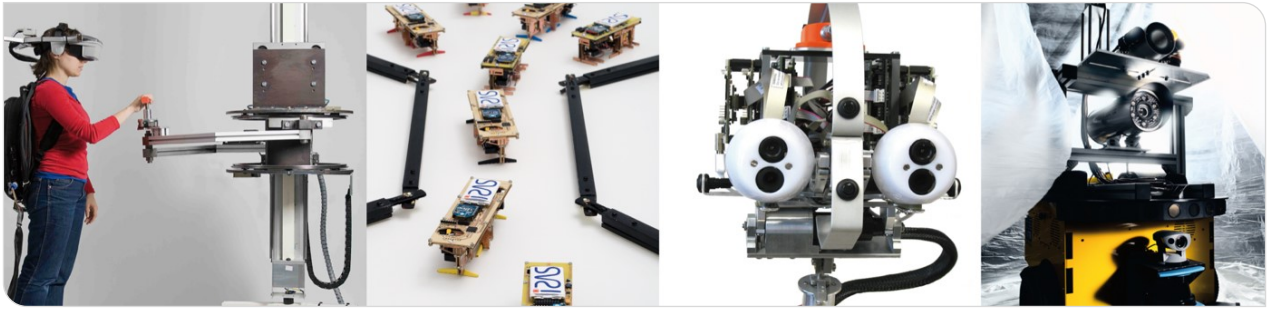
Karlsruhe Institute of Technology

confertus: a dynamic bitvector implementation in rust

Felix Karg | 25. Juli 2022



Automatically Generated documentation

Click or press 'S' to search, '?' for more options...

Struct confertus::dynamic_vector::DynamicBitVector

source · [-]

```
pub struct DynamicBitVector {
    pub root: usize,
    pub nodes: Vec<Node>,
    pub leafs: Vec<Leaf>,
}
```

[-] Implementation of Dynamic Bit Vector using self-balancing AVL tree.

Instance bit size: 56 bytes = 448 (not included: bit sizes of instances in Vector structures)

Fields

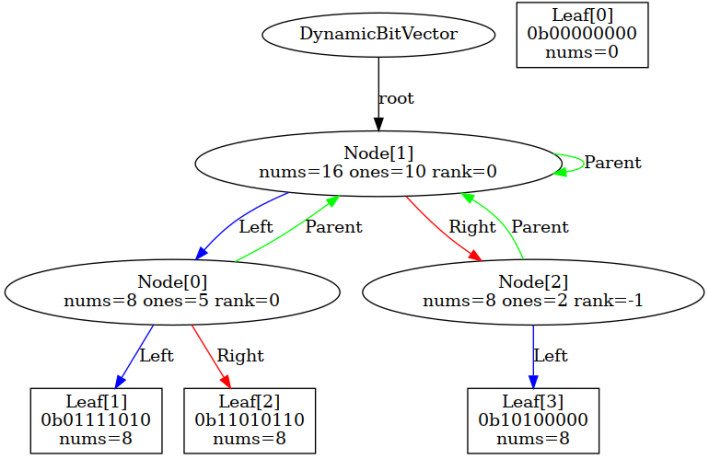
`root: usize`
index to root `Node`, 8 bytes

`nodes: Vec<Node>`
Vector containing `Node`, 24 bytes

`leafs: Vec<Leaf>`
Vector containing `Leaf`, 24 bytes

Documentation can be found at `https://fkarg.me/confertus`

Structure Visualization with dotviz



DynamicBitVector

```
+-----------------+
|DynamicBitVector |
+-----------------+
|root: usize      |
|nodes: Vec<Node> |
|leafs: Vec<Leaf> |
+-----------------+
```

```rust
/// Implementation of Dynamic Bit Vector based on
/// a self-balancing [AVL tree]. ...
#[derive(Debug, PartialEq, Clone, Default)]
pub struct DynamicBitVector {
    /// index to root [`Node`]
    pub root: usize, // 8 bytes
    // positively indexed, usize
    /// Vector containing [`Node`]
    pub nodes: Vec<Node>, // 24 bytes
    // negatively indexed, isize
    /// Vector containing [`Leaf`]
    pub leafs: Vec<Leaf>, // 24 bytes
}
```

Node

```
+---------------------+
|Node                 |
+---------------------+
|parent: Option<usize>|
|left: Option<isize>  |
|right: Option<isize> |
|nums: usize          |
|ones: usize          |
|rank: i8             |
+---------------------+
```

```rust
pub struct Node {
    /// index of parent Node
    pub parent: Option<usize>, // 8 bytes + 1bit
    /// left side subtree child index
    pub left: Option<isize>, // 8 bytes + 1bit
    /// right side subtree child index
    pub right: Option<isize>, // 8 bytes + 1bit
    /// number of filled bits on the left subtree
    pub nums: usize, // 8 bytes
    /// number of ones on the left subtree
    pub ones: usize, // 8 bytes
    /// difference of height between left and
    /// right subtree. Valid values are -1, 0, 1
    pub rank: i8, // 1 byte
}
```

Leaf

```
+-------------+
|Leaf         |
+-------------+
|parent: usize|
|value: u128  |
|nums: u8     |
+-------------+
```

```rust
/// Leaf element of [`DynamicBitVector`]
/// ...
#[derive(PartialEq, Clone, Default)]
pub struct Leaf {
    /// reference to parent [`Node`]
    pub parent: usize, // 8 bytes
    /// container for actual bit values
    pub value: u128, // 16 bytes
    /// number of bits used in `value`-container
    pub nums: u8, // 1 byte
    // realistically below u128::BITS,
    // so u8::MAX = 255 is sufficient
}
```

Implementation of rank and select for u64

```rust
use core::arch::x86_64::{_pdep_u64, _popcnt64, _tzcnt_u64};

#[inline]
#[cfg(all(target_arch = "x86_64", target_feature = "bmi1", target_feature = "bmi2"))]
unsafe fn select_internal(&self, bit: bool, n: usize) -> usize {
    _tzcnt_u64(_pdep_u64(1 << n, if bit { *self } else { !self })) as usize
}

#[inline]
#[cfg(target_arch = "x86_64")]
unsafe fn rank_internal(&self, bit: bool, index: usize) -> usize {
    _popcnt64({if bit {*self} else {!self}}.overflowing_shl(u64::BITS - index as u32).0 as i64)
        as usize
}
```

(I also have respective non-architecture-specific fallbacks for both)

Index for DynamicBitVector

I have **two** implementations for indexing!

```
fn index(&DynamicBitVector self, index: usize) -> Node { ... }
fn index(&DynamicBitVector self, index: isize) -> Leaf { ... }
```

This way, I can do type-safe indexing:

- `self[+3 as usize] -> Node`
- `self[+3 as isize] -> Leaf`
- `self[-3 as isize] -> Leaf`

Unified Traversal: Apply

```rust
pub fn apply<T>(&mut self, mut f: FnMut(...) -> T, index: usize) -> T {
    self.apply_node(self.root, f, index)
}
fn apply_node<T>(&mut self, node: usize, mut f: FnMut(...) -> T, index: usize) -> T {
    if self[node].nums <= index {
        let right_id = self[node].right.unwrap();
        if right_id >= 0 {
            self.apply_node(right_id as usize, f, index - self[node].nums)
        } else {
            f(self, right_id, index - self[node].nums)
        }
    } else {
        let left_id = self[node].left.unwrap();
        if left_id >= 0 {
            self.apply_node(left_id as usize, f, index)
        } else {
            f(self, left_id, index)
        }
    }
}
```

Beispielimplementation: flip

```rust
impl DynamicBitVector {
    ...

    #[inline]
    pub fn flip(&mut self, index: usize) {
        let leaf = self.apply(Self::flip_leaf, index);
        self.update_left_values(self[leaf].parent, leaf);
    }

    #[inline]
    fn flip_leaf(&mut self, leaf: isize, index: usize) -> isize {
        self[leaf].flip(index);
        leaf
    }
}
```

LOC-Stats

```
$ tokei
===============================================================================
 Language            Files        Lines         Code     Comments       Blanks
===============================================================================
 Rust                   18         3510         2740          389          381
 |- Markdown            15          434            0          389           45
 (Total)                           3944         2740          778          426
===============================================================================
```

Time invested

Does not include time needed to create this presentation

Results

Unfortunately, I didn't fully get it to run until the end.

```
$ RUSTFLAGS="-C target-cpu=native" cargo run --release bv input/example_bv_10k.txt out.txt
   Compiling confertus v0.1.0 (/home/pars/Coding/confertus)
    Finished release [optimized] target(s) in 4.11s
     Running `target/release/confertus bv input/example_bv_10k.txt out.txt`
index 7 out of bounds for 5
Error: "Leaf.insert: Index out of bounds `index > self.nums`"
```

Turns out, it doesn't uphold all invariances regarding `nums` and `ones`, causing them to drift – which eventually causes frustrating errors like this.

What are your Questions?

Abstraction: Static Bit Vector

```rust
/// Functions associated with static bit vectors. ...
pub trait StaticBitVec {
    ...

    /// Return number of on-bits in container or on left half for tree-based elements
    fn ones(&self) -> usize;

    /// Access bit value at position `index`
    fn access(&self, index: usize) -> bool;

    /// Returns number of `bit`-values up to `index` in container
    fn rank(&self, bit: bool, index: usize) -> usize;

    /// Return index of `n`-th `bit`-value in container
    fn select(&self, bit: bool, n: usize) -> usize;
}
```

Abstractions

Abstraction: Dynamic Bit Vector

```rust
/// Functions associated with dynamic bit vectors.
pub trait DynBitVec: StaticBitVec {
    /// Insert `bit` at position `index` in underlying container
    fn insert(&mut self, index: usize, bit: bool) -> Result<(), &'static str>;

    /// Remove bit value at position `index`
    fn delete(&mut self, index: usize) -> Result<(), &'static str>;

    /// Flip bit at position `index`, updates `ones` and `num` values accordingly
    fn flip(&mut self, index: usize);

    /// Return used capacity of underlying container
    fn nums(&self) -> usize;
}
```

Abstractions