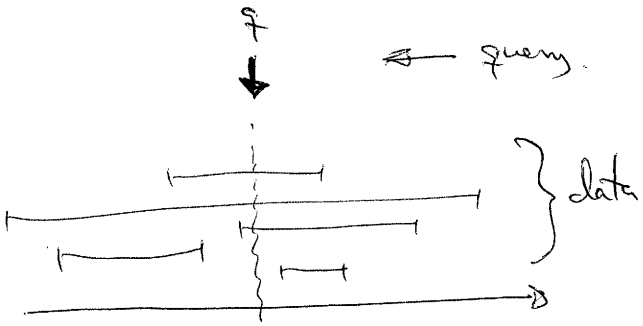
External Interval Trees

A geometry problem in 1D : Stabbing queries

Data stored : 1D intervals

Query : a point in 1D

> Returns all stored intervals containing query point.

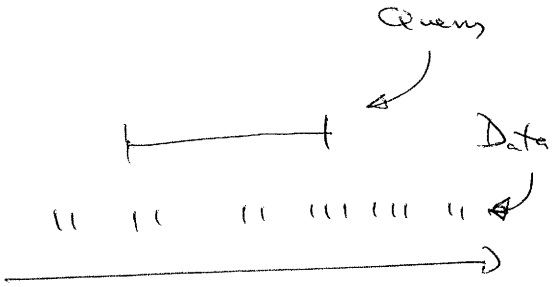


Kind of dual problem to **range queries** in 1D.

Data stored : points in 1D

Query : an interval in 1D

> Returns all stored points contained in query interval



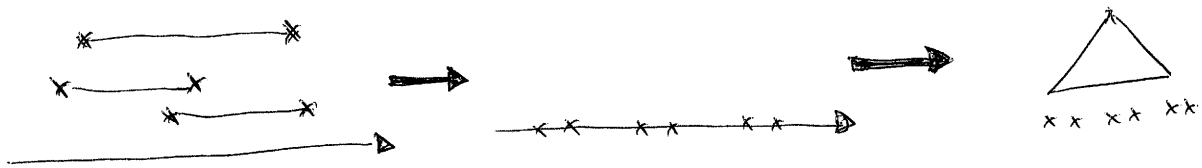
Motivation :

1) Direct applications, such as : intervals ~ objects life spans, query ~ find objects alive at time q.

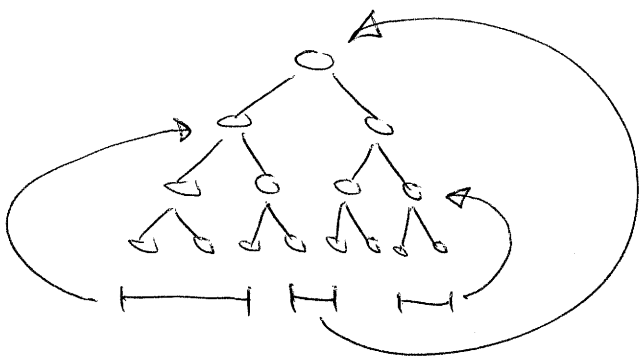
2) Efficient external solution displays a lot of good ideas.

Internal solution : Interval Trees [Edelsbrunner, 1983]

1) Binary tree on all endpoints of stored intervals



2) The intervals are stored in internal nodes of the tree. Each interval is stored in one node, namely the highest node whose key intersects (lies in) the interval:

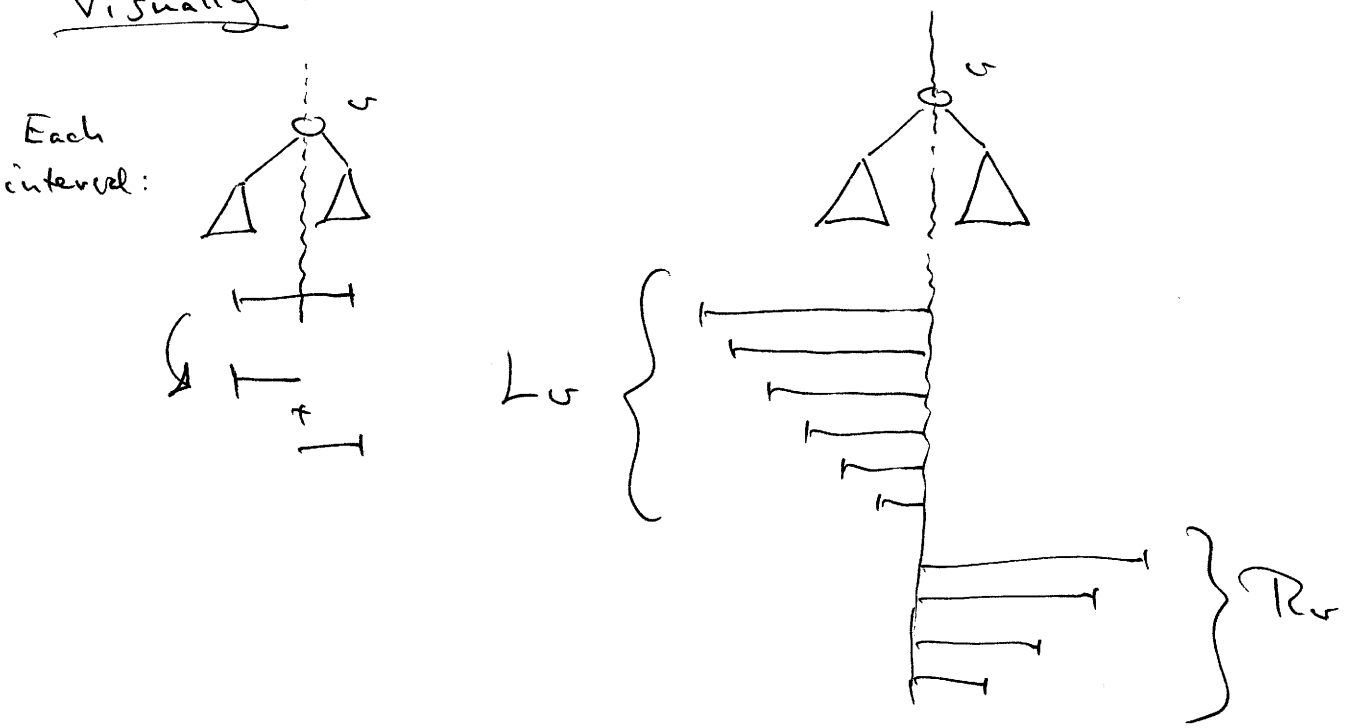


3) Each interval is stored two times in this node v. Once in each of the node's two side structures, L_v and R_v. Both of these are just lists.

L_v : stored intervals sorted by left endpoint

R_v : ——————— right ———

Visually:



Space: # endpoints $\leq 2 \cdot$ # intervals $= 2n$

= size of Lin tree

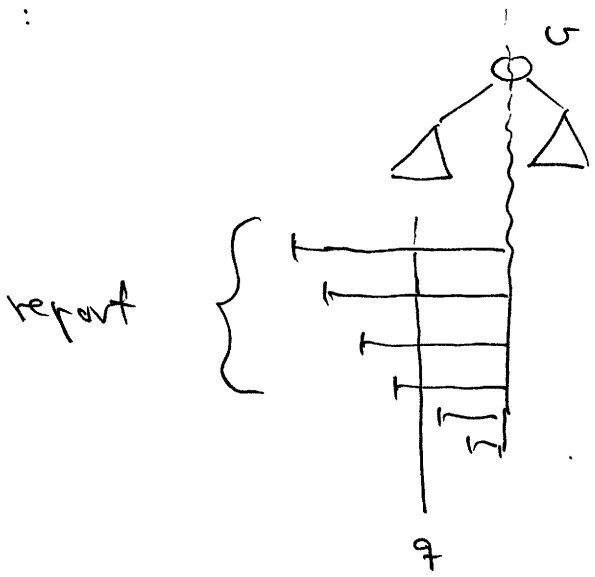
Side structures in total $= 2 \cdot n$

Total space is $O(n)$.

Query: Search for q in Lin. tree. In a node v during search (wlog. search goes left in v):

- No intervals in right subtree of v should be reported.
- Intervals in left subtree of v are reported during rest of search
- Intervals stored in v:

Traverse L_v in sorted order [scan], stop at first (left) endpoint to the right of q:



Time spent in v is $O(1) + O(\#\text{reported intervals})$.

Total time for query: $O(\log(n) + T)$

T = # reported intervals (= size of output).

[Dynamization (updates): use weight-balanced binary trees to give good [log n] amortized cost, even though each rotation means rebuild of nodes' side structures]

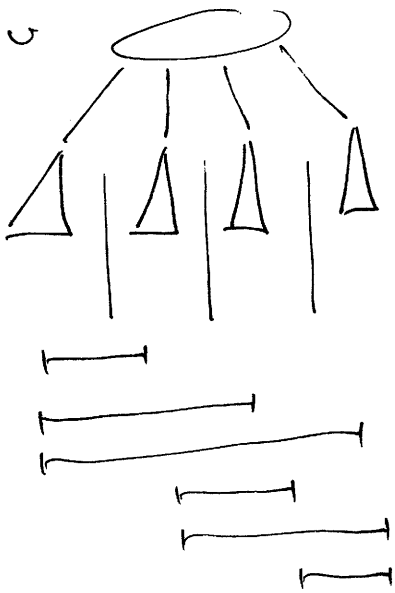
External variant ?

A direct use of the internal structure above gives queries in $O(\log_2(n) + T/B)$ I/O's.

We want $O(\log_B(n) + T/B)$. [Since $\log_2(n)$ is not optimal for external searching.]

Arge, Vitter [1996/2003] solution :

Idea 1 High-degree nodes



Fanout k

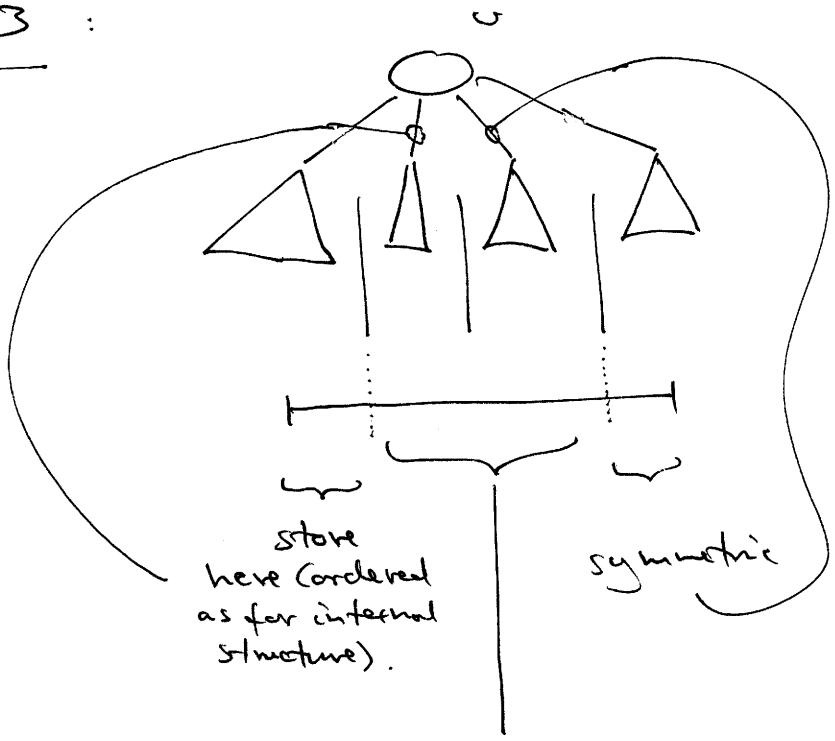
$\sim k^2$ types of intervals belongs to v.

Idea 2 : $\log_{\sqrt{B}}(n) = \frac{\log_2(n)}{\log_2(B^{1/2})} = \frac{\log_2(n)}{\frac{1}{2}\cdot\log_2(B)} = 2\cdot\log_B(n)$

So fanout $\sqrt{B}$ is OK for our goal.

$\Rightarrow k^2 = (\sqrt{B})^2 = B$.

Idea 3 :



single slab lists

if $\Omega(B)$ such intervals (for this type (out of $\sim k^2$))

store in list in v (head of list has ID = the two children giving type).

else

store in <u>underflow structure</u> [common for all interval types] of v.

multi slab lists

Size of underflow structure $= O(k^2 \cdot B)$

$= O(B^2)$.

Underflow structure is a static external persistent B-tree [see below].

(partially)

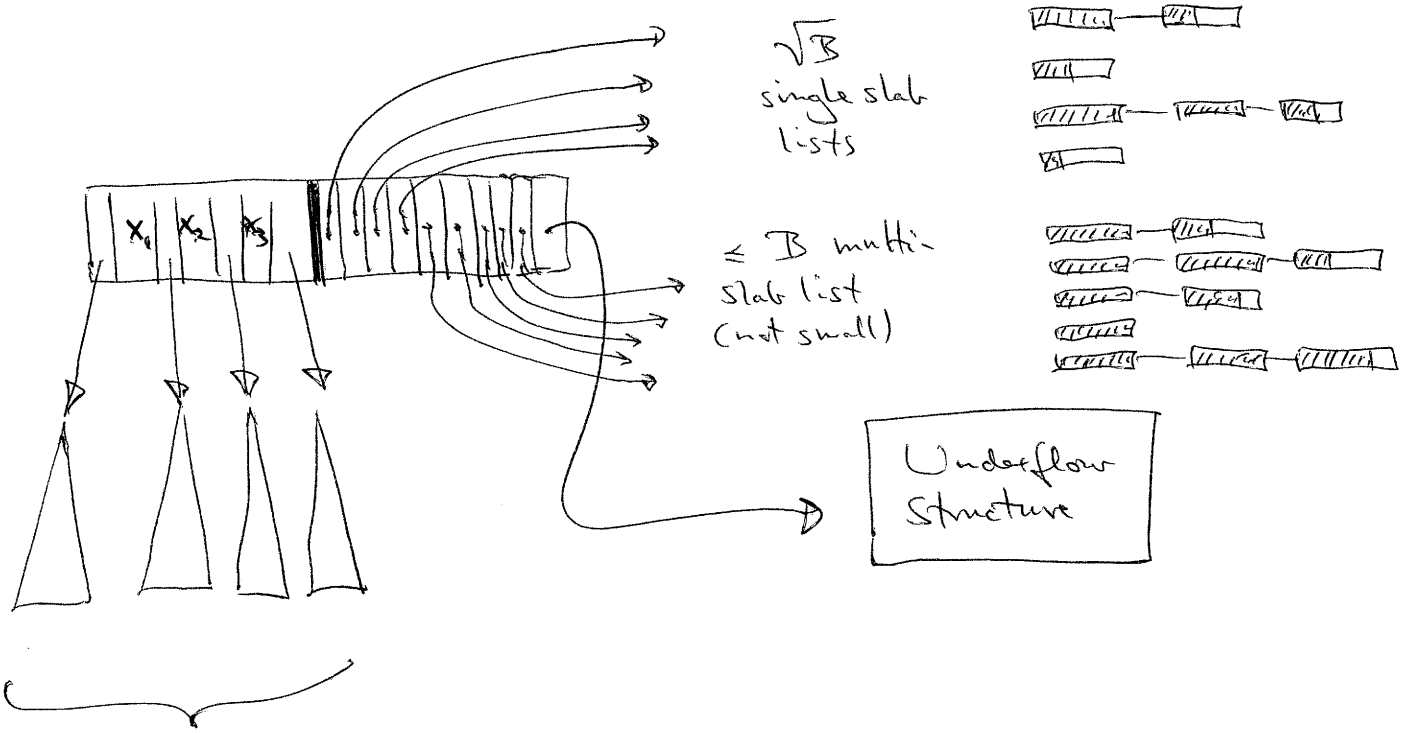
Idea 4: Thus, a node has 2 single-slab list per child (2·k in total), and one multi-slab list per pair of children ($\binom{k}{2} = \frac{k \cdot (k-1)}{2} = \Theta(k^2)$ in total).

A single-slab list is just stored as a list, sorted by endpoints of its intervals similar to the internal case.

A multi-slab list is just stored as a list if it contains $\Omega(B)$ elements. It is ordered by ID's of its intervals. Else it is small. All small multi-slab lists of a node are stored together in the underflow structure.

By a "list" above, we mean a linked list of blocks (containing the elements in sorted order). Actually, for the dynamic version of the external interval tree, the "lists" will be B-trees to allow efficient updates to lists. The node itself contains a pointer to the first block [root] of each list [B-tree].

Figure of a node (internal):



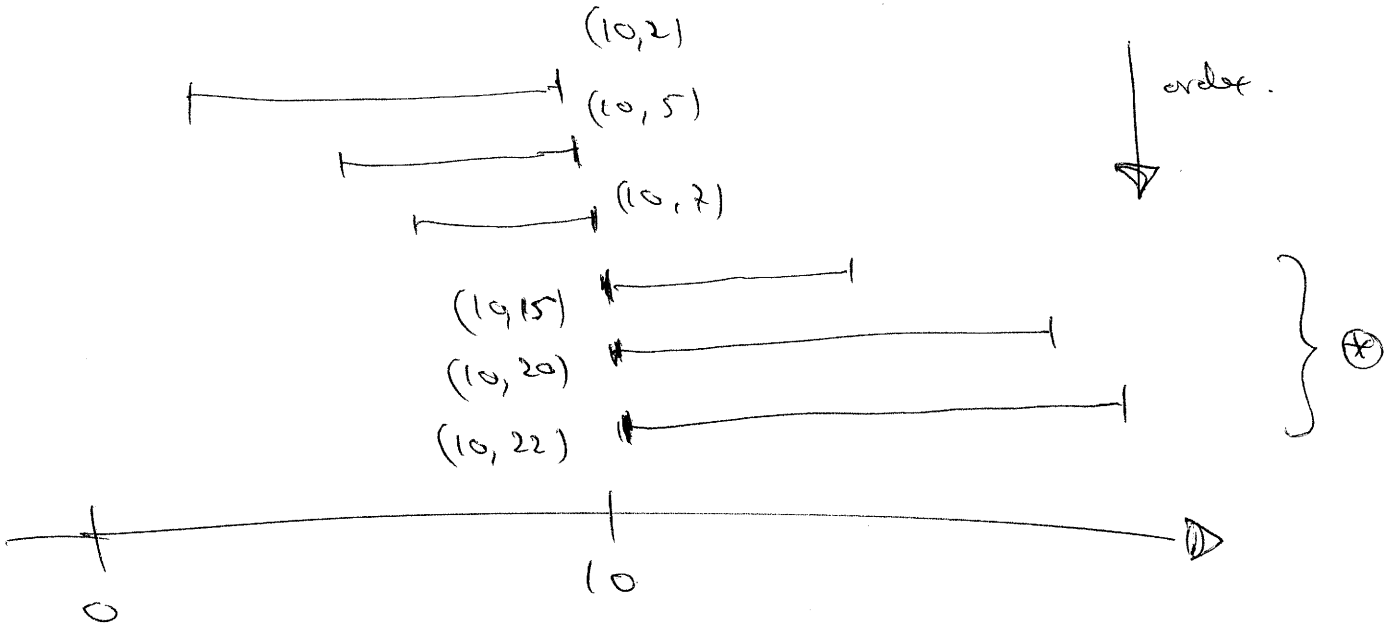
A leaf node just contains a list of all its intervals.

Idea 6: We can assume all endpoints of all intervals are different. By a standard trick: endpoint $\longrightarrow$ (endpoint, intervals other endp.).

So $\vdash\!\!\!\dashv$ becomes $\vdash\!\!\!\dashv$. These
2 5 (2,5) (5,2)

tuples are ordered lexicographically. If there are no duplets among intervals [which we will assume, is a fair ~~assum~~ restriction], there are now none among endpoints.

So if endpoint 10 is shared by many intervals, all the "10's" are being separated into following sorted order :

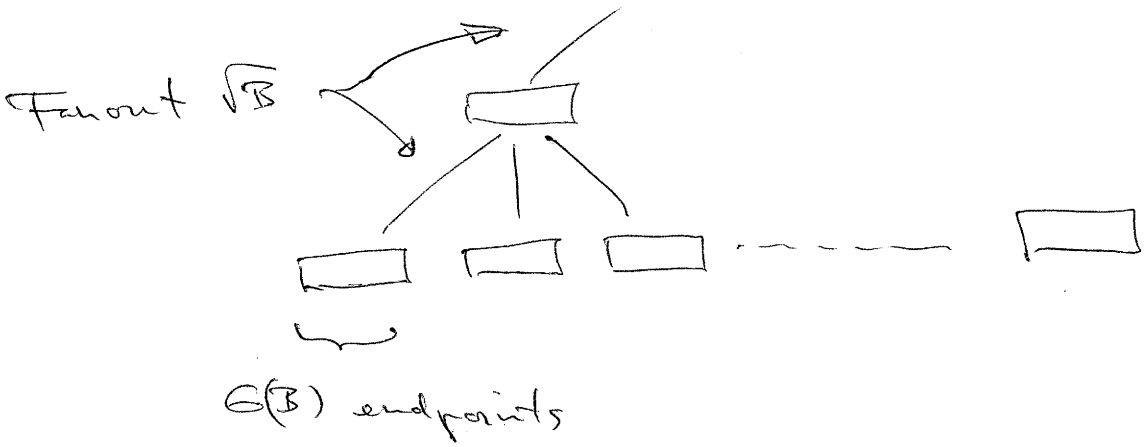


A query on 7 must now be made with tuple $(7, -\infty)$. This will report all intervals intersecting 7, except those of form ⊛. These can be found by another query with tuple $(7, +\infty)$. That second query should skip reporting all intervals found which are not of type ⊛ [to avoid reporting ⟝ 9 11 twice for $7 = 10$ in above example].

This at most doubles the search work.

In analogy with the internal structure, our main tree is a multiway search tree built on the endpoints [transformed to tuples, as above], with side structures in internal nodes as described above.

Idea 7: Even though we chose fanout $k = \sqrt{B}$, we let sizes of leaves be $\Theta(B)$.



This gives $O(\frac{N}{B})$ leaves and $O(\frac{N}{B} \cdot \frac{1}{\sqrt{B}})$ internal nodes [this is the number of nodes on lowest internal level, which dominates the total number of internal nodes since their number per level decreases exponentially (by a factor of $\frac{1}{\sqrt{B}}$)].

Space analysis

An internal node uses 1 block plus space for side structures. If a side structure (single slab list, multi slab list, ~~the~~ underflow structure) stores $t = \Omega(B)$ intervals, it uses $O(t/B)$ blocks (*) [for the underflow structure, we are here using that it takes linear space – see below].

A single slab list and the underflow structure may contain $< B$ elements. They still need 1 entire block [in the dynamic case, their varying sizes makes it hard to let the efficiently share blocks].

Thus, an internal node may contain/use $O(\sqrt{B})$ blocks which are nearly empty.

These sum to $O(\frac{N}{B} \cdot \frac{1}{\sqrt{B}} \cdot \sqrt{B}) = O(N/B)$ blocks.

By (*), the remaining blocks used sum to $O(N/B)$ in the tree. [Since each interval is stored at most 3 times]

So the data structure uses $O(N/B)$ blocks

(N = no. of intervals).

Persistent B-trees

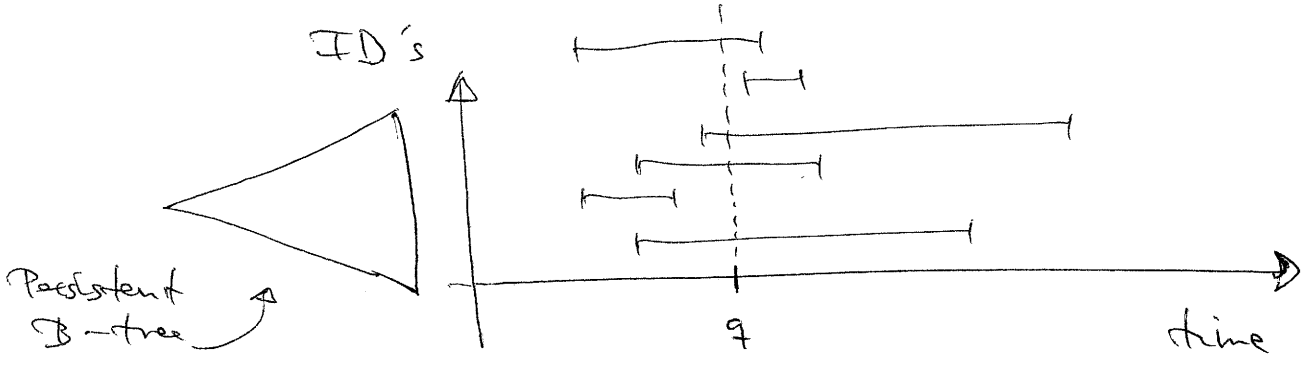
In a data structure, an update gives a new version of it. The data structure is partially persistent if all previous versions are available to search in.

(Starting from empty tree)

Fact: there exist a partially persistent version of B-trees which i) after a sequence of n updates occupies $O(n/B)$ blocks ii) allows (range) searches in $O(\log_B(n) + \frac{|output|}{B})$ I/O's iii) can be built in $O(\frac{n}{B}\log_{M/B}(n/M))$ I/O's given the sequence of updates.

Proof: Omitted.

Now consider a set of intervals. Let their ID's be keys in a persistent B-tree, and let their start and end points be insertion and deletion times



Each endpoint is an update time. By another B-tree storing the update times (i.e. endpoints), we can for a query point find the update time just preceding q [by a predecessor search].

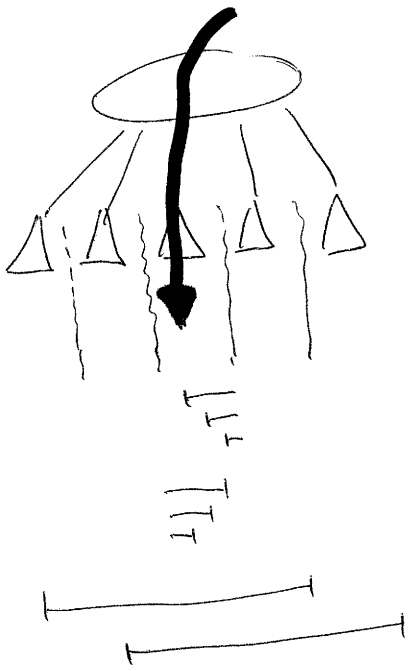
If we in the version [of the persistent B-tree] corresponding to this version does RangeSearch$(-\infty, +\infty)$, we get the intervals stabbed by q.

The underflow structure of a node is such a persistent B-tree (on all intervals in all the nodes small multislab lists, in a single tree). Hence, it has $n \leq B \cdot B = B^2$ (# of multislab list $\cdot$ max size for small lists).

So it answers stabbing queries on these intervals in time

$$O\left(\underbrace{\log_B(B^2)}_{2} + \frac{|\text{output}|}{B}\right) = O\left(1 + \frac{|\text{output}|}{B}\right).$$

Query : Like in internal data structure, search normally for q [$(q, -\infty)$] in the search tree and report in each node passed the stabbed intervals among those stored in node.



In each node do this by:

I/O s

i) Scan one left type single slab list $O(1 + \frac{|output|}{B})$

i) —"— right type —"— $O(1 + \frac{|output|}{B})$

ii) Scan the relevant multislab list (those where q is contained) which are not small [up to $k^2 = B$ lists] $O(\frac{|output|}{B})$

Report from all small multislab list by a single stabbing query to the over flow structure $O(1 + \frac{|output|}{B})$

i) : Scan as in internal data structure side list (stop when [sorted lists of endpoints] passes q).

ii) : The entire multislab list should be reported.

Total : $O(\text{height of tree} + \frac{|output|}{B}) = O(\log_B(N) + \frac{|output|}{B})$