2-3 Search Tree algorithms

Describing a NODE (in the asgn)
Generic 2-3 search trees use a node which looks like:

LChPtr (KV1 DRP1) MChPtr (KV2 DRP2) RChPtr

It's an M-way tree (a Many-way tree), where M=3. That is, each node can have at most 3 children.
For the asgn we’re using a 2-3 search tree for an INDEX, so:
- The Data fields would be the index’s KeyValues (KV) = the Titles (after capitalizing).
  [This is the “comparison field”].
- And, since it’s a dense index, every KV has a DataRecordPtr (DRP) paired up with it.
- So a node looks like:
  LChPtr (KV1 DRP1) MChPtr (KV2 DRP2) RChPtr
But to make this more modifiable for the future (e.g., change to a B-Tree) asgn3 does the following:
- Since there are 2 kinds of “pointers” involved, I call the child pointers “TreePtrs” (TP) (vs. DRP).
- And rather than give them each a special names, just call them TP1, TP2, TP3.
- So a node looks like:
  (TP1  KV1 DRP1)  (TP2  KV2 DRP2)  TP3
  (that’s 3 “triples”, plus an extra TP) (that’s M-1 triples, plus the extra Mth TP)
- But if this were a fatter tree with more triples (like a B-Tree might have M=43 instead of M=3)
  I wouldn’t specifically name any of the fields, but rather use arrays (and start at [0]), so:
- But to put this in a program with arrays, it’d be easier to re-arrange the fields as:
  a Node is a structure defined as:
  An array of M TPs  integers
  An array of M-1 KVs  20-char titles (which have been capitalized)
  An array of M-1 DRPs integers

Node STORAGE
Asgn 3 uses a LINKED, ARRAY-based implementation of the tree, so needs:
- an array of nodes (static, of size MaxNodes=50 will do for now – or dynamic)
- NextEmpty (which = NTitleNodes)
- an explicit RootPtr
So “pointers”, like TP’s, NextEmpty, RootPtr, are all subscripts (integers)
- and “points nowhere” is a -1 (rather than a NULL).

Other Issues
- RootPtr will change as Data is inserted in the tree (for plain BST’s, it wouldn’t)
- RootPtr is initialized to -1, NextEmpty to 0
- a node is a leaf if TP[0] = -1 (NOTE: both leaf nodes & non-leaf nodes might have other TP[2] = -1)
- “empty” KV’s contain 20 ‘^’s and so are greater than “Z…” (in terms of their ASCII codes). The algorithms take advantage of this - (i.e., any newKV or targetKV is < an “empty” KV
  - This assumes Titles start with a letter (smalls converted to caps), digit, or most special char’s.
- Problems if a Title starts with 1 of 6 “other” special char or uses “extended ASCII”
- Algorithms assume that there will be no duplicate KV’s in the tree.
- for debugging, use a Snapshot of RootPtr, NextEmpty, the NodeArray [0 .. NextEmpty-1]

Create
If RootPtr = -1 then /*special case */
  Use NextEmpty as the storage location (which is [0])
  Put these values in node[NextEmpty]:
    newData in KV[0] and DRP[0]
  Fix RootPtr := NextEmpty
  Increment NextEmpty
Else /* normal case */
  insertInTree (newData, RootPtr)

InOrder Traversal
The Call: InOrder (RootPtr)

InOrder (IN: Ptr)
  If Ptr != -1 call InOrder (using TP[0])
  Visit KV[0]  (i.e., use DRP[0]. . .)  
  ...       If KV[1] != "empty"
  Visit KV[1]  (i.e., use DRP[1]. . .)  call InOrder (using TP[2])
  Else
  unsuccessful search - do . . .
  Else  successful search - use DRP to . . .

Search (which returns DRP)
The Call: Search (targetKV, RootPtr) which returns DRP
  If DRP = -1 /* empty tree OR fell off bottom of tree without a match */
  Return -1 for DRP
Else
  Case targetKV
    < KV[0]  : call Search(targetKV, TP[0])
  = KV[0]  : return DRP[0] for DRP
  < KV[1]  : call Search(targetKV, TP[1])
  Otherwise: call Search(targetKV, TP[2])
Insert KV/DRP in Tree

NOTE: A newKV (& newDRP) is always inserted in a leaf – whether it fits or not!
NOTE: This algorithm assumes:
- no deletes can happen, so there can never be an empty tree after CREATE
- there will always be at least 1 node in the tree since CREATE handled
  the initial empty tree situation

NOTE: BigNode is “1 triple bigger” than a regular node – in generic terms:

NOTE: Split will refer to the above as the left “half”, the middle, and the right “half”
NOTE: There are many different algorithms for doing this, including some recursive ones. Any would
be acceptable. Some are more/less efficient (run-time? Design-time? Test/debug-time?)

The Call: InsertInTree (newKV, newDRP, RootPtr)

InsertInTree (IN: newKV, IN: newDRP, IN: Ptr)
  Search tree from root to leaf for correct leaf
  pushing each node location onto LocStack
  (in case you need to go back up the tree in “Split”)
  - [RootPtr was sent in, so you know where to start]
  - [and you know it’s a leaf when TP[0] = -1]
  Call Prepare&Insert with (newKV, newDRP, -1, . . .) which does:
    1) move node at [top of LocStack] to BigNode (the left end)
       & initialize the rightmost triple: KV[2], DRP[2], TP[3]
    2) put newKV & newDRP & 3rd parameter (the TP)
       into BigNode in the correct spot
       (i.e., keeping the KVs in BigNode in sequence,
         shifting KV[1] DRP[1] TP[i+1] to right til…)
  while BigNode has spill-over into rightmost triple
  [= see if rightmost triple still looks like your initialization]
  [NOTE: this is a pre-test loop – so body won’t get executed if
   the prior insert did NOT cause a spill-over]
  {  
    split BigNode:
      1) left “half” gets put back in original location
      2) right “half” gets put into a new location
      3) middle KV/DRP & #2’s new location get sent up to parent
         [NOTE: it doesn’t matter if parent has room for the new triple or not – at this point]
         - This step fills BigNode, so it’s ready for while loop check
  }

  Move BigNode to the node at [top of LocStack]
  (except you’re not moving that rightmost triple of BigNode)

  Split BigNode details:
  1) left “half” of BigNode includes TP[0] KV[0] DRP[0] TP[1]
     Put those in the left end of node at original location (which is at [top of LocStack])
     & the right end of that node needs initializing
     Pop the location off the LocStack since you’re done with it
     Put those in the left end of a NEW node (which is at [NextEmpty])
     & the right end of that node needs initializing
     increment NextEmpty
  3) middle KV & DRP & NextEmpty-1 (since it’s already incremented in #2)
     are sent up to the original node’s parent.
     That parent’s location is at the top of LocStack
     (unless there’s no parent, if original node was the root)

So…
If LocStack empty /* special case */
  Create a new root node using NextEmpty location
  Put RootPtr into TP[0] of this new node
  Put BigNode’s KV[1] DRP[1] along with NextEmpty-1
  into left end of this new node
  Initialize the right end of this new node
  Fix RootPtr to be NextEmpty
  Increment NextEmpty

Else /* normal case */
  Call Prepare&Insert (KV[1], DRP[1], NextEmpty-1, . . .)