BINARY SEARCH TREE (BST) Algorithms

Implementations of a Binary Tree (a BT, not a BST):

A. LINKED (explicit ptrs – i.e., actually stored) – good for BSTs
   1. Dynamic-Memory-based
      Node storage: allocated by system upon request
      Ptrs: references to memory locations
   2. ARRAY-BASED
      Node storage: allocated by the program from set of static memory locations – an array of nodes
      Ptrs: subscript values

B. LINKED (implicit ptrs – i.e., formulas to calculate) – good for Heaps
   1. Node storage: array
      Ptrs: formulas to calculate where child/parent node is

A BST needs:
1. Node storage space
2. A RootPtr
3. N (if using manual node-storage allocation)

A single BST node consists of:
LeftChildPtr, KeyValue (KV), (Other data ???), RightChildPtr

Ptr's include:
- all LeftChildPtr's & RightChildPtr's in the nodes
- the single RootPtr
- [and, for a static array implementation] NextEmpty, which = N since array locations start at 0.
  [assuming dynamic delete algorithm (below) is used. If static delete algorithm is used, - But NextEmpty != N]

If static array locations (A2 above) are used, ptrs are subscript values (integer).
The "null" ptr situation uses -1 to indicate "points nowhere" since 0 is a valid array location.

Alternative algorithms sometimes use recursion since BST are recursive structures
I show iterative algorithms below.

AN ASIDE: Basic template for event-controlled loops

HUMAN ALGORITHM:

```
initialize i
loop:  stop when you EITHER find success (at i)
                OR decide the search is unsuccessful
{   some processing perhaps ?
    increment i
}
if found success
    do processing for SUCCESSFUL case at i
else
    do processing for UNSUCCESSFUL case
```

Two issues on LOOP's conditions:

1 – Humans use: STOPPING condition: A OR B
Programs typically use: while's CONTINUE condition: ! (A OR B) = !A AND !B

2 - Program's order of the 2 conditions may need reversing:

```
while (! fall off end of DataStructure AND ! find match at i)
```

because it's not legal to check "`find a match at` i" when you're already off the end of the DataStructure

One issue on IF's condition:

Often times, you could ask EITHER: if found success... else...
OR: if decided it's unsuccessful... else...

But for some particular uses of the looping algorithm,
BOTH conditions COULD be true at the same time,
& thus both are reasons why the loop is stopping.
So would that specific case fall into successful or unsuccessful handling?
So the if condition would have to funnel control into the appropriate case.

a BST SEARCH Algorithm

1) Do Search

```
i = RootPtr
while (i != -1) and (target != BST.data [i])
{   if target < BST.data [i]
        i = LChPtr [i]
    else
        i = RChPtr [i]
}                                                    2) Handle it
if i == -1
    handle UNSUCCESSFUL search situation
else
    handle SUCCESSFUL search situation
```

(NOTE: i now points to the desired node)

BST_VISIT-ALL-NODES-IN-SEQUENCE_algorithm
use INORDER TRAVERSAL of a Binary Tree (recursive algorithm)

a BST_INSERT_algorithm

1) physically insert data into a node

```
BST.data   [N] = new data
BST.LChPtr [N] = -1
BST.RChPtr [N] = -1
```

2) hang node on BST in correct spot

USES variables: int parentI *AND* boolean LorR

```
if RootPtr == -1            [special case - no nodes in BST yet]
    RootPtr = N
else                        [normal case]
{   i = RootPtr
    while i != -1
    {   parentI = i
        if BST.data [N] < BST.data [i]
             i = BST.LChPtr [i]
             LorR = L
        else
             i == BST.RChPtr [i]
             LorR = R
    }
    if LorR == L
        BST.LChPtr [parentI] = N
    else
        BST.RChPtr [parentI] = N
}
```

3) *fix N & thus the implied NextEmpty location*

```
increment N
```

a BST_(Static) DELETE Algorithm

Do BST SEARCH
If you find target, then tombstone it

NOTE: Search would need to be modified to disallow matches where the BST node was previously tombstoned. Similarly, VisitAllNodesInOrder would need to be modified to disallow tombstones being considered usable nodes.

a BST (Dynamic) DELETE Algorithm

NOTE: "Dynamic Delete" algorithm is what's typically meant by "BST Delete")

1) Do Search & keep track of parentI & LorR]

```
i = RootPtr
while (i != -1) and (target != BST.data [i])
{   parentI = i
    if target < BST.data [i]
         i = LChPtr [i]
         LorR = L
      else
         i = RChPtr [i]
         LorR = R
}
if i == -1                                   2) Handle it
    handle UNSUCCESSFUL query
else [handle successful deletion – see Final sub-algorithm in the right-hand column]
    if it's case 1 or case 2
        . . .
    else if it's case 3
        . . .
    else it's case 4
        . . .
```

3) return node to available node pool - procedure not specified here

---

[A NOTE on CASES: determine: 1 - how do you RECOGNIZE you're in case X ?
2 - what do you DO when you've got a case X?
After developing all the pseudocode, can some cases be combined into same code?]

CASE 1 (i.e., node [i] is a leaf)

```
if LChPtr[i] == -1 and RChPtr[i] == -1        it's case 1
    if LorR == L                              so handle it
         LChPtr[parentI] = -1
    else
         RChPtr[parentI] = -1
```

CASE 2 (i.e., node [i] has a RChild by no LChild)

```
if LChPtr[i] == -1 and RChPtr[i] != -1        it's case 2
    if LorR == L                              so handle it
         LChPtr[parentI] = RChPtr[i]
    else
         RChPtr[parentI] = RChPtr[i]
```

CASES 3 & 4 (i.e., node [i] has a LChild) so:

1st) find the predecessor value of node [i]
2nd) then that node will replace node [i] by fixing 4 pointers

Where's the predecessor value?

In the left subtree of node [i], then go right, go right,... til you can't.

Case 3, node [i]'s LChild has no RChild, so no looping needed to find predecessor
- predecessor is at [LChPtr[i]]

Case 4, you need to loop to find the predecessor

```
parentP = LChPtr[i]
p = RChPtr[parentP]
while RChPtr[p] != -1
{   parentP = p
    p = RChPtr [p]
}
```

- predecessor is now at [p]

Final sub-algorithm for "handle successful deletion"

```
if LChPtr[i] == -1                                [CASE 1 & 2 combined]
    1)  if LorR == L
              LChPtr[parentI] = RChPtr[i]
        else
              RChPtr[parentI] = RChPtr[i]
else if RChPtr[LChPtr[i]] == -1                   [CASE 3]
    1)  if LorR == L
              LChPtr[parentI] = LChPtr[i]
        else
              RChPtr[parentI] = LChPtr[i]
    2)  RChPtr[LChPtr[i]] = RChPtr[i]
else                                              [CASE 4]
    0)  find predecessor:
            parentP = LChPtr[i]
            p = RChPtr[parentP]
            while RChPtr[p] != -1
            {   parentP = p
                p = RChPtr [p]
            }
    1)  RChPtr[p] = RChPtr[i]
    2)  if LorR == L
              LChPtr[parentI] = p
        else
              RChPtr[parentI] = p
    3)  RChPtr[parentP] = LChPtr[p]
    4)  LChPtr[p] = LChPtr[i]
```

Three Save / Restore BST algorithms (Memory ← → File)

1) for array-based implementations.

This produces the same exact BST when restored.
It does NOT correct a skewed BST.

SAVE:

```
write N
for loop   i: 0 . . N-1
      write node [i]          [i.e., LChPtr, KV, (???) RChPtr]
```

RESTORE:

```
read N
for loop   i: 0. . N-1
      read node [i]           [i.e., LChPtr, KV, (???), RChPtr]
```

- -

2) for array-based or dynamic-memory-based implementations.

This produces the same exact BST when restored.
It does NOT correct a skewed BST.

SAVE:

```
write N
```
do PREOrder Traversal (i.e., "PLR")
[where "visit node P" = write KV (& ???) at node P]
[do not write LChPtr & RChPtr]

[NOTE: File now contains just KVs (& ???s) but no ChPtr's.
File is NOT in SORTED order,
nor is it in the same order as the original input file.
But it will produce the same BST as the original CREATE did.]

RESTORE:

```
read N
for loop   i:  0. . N-1
      read a node
      call BST Insert algorithm
```

- -

3) for array-based or dynamic-memory-based implementations.

This produces a Balanced BST (of minimum height) when restored.
It is thus NOT the same exact BST as the original one.

SAVE:

```
write N
do INOrder Traversal
```
where "visit node P" means:
write KV (& ???) at node P
but do NOT write LChPtr & RChPtr

NOTE: The file is now in SORTED order by KV.

RESTORE (a recursive algorithm):

```
read N
readTree (IN N:int, OUT P:"ptr" [an int or ptr] )
      if N > 0
            P ← "ptr" to new node
                  with "NULL" LChPtr & RChPtr
            readTree (N/2, LChPtr at P)
            read KV (& ???) from file
                        into KV (& ???) at P
            readTree ((N-1)/2), RChPtr at P)
rootPtr ← P
```