BTree Insert Algorithm

Kaminski / 3310

Some Notes about BTrees:

For an M-way B Tree, **M**'s value is determined based on what will physically fit in the file's physical block (the I/O unit) – bigger is better.

So **NormalNode** size is dictated by what M's value is.

If M came out to be = 7, then a node in the file contains:

an array of 7 TP's	i.e.,	TP[0] through TP[M-1]
an array of 6 KV's	i.e.,	KV[0] through KV[M-2]
an array of 6 DRP's	i.e.,	DRP[0] through DRP[M-2]

Note that there's 1 more TP than there is KV/DRP.

What data types of these?

TP's are probably int's (or long's, if the number of BTree nodes could exceeds a 9-digit number)
(or short's, if it's just a small BTree, and disk storage space is an issue)

KV's are based on the KEY field that this index is based on.
(an int for SSN or WIN; a short for internationalPhoneCode, 3 char's for a 3-char charArray like countryCode,
4 char's for a 3-char string like countryCode)

DRP's are probably int's (or long's, if the number of data/DB records exceeds a 9-digit number)
(or short's, if it's just a small data file, and disk storage space is an issue)

BigNode is 1 triple bigger than a NormalNode: it's got 8 TP's, 7 KV's, 7 DRP's: TP[0] to TP[M], KV[0] to KV[M-1], DRP[0] to DRP[M-1].

Some B Tree "**RULES**" (which the Insert algorithm needs to maintain):

1) The good data is left-justified within a node.
2) Unused locations on the right end of a node contain:
   - 0's for TP's (which "points nowhere", assuming that RRN's in the BTreeIndex file start at 1, not 0)
   - HighValue for KV's (the HighValue must be > any possible legal KV, ASCII-code-wise, which assumes comparisons use C#'s CompareOrdinal rather than just Compare)
     (for example, if KV's were 3 char's of all capital letters, one could use "]]]" as the HighValue)
   - 0's for DRP's (which "points nowhere", assuming that RRN's in the main data/DB file start at 1, not 0).
3) The KV's in a node are always in non-descending (~ ascending) order.
4) Any new KV (and its accompanying DRP) is always inserted in a leaf node.
5) All leaves are at the same level, so when the tree grows taller, it's at the root end, not the leaf end.
6) All nodes are at least half full of good data (the left-most 3 KV's, 3 DRP's, 4 TP's, if M = 7)
   – except the root which can have a minimum, 1good KV/DRP with the 2 TP's (on the left end)

The program handles file space management, so nextEmpty is used to keep track of the next RRN value to be used.
(One could instead keep track of N, which is the same as nextEmpty-1).

The BTree's **initialization**: make a tiny BTree with nothing in it -that is, initialize: rootPtr = 0 and nextEmpty = 1.

A **stack** is needed to store RRN's during the search part which travels DOWN from root to leaf,
so as to allow travelling back UP the tree when a split is needed.

Insert's **caller** sends in: the newKV and its newDRP plus a 0 for the newTP.
(This newTP value of 0 wouldn't be necessary on the first call, but for reusability of the methods in the Split, a newTP value is needed).

A B Tree index is an **external** data structure, and so is created/stored/maintained on a **file** rather than in memory.

Individual node creation/manipulation is done in memory, however. (So there's generally never more than 1 node in memory at once).

The B Tree is stored in a **RANDOM ACCESS, BINARY** file – and hence needs fixed-length records. So calculate NODE_SIZE
(in bytes, needed for calculating offset for use in seeking) once and for all (a defined constant):

```
( (M-1) * ( sizeof(int) + sizeof(KV) + sizeof(int) ) )  +  sizeof(int)
```

(since presumably the TP's and DRP's are both int's).

A HeaderRec is generally used for storing singleton values related to the B Tree, e.g.,
M (a defined constant value), rootPtr (an RRN value), nextEmpty (an RRN value), nKV (number of KV's in BTree).
So calculate HEADER_SIZE (in bytes, needed for calculating offset for use in seeking) once and for all (a defined constant):

```
4 (or whatever) * sizeof(int)
```

(since presumably all these are int's).

BTree's INSERT algorithm

if rootPtr == 0 then it's a SPECIAL CASE, so call **MakeBTreeGrowTaller**
else it's the NORMAL CASE, so

1. call **GetCorrectLeaf**
2. call **InsertTripleInBigNode**
3. *[We now want to put the node back into the file, BUT we first have to check if it will fit back].* That is,
   if BigNode is FULL (that is, `KV[M-1] != HighValue` anymore)
   then call **SplitBigNode**
   else call **WriteANode**, sending in the RRN from the top of the stack. *[And we're done with this INSERT !!]*

SplitBigNode *(which is recursive algorithm):*

[BigNode is 1 triple too big to fit into the file's storage spot for a normal node.
So, since it's FULL, it needs to be split up before putting the data it contains into the file.
It's split into 3 parts].

1. the left "half" of BigNode gets written back to the spot where it came from. That is,
   (Don't wreck BigNode - you still need to use it below. Do your fixing up of a node to Write in NormalNode).
   a. move BigNode's left-"half" data to the left half of NormalNode,
   & "clean up" the right "half" of NormalNode.
   b. call WriteANode, sending in RRN from the top of the stack and pop that RRN off the stack.

2. the right "half" gets written to a NEW location.
   a. fix up NormalNode with BigNode's right-"half" data, putting that in the left "half" of NormalNode,
   & "clean up" the right "half" of NormalNode *(same as clean-up in 1a above).*
   b. call WriteANode, sending in nextEmpty as the RRN.
   c. so increment nextEmpty.

3. the middle KV, DRP & nextEmpty-1 *(since you just incremented it above)* get pushed up to be inserted into the parent node.
   (NOTE: nextEmpty-1 is the TP for the NEW location in step 2 above, so that NEW node will get hung on the BTree here).
   a. assign the middle KV, middle DRP & nextEmpty-1 to newKV, newDRP & newTP
   b. to get the parent node, pop off the RRN from the top of the stack
   IF you can NOT POP (since the stack is empty), then
   call **MakeBTreeGrowTaller**
   and EXIT RECURSION (INSERT is now done!)
   ELSE
   call **ReadANode** giving it the RRN you just popped off the stack which gets a NormalNode,
   put the NormalNode into BigNode (on the left end)
   put HighValue into KV[M-1] as a sentinel.
   c. call **InsertTripleInBigNode**
   d. if BigNode is FULL (that is, `KV[M-1] != HighValue` anymore)
   then call **SplitBigNode** (a recursive call)
   else call **WriteANode**, sending in the RRN from the top of the stack
   and EXIT RECURSION (INSERT is now done!)

ReadANode (random access version which needs RRN, NODE_SIZE, HEADER_SIZE)
reads in a single designated NormalNode from the file

WriteANode (random access version which needs RRN, NODE_SIZE, HEADER_SIZE)
writes out a single NormalNode to a designated location in the file

MakeBTreeGrowTaller does these steps:

1. the Physical Storage step:
   - call **InitializeANormalNode** (in memory), that is: 0's in the M TP's, [0] to [M-1]
     HighValue's in the M-1 KV's, [0] to [M-2]
     0's in the M-1 DRP's, [0] to [M-2]
   - put the new parameter values into the node:
     - newKV & newDRP into KV[0] & DRP[0] *(so for loop above could start at 1, not 0)*
     - newTP into TP[1] *(note that it's [1], not [0])*
   - put the rootPtr into TP[0]
   - call **WriteANode**, sending in nextEmpty for the RRN
2. hang this new node on the BTree: that is, change rootPtr to nextEmpty's value
3. increment nextEmpty

GetCorrectLeaf does a root-to-leaf search
calling **ReadANode**, as needed, based on the correct RRN found at the prior level in the tree.
While travelling down the tree, push the RRN's of the nodes visited onto a stack which will be needed later.
Once a leaf is reached (knowable because all leaf nodes have TP[0] == 0)
put the leaf node (in NormalNode) into BigNode (on the left end)
& put HighValue into KV[M-1] as a sentinel.
NOTE: The root node may actually be a leaf node, so use appropriate initializations/pushing/reading/checking….

InsertTripleInBigNode puts the newKV, newDRP, and newTP in the correct spot in BigNode
using the shift-triple-to-right approach, starting at the right end
where the comparisons (C#'s CompareOrdinal, not Compare) are made based on KV's
and the DRP & TP in the triple just go along for the ride when shifting to the right.
NOTE: Inside the loop, when shifting a triple, TP's subscript value is always 1 > KV's & DRP's subscript value.
NOTE: Since this is using BigNode (which is 1 triple bigger than NormalNode),
there will ALWAYS be a spot for inserting newKV, newDRP and newTP),
though allow for the rightmost triple (containing HighValue 0 0) to be shifted off the edge out of existence.