Robust Search Methods for B-Trees

Kikuo Fujimura, Pankaj Jalote

18th International Symposium on Fault-Tolerant Computing
(FTCS-18), 1988

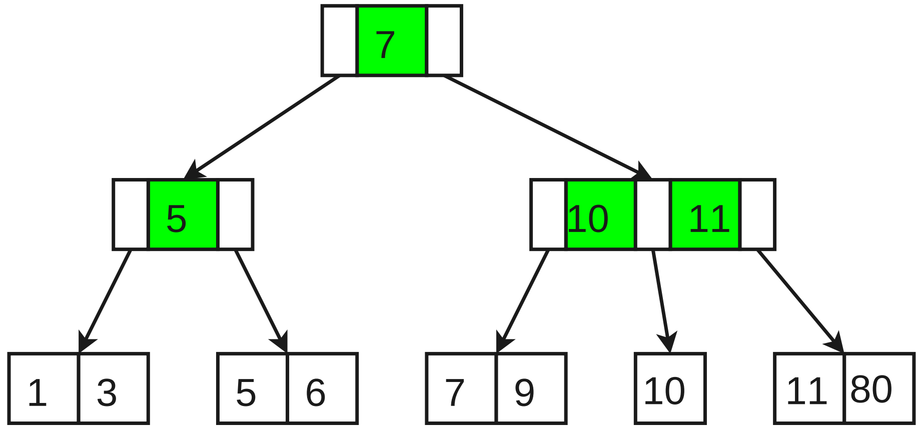
Presented by Zheng Zhang

Software fault tolerance

- Recovery block based schemes[1]
- n-version programming[2]
- Exception handling[3]
- Robust data structures[4]
  - By adding redundancy
- *With unreliable data structures*: this paper, [on B-Tree]
  - Explore semantic information (built-in redundancy)
  - No additional redundancy needed

What did this paper accomplish?

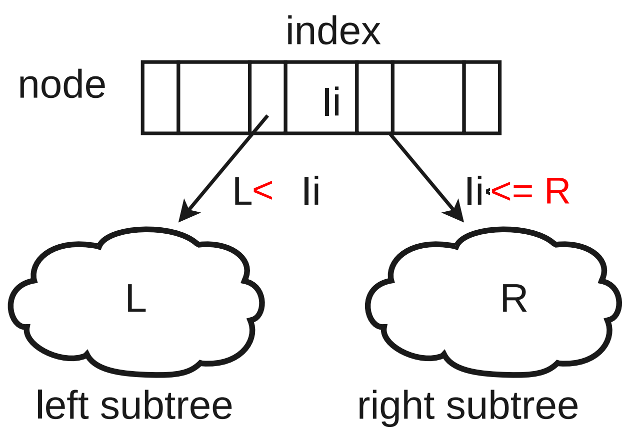
A robust search method on B^+-tree



- Fault model: Only index corruption. Structure is correct
  - Basic: single index corrupted
  - Extended: multiple indices corrupted
- Search returns "yes" or "no". No false report.

Index corruption

BTP:



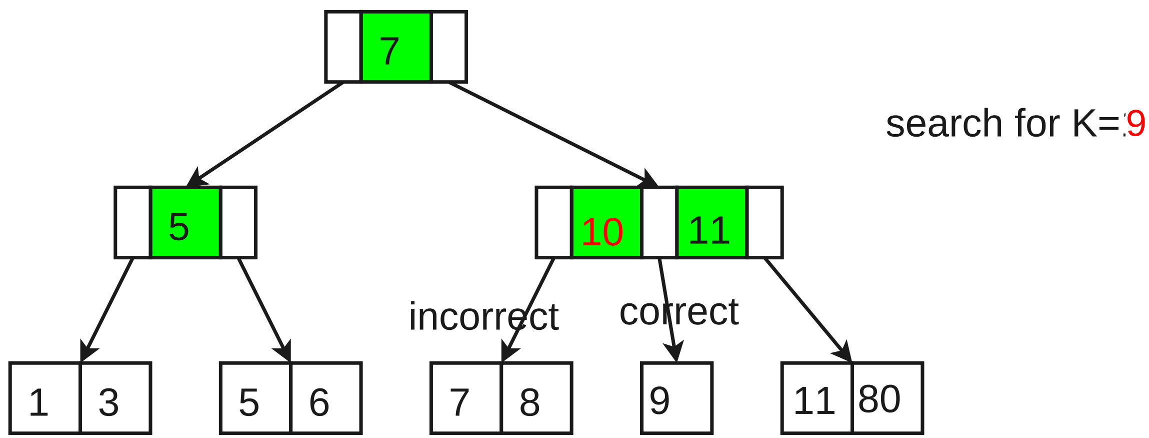
- An index I_i is corrupted if I_i does not satisfy BTP
- Suppose a corrupted index does not break the ascending order on the node.

Observations:

- index corrupted $\Rightarrow$ index changed
- index changed $\not\Rightarrow$ index corrupted

Misdirected Search

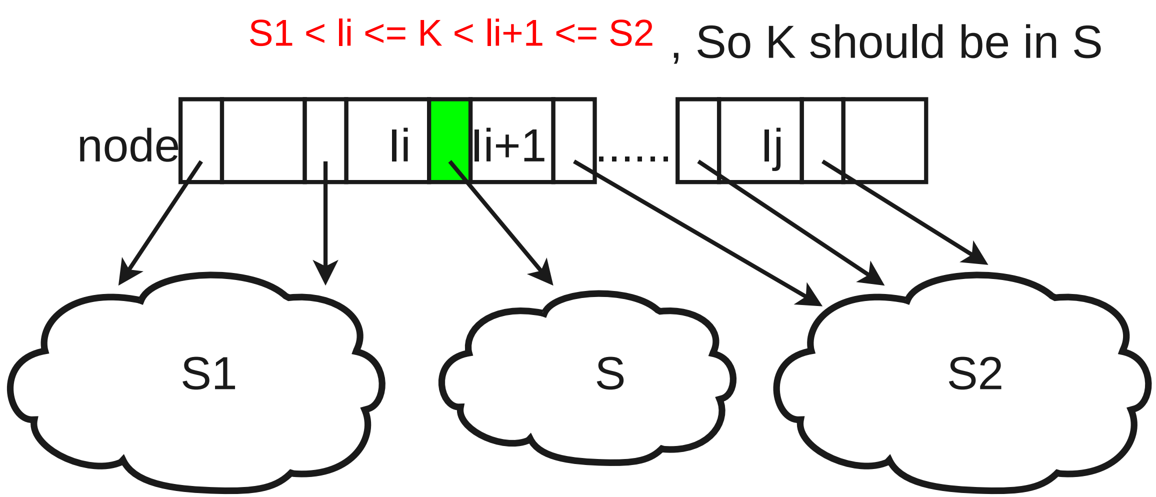
- A corrupted index MAY misdirect a search



- Consequence: you search for a existing key, but search returns failure ("key not exist")

Suspicious set

- A search is misdirected only if there is a corrupted index sitting along the search trace.



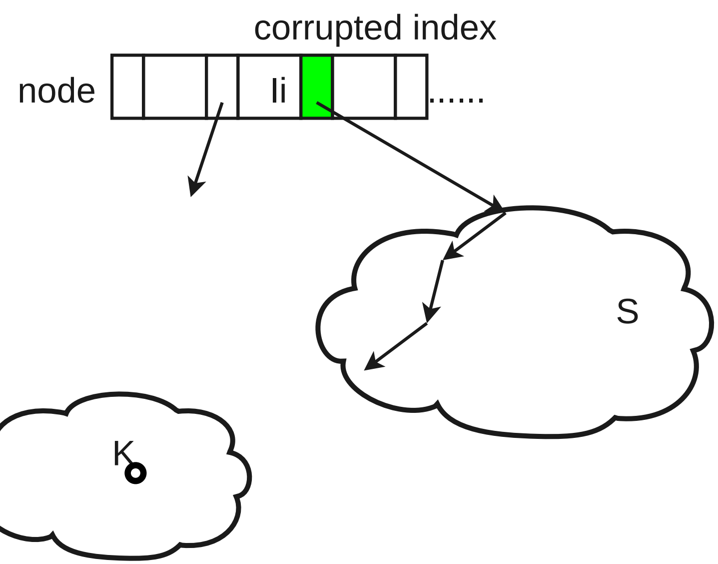
The branching decision

$\Rightarrow$ Robust search solution: remember all the indices along the trace. Those indices are called "suspicious set". If search fails, check if the indices in suspicious set are corrupted.

- If a corrupted index misdirected the search, the correct branching should be the alternate branch.

A closer look

- So check each index in suspicious set? No, expensive.
- Assume single error. There is a smart solution.
  - What happens after a corrupted index misdirected the search?



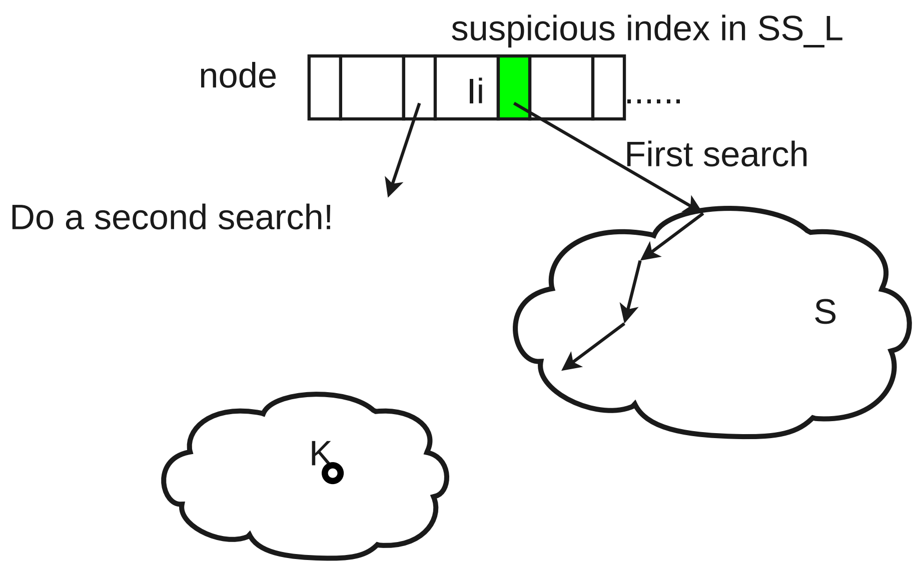
If a previous index which directs the search to R was corrupted, during the rest of the search, indices chosen in the nodes must be the smallest

Maintaining suspicious set

- So, if an index encountered during the search is not the smallest one, the direction R given by the previous index must be correct.
-
```
Procedure UPDATE_SS(n: node; i: index)
if I_i not the smallest index then
    delete(SS_R)
    if n is not a leaf then
        add(SS_R, (n, i, R))
    end if
end if
if I_i not the largest index then
    delete(SS_L)
    if n is not a leaf then
        add(SS_L, (n, i + 1, R))
    end if
end if
```

Error detection

- Observation: For an unsuccessful search, $SS = (SS_R + SS_L)$ contains at most one index.
- Do a second search on the alternate branch of the suspicious index



  - If found, correct the error.
  - If not found, check BTP again, if it is corrupted, correct the error.

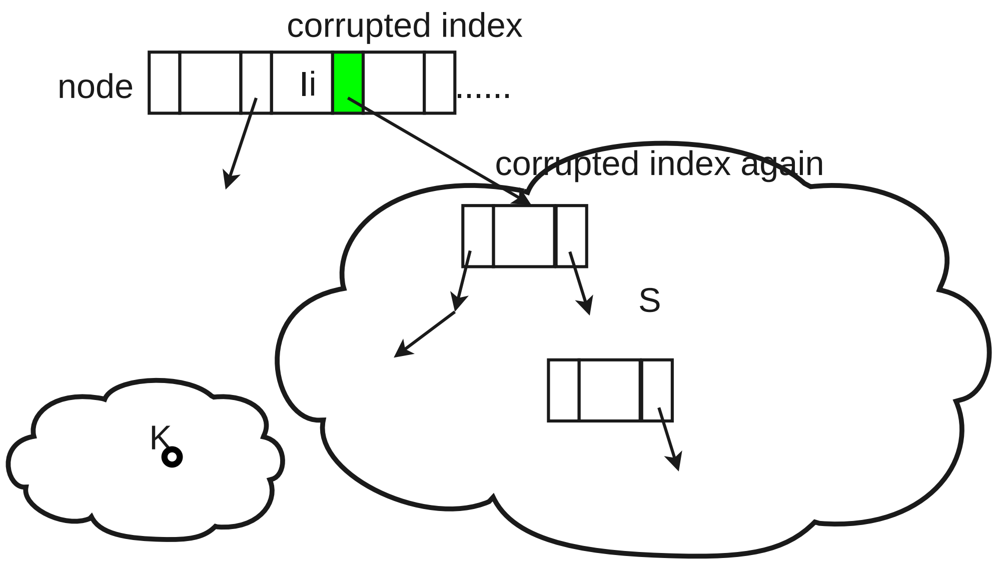
Error correction

How to correct? Change the corrupted index to a value between the largest in left-subtree and smallest in right-subtree.
Error correction comes after a unsuccessful first search.

- If the suspicious index I is in SS_L, you have already reached the leftmost index r in the right branch,
  Let $I = r$
- If I is in SS_R, you have already reached the rightmost index l in the left branch,
  Let $I = l + 1$

Multiple errors

at most m errors.

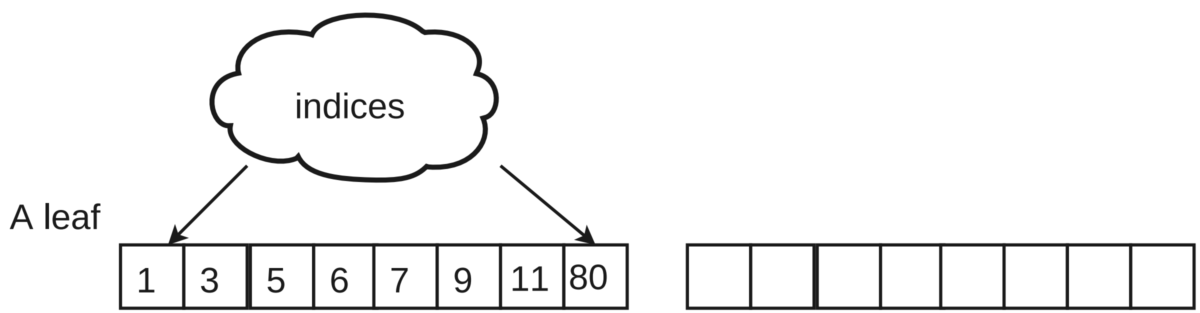


UPDATE_SS_m: delete an element from the queue only if $|queue| == m$.

Discussion

Overhead:

- Storage overhead: a queue, size of 1 (single error), size of m (multiple errors)
- Time overhead when there is no corruption.
  - Maintaining suspicious set (It's no I/O operation).
  - If first search fail and suspicious set not empty, a second search and .... (this probability is low when leaf size is large, not common case).



Suspicious set is non-empty only if you hit the smallest or largest key in the leaf

References

[1] B. Randell, “System structure for software fault tolerance”, IEEE Trans. on Software Eng., June 1975, Vol.SE-1, No.2, pp.220-232

[2] A. Avizenis, "The N-version appraoch to fault tolerance", IEEE Trans. on Software Eng., Dec. 1985, Vol.SE-11, No.12, pp.1491-1501

[3] F. Cristian, "Exception handling and software fault tolerance", IEEE Trans. on Computers, Vol.C-31, No.6, June 1982, pp.531-540

[4] D. J. Taylor, D. E. Morgan, and J. P. Black, "Redundancy in data structures: Improving software fault tolerance", IEEE Trans. on Software Eng., Nov. 1980, Vol.SE-6, No.6, pp.585-594