Efficient Locking for Concurrent Operations on B-Trees.

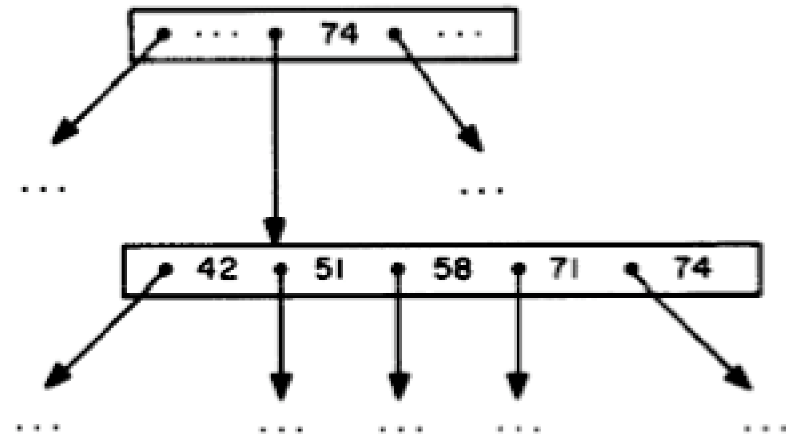
Lehman81: Philip Lehman, S. Bing Yao. ACM Trans. on Database Systems (TODS), vol 6, no 4, December 1981.

Lock-based Concurrency

- B-Trees common concurrent data structure
  - Indices for databases of all kinds
  - Good at fixed, short depth of tree for fast lookup
  - Insertion & deletion can change many nodes (especially if the tree is rebalanced)
  - Simple: lock all nodes that might change as you look for point of insertion
    - But this locks top of tree, blocking everything else
  - Bayer77: don’t read lock top of tree on first try, and hope splitting will not need to change top of tree
    - If wrong, abort and retry holding all locks

Eg. Splitting a B-Tree

- Add item with key 47
  - Node for 47 is full
  - Split node into two
  - Add entry in parent
- If parent is full
  - Propogate up
  - Rotate nodes from deeper to shallow side to preserve depth



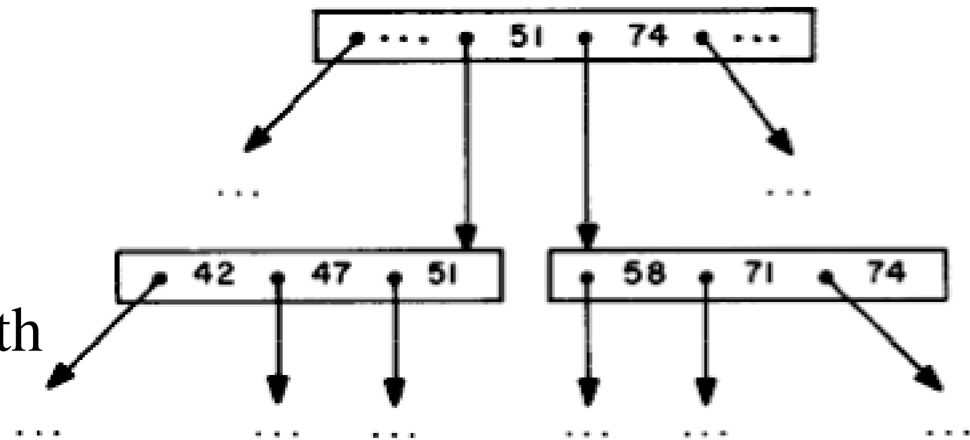


Fig. 4. Splitting a node after adding "47" ($k = 2$).

Lehman81: more concurrency

- Small maximum number of locks held (3)
- Readers are never blocked; tree is always valid
  - Reader may “miss” concurrent insertion
- B-link-tree adds depth-first next-node link
  - Reader searching a node being split may see “highest value” smaller than search key, indicating real node is next in depth-first search
  - Writers only lock an individual node wrt other writers
  - Careful coding of split/rebalance needed
- “Disk” ops get(), put() are indivisable (atomic)

B-Link-Tree Example

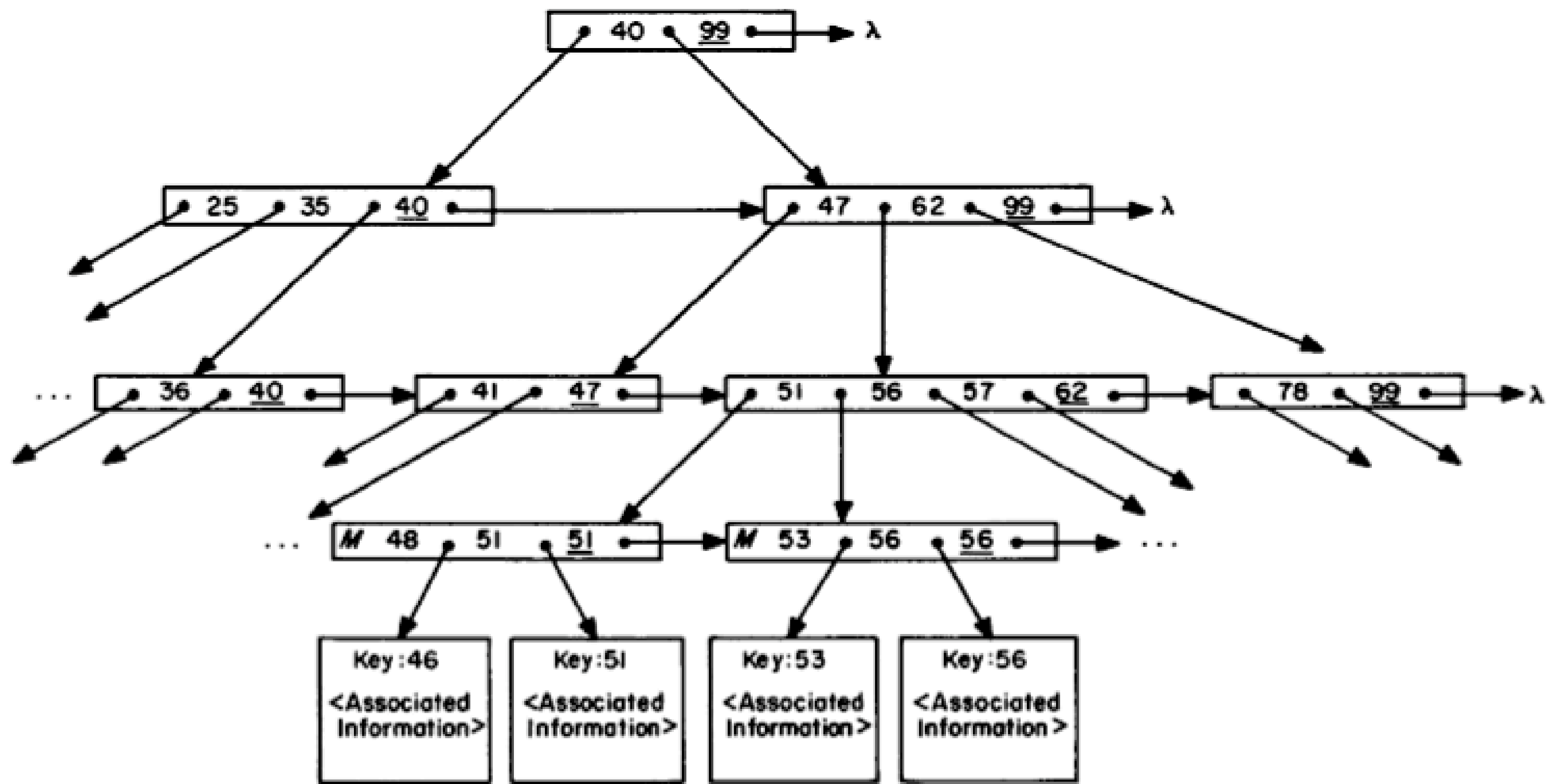


Fig. 7. A B^{link}-tree.

Readers never lock!

$x \leftarrow scannode(v, A)$ denotes the operation of examining the tree node in memory block A for value v and returning the appropriate pointer from A (into x).

```
procedure search(v)
current ← root;                                   /* Get ptr to root node */
A ← get(current);                                 /* Read node into memory */
while current is not a leaf do
begin                                             /* Scan through tree */
  current ← scannode(v, A);                       /* Find correct (maybe link) ptr */
  A ← get(current)                                /* Read node into memory */
end;
                                                  /* Now we have reached leaves. */
while t ← scannode(v, A) = link ptr of A do
                                                  /* Keep moving right if necessary */
begin
  current ← t;
  A ← get(current)                                /* Get node */
end;
                        /* Now we have the leaf node in which v should exist. */
if v is in A then done "success" else done "failure"
```

Insertion

- Inserted value appears at step c, although f still not updated
- Deletion?
- Does tree stay balanced?

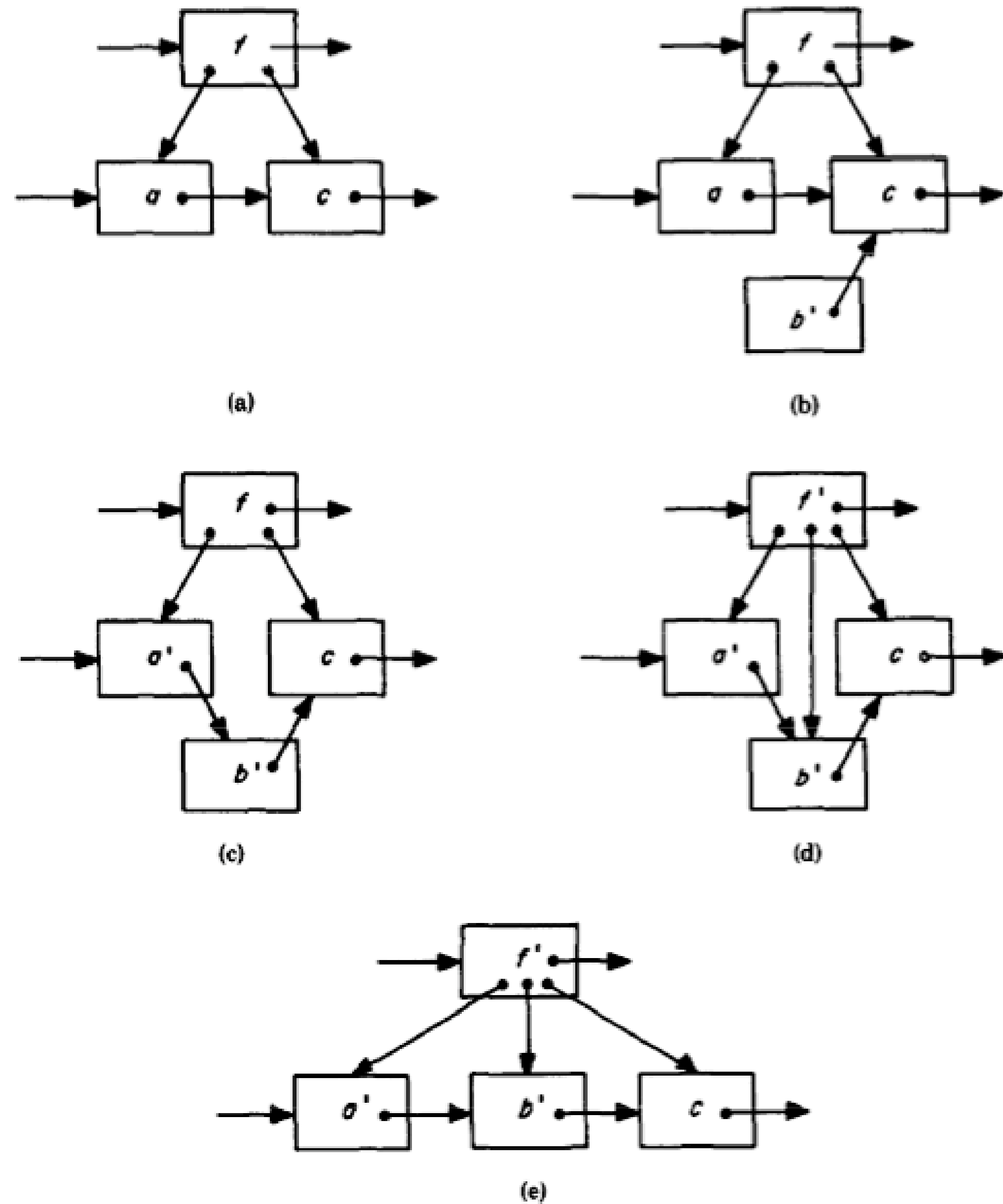


Fig. 8. Splitting node a into nodes a' and b'. (Note that (d) and (e) show identical structures.)

Insertion

- Nasty code!
- Eraser would ….

```
procedure insert(v)
initialize stack;                                          /* For remembering ancestors */
current ← root;
A ← get(current);
while current is not a leaf do
begin                                                      /* Scan down tree */
  t ← current;
  current ← scannode(v, A);
  if new current was not link pointer in A then
    push(t);                                               /* Remember node at that level */
  A ← get(current)
end;
lock(current);                                             /* We have a candidate leaf */
A ← get(current);
move.right;                                                /* If necessary */
if v is in A then stop "v already exists in tree";         /* And t points to its record */
w ← pointer to pages allocated for record associated with v;
Doinsertion:
if A is safe then
begin
  A ← node.insert(A, w, v);                  /* Exact manner depends if current is a leaf */
  put(A, current);
  unlock(current);                                         /* Success—done backtracking */
end else begin                                             /* Must split node */
  u ← allocate(1 new page for B);
  A, B ← rearrange old A, adding v and w, to make 2 nodes,
    where (link ptr of A, link ptr of B) ← (u, link ptr of old A);
  y ← max value stored in new A;                           /* For insertion into parent */
  put(B, u);                                               /* Insert B before A */
  put(A, current);                                         /* Instantaneous change of 2 nodes */
  oldnode ← current;                                       /* Now insert pointer in parent */
  v ← y;
  w ← u;
  current ← pop(stack);                                    /* Backtrack */
  lock(current);                                           /* Well ordered */
  A ← get(current);
  move.right;                                              /* If necessary */
  unlock(oldnode);
  goto Doinsertion                                         /* And repeat procedure for parent */
end

procedure move.right
while t ← scannode(v, A) is a link pointer of A do
begin                                                      /* Move right if necessary */
  lock(t);                                                 /* Note left-to-right locking */
  unlock(current);
  current ← t;
A ← get(current);
end
```