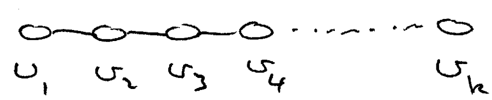
Persistent B-trees

Persistence: Each update gives a new version of tree
Give access to all versions (created so far).
(so similar in spirit to version control).

Partial persistence: search in all versions
update in newest only

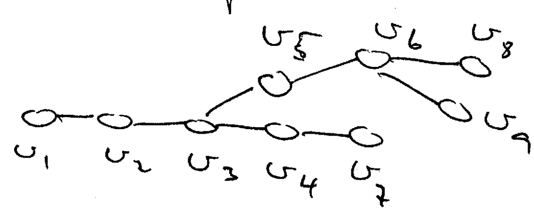
(⇒ versions form a list:



)

Full persistence: Search in all versions
Update — " —

(⇒ versions form a tree:



)

+ Naive solution: copy structure (entire) at each update
+ We want more efficient solutions (in terms of time and space).

[Generic solutions for pointer based data structures (in RAM model) developed by Driscoll et al [1989]]

TODAY: I/O-efficient partial persisten B-trees

$t_1 \; t_2 \quad t_3 \; t_4 \quad t_5 \ldots$

Time:

Version: 1 2 3 4 5

Version i ~~exists~~ in time interval $[t_i ; t_{i+1}[$

(I.e., i'th update starts at time t_i)

Using (t_i, i) as (key, value)-pairs stored in an ordered dictionary, we can find the ~~current~~ number of the version existing at any time t (by predecessor search on t).

i.e., non-persistent

Below, we will use a (standard) B-tree as this ordered dictionary, and will call it the access structure.

Except the value stored will not be version number, but a pointer into the rest of our (persistent B-tree) structure — namely to the correct place to start the search in the version in question

Access structure:

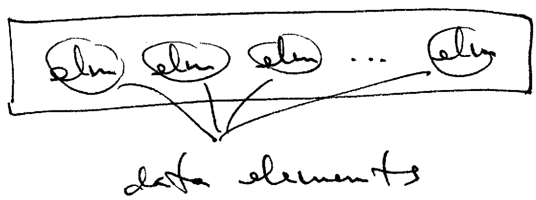
Rest:

Persistent B-tree

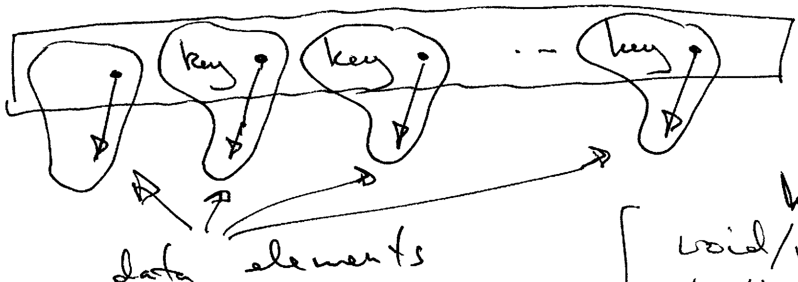
<u>The rest</u> : Let $b \geq 16$ be a parameter.

The rest of our structure consists of <u>nodes</u> containing <u>up to b</u> <u>data elements</u>.

For <u>leaf nodes</u>, the data elements are the elements stored in our persistent B-tree (i.e., are (key, value)-pairs)



For <u>internal nodes</u>, the data elements are routers (i.e., (key, node-pointer)-pairs).



[Keys may be void/missing (or, equivalently, be $-\infty$)]

Hence, "the rest" forms a graph.
Unlike a (standard) B-tree, the graph is <u>not</u> restricted to be a (multiway) <u>tree</u>. In fact, it will be a <u>DAG</u>.

We choose b such that a disk block can hold b data elements (b is B).

Also different from (std.) B-trees : each data element has a <u>time stamp</u>

A <u>time stamp</u> is an interval of the form

i) $[t_i ; t_j[$

or

ii) $[t_i ; \infty[$

for t_i, t_j update times. It stores the versions for which the data element is valid.

Let t be the current time (ie., $t \in [t_i ; \infty[$ for i the (currently) last version).

A data element is said to be <u>alive</u> if t is contained in its timestamp, and <u>dead</u> otherwise

(In fact, it will turn out that alive data elms. have the $[t_i ; \infty[$ form of the timestamp, and dead elements the other form).

<u>No data element will ever get deleted</u> from a node. Timestamps (of alive nodes) may change, however (by setting $\infty \to t_k$) We call this <u>closing</u> the timestamp.

A data element is <u>valid at</u> (some) <u>time t</u> if t is contained in its timestamp (so alive = valid at current time).

A node is <u>valid</u> at time t if it contains any data elements valid at time t.

The main invariant maintained by the "rest" part of our structure is:

<u>INVARIANT 1</u>: For any t, the <u>nodes</u> valid at t and the <u>data elements</u> valid at t together constitute a $(\frac{b}{4}, b)$-tree with leaf parameter b, and this tree contains in the leaf nodes exactly the elements stored in the version existing at time t.

In our access structure, the pointer stored as the value attached to t_i points to the root of the tree of the invariant for $t = t_i$.

<u>Searches</u> : The operation $\boxed{\text{Search}(k, t)}$

that searches for the element of key k in version existing at time t, is now straight-forward :

> Search in access structure with t (predecessor search).
> Search as usual in (a,b)-trees with key k, starting at root pointed to by pointer returned by first search.

<u>In details:</u> Each time a node is read, extract the data elms. valid at time t, sort these, view the result as an (a,b)-tree node and proceed

Each node access is one I/O. Hence, entire search takes $O(\log_B(I)) + O(\log_B(N_t))$

$$\cancel{\left(= O(\log_B(I + N_t))\right)} \quad \boxed{= O(\log_B(I))}$$

where I is the current number of versions (= number of updates so far) and N_t is the number of elms. in version existing at time t.

$I \geq N_t$ if we start with empty tree (assumed here)

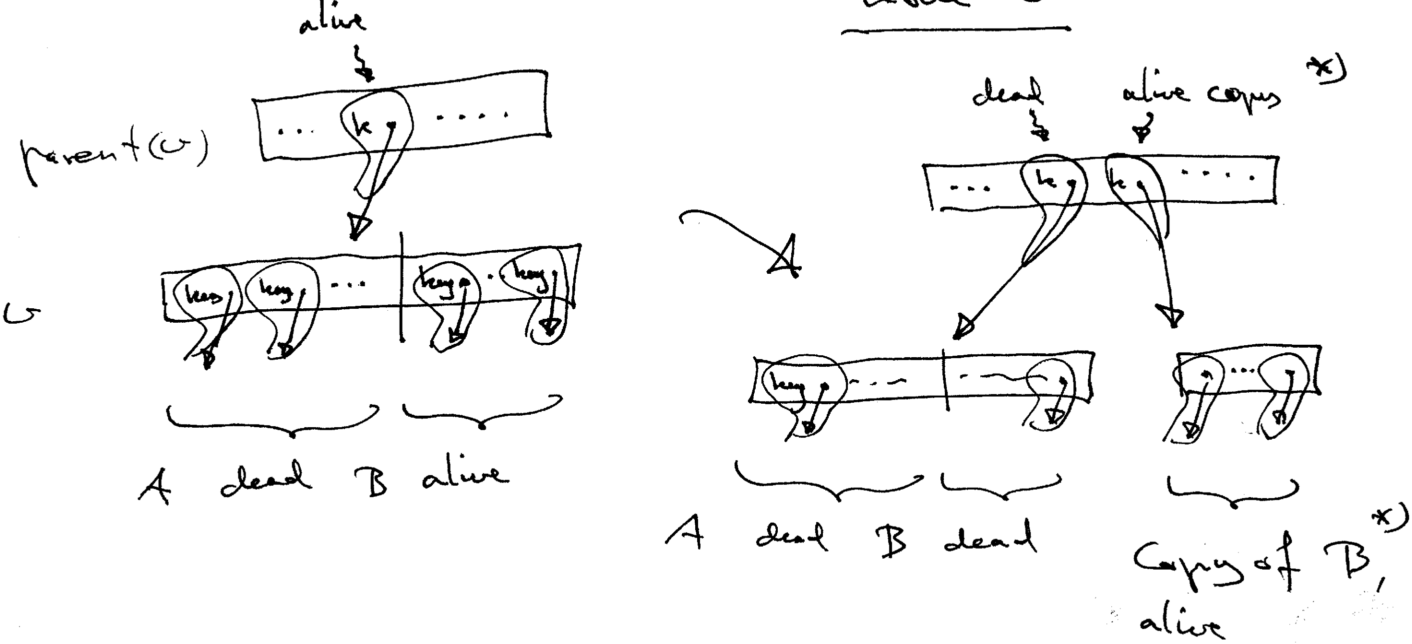
<u>Notes</u>: i) the above is I/O-~~efficient~~ (optimal), but not CPU-optimal. (For this, nodes should contain persistent binary search trees, made using methods from Driscoll et al., which is out of scope here), due eg. to the sorting.

ii) If searches are based on ~~versions~~ version numbers (instead of time t) directly, the access structure can be simply an array of pointers (using $O(1)$ I/O's to look up).

Updates

[Note: for updates t = current time (as partial persistence)]

A basic operation will be the Version Split of a node v:



+ Current time is t_i (i the number of the new version made by the update)
+ Existing dead timestamps are left untouched
+ The existing alive timestamps are made dead by changing $[t_j ; \infty[$ to $[t_j , t_i[$
+ New alive copies get timestamp $[t_i , \infty[$
  ↑
  *)

+ Operation will only be invoked on search path for update position in current version. Hence the data element of the parent is alive (is assumed in drawing above).

__Observation__ : A version split does not violate the invariant INV 1.

(Proof: only two previous / three new nodes need to be considered. Observe that for any t, what is valid at time t is not changed).

To obtain a good space analysis in the end, we introduce the following invariant:

$$\boxed{\text{\underline{INV 2}: New nodes ~~are~~ created with } \tfrac{3}{8}b \le \#\text{ data elements} \le \tfrac{7}{8}b}$$

Also recall the already introduced constraint:

$$\boxed{\text{\underline{INV 3}: No node ever contains more than } b \text{ data elements.}}$$

We now proceed to define the update operations.

Insert:

Increment version counter i
Log the update time t_i
Search in persistent B-tree for insertion point (using time t_{i-1}).
Insert data element in found leaf node with time stamp $[t_i ; \infty[$

If <u>needed</u> : <u>rebalance</u>.

(details below)

Needed: INV 3 may be violated ($b+1$ data elements in leaf)

INV 1 may also be violated for $t_i \leq t$ (a B-tree overflow) – but then the above violation also occurs. So enough to check that.

Rebalance: Make VersionSplit on leaf.

If INV 2 is violated by <u>overflow</u> ($> \frac{7}{8}b$ data elm. in new node (all alive)): make a <u>standard split</u> (B-tree split) on new node (inserting a new data elm. in parent (with timestamp $[t_i ; \infty[$)).

New nodes will have at least

$$\left\lfloor \frac{\frac{7}{8}b+1}{2} \right\rfloor \geq \frac{7}{16}b > \frac{3}{8}b \text{ data elms.}$$

and at most

$$\left\lceil \frac{b+1}{2} \right\rceil \leq \frac{b}{2}+1 = b\left(\frac{1}{2}+\frac{1}{b}\right) \leq b\left(\frac{1}{2}+\frac{1}{16}\right) < \frac{7}{8}b \text{ data elms.}$$

If INV2 is violated by underflow

($< \frac{3}{8}b$ data elms. in new node (all alive)):

Find a sibling in t_i-B-tree (via alive data elms in parent).

Make a VersionSplit on this. (this gives between $\frac{1}{4}b$ and b alive data elms. due to sibling being a valid B-tree node at time t_i (INV 1)).

Make a standard B-tree fuse between the two new nodes.

If result has $> \frac{7}{8}b$ data elms, make a standard B-tree split.

Update parent. ~~("close" a timestamp for the fuse, add new data elm. with timestamp $[t_i, \infty[$ for split).~~ ~~$[t_k, \infty[\to [t_k, t_i[$~~

The two version splits introduced two new data elements (with timestamps $[t_i ; \infty[$).

Now adjust these:

Fuse : remove one

Split : add one again.

After the fuse, the new node has at least

(Fused nodes are valid B-tree nodes at time t_i) $\longrightarrow$

$$\frac{1}{4}b + \frac{1}{4}b = \frac{1}{2}b > \frac{3}{8}b$$

data elms. With no split, it has at most $\frac{7}{8}b$.

After a split, the two new nodes have at least $\lfloor \frac{7}{8}b + 1 \rfloor > \frac{3}{8}b$ data elms (see previous page). and at most

$$\left\lceil \frac{\frac{3}{8}b - 1 + b}{2} \right\rceil \leq b\left(\frac{3}{16} + \frac{1}{2}\right) = \frac{11}{16}b < \frac{7}{8}b.$$

12

<u>Observe</u>: Now, No violations of INV 2 on current level (B-tree level) [i.e. on newly produced nodes]. See calculations above

No violations of INV 1 (only $t \geq t_i$ need to be considered) or INV 3 (as bounds of INV 2 are stronger). on the current B-tree level.

For $t_i \geq t$, INV 1 <u>can</u> be violated on B-tree parent level / node

*) (+1 or ∸1 alive elms are possible [check cases])

INV 3 <u>can also</u> be violated at that node.

**) (Between 0 and 2 new data elm. [check cases]).

Is fixed by <u>recursion</u> on parent, starting with a Version Split on this.

Except that some node size counts may be larger / smaller by one (which doesn't invalidate the conclusions of calculations [check]), the same analysis applies.

Recursion may propagate to root of t_i - B - tree, creating a new t_i - root.

Finally (at end of insertion) update access structure [insert (t_i, pointer to root)]
new or not.

<u>Deletion</u> is very similar. Start by closing a timestamp in leaf. Analysis and actions are very similar to insert (actually, ~~fewer~~ cases)

[Better : redo the entire analysis based on a change at current level [initially leaf level] of

± 1 live data element change

$0 \le x \le 2$ data element change

(Then analysis of recursive cases and of deletion is built in from beginning.)]

Note: During the recursive rebalancing, rebal. at a given level is triggered by a i) violation of INV 1 for t_i (current, alive tree), where B-tree node sizes may be violated, or ii) violation of INV 3 (too many data elm. in node).

0) New nodes contain between $\frac{3}{8}b$ and $\frac{7}{8}b$ alive data elements (and nothing else). So needs $\frac{1}{8}b$ new data elements or $\frac{1}{8}b$ changes of number of alive data elements before they trigger.

Denote these changes steps.

A node can only trigger once (then made dead by a version split, and only alive nodes (ie, nodes in current tree) are part of updates.

1) Each update gives one step on level 0 (in current tree)

2) Each rebal. at level i gives at most two steps on level $i+1$ [See *) and **) some pages above – the relevant count is max (not sum) of the two (numeric value of) types of steps.]

With I updates (= # versions):

$\overset{1)}{\Longrightarrow}$ I steps on level 0.

$\overset{3)}{\Longrightarrow} \leq I / \frac{b}{8} = I \cdot \frac{8}{b}$ rebal. on level 0

$\overset{2)}{\Longrightarrow} \leq I \cdot \frac{8}{b} \cdot 2$ steps on level 1

$\overset{3)}{\Longrightarrow} \leq I \cdot \frac{8}{b} \cdot 2 / \frac{b}{8} = I \cdot \left(\frac{8}{b}\right)^2 \cdot 2^1$ rebal on level 1

$\overset{2)}{\Longrightarrow} \leq I \cdot \left(\frac{8 \cdot 2}{b}\right)^2$ steps on level 2

$\overset{3)}{\Longrightarrow} \leq I \left(\frac{8 \cdot 2}{b}\right)^2 / \frac{b}{8} = I \cdot \left(\frac{8}{b}\right)^3 \cdot 2^2$ rebal on level 2

$\vdots$

Total: $\leq I \cdot \frac{8}{b} \cdot \sum_{i=0}^{\infty} \left(\frac{8 \cdot 2}{b}\right)^i$

$= I \cdot \frac{8}{b} \cdot O(1)$ as $b > 16$

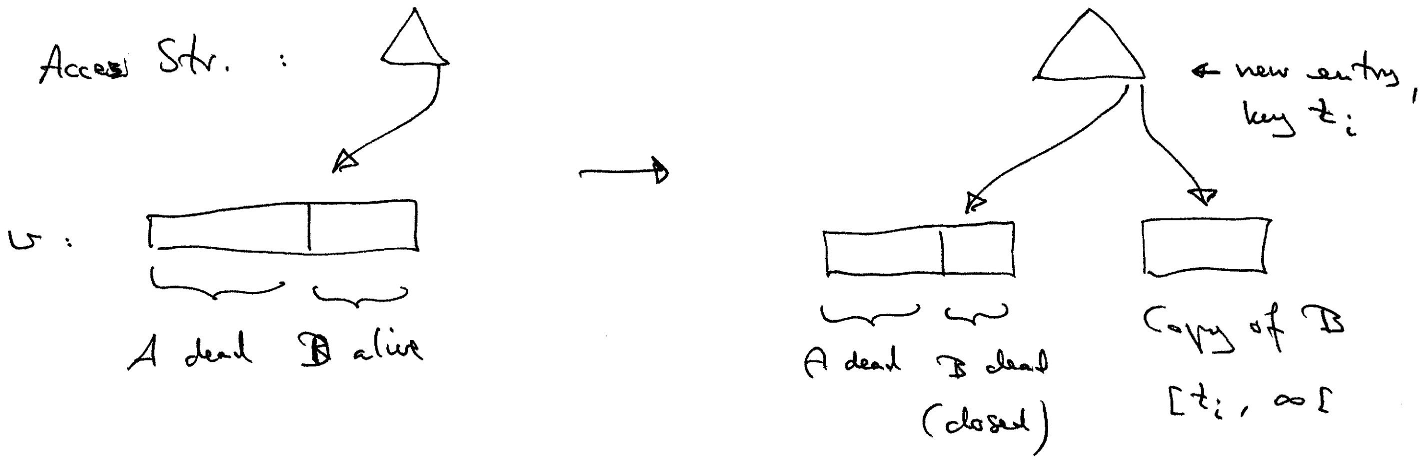
Time: Amortized $O\left(\frac{1}{b}\right)$ rebal. (= I/Os) pr. update.

Space: Each rebal. $\Longrightarrow$ at most 2 new nodes. So total space $\leq 2 \cdot$ total rebal.

$= O(I/b)$

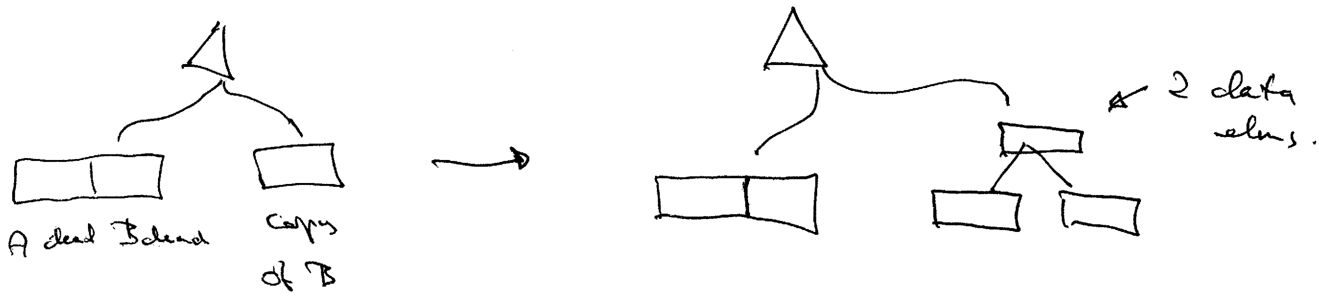
Special conditions at root :

Version Split at root :



Note that underflow $\left(< \frac{3}{8} b\right)$ cannot be handled (there is no sibling to fuse with).

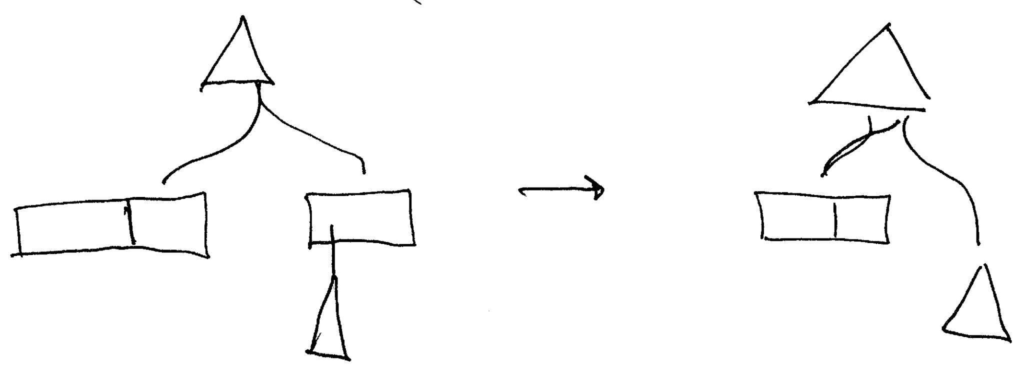
Also, the overflow case $\left(> \frac{7}{8} b\right)$ has a split which at root gives a node with two data elms :



So INV 2 has to have the following addition :

$\boxed{\text{INV2}' : \text{If new node is root of current tree, then } 2 \leq \# \text{ data elms} \leq \tfrac{7}{8} b}$

However,
If the underflow of new root (after Version Split) is 1, we do as follows :
(data elems present)



I.e. we remove the node.

(Note that underflow cannot be 0 data elems present, as the current tree was a valid B-tree before update, and at most 1 child (so root there has ≥ 2 children) was removed by the recursive rebalancing reaching the root.)

The changed INV 2 affects the analysis of time and space:

The analysis above really does not covers rebalancing at current root triggered by underflow.

Such triggering could happen after 1 (not 1/8 · b) steps in the node [So c) does not hold].

However, such rebal. also <u>does not imply steps on the level above</u>.

$\boxed{\text{So analysis on page 15 is valid for the rest of the rebal. ops.}}$ (root rebal. triggered by underflow)

As such triggering happens at levels ≥ 1 (Really, a root at level 0 in B-tree must be allowed down to 0 (not 2) elms), and requires $\Omega(1)$ steps, line 4 on page 15 shows that such rebal. is $O(I/b)$.

Hence, time ~~and~~ $\underbrace{\text{space}}_{\text{cf page 15, bottom}}$ bounds are still valid.