1 List-order DS [1]

List-order is a hybrid abstract data structure combining advatages of linked lists (quick insertion anywhere) and arrays (we can compare indices quickly).

This is also called *order maintenance problem.*

Operations:

- INSERTION after a given element in $\mathcal{O}(1)$ amortized time,
- PREDECESSOR and SUCCESSOR of an element in $\mathcal{O}(1)$ time, and
- finding the ORDERING of two given elements in $\mathcal{O}(1)$ time.

We are always given pointers to the queried elements. The space is linear in the length of the list.

ToDo: extend by deletion, splicing?
Ref: de-amortized variant (all worst-case).

Overview We first construct a *list-labeling* algorithm that maintains a label on every element of a linked list, which will need $\mathcal{O}(\log n)$ amortized time per insertion (1.1). Comparing labels will tell us the order. The labels will be integers with values bounded by $n^{\mathcal{O}(1)}$, so we can handle each label in constant time.

Then we use *indirection*: we create groups of size $\Theta(\log n)$ and maintain labels of the groups (1.3). The small size of groups allows us to use labels of exponential size, so the list-labeling within any group is easy. Combining the information from both labels gives us the order in $\mathcal{O}(1)$ time. Overflowing groups are split in halves, so the amortized time per insert is also $\mathcal{O}(1)$.

1.1 Polynomial list labeling

- We are given a parameter n which is an upper bound on the length of the list.
- We maintain integer labels on the elements such that their order corresponds to the list order.
- Only insertions are allowed (we are given a pointer where to insert).

The algorithm

- Elements implicitly form hierarchical ranges: labels sharing the first i bits make a range of level i (ranges are consecutive).
- Choose $\alpha \in (1, 2)$; for ranges of level i the threshold is α^i. We use at most $\log_\alpha n$ bits for labels, as we never get over $n = \alpha^{\log_\alpha n}$ elements.
- Insertion: if possible, take any valid label; otherwise find the smallest range containing the position and relabel it uniformly.

Also works: select the rearranged range anywhere around the insertion point.

Lemma. *Insertion costs $\mathcal{O}(\log n)$ amortized time.*

Proof. When we assign a label, we pay 1 coin to each of the nested ranges containing the label, that is $\leq \log_\alpha n$. On collisions we find the shortest containing range that is under its threshold, and we relabel it evenly. When we relabel a range of length 2^i, it can pay for itself, because:

Case 1. If it's the first relabeling of the range, then we have 2^i coins.

Case 2. Otherwise we show the range has accumulated enough coins since its last relabel containing it. The number of elements in the overflowing child clearly went from $\leq \lceil \alpha^i/2 \rceil$ to $\lfloor \alpha^{i-1} \rfloor + 1$, which means at least $\alpha^{i-1}(1-\alpha/2)-1$ elements were added in the meantime. The number of relabeled elements is at most α^i, which gives us at most

$$\alpha/\left(1-\alpha/2\right) = 2\alpha/\left(2-\alpha\right) \in \Theta\left(1\right)$$

elements per coin spent (we disregarded that insignificant -1).

As a consequence, the total number of relabels is bounded by

$$\left(n \log_\alpha n\right) \frac{2\alpha}{2-\alpha} = \left(n \log n\right) \frac{2\alpha}{\left(2-\alpha\right)\log\alpha} \quad ,$$

which is lowest for $\alpha \doteq 1.37$ and gives us bound $< 14 n \log n$. ☐

ToDo: connection of labeling to weight-balanced trees (sec. 2.2)?

1.2 Labeling with unknown n

We could use the traditional method of rebuilding the whole structure in geometrically increasing intervals, but it's not necessary:

- only insertions are allowed;
- we start with 1-bit labels;
- whenever the root segment overflows, we relabel all with labels one bit longer (we would relabel them anyway);
- together the item x_i costs us less than $14\left(1+\log i\right)$ relabelings.

1.3 Indirection

- If we allow labels of $b + \log a$ bits, we can easily process b insertions into a list with a equidistant labels without any relabeling.
- Choose $a, b \in \Theta\left(d\right)$ where d is the current label length in the big structure.

ToDo: more details?

2 Persistent data structures

2.1 Definitions

Types of persistence:

- partial persistence: linear history (only the last version is modifiable);
- undo: linear history, but backtrackable (usually not mentioned in persistence);
- full persistence: tree history (any version is modifiable);
- confluent persistence: DAG history (allow combining more versions).

+ modifying several versions at once, retroactivity

We suppose that the versions are identified by opaque identifiers, usually pointers to a particular version of the structure. Updates create new such identifiers.

2.2 Applications

- Geometric data structures, e. g. planar point location. We are given a set of lines in the Euclidean plane and every query asks for the list of lines bounding a particular point. We do a plane sweep with one coordinate represented as the time (partial persistence), where ordereded dictionary represents the order of the lines at that time point, and line crossings are done via updates.
- Run-time dispatching for subtyped multi-parameter methods. A query asks for the most specialized method for the given type-tuple of parameters. That corresponds to finding the first marked ancestor in a partial order (the order can be represented as the time).

2.3 Universal techniques

- Note connections to construction of dynamic structures from pieces of static structures, e. g. Overmars: decomposable searching problems.
- One way: *purely functional* structures – we define the values of memory locations on their allocation an never change them (typically freed by garbage collection). For example we can do path-copying on tree-structure updates. These methods typically pay non-optimal memory per update, which is too much if we need to keep the history. On the other hand, we get *confluent* persistence for free, which has no general efficient techniques.

ToDo: notes on amortization in full persistence with strict functionality? (Okasaki)

2.3.1 Full persistence in pointer model [2]

- The original structure is composed of $\mathcal{O}(1)$-sized boxes linked with pointers.
- We construct an efficient persistent version.

Version tree representation

- Versions are represented by opaque pointers into a shared structure (simple integers would only suffice for partial persistence).
- The tree is linearized. New versions are inserted into the ordering right after the parent version (so the resulting linear list is some preorder sequence on the tree).
- We use a shared list-order data structure (1) to maintain all timestamps.

Fat-node method

Idea: let every node remember all its versions.
Overhead of operations performed by the original structure (multiplicative):

- update: $\mathcal{O}(\log n)$ time and $\mathcal{O}(1)$ space
- access: $\mathcal{O}(\log n)$ time

The method

- Field values for all versions are represented by an ordered map: (field ID, timestamp) $\mapsto$ value. (We need logarithmic time per operation.)
- A field's value from the map with timestamp t is *valid* in the interval between t (inclusive) and the next version for the same field in node's map (exclusive).
- The only *valid* value of a field in a time t can be found by a simple predecessor search on the pair (field ID, t).
- A field update in time t results into two insertions into the map – one storing the new value with timestamp t and another one undoing the effect for the version following t in the shared version list (not needed in partial persistence).

Node splitting

Idea: use a hybrid between path-copying and fat-node approaches. We need a constant bound on the in-degrees of nodes.
Overhead of operations performed by the original structure (multiplicative):

- update: $\mathcal{O}(1)$ amortized time and space
- access: $\mathcal{O}(1)$ time

The method

- First we modify the ephemeral structure by adding p *inverse pointer* slots to every node (not versioned), where p is an upper bound on node's in-degree (including access pointers). We maintain by these that all links are bidirectional. Then let k denote the maximal number of pointers from the node (including the new inverse pointers).
- We add a field for default version to every node, reserve $2e$ additional slots for *extra fields* ($e \geq k$, annotated by field IDs and versions) and one *copy pointer*.
- The copy pointers form a singly-linked list, one for every node family. Each family represents all versions of the node in growing version order.
- The *valid interval* of a node is the interval from its version stamp (inclusive) up to the version stamp of the successor in its family (exclusive). (It is unbounded if the node is the last in ist family.)
- Pointer properties:
  - *proper*: it's version is within the valid interval of the target
  - *overlapping*: pointer's valid interval isn't contained within the valid interval of the target node (we'll only consider this property for proper pointers)

  We want all pointers to be proper and non-overlapping after completing any update. This ensures that following a pointer only takes $\mathcal{O}(1)$ time in the worst case.
- On updating any of the fields we use the extra fields to store changes. After filling up we have to split the node in two. After node splits we first make all pointers proper (by jumping over copy pointers) and collect the set S of all nodes containing overlapping pointers. Then we always take a node $x \in S$, remove it and:
  - Split all overlapping pointers from x, making them proper and non-overlapping.
  - Split x if overflowing, assigning pointers from right to left and always leaving e free extra fields. The only exception is the last=beginning chunk which can be left larger (just not overflowing) and it occupies the original address of x. Note that splitting might've made some pointers to x improper.
  - Find all pointers to x and make them proper; add to S all nodes with newly overlapping pointers. Newly created nodes from x can't contain overlapping pointers, so every node can only be added to S at most once per update.

Analyzing time and space

- Note that all nodes still have $\mathcal{O}(1)$ size.
- Accessing a field only needs to examine one node, because all pointers are made proper and non-overlapping, so it is $\mathcal{O}(1)$.
- The total work is proportional to the number of newly created nodes, because improper and overlapping pointers can only be created by a split and every split generates at most p such pointers. Therefore it suffices to show that the number of newly created nodes is $\mathcal{O}(1)$ amortized per a field update done by the original structure, which also shows the space overhead.
- Space: we define the potential of a node

  $$\Phi\left(v\right) := \frac{\max\left\{e-f,0\right\}}{e-k+1}$$

  where f is the number of free extra fields (f is temporarily negative for overflowing nodes). The amortized space cost is then the number of new nodes plus the change in Φ.
- First, if the original algorithm creates a new node v, it will have $\Phi\left(v\right)=0$, so the amortized space complexity isn't affected.
- Let u be the number of fields changed by an update of the original structure. Apart from splitting, every field update can consume at most $2k$ extra fields to the nodes (because of inverse pointers).
- Let s be the total number of new nodes created by splitting during the update. Note that splitting a node can force splitting of up to k pointers (pointers to the node being split). Every node-split lowers the potential by lowering the number of extra fields by $2e+1-e=e+1$ (and the new node has e free fields, so its potential is zero). Therefore the potential change is bounded by

  $$\frac{1}{e-k+1}\begin{bmatrix} 2ku & \ldots\text{for field updates} \\ +sk & \ldots\text{for pointer splits} \\ -s\left(e+1\right) & \ldots\text{for node splits}\end{bmatrix} =$$
  $$= \frac{2ku+s\left(k-e-1\right)}{e-k+1} = \frac{2ku}{e-k+1} - s$$

  The amortized space complexity is $\leq u\frac{2k}{e-k+1}$ which is $\mathcal{O}(u)$.

Note: the method was de-amortized.

2.3.2 Persistent arrays

Time $\mathcal{O}(\log |\text{history}|)$ or $\mathcal{O}(\log\log n)$. Construction builds on list-order (1) and vEBT, but many improvements are needed. The times are asymptotically optimal [3].

The arrays can be easily used to make any RAM data structure fully persistent with a $\mathcal{O}(\log\log n)$ slowdown.

? redundant number representations, persistent deques, confluent persistence? More explanation in Okasaki's book.

References

[1] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito, "Two simplified algorithms for maintaining order in a list," in *Algorithms — ESA 2002*, R. Möhring and R. Raman, Eds. Springer Berlin Heidelberg, vol. 2461, pp. 152–164, 00000. [Online]. Available: http://www.springerlink.com/content/gm8mjdfhaa4c6fr2/

[2] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, "Making data structures persistent," vol. 38, no. 1, pp. 86–124, 00662. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0022000089900342

[3] M. Straka, "Optimal worst-case fully persistent arrays." [Online]. Available: http://fox.ucw.cz/papers/perarray/perarray.pdf

[4] H. Kaplan and R. E. Tarjan, "Purely functional representations of catenable sorted lists," in *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, ser. STOC '96. ACM, p. 202–211, 00030. [Online]. Available: http://doi.acm.org/10.1145/237814.237865

[5] H. Kaplan, C. Okasaki, and R. Tarjan, "Simple confluently persistent catenable lists," in *Algorithm Theory — SWAT'98*, ser. Lecture Notes in Computer Science, S. Arnborg and L. Ivansson, Eds. Springer Berlin / Heidelberg, vol. 1432, pp. 119–130, 00004. [Online]. Available: http://www.springerlink.com/content/g1w4422882561038/abstract/

2.4 Catenable lists

Operations:

- deque operations PUSH and POP: add/split an element to/from **either** end;
- CAT: catenate two lists;
- sometimes also SLICE: construct sub-list covering a given range of indices (that implies reading/changing/inserting/removing elements at a given index).

State of the art:

1. PUSH, POP and CAT in $\mathcal{O}(1)$ time and space.
2. PUSH and POP in $\mathcal{O}(1)$, CAT in $\mathcal{O}(\log\log n)$, and SLICE in $\mathcal{O}(\log n)$.[1] This works by adapting finger search $(2,3)$-trees [4].

- Both structures are worst-case and strictly functional: it is enough to define new nodes and never change them. (No techniques like laziness or memoization. Today's engineers often call it Copy-On-Write or Read-Copy-Update.)
- We show a simplified amortized version of the first structure [5].

2.4.1 Simple lists

First we show some important techniques on a simple version of non-catenable lists.

- The structure of sList(A) is either simply buf: Buffer(A), or a triple (pb: Buffer(A), cl: sList(A×A), sb: Buffer(A)).
- Each buffer contains between zero and three elements.
- We allow re-writing a node by a different representation of the *same* sequence.[2]
- POP (others are very similar) operation is easy without nonempty cl; otherwise:
  - If pf is empty, we POP a pair from cl into it, and *rewrite* the node.
  - We return the element from pf and a *copy* of the node with shortened pf.

Analysis:

- Buffers with zero or three elements are called *red*, others are *green*.
- Potential: 3×(#nodes with two red buffers) + (#nodes with one red buffer).
- If we perform k recursive calls, the actual cost is $\Theta(k+1)$, but the potential decreases by k: if a rewritten node had two red buffers, we replace it by two nodes with one (unchanged) red buffer; if that node only had one red buffer, we replace it by two nodes with green buffers. The non-recursive tail creates a new node in arbitrary state, but the amortized cost is still $\mathcal{O}(1)$.

[1] We will denote the lengths of lists by n. More precisely, for CAT the length of the shorter catenated list is enough, and indexing is logarithmic in the distance from the nearer end.

[2] This allows us to use standard amortization techniques which aren't possible if nodes can't be changed. That is the reason why there was so much focus on worst case in confluent persistence.

2.4.2 Catenable deques

- The structure $\mathsf{List}(A)$ is either simply buf: $\mathsf{Buf}(A)$, or a 5-tuple (pb: $\mathsf{Buf}(A)$, lcl: $\mathsf{List}(\mathsf{Triple}(A))$, mb: $\mathsf{Buf}(A)$, rcl: $\mathsf{List}(\mathsf{Triple}(A))$, sb: $\mathsf{Buf}(A)$) where Triple(A) is a triple (pft: $\mathsf{Buf}(A)$, cl: $\mathsf{List}(\mathsf{Triple}(A))$, sbt: $\mathsf{Buf}(A)$).
- Buffer lengths in List(A) are 1–8 for buf, 3–6 for pb and sb, 2 for mb. In $\mathsf{Triple}(A)$ the buffers have 2–3 elements, but sb can be empty if cl is empty.

Potential

- Only pb and sb buffers affect the potential; they are red if containing 3 or 6 elements, and green otherwise.
- Potential: $3\times$(#nodes with two red buffers) + (#nodes with one red buffer).

PUSH$(x \to l)$

Case 1. List l is a 5-tuple.

- If $|l.\mathsf{pb}|=6$, we split away the last two elements, form a one-buffer Triple from them, and PUSH it into $l.\mathsf{lcl}$ (we overwrote fields $l.\mathsf{pb}$ and $l.\mathsf{lcl}$).
- We assemble a copy of node l with x added to $l.\mathsf{pb}$.

Case 2. List l is just single buffer and $|l.\mathsf{buf}|=8$. We overwrite l by a 5-tuple: the buffers take (3,2,3) elements, respectively, and the child lists are empty. Then we continue as in *Case 1*.

Case 3. Otherwise, l is simple and $|l.\mathsf{buf}|<8$, so we can simply extend it.

Analysis is exactly the same as for POP on simple lists – recursion of depth k gets at least $k - \mathcal{O}(1)$ decrease in potential. Thus, amortized complexity of PUSH is $\mathcal{O}(1)$.

CAT(l_1, l_2)

Case 1. Both lists are 5-tuples. We construct a new node l. The last element from $l_1.\mathsf{sb}$ and the first from $l_2.\mathsf{pb}$ will make $l.\mathsf{mb}$; $l.\mathsf{pb} := l_1.\mathsf{pb}$ and $l.\mathsf{sb} := l_2.\mathsf{sb}$. Then it only remains to create $l.\mathsf{lcl}$, and $l.\mathsf{rcl}$ which is just mirrored. We split l_1.sf into one or two buffers of 2–3 elements, where the first one forms a triple with ($l_1.\mathsf{mb}$,$l_1.\mathsf{lcl}$,*) and the second one can be triple by itself. We PUSH these 1–2 triples to the right end of $l_1.\mathsf{lcl}$ and thus obtain our $l.\mathsf{lcl}$.

Case 2. Either list is a simple buffer. We PUSH the elements of the smaller list into the larger list, one by one.

Analysis: the operation by itself just allocates one new node, performs constant work, and then it calls PUSH at most four times. Thus, CAT is $\mathcal{O}(1)$ amortized, too.

ToDo: (NAIVE)POP and analyses.

naivePop(l)

We simply assemble a copy of node l without the first element in l.pb. This may create an invalid node, but we can afford that if it's just a temporary node and we want to PUSH something back on that place immediately.

pop(l) x

References

[1] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito, "Two simplified algorithms for maintaining order in a list," in *Algorithms — ESA 2002*, R. Möhring and R. Raman, Eds. Springer Berlin Heidelberg, vol. 2461, pp. 152–164, 00000. [Online]. Available: http://www.springerlink.com/content/gm8mjdfhaa4c6fr2/

[2] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, "Making data structures persistent," vol. 38, no. 1, pp. 86–124, 00662. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0022000089900342

[3] M. Straka, "Optimal worst-case fully persistent arrays." [Online]. Available: http://fox.ucw.cz/papers/perarray/perarray.pdf

[4] H. Kaplan and R. E. Tarjan, "Purely functional representations of catenable sorted lists," in *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, ser. STOC '96. ACM, p. 202–211, 00030. [Online]. Available: http://doi.acm.org/10.1145/237814.237865

[5] H. Kaplan, C. Okasaki, and R. Tarjan, "Simple confluently persistent catenable lists," in *Algorithm Theory — SWAT'98*, ser. Lecture Notes in Computer Science, S. Arnborg and L. Ivansson, Eds. Springer Berlin / Heidelberg, vol. 1432, pp. 119–130, 00004. [Online]. Available: http://www.springerlink.com/content/g1w4422882561038/abstract/