# Persistent Data Structures

Andrew Stankevich

September 28, 2019

## 1 Persistence

Main idea of persistence is to keep older versions of data structures to have access to them later.

Levels of persistence:

- Partial — read only access to older versions.
- Full — read access to older versions, but can also modify them to create new versions, the versions form a tree.
- Confluent — read and write access to older versions, also can combine different versions of the same structure in one operation (for example, merge treaps).
- Purely functional — all fields are unmodifiable, automatically persistent at all levels (but take care about time complexity).

Note: amortized analysis of time complexity and persistence are not compatible. In amortized analysis you check the average time of some operation. Expensive operations are compensated by lots of cheap ones. In persistent version you can do expensive operation again and again.

## 2 Node Copying

Main idea to make data structure persistent is not to destroy nor modify older versions. One way to do it is *node copying*.

Consider some data structure composed of nodes. If you need to change some value, create a new node instead, copy all preserved data, and put new values to the modified fields.

There is a problem, however, that some other nodes could have links to the modified one, that now has been copied. Their links must be recursively updated, so actualy changing a value in a node can cascade to the whole set of nodes copied. Actually, all nodes, such that the modified node can be readched from them, must be copied.

Sometimes it is OK (stacks, segment trees), sometimes it doesn't work (queues). Let us move to examples.

## 3 Persistent stacks

Stack data structure.

- `push(`$x$`)` — add value $x$ to the top of the stack.
- `pop()` — remove the topmost value from the stack.

In persistent setting we also to specify which version should be modified. This can be achieved by giving additional argument to all operations.

```
int push(int v, int x) // v - version, x - value
                       // return new version
    int r = create new node
    nodes[r].value = x
    nodes[r].next = v
    return r

int pop(int v) // v - version, return new version
    return nodes[v].next
```

Alternative: use methods, but all methods return new versions of the structure. Pseudocode below uses Java-style classes, in C++ add "`*`" to most class references to work with pointers.

```
class Stack
    int value
    Stack next

    Stack push(int x)
        return new Stack(value = x, next = this)

    Stack pop()
        return next
```

Note: `pop` operation might want to return the popped value. Need to return `pair<new version, returned value>` in this case.

Note: versions create a tree, this tree is isomorphic to a tree formed by `next` references interpreted as `parent` references.

# 4 Persistent queues

We cannot directly use node copying here. No matter which direction we direct links in a list that stores the queue, all nodes can reach either head, or tail. So either for `pop` or for `push` the whole queue would need to be copied.

Nice trick to implement a queue using two stacks also doesn't work for persistent versions, because it uses amortized analysis.

The following structure is due to Tarjan et al. Consider a tree-like structure, each node has two buffers of size 1 that store the first and the last element of the queue, and also a child element that recursively stores the same sturcture, but for pairs of elements.

```
class Queue<T>
    T front
    Queue<Pair<T, T>> child
    T back

    Queue<T> push(T x)
        if back == NIL then
            return new Queue(
                front = this.front
                child = this.child
                back = x
            )
        else
            return new Queue(
                front = this.front
                child = this.child.push( (back, x) )
                back = NIL
            )


    pair<T, Queue<T>> pop()
        if front != NIL then
            return (front, new Queue(
                front = NIL,
                child = this.child,
                back = this.back
            ) )
        else if child = NIL then
            return (back, NIL)
        else
            ((a, b), new_child) = child.pop()
            return (a, new Queue(
```

```
            front = b,
            child = new_child,
            back = this.back
        ))
```

Note: you cannot implement data structure like this in C++, because of infinite recursion for templates (they are resolved at compile time). See appendix to learn how to fix it.

Another way is to use general approach to make any array-based data structure persistent, see below for details.

## 5 Persistent Segment Trees

Here we can use node copying if we store segment tree as a rooted tree, vertices have links to their children, but not to their parents. In this case $O(\log n)$ vertices can reach a vertex, so they should be copied in any modification operation.

```
int set(int v, int l, int r, int p, int nx)
    if l == r - 1 then
        nv = create new vertex
        tree[nv] = nx
        return nv
    m = (l + r) / 2
    if p < m then
        nl = set(left[v], l, m, p, nx)
        nv = create new vertex(left = nl, other copy from v)
        return nv
    else
        nr = set(right[v], m, r, p, nx)
        nv = create new vertex(right = nr, other copy from v)
        return nv
```

Group operations that modify a segment can also be supported. When you modify a vertex, either in `push` operation, or directly setting some field values, create a node copy with new values, and also copy all nodes on the path from the root.

## 6 Persistent Arrays

Another usage of persistent segment trees is implementing persistent array. We ignore segment operation, and just store values in leaves, internal nodes contain no information. Assignment of an element is just as change operation in segment tree.

Persistent arrays allow making persistent almost any data structure that has array based implementation. It gives logarithmic slowdown in time complexity, and requires $O(\log n)$ memory per array assignment, but if the cost is acceptable, the method is quite universal.

# 7 Persistent Treaps

In a similar to segment trees way we can make treaps persistent.

However, a special care should be taken in order to make them confluently persistent. If we try to merge a treap to itself, the $y$-keys are the same. So it ruins the randomization, and the treap can degrade to a bamboo.

To avoid it, let us use RBST (randomized binary search trees) instead of treaps. In RBST when merging two trees we use the following algorithm to choose which node would become the root of the combined tree. Let one tree have size $a$, and another one have size $b$. Generate a random number $r$ from 0 to $a+b-1$, inclusive. If $r < a$, the root of the first tree becomes the combined root, if $a \le r < a+b$, the root of the second one does.

The same analysis as in treaps can be applied to prove that the expected depth of a node, as well as the expected height of the tree, is $O(\log n)$. The only property needed is the fact that each vertex has equal chance to be the root of the tree, and it can easily be shown by induction in case of RBST.

Thus we get the following merge operation.

```
Tree merge(Tree a, Tree b)
    if a == NIL then return b
    if b == NIL then return a
    r = random(0..size(a) + size(b) - 1)
    if r < size(a) then
        res = Tree(right = merge(right(a), b),
            copy other from a)
        return res
    else
        res = Tree(left = merge(a, left(b)),
            copy other from b)
        return res
```

Split doesn't use $y(v)$, so it is the same as in treaps, with standard modification to make things persistent.

```
pair<Tree, Tree> split(Tree a, int k) // size(first) = k
    if a == NIL then return (NIL, NIL) // assert k == 0
    if size(left(a)) >= k then
        (u, v) = split_k(left(a), k)
        return (u, Tree(left = v, copy other from a))
```

```
    else
        (u, v) = split(right(a), k - size(left(a)) - 1)
        return (Tree(right = u, copy other from a), v)
```

# 8 Using Persistence to Convert Offline to Online

One way to use persistence is the following. Consider a solution to some problem in offline setting that uses some data structure. Usual usage of offline setting is to create a common list of *events* that contains both some input objects, and queries, sort them by some key, and process one after another. Usually processing events that correspond to input objects modifies the contents of the data structure, and processing events that correspond to queries makes some requests to it.

If the queries are not known beforehand, it is impossible to sort events and queries together. Let us instead take a persistent version of the data structure and use it. Take all input object and sort them. Now process events, as if there were no queries, but save the version of the data structure at every possible query location. Now to process the online query just make request to the corresponding version of the data structure.

Example: consider the problem of counting given points in query rectangles. Events: points, and vertical sides of rectangles, sort by $x$ coordinate. Scanline with segment tree for sum operation, indices in segment tree correspond to $y$ coordinates of the given points, compressed. Value in segment tree is the number of points with such $y$ coordinate.

Processing point: add 1 to the corresponding element in the segment tree. Processing left side of rectangle: subtract sum of elements on the corresponding segment from the answer for this rectangle. Processing right side of rectangle: add sum of elements on the corresponding segment to the answer for this rectangle.

Moving to online setting: use persistent segment tree, process points as before, store version of the segment tree after processing every $x$ coordinate. Query: extract versions of the segment tree for left and right side of the rectangle to make the required queries.

# Appendix

## Level templates in C++

Need to introduce maximum level like this

```
template<typename T, int level>
struct Queue {
    T front;
    Queue<pair<T, T>, level - 1>* child;
```

```
        T back;
        ...
    };
```

and introduce bottom level like this:

```
    template<typename T> struct Queue<T, 0> {
        Queue() {
            // throw error here, too many levels
        }
    };
```

Now you can create an istance:

```
int main() {
    Queue<int, 30>* x;
}
```

Not so nice, yeah...