University of Toronto
Scarborough Campus

Computer Science B28
Spring 1990

ALGORITHMS FOR B+ TREES

FIND(K)
% This algorithm returns the block where the record with key K is
% stored, if it is in the file at all.

```
b := root of the B+ tree (this may be stored in a special location)
loop
    read b into main memory (if it is not already there)
    exit when b is a data (not index) block
    let b contain P_0 K_1 P_1 ... P_{r-1} K_r P_r
    let i be the number in the range 0 ≤ i ≤ r such that K_i ≤ K < K_{i+1}
        (where we take K_0 = -∞ and K_{r+1} = +∞)
    let b be the block pointed to by P_i
end loop
return b
```

INSERT(R)
% This algorithm inserts record R to the appropriate data block.
% If the insertion of R causes the block to overflow, that block
% is "split" by moving half of its records to a new block that
% becomes its sibling in the B+ tree. To make the new block accessible
% through the index the algorithm INSERTTOINDEX is called to add
% to the index a pointer to the new block and the key that separates the
% block that overflowed and its new sibling

```
K := key of R
b := FIND(K)
read b into main memory (if it is not already there)
if b contains a record with the same key as K then duplicate key error
elsif b has room for one more record then
    put R in b, rearranging records to maintain key sequence
    write b back to disk
else            % b would overflow with the insertion of R
    let R_1, R_2, ..., R_n be the records of b together with R in key sequence
    get a free block b' from the free pool
    put R_1, R_2, ..., R_⌈n/2⌉ in b
    put R_⌈n/2⌉+1 R_⌈n/2⌉+2 ... R_n in b'
    write b, b' back to disk
    let K be the minimum key of a record in b' (i.e. the key of R_⌈n/2⌉+1)
    INSERTTOINDEX(K, b, b')
end if
```

Note: In the algorithm below, the symbol M refers to the order of the B+ tree.

INSERTTOINDEX(K, *splitBlock*, *newBlock*)
% This algorithm makes an insertion to an index block, to account for the
% insertion of *newBlock* which is a block created to accommodate overflows
% that occurred as a result of an earlier insertion to *splitBlock*.
% Hence, *splitBlock* and *newBlock* will be sibling nodes in the B+ tree
% when the insertion is completed. The key that will separate *splitBlock*
% and *newBlock* in their (common) parent will be K. Note that *splitBlock*
% can be either a data block (the first time INSERTTOINDEX is called) or an
% index block (in the recursive calls).

```
if splitBlock is presently the root of the B+ tree then
    get a block newRoot from the free pool
    put in newRoot a pointer to splitBlock and a pointer to newBlock separated by K
    write newRoot back to disk
else
    let parBlock be the parent of splitBlock
    read parBlock into main memory (if it is not already there)
    let P_0 K_1 P_1 ... P_{r-1} K_r P_r be the contents of parBlock together with
       K followed by a pointer to newBlock spliced in so as to maintain key sequence
    if r < M then   % parBlock has room for one more child
        put P_0 K_1 P_1 ... P_{r-1} K_r P_r in parBlock
        write parBlock back to disk
    else            % parBlock was full and must be split; in this case r = M
        get block newParBlock from free pool
        put P_0 K_1 P_1 ... P_{⌈M/2⌉-1} K_{⌈M/2⌉} P_{⌈M/2⌉} in parBlock
        put P_{⌈M/2⌉+1} K_{⌈M/2⌉+2} P_{⌈M/2⌉+2} ... P_{M-1} K_M P_M in newParBlock
        write parBlock, newParBlock back to disk
        INSERTTOINDEX(K_{⌈M/2⌉+1}, parBlock, newParBlock)
    end if
end if
```

Note: In the algorithm below, the symbol M refers to the order of the B+ tree.

```
DELETEFROMINDEX(K, b, emptyBlock)
% This algorithm removes from index block b the pointer to emptyBlock
% and the separator key K (which used to separate emptyBlock from a
% sibling with which emptyBlock was merged). If this removal causes b
% to underflow, we resolve the problem by borrowing or merging, as in the
% DELETE algorithm described above. In the case of a merge,
% DELETEFROMINDEX is called recursively, to remove the reference to a newly
% empied index block. Thus, emptyBlock may have been either a data block
% (the first time DELETEFROMINDEX is called) or an index block
% (in the recursive calls).
  read b into main memory (if it is not there already)
  remove K and the pointer to emptyBlock from b
  return emptyBlock to the free pool
  if b now has at least ⌈M/2⌉ children then    % no underflow
    write b back to disk
  else                                          % underflow
    if b has a sibling b' with more than ⌈M/2⌉ children then % resolve underflow by borrowing
      let parBlock be the parent of b (and b')
      read parBlock into main memory (if it is not already there)
      let L be the separator if b and b' in parBlock
      if b is to the left of b' then
        let P be the leftmost pointer of b'
        let N be the leftmost key of b'
        add L as the rightmost key of b
        remove P from b' and move it as the rightmost pointer of b
        remove N from b' and replace L by N in parBlock
      else      % b is to the right of b'
        let P be the rightmost pointer of b'
        let N be the rightmost key of b'
        add L as the leftmost key of b
        remove P from b' and move it as the leftmost pointer of b
        remove N from b' and replace L by N in parBlock
      end if
      write b, b', parBlock back to disk
    elsif b has a sibling b' then                 % resolve underflow by merging
      let parBlock be the parent of b (and b')
      let L be the separator of b and b' in parBlock
      move the pointers and keys of b' into b, separating them from those already there by L
      if parBlock was the root and b, b' were its only children then
        b becomes the new root of the B+ tree
        return b' and parBlock to the free pool
      else
        DELETEFROMINDEX(L, parBlock, b')
      end if
    end if
  end if
```

```
DELETE(K)
% This algorithm deletes the record with key K, if one exists.
% If, upon deletion, the block that contained the record underflows
% (i.e. is less than half full) that condition is fixed in one of two
% ways: borrowing some records from a sibling, or merging the block
% with a sibling. In the latter case, the algorithm DELETEFROMINDEX
% is called to remove from the index the pointer to the emptied sibling
% block and the key that separates the two blocks that were merged
% into one.
    b := FIND(K)
    read b into main memory (if it is not already there)
    if b does not contain a record with key K then
        return        % done: nothing to delete!
    else
        delete the record with key K from b, maintaining the remaining records in key sequence
        if b is now at least half full then % no underflow
            write b back to disk
        else                                % b underflows
            if b has a sibling b' which is more than half full then   % resolve underflow by borrowing
                move a record from b' to b
                % find the key L that should separate b and b' in their parent
                if b is to the left of b' then let L be the minimum key in b'
                else let L be the minimum key in b
                end if
                let parBlock be the parent block of b (and b')
                read parBlock into main memory (if it is not already there)
                change the key in parBlock that separates b and b' to L
                write b, b', parBlock back to disk
            elsif b has a sibling b' then                              % resolve underflow by merging
                let parBlock be the parent block of b (and b')
                read parBlock into main memory (if it is not already there)
                let L be the separator of b and b' in parBlock
                put all records in b and b' into b, maintaining key sequence
                write b back to disk
                DELETEFROMINDEX(K, parBlock, b')
            end if          L
        end if
    end if
```