RRB-Trees: Efficient Concatenation of Immutable Vectors

Elias Khoury

12th January 2017

1 Abstract

Vectors are a common datastructure used in everyday programming to represent an indexed sequence of data. The purpose of this report is to present a novel implementation of vectors which focuses on making the concatenation operation as efficient as possible. This is achieved by using a balanced tree structure to store the elements of the vector, the main challenge of this method is to maintain the balance of the tree across multiple concatenations. SOMETHING ABOUT BATCH UPDATE

2 Typical Uses

A typical example of when efficient concatenation is useful is in the area of server-side dynamic code generation, where either SQL scripts need to be generated on the fly, or HTML web pages for users.

3 Overview of the Datastructure

In this implementation Vectors are represented using a tree structure, with each leaf being an element in the sequence. Indexing is done by traversing the tree in order to reach the desired leaf node, concatenation is done by taking two trees and joining them into a new tree containing the leaves from both trees. With that in mind it is important to make sure that the speed of the concatenation operation should not impact the accessibility of the leaves since that will negatively impact the speed of other operations.

One way to reliably access a desired leaf node is by utilising a radix search. For a balanced tree with branching factor m, it is possible to calculate the number of leaf nodes at each level using the formula: RADIX SEARCH FORMULA However, this requires a consistent branching factor to be maintained to ensure the reliability of each leaf's position. The issue with having the branching factor as an invariant is that it constrains the join operation to a linear copy of each element in both trees into a new tree, since neither tree can be modified to accommodate the new leaves. Since the focus is on making the concatenation operation as quick as possible, a linear copy is not ideal.

Still focusing on indexing operation, it is possible to modify the tree structure to allow some flexibility with concatenation, at a slight cost of finding leaves. Relaxed Radix search is a method which allows for a variable branching factor while still ensuring random access to leaf nodes. The overhead this adds is a consequence of the fact that no sub tree is perfectly balanced any more. Therefore, to ensure the location of a leaf node, for any unbalanced node, the number of leaves in each child needs to be stored.

- Tree structure - Radix Search - Relaxed Search - Invariant - Branching Factor - Max Depth (for 32 bit index) - Balance and relation to other operations

4 Main Features

- Indexing

- Take, Drop

- Concat

- First

- Last+

5 Analysis of Runtime

The task of concatenating two trees can be done in a multitude of ways. However, how the tree is structured can restrict which methods can be applied. For example, given two trees with a constant branching factor, the only way to get a perfect concatenation of the trees is to do a linear copy of each element into a new tree. Likewise, if speed is the only concern then an O(1) solution is possibly by simply making a new root and joining that to the two new trees. For this application both solutions have their disadvantages, firstly a linear cost is far too slow for large scale applications, however simply making a new root each time is dangerous because after multiple concatenations the tree will simply degenerate into a linked list. This will negatively impact other operations like indexing as they will become constant time. The balance that needs to be struck is a method which is fast, but doesn't degenerate the data structure.

Relaxed Radix Structure

The relaxed radix structure is a variant of the Radix tree which allows for the possibility of nodes not being full. As such, the height is no longer bounded by log(n) like in a typical balanced tree, but by an additional invariant based on the minimum and maximum branching factor. Given two branching factors: b_{min}, b_{max}, the height of the tree is bounded by the two branching factors, giving b_{min}, b_{max}. SHOW CALCULATIONS

Given this new structure, it is no longer possible to calculate how many leaves are in each subtree, since that subtree might be unbalanced. In that case, any unbalanced node needs to have some metadata attached to it indicating how many leaves are in each subtree. This adds a constant time factor to the radix search. The complexity of finding a leaf is now $O(m * Log(n))$ where m is a constant cost associated with an unbalanced node.

The extra linear

Concatenation

The concatenation operation starts at the leftmost branch of the right tree and the rightmost branch of the left tree. It takes those two subtrees and creates a new tree containing the leaves from both subtrees. In order to maintain balance, the subtrees are not simply copied, but they are broken down to their respective leaves and then reconstructed.

Once this is done, the algorithm moves up the two trees and joins the nodes at each level, each time creating a dummy root of size two which will be deleted in the subsequent iterations. At each level it is important to rebalance the nodes. This is done by taking the children of the nodes being merged and copying them to the new dummy root. Thanks to lazy evaluation and sharing, this copying operation should have little overhead.

The next few diagrams show how the algorithm moves up the tree. This variant of the algorithms is an alteration of the one presented in the original paper[Bagwell and Rompf, 2011], Stucki et al. [2015] present an alternative method which focuses on the practicality of the datastructure as a whole. In the original paper, the method described did not prioritise each node having exactly m children. Instead, tree balance was the main factor taken into account, which results in trees containing nodes with fewer than m children. Although the method presented in the original paper is serveral times quicker than the chosen method, it would often result in trees that were very tall, this directly affects the speed of the indexing operations which is bounded by the depth of the tree. Therefore, in order to have a practical, general purpose data structure, it is important to compromise the speed of the concatenation operation in order to not negatively effect the other operations.

5.1 Branching Factor

Choosing the branching factor, m is a crucial factor for calculating the runtime of each operation. As stated, accessing leaf nodes requires a full traversal of the tree, for n elements in a perfectly balanced tree the runtime is bounded by $O(log_m n)$. There is a small caveat with a tree structure which allows us to exploit the branching factor in order to make tree traversal trivial. In the two papers presented, a branching factor of 32 was chosen, one good reason is that this allows cache sizes to be exploited since each node can fit perfectly in a cache line[Stucki et al., 2015] (although this assumes the datastructure itself has no overhead which may not be the case for high level languages). Although having a high branching factor has another, much more obvious effect on efficiency. With a branching factor of 32, storing 10,000,000 items in the tree will only create a tree of depth 5. Infact, considering the case of indexing into the vector using a 32 bit integer, the maximum value accessible is 4294967296 (2^{32}) meaning that if that were the size of the entire vector then the tree would still only be of depth 7. With the assumption that indexing is bounded by a 32 bit integer, any operation which runs in $O(log_m(n)$ is bounded by the maximum depth of the tree which is a constant of 7, effectively making the operation run in constant time.

	RRB-Vector	With m = 32
indexing	log_m	eC
update	aC	eC
insert ends	$m \times log_m$	aC
concat	$m^2 \times log_m$	L v.s. eC
split	$m \times log_m$	eC

Empirical tests have also shown that a branching factor of 32 is also the most ideal. The figure below shows the index and update times for several branching factors, the choice to be made is a trade off between the two operations, the values of m with the best ratios are 8,16,32 meaning they all lead to good solutions, however 32 was chosen simply because it leads to slightly faster index times which is worth the cost of the slower updates.

6 Chosen Language

RRB-Trees are a purely functional datastructure, they rely heavily on the concept of sharing and lazy evaluation when copying and manipulating segments of the tree. In order to take advantage of these features, it would be ideal to use a functional language to write the implementation. The concepts behind RRB-Trees emerged from research done on developing Scala and OCaml, as such it would be interesting to compare the difference with an implementation in a similar functional language.

Haskell makes a good candidate for such a comparison. - Lazy evaluation therefore efficient copying - Used in Facebook - Strongly typed, less overhead - Fast, as fast as C - Concise code, easy to present - Never been done before

7 Implementation

Tree Structure

The tree is represented using a Haskell datastructure, each node contains its height in the tree, along with a list of its children and the sizes metadata should the node be unbalanced. The leaves themselves contain the elements of the vector, and since leaves are always balanced they have no size metadata.

```haskell
branchingFactor :: Int
branchingFactor = 32 -- Branching factor of tree

type Height = Int
type Sizes = [Int]

data Tree a = Node Height [Tree a] Sizes | Leaf [a]
   deriving (Show)
```

Indexing

Indexing into the tree is performed using the relaxed radix search algorithm. For efficiency, the function alternates to using radix search if it encounters a balanced subtree. How this is done is by pattern matching on the sizes metadata. If there is no metadata then the distribution of leaves in each subtree must be equal. In this case it is possible to calculate which subtree the desired leaf is in using the formula: $\lfloor i/(m^l) \rfloor \ \% \ m$ (where l is the current height of the tree). The function recurses on the desired subtree, which means that the search needs to be localised on that specific subtree. In order to do this, the index of the desired leaf (iOffset) is calculated relative to the chosen subtree by taking the original index and subtracting from it all the leaves that belong in the subtrees before the chosen one.

```
relaxedSearch :: Int → Tree a → a
relaxedSearch i (Leaf as) = as !! (i `mod` branchingFactor)
relaxedSearch i (Node l ts []) = relaxedSearch (i - iOffset) (ts !! nodeCount)
             where nodeCount = (i `div` (branchingFactor ^ l)) `mod` branchingFactor
                   iOffset  = (branchingFactor ^ l) * nodeCount
relaxedSearch i (Node l ts ss) = relaxedSearch (i - iOffset) (ts !! subTree)
             where iOffset  = if subTree == 0 then 0 else ss !! (subTree - 1)
                   subTree  = length (takeWhile (< i) ss)
```

In the case where the node is unbalanced, the sizes metadata needs to be accessed, this keeps a running total of all the leaves in each subtree. Firstly the correct subtree is located by indexing into sizes list until an element is found where the total number of leaves so far exceeds the desired index. The previous value in the sizes list contains the number of leaves that come before the desired subtree, this is used to calculate the new index, and the function recurses on the desired subtree.

Take/Drop

Concatenation

The concatenation function begins by merging the leftmost branch of the right tree with the rightmost branch of the left tree into a new tree of height two. The leaves are redistributed to ensure that the left branch of the new tree is full. if there are not enough leaves to fill both branches, then the tree is unbalanced and the sizes need to be calculated and stored.

```
mergeEnds :: Tree a → Tree a → Tree a
mergeEnds t1 t2 = Node 2 [Node 1 leftChild [], Node 1 rightChild []] (if lSize == rSize then [] else
    ↪ [lSize,rSize])
      where Node _ leftLeaves _ = getRightChild t1 1
            Node _ rightLeaves _ = getLeftChild t2 1
            leaves       = leftLeaves + + rightLeaves
            leftChild    = take branchingFactor leaves
            rightChild   = drop branchingFactor leaves
            [lSize,rSize] = map (length.stripLeaf) [last leftChild,last rightChild]
            stripLeaf    = \ (Leaf a) → a
```

A similar method is done for the subsequent heights. At each level of the merge process, the mergeAt function is called. This takes the appropriate subtrees from the left and right tree for some arbitrary height n, and merges them into the new tree. The merge process is done my the mergeRebalance function, its purpose is to ensure that when merging into the middle tree, that as many nodes as possible are filled, leaving only the rightmost node to possible having fewer children than the branching factor.

```
mergeAt :: Int → (Tree a,Tree a,Tree a) → (Tree a,Tree a,Tree a)
mergeAt n (t1,t2,t3) = (leftTree,mid,rightTree)
             where leftTree = dropRightChild t1 n
                   leftChild = getRightChild t1 n
                   mid       = mergeRebalance (leftChild,t2,rightChild)
                   rightTree = dropLeftChild t3 n
                   rightChild = getLeftChild t3 n
```

The nodes from all three trees need to be merged in a way that maintains the balance of the tree. In order to do this, each node is replaced by its children, and those children are replaced by their children. Then the parent nodes are reconstructed in a way where each new parent is now full with the exception of possibly the last parent. At this stage the sizes metadata needs to also be recalculated.

```
mergeNodes :: [Tree a] → [Tree a]
mergeNodes ns = case firstMerge of
                [] → []
                ((Leaf _):ns) → balanceLeaves secondMerge
                ((Node l _ _):ns) → map (\ts → Node l ts (computeSizes ts l)) (collect
                    ↪ branchingFactor secondMerge)
              where firstMerge = concat (map getKids ns)
                    secondMerge = concat (map getKids firstMerge)
```

The mergeRebalance function takes the three trees that need to be merged and merges them using the mergedNodes function. It then creates a dummy node with two children containing the output of mergedNodes.

```
mergeRebalance :: (Tree a,Tree a,Tree a) → Tree a
mergeRebalance (t1,t2,t3) = Node (l + 1) [left,right] (computeSizes [left,right] (l+1))
                  where mergedNodes = mergeNodes[t1,t2,t3]
                        Node l _ _ = t1
                        leftChildren = (take branchingFactor mergedNodes)
                        rightChildren = (drop branchingFactor mergedNodes)
                        left = Node l leftChildren (computeSizes leftChildren l)
                        right = Node l rightChildren (computeSizes rightChildren l)
```

The whole algorithm is tied together by the mergeAt function. Given an input of a particular height of a tree, the mergeAt function will firstly remove the appropriate branches at that level from the left and right tree. It will then take those branches that were removed and feed them into the mergeRebalance function, along with the current state of the middle tree. The result is a new version of the middle tree with the branches merged correctly. By calling the mergeAt function for each level of the trees that are being merged, the middle tree will slowly be built up until it contains all leaf nodes from both trees.

```
mergeAt :: Int → (Tree a,Tree a,Tree a) → (Tree a,Tree a,Tree a)
mergeAt n (t1,t2,t3) = (leftTree,mid,rightTree)
              where leftTree = dropRightChild t1 n
                    leftChild = getRightChild t1 n
                    mid       = mergeRebalance (leftChild,t2,rightChild)
                    rightTree = dropLeftChild t3 n
                    rightChild = getLeftChild t3 n
```

8 Empirical Evaluation

9 Conclusion

References

P. Bagwell and T. Rompf. Rrb-trees: Efficient immutable vectors. Technical report, 2011.

N. Stucki, T. Rompf, V. Ureche, and P. Bagwell. Rrb vector: a practical general purpose immutable sequence. In *ACM SIGPLAN Notices*, volume 50, pages 342–354. ACM, 2015.