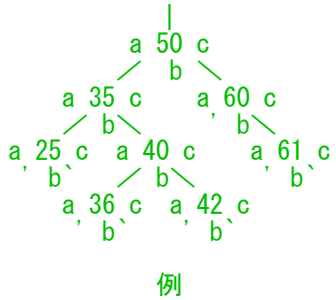
```
0001: !*****************************************************************************************
0002: !***  「木(Tree)」へのデータの逐次追記 ＆ 一括取出し                                    ***
0003: !***                                  使用データ構造：「2分探索木 (Binary Search Tree)」 ***
0004: !***                                使用アルゴリズム：「挿入(insert)」＆「横断(traversal)」 ***
0005: !*****************************************************************************************
0006: !
0007: ! 用途）FEMにおける「隣接行列(adjacency matrix)」（全体自由度間の接続関係）の組立て
0008: !       （ 隣接行列の各行を2分探索木で表現                                          ）
0009: !       （ よって，行列全体は多数の2分探索木から成る「森(forest)」で表現される ）
0010: !
0011: ! 注）「データ構造とアルゴリズム」の基礎である「リスト(list)」，「再帰(recursion)」，
0012: !     さらには「スタック(stack)」の知識があるとよい．
0013: !
0014: !          スタック：行き掛けに何らかの目印を記憶しておき（push），
0015: !                    帰り掛けにその記憶を取り出す（pop）ことができるようにするための記憶域
0016: !
0017: !
```

```
!*****************************************************
! 2分探索木(Binary Search Tree)
!
! データ構造の一種であり，その基本単位(「頂点(vertex, node)」)は構造型とポインタで実現される.
!
!   長) 繰り返しのデータ追記（＆削除）に対して，*見かけ上*のデータ列を常に整列状態とできる.
!        (素直な配列では毎回の再整列に伴う大量のデータ移動が必要だが，こちらはメモリ割付のみ)
!       これはデータ探索に有利であり，O(n)の線形探索でなくO(log(n))の2分探索とできる.
!
!   短) 構造型とポインタの考え方によるため，分かりやすい配列表現に比べ，実現が複雑化.
!
!  ・「2分木」とは，上から下に「根(親)」から左右に2分された「枝(子)」を伸ばしていくもの.
!      (ゆえに本質的に再帰での記述が，ラクかつ本質的)
!  ・「2分探索木」とは，2分木の一種であり，左の枝(子)に根(親)より小さな値を，
!      右の枝(子)に根(親)より大きな値を入れていく制限を課したもの.
!     (この制約により，見かけ上のデータ列を常に整列状態とできて，データ探索がラクになる)
!
!   例)                 |                              |
!                      50                             25
!                    ／  ＼                             ＼
!                  35      60                             35
!                ／  ＼  ／  ＼                             ＼
!              25     40  55     61                            40
!                          ＼                                    ＼
!                           58                                    50
!              (バランスが良い場合)          (バランスが最悪の場合：一方向連結リストと同じ)
!
!   補足)
!        ・「木」は，閉路のない有向グラフともいえる
!        ・ある頂点以下の部分に注目するとき，特に「部分木」という
!
! なお，基本単位（頂点）のデータ構造と，その一般的な表現は下記だが，
!
!    TYPE, PRIVATE :: vtx                                 !        _______
!       INTEGER :: c                       ! データそのもの  !       |___c___|
!       TYPE(vtx), POINTER :: L => NULL()  ! 左の子を指す    !       |_L_|_R_|
!       TYPE(vtx), POINTER :: R => NULL()  ! 右    〃       !      ／       ＼
!    END TYPE vtx                                         !     v           v
!
! 再帰アルゴリズムの理解を容易化するために，基本単位のデータ構造とアルゴリズムを次のように融合
! 表記しておく. (再帰はらせん階段を降っていくようなイメージ. どこで一周したのかが分かり辛い)
!
!    RECURSIVE SUBROUTINE kernel ( ptr )
!       TYPE(vtx), POINTER, INTENT(inout) :: ptr  ! ポインタ結合状態へのINTENT指定は F2003
!       IF ( .NOT. ASSOCIATED(ptr) ) RETURN
!       ! a) ここに ptr%c が入れば，行きがけ順(前順)に       !      ptr  v              ←仮引数
!       CALL kernel ( ptr%L )                           !    a   ___|___   c
!       ! b) ここに ptr%c が入れば，通りがけ順(中順)に       !       |_ptr%c_|
!       CALL kernel ( ptr%R )                           ! ptr%L ／     ＼ ptr%R  ←実引数
!       ! c) ここに ptr%c が入れば，帰りがけ順(後順)に       !     v    b    v
!    END SUBROUTINE kernel
!
! このとき，2分木の再帰処理は左図のように表現できる.              root |
!                                        __0th_FL__             v              _____
!                                                               v
!                                                           a ___|___ c
!                                                            |___50__|
!                                                           ／  b   ＼
!                                        __1st_FL__        v         v         _____
!                                                          v         v
!                                                  a ___／__ c       ＼ NULL()
!                                                   |___35__|
!                                                  ／  b   ＼
!                                        __2nd_FL__ v         v                _____
!                                                   v         v
!                                          NULL() ／           ＼ NULL()
!
```

```
0086:
0087: !*****************************************************
0088: ! データ逐一追記のための 2分探索木への「挿入(insert)」 アルゴリズム
0089: !
0090: !    まずは該当データがストア済みであるか否かを根から再帰的に順に調べる（データ探索）.
0091: !    すなわち，該当データが自身と一致すれば追記せずに終了するが，より小さければ左の子へ，
0092: !    大きければ右の子へ繰り返し調べていく（3択問題）．根よりはじめて末端まで調べきると，
0093: !    ストア済みでない新規データであることが確定するので，最後に調べたものを新たに親として，
0094: !    その子としてストア．その際，親より小さければ左の子，大きければ右の子となる.
0095: !
0096: !    補足）上の例からも分かるように，データのストア順により木の形が変わる.
0097: !          左右にバランスよくストアできると，値の挿入・探索ともに O(log(n)) で処理できる.
0098: !          しかし，整列済みのデータを逐次追記すると，左右どちらか一方にのみ伸びるので，
0099: !          最悪 O(n) の線形探索となるので要注意.
0100: !
0101: !    蛇足）バランスを保った状態で2分探索木を作っていく高級アルゴリズム（例：赤黒木，AVL木）
0102: !          もあるが，FEM用途ではほぼ不要.
0103: !          理由：FEMでは，そもそもバランスが悪くなる木は少なく，木の頂点数（各自由度が
0104: !          接続する自由度の数，隣接行列の1行あたりの非ゼロ数）も平均で数10から多くとも100未満.
0105: !          また，木の本数（総自由度数）は数千から数百万なので，バランスが悪い木の計算コスト
0106: !          は無視できる.
0107: !
0108: !
0109: !*****************************************************
0110: ! 全データー括取出しのための 2分探索木の「横断(traversal)」アルゴリズム
0111: !
0112: !    「横断(traveral)」とは，一定の手順で木の全ての「頂点」を訪れることをいう.
0113: !    すなわち，木の根の左側からぐるっと木を眺めてまわり，木の根の右側に至る．このとき，
0114: !    各頂点周りの領域に a,b,c と名付けておくと，各頂点には a,b,c の順に必ず3回立ち寄る
0115: !    ことになる．木を左側に眺めつつ一巡するとき，各頂点の領域 a を通過するときにだけ
0116: !    何らかの処理をする手順のことを，「行きがけ順(前順)」，b を「通りがけ順(中順)」，
0117: !    c を帰りがけ順(後順) という．こららの手順は次に示すように本質的に再帰である.
0118: !
0119: !      a) 行きがけ順(前順：pre-order traversal)
0120: !           (1)頂点データの表示
0121: !           (2)左の部分木を訪問する再帰呼出し
0122: !           (3)右の部分木を訪問する再帰呼出し
0123: !           表示データ順：50,35,25,40,36,42,60,61
0124: !
0125: !      b) 通りがけ順(中順：in-order traversal)
0126: !           (1)左の部分木を訪問する再帰呼出し
0127: !           (2)頂点データの表示
0128: !           (3)右の部分木を訪問する再帰呼出し
0129: !           表示データ順：25,35,40,42,50,60,61（昇順，ascend）
0130: !
0131: !      c) 帰りがけ順(後順：post-order traversal)
0132: !           (1)左の部分木を訪問する再帰呼出し
0133: !           (2)右の部分木を訪問する再帰呼出し
0134: !           (3)頂点データの表示
0135: !           表示データ順：25,36,42,40,35,61,60,50
0136: !
0137: !
0138: !*****************************************************
0139: ! 補足）再帰(recursive)プログラムの注意点と，その非再帰化
0140: !
0141: ! ○ 再帰プログラムは簡潔であるが，次のような問題があるため，汎用ライブラリでは避けるべき
0142: !     ・再帰が深くなると，スタックオーバーフローの可能性がある
0143: !     ・関数呼び出しにはオーバーヘッドがあるため，いくらか速度が落ちる
0144: !
0145: ! ○ 一般に再帰は非再帰化できることが多い．末尾再帰は必ず非再帰化できる.
0146: !     （ただ，末尾再帰をコンパイラが自動で非再帰化してくるかどうかは別問題）
0147: !
0148: ! 再帰プログラムでは，コンパイラが勝手に裏側でスタックを作成し，push と pop 処理もしてくれる.
0149: ! しかし，非再帰化する場合には自分でスタック配列を作成し（大きさも決めるなければならない），
0150: ! push と pop処理もしなければならない.
0151: !
0152: !
```



例

```
0153:
0154: !*****************************************************
0155: ! 補足) Open-MP マルチスレッド環境下での ALLOCATE / DEALLOCATE (malloc/free) には要注意
0156: !
0157: ! ○ cygwin での multi-threading では malloc の性能が劇的に悪いまま.
0158: !    https://www.cygwin.com/ml/cygwin/2013-02/msg00308.html
0159: !    "Re: wrong performance of malloc/free under multi-threading"
0160: !    (Linux gcc では問題ない様子)
0161: !    なぜならば, 全スレッドで共通のヒープを使うことから, ロック制御をしているため.
0162: !
0163: ! ○ Intelコンパイラ では, multi-threading の malloc に問題はない.
0164: !    http://www.isus.jp/article/intelguide/3-1/
0165: !    なぜならば, スレッド毎にローカルなヒープを準備しているため.
0166: !    一方で, malloc した時と別スレッドがfreeすると, 性能が悪くなる.
0167: !    "User and Reference Guide for the IntelR Fortran Compiler 15.0"の
0168: !    "IntelR Compiler Extension Routines to OpenMP*" に以下の記述あり.
0169: !    "  The IntelR Fortran Compiler implements a group of memory allocation routines
0170: !     as an extension to the OpenMP* run-time library to enable threads to allocate
0171: !     memory from a heap local to each thread. These routines are: KMP_MALLOC(),
0172: !     KMP_CALLOC(), and KMP_REALLOC().
0173: !       The memory allocated by these routines must also be freed by the KMP_FREE() routine.
0174: !     While you can allocate memory in one thread and then free that memory in a different
0175: !     thread, this mode of operation incurs a slight performance penalty.
0176: !
0177: !
```

```
0178:
0179: MODULE BS_tree_class
0180:
0181:    IMPLICIT none
0182:    PRIVATE
0183:
0184:    TYPE, PRIVATE :: vtx                    ! 2分木の基本構成要素：頂点 (vertex, node)
0185:       INTEGER :: c                         !        ___|___
0186:       TYPE(vtx), POINTER :: L => NULL()    !       |___c___|       c: Cell data, Contents
0187:       TYPE(vtx), POINTER :: R => NULL()    !       |_L_|_R_|       L: Left
0188:    END TYPE vtx                            !         /     \        R: Right
0189:                                            !        v       v
0190:
0191:    TYPE, PUBLIC :: tree_c                  ! 抽象データとしての2分探索木
0192:       PRIVATE
0193:       TYPE(vtx), POINTER :: root => NULL() ! 木の根
0194:       INTEGER :: nvtx = 0                  ! 頂点数=データの個数（効率化のために）
0195:    END TYPE tree_c
0196:
0197:    PUBLIC :: insert
0198:    INTERFACE insert
0199:     ! MODULE PROCEDURE insert_datum_R      ! 学習用の再帰(Recursive)版
0200:       MODULE PROCEDURE insert_datum_NR     ! 非再帰(Non-Recursive)版
0201:       MODULE PROCEDURE insert_data_NR      ! 一括登録版
0202:    END INTERFACE
0203:
0204:    PUBLIC :: get_ndat
0205:    INTERFACE get_ndat
0206:        MODULE PROCEDURE get_ndat
0207:    END INTERFACE
0208:
0209:    PUBLIC :: get_dat
0210:    INTERFACE get_dat
0211:     ! MODULE PROCEDURE traversal_asend_R
0212:       MODULE PROCEDURE traversal_asend_NR
0213:    END INTERFACE
0214:
0215:    PUBLIC :: display
0216:    INTERFACE display
0217:       MODULE PROCEDURE display_R
0218:    END INTERFACE
0219:
0220:    PUBLIC :: final
0221:    INTERFACE final
0222:     ! MODULE PROCEDURE dealloc_R
0223:       MODULE PROCEDURE dealloc_NR
0224:    END INTERFACE
0225:
0226:    PUBLIC :: renumber
0227:    INTERFACE renumber
0228: !     MODULE PROCEDURE renumber_R
0229:      MODULE PROCEDURE renumber_NR
0230:    END INTERFACE
0231:
0232:    TYPE, PRIVATE :: vtx_ptr                ! Fortranでポインタ配列を表現するためのトリック
0233:       TYPE(vtx), POINTER :: ptr            !  （非再帰化時に明示的に必要となるスタックのために）
0234:    END TYPE vtx_ptr                        ! 本当はダブルポインタ（Fortran規約外）があると良い
0235:
0236: CONTAINS
0237: !
```

```
0238:
0239: !*****************************************************************************************
0240:    SUBROUTINE insert_datum_R ( this, a )  ! datum の複数形が data
0241:
0242:       ! 2分探索木に単一データ a を追記する（学習用の再帰版）
0243:
0244:       TYPE(tree_c), INTENT(inout) :: this
0245:       INTEGER,      INTENT(in   ) :: a
0246:
0247:       CALL insert_kernel( this%root )
0248:
0249:    CONTAINS
0250:
0251:       RECURSIVE SUBROUTINE insert_kernel( ptr )
0252:          TYPE(vtx), POINTER, INTENT(inout) :: ptr ! ポインタ結合状態へのINTENT指定は F2003
0253:          IF ( ASSOCIATED(ptr) ) THEN       ! 何らかのデータが登録されている場合,
0254:             IF      ( a == ptr%c ) THEN    !    a が既に登録済みならば探索完了
0255:                RETURN                      !                        （再帰を抜ける）
0256:             ELSE IF ( a <  ptr%c ) THEN    !    さもなくば，まずは左の子を探りにいく
0257:                CALL insert_kernel( ptr%L )
0258:             ELSE                           !                 つぎに右の子を探りにいく
0259:                CALL insert_kernel( ptr%R )
0260:             END IF
0261:          ELSE                              ! ここにたどり着くのは，未登録以外にない
0262:             ALLOCATE(ptr)                  ! なので登録作業
0263:             ptr%c = a
0264:             this%nvtx = this%nvtx +1
0265:             RETURN
0266:          END IF
0267:       END SUBROUTINE insert_kernel
0268:
0269:    END SUBROUTINE insert_datum_R
0270:
0271:
0272:    SUBROUTINE insert_datum_NR ( this , a )
0273:
0274:       ! 2分探索木に単一データ a を追記する（非再帰版）
0275:
0276:       TYPE(tree_c), INTENT(inout) :: this
0277:       INTEGER,      INTENT(in   ) :: a
0278:
0279:       TYPE(vtx), POINTER :: ptr
0280:
0281:       ! 下記 BS_loop 本体にたどり着く前に，例外処理が必要
0282:       IF ( .NOT. ASSOCIATED(this%root) ) THEN  ! 初回ならば根がないので，根を作る
0283:          ALLOCATE(this%root)
0284:          this%root%c = a
0285:          this%nvtx   = 1
0286:          RETURN
0287:       END IF
0288:
0289:       ptr => this%root
0290:       BS_loop: DO      ! 2分探索(Binary Search)していく，そのデータがない場合のみ追記
0291:
0292:          IF      ( a == ptr%c ) THEN      ! a が既に登録済みならば探索完了
0293:             EXIT BS_loop
0294:          ELSE IF ( a <  ptr%c ) THEN      ! さもなくば，まず左の子を探りにいく
0295:             IF ( ASSOCIATED(ptr%L) ) THEN !    子があれば，
0296:                ptr   => ptr%L
0297:             ELSE                          !    さもなくば，a は未登録
0298:                ALLOCATE(ptr%L)            !       なので登録作業
0299:                ptr   => ptr%L
0300:                ptr%c =  a
0301:                this%nvtx = this%nvtx +1
0302:                EXIT BS_loop
0303:             END IF
0304:          ELSE  ! ( a >  ptr%c )           ! ここにたどり着くのは，右の子の場合のみ
0305:             IF ( ASSOCIATED(ptr%R) ) THEN !   以下は直上の場合と同じ
0306:                ptr   => ptr%R
0307:             ELSE
0308:                ALLOCATE(ptr%R)            !   注）Fortranにはダブルポインタがないので
0309:                ptr   => ptr%R             !       L, Rを分けざるを得ない
0310:                ptr%c =  a
0311:                this%nvtx = this%nvtx +1
0312:                EXIT BS_loop
0313:             END IF
```

```
0314:          END IF
0315:
0316:        ! ! 注) 下記は問題なさそうに見えるが, 間違い
0317:        !       (これをやるにはFortran規格にないダブルポインタが必須)
0318:        ! IF      (a==ptr%c) THEN
0319:        !    EXIT BS_loop
0320:        ! ELSE IF (a< ptr%c) THEN
0321:        !    ptr = ptr%L              ! こんなダブルポインタはFortranでは無理 (記述も規格違反)
0322:        ! ELSE  ! (a> ptr%c)
0323:        !    ptr = ptr%R
0324:        ! END IF
0325:        ! IF ( .NOT. ASSOCIATED(ptr) ) THEN
0326:        !    ALLOCATE(ptr)            ! ptrが指す先に対してalloc するのでなく,
0327:        !    ptr%c =  a               ! ptr自体に対してalloc するので, 結果ptrの指す先が変わる
0328:        !    this%nvtx = this%nvtx +1
0329:        !    EXIT BS_loop
0330:        ! END IF
0331:
0332:       END DO BS_loop
0333:
0334:    END SUBROUTINE insert_datum_NR
0335:
0336:
0337:    SUBROUTINE insert_data_NR ( this , buf )
0338:
0339:       ! 2分探索木にデータ列 dat(:) を追記する (単一データの追記の繰り返し)
0340:
0341:       TYPE(tree_c), INTENT(inout) :: this
0342:       INTEGER,      INTENT(in   ) :: buf(:)  ! データ格納用のバッファ
0343:
0344:       INTEGER            :: n       ! データ数
0345:       TYPE(vtx), POINTER :: ptr
0346:       INTEGER            :: a
0347:       INTEGER            :: i, st, j, sgn
0348:
0349:       n = SIZE(buf)
0350:
0351:       ! dat(:)が整列済みデータならば ― ありがち ―, 前から1つずつに2分探索木に
0352:       ! 追記していくと線形リストになる. これは最悪パターンである.
0353:       ! そこで, それを回避するため, 下記 For_each_dat_loop では簡易シャッフルを実施
0354:       ! 簡易シャッフルの方針 : 中央からはじめて左右交互の順で
0355:       ! 例) dat(1:10) ならば, 5番目 からはじめて 6,4, 7,3, 8,2, 9,1, 10
0356:       ! 例) dat(1:11) ならば, 6番目 からはじめて 7,5, 8,4, 9,3,10,2, 11,1
0357:
0358:       ! 下記 For_each_dat_loop のための初期値設定
0359:       j   = (n+1)/2                  ! dat(j) の j は中央からはじめる
0360:       st  = 1                        ! ループカウンタ i の開始値
0361:       sgn = 1                        ! 符号入れかえのための補助変数
0362:
0363:       ! 下記 For_each_dat_loop 本体にたどり着く前に, 例外処理が必要
0364:       IF ( .NOT. ASSOCIATED(this%root) ) THEN ! 初回ならば根がないので, 根を作る
0365:          ALLOCATE(this%root)
0366:          this%root%c = buf(j)
0367:          this%nvtx   = 1
0368:          ! この例外処理にともない, For_each_dat_loop の初回もここで実施せざるを得ない
0369:          sgn = -sgn
0370:          j   = j + sgn*(st-1)        ! j=j なのだが, 分かりやすさのために…
0371:          st  = st +1                 ! 初回実施済みゆえ, ループカウンタ i の開始値を繰下
0372:       END IF
0373:
0374:       For_each_dat_loop: DO i = st, n
0375:
0376:          sgn = -sgn                  ! 逆符号にする
0377:          j   = j + sgn*(i-1)         ! この処理により j は中央からはじめて左右交互順となる
0378:          a   = buf(j)
0379:
0380:          ! 以下は insert_datum_NR と同一
0381:          ptr => this%root
0382:          BS_loop: DO
0383:             IF      ( a == ptr%c ) THEN
0384:                EXIT BS_loop
0385:             ELSE IF ( a <  ptr%c ) THEN
0386:                IF ( ASSOCIATED(ptr%L) ) THEN
0387:                   ptr   => ptr%L
0388:                ELSE
0389:                   ALLOCATE(ptr%L)
```

```
0390:                   ptr   => ptr%L
0391:                   ptr%c =  a
0392:                   this%nvtx = this%nvtx +1
0393:                   EXIT BS_loop
0394:                END IF
0395:             ELSE  ! ( a >  ptr%c )
0396:                IF ( ASSOCIATED(ptr%R) ) THEN
0397:                   ptr   => ptr%R
0398:                ELSE
0399:                   ALLOCATE(ptr%R)
0400:                   ptr   => ptr%R
0401:                   ptr%c =  a
0402:                   this%nvtx = this%nvtx +1
0403:                   EXIT BS_loop
0404:                END IF
0405:             END IF
0406:          END DO BS_loop
0407:
0408:       END DO For_each_dat_loop
0409:
0410:    END SUBROUTINE insert_data_NR
0411:
0412: !
```

```
0413:
0414: !*****************************************************************************************************
0415:    FUNCTION get_ndat ( this ) RESULT( ans )
0416:        ! 2分探索木の頂点(データ)数を返す
0417:        TYPE(tree_c), INTENT(in   ) :: this
0418:        INTEGER                     :: ans
0419:        ans = this%nvtx
0420:    END FUNCTION get_ndat
0421:
0422:
0423: !*****************************************************************************************************
0424:    SUBROUTINE traversal_asend_R ( this, buf, n_ )
0425:
0426:       ! 2分探索木中の全データを昇順(asend)で配列 buf に取出す (学習用の再帰版)
0427:
0428:       TYPE(tree_c),       INTENT(in   ) :: this
0429:       INTEGER,            INTENT(out  ) :: buf(:)  ! 全データ取出し用の配列  SIZE(buf) >= n_
0430:       INTEGER, OPTIONAL,  INTENT(out  ) :: n_      ! データ数
0431:
0432:       INTEGER :: cnt   ! 訪問済みの頂点数
0433:
0434:       IF ( SIZE(buf) < this%nvtx ) STOP "BS_tree_class: 全データ取出しのためのバッファが不足"
0435:       IF ( PRESENT(n_) ) n_ = this%nvtx
0436:
0437:       ! 通りがけ順(中順)による横断 (in-order traversal)
0438:       cnt = 0
0439:       CALL trvsl_inorder( this%root )
0440:
0441:    CONTAINS
0442:
0443:       RECURSIVE SUBROUTINE trvsl_inorder( ptr )
0444:          TYPE(vtx), POINTER, INTENT(in   ) :: ptr ! ポインタ結合状態へのINTENT指定は F2003
0445:          IF ( .NOT. ASSOCIATED(ptr) ) THEN
0446:             RETURN
0447:          ELSE
0448:             CALL trvsl_inorder( ptr%L )
0449:             cnt = cnt +1
0450:             buf(cnt) = ptr%c
0451:             CALL trvsl_inorder( ptr%R )
0452:          END IF
0453:       END SUBROUTINE trvsl_inorder
0454:
0455:    END SUBROUTINE  traversal_asend_R
0456:
0457:
0458:    SUBROUTINE traversal_asend_NR ( this, buf, n_ )
0459:
0460:       ! 2分探索木中の全データを昇順(asend)で配列 buf に取出す (非再帰版)
0461:
0462:       TYPE(tree_c),       INTENT(in   ) :: this
0463:       INTEGER,            INTENT(out  ) :: buf(:)  ! 全データ取出し用の配列  SIZE(buf) >= n_
0464:       INTEGER, OPTIONAL,  INTENT(out  ) :: n_      ! データ数
0465:
0466:       TYPE(vtx_ptr) :: stack(this%nvtx+1)          ! 自動配列 (コンパイラ標準ではスタック域に)
0467:       INTEGER       :: sp, cnt                     ! Stack Pointer & CouNTer
0468:
0469:       IF ( SIZE(buf) < this%nvtx ) STOP "BS_tree_class: 全データ取出しのためのバッファが不足"
0470:       IF ( PRESENT(n_) ) n_ = this%nvtx
0471:
0472:       IF ( .NOT. ASSOCIATED(this%root) ) RETURN
0473:       ! 通りがけ順(中順)による横断 (in-order traversal)
0474:       sp = 1
0475:       stack(sp)%ptr => this%root
0476:       DO cnt = 1, this%nvtx
0477:          DO WHILE( ASSOCIATED( stack(sp)%ptr  ) ) ! 左に行けるだけ進みつつ,
0478:             stack(sp+1)%ptr => stack(sp)%ptr%L    ! 順次 push (スタックに戻りに必要な情報を積む)
0479:             sp = sp +1
0480:          END DO
0481:          sp = sp -1                               ! 1つ pop (スタックから戻りに必要な情報を取出)
0482:          buf(cnt)       =  stack(sp)%ptr%c
0483:          stack(sp)%ptr => stack(sp)%ptr%R
0484:       END DO
0485:       ! 終了時には sp = 1
0486:
0487:    END  SUBROUTINE traversal_asend_NR
0488:
```

```
0489: !*****************************************************************************************************
0490:    SUBROUTINE display_R ( this, width_ )
0491:
0492:       ! 木を画面に簡易出力（左に90度回転した状態で）
0493:
0494:       TYPE(tree_c),       INTENT(in   ) :: this
0495:       INTEGER, OPTIONAL, INTENT(in   ) :: width_   !  深さ1つ分の印字長さ
0496:
0497:       INTEGER       :: width
0498:       CHARACTER(80) :: fmt
0499:
0500:       IF ( PRESENT( width_ ) ) THEN
0501:          width = width_
0502:       ELSE
0503:          width = 5          ! デフォルト値
0504:       END IF
0505:
0506:       WRITE(*,*) ''
0507:       WRITE(*,*) '木を左に90度回転した状態で簡易出力'
0508:       CALL trvsl_in( this%root, 0 )          ! 逆進での通りがけ順(中順)で
0509:       WRITE(*,*) ''
0510:
0511:    CONTAINS
0512:
0513:       RECURSIVE SUBROUTINE trvsl_in( ptr, depth )
0514:          ! 逆進での通りがけ順(中順)走査   (通常は L → R だが, ここでは逆の R → L )
0515:          TYPE(vtx), POINTER, INTENT(in   ) :: ptr
0516:          INTEGER,            INTENT(in   ) :: depth
0517:          IF ( .NOT. ASSOCIATED(ptr) ) THEN
0518:             RETURN
0519:          ELSE
0520:             CALL trvsl_in( ptr%R, depth+1 )   ! Left側でなく Right側を先に CALL
0521:             WRITE(fmt,"('(',I3,'x, I',I2,')')") 1+width*depth, width  ! 内部ファイル
0522:             WRITE(*,fmt) ptr%c
0523:             CALL trvsl_in( ptr%L, depth+1 )
0524:          END IF
0525:       END SUBROUTINE trvsl_in
0526:
0527:    END SUBROUTINE  display_R
0528:
0529: !
```

```
0530:
0531: !*************************************************************************************************
0532:    SUBROUTINE dealloc_R ( this )
0533:       ! メモリ解放（学習用の再帰版）
0534:       TYPE(tree_c), INTENT(inout) :: this
0535:       ! 帰りがけ順(後順)による横断 (post-order traversal)
0536:       CALL trvsl_post( this%root )
0537:       this%nvtx = 0
0538:    CONTAINS
0539:       RECURSIVE SUBROUTINE trvsl_post( ptr )
0540:          TYPE(vtx), POINTER, INTENT(inout) :: ptr
0541:          IF ( .NOT. ASSOCIATED(ptr) ) RETURN
0542:          CALL trvsl_post( ptr%L )
0543:          CALL trvsl_post( ptr%R )
0544:          ! ここにたどり着いた時には，もはや帰りがけ
0545:          ! 理想的には，メモリ解放に先立ち次のようにゴミを詰めておくべき
0546:          ! ptr%c = （ごみ）
0547:          ! ptr%L => NULL()          ! NULLIFY(ptr%L) と同じ
0548:          ! ptr%R => NULL()
0549:          DEALLOCATE( ptr )          ! ptr が指す先のメモリが解放され，指す先は null となる
0550:       END SUBROUTINE trvsl_post
0551:       ! 注）行きがけ順などでも deallocate できるが，帰りがけ順では不要であった
0552:       !     自動変数 ptrL, ptrR (=スタックに積まれる）が必要となる（下記）.
0553:       !  RECURSIVE SUBROUTINE trvsl_pre ( ptr )
0554:       !       :
0555:       !     ptrL => ptr%L
0556:       !     ptrR => prr%R
0557:       !     DEALLOCATE( ptr )
0558:       !     CALL trvsl_pre( ptrL )
0559:       !     CALL trvsl_pre( ptrR )
0560:    END SUBROUTINE  dealloc_R
0561:
0562:
0563:    SUBROUTINE dealloc_NR ( this )
0564:
0565:       ! メモリ解放（非再帰版）
0566:
0567:       TYPE(tree_c), INTENT(inout) :: this
0568:
0569:       TYPE(vtx_ptr) :: stack(this%nvtx+1)          ! 自動配列（コンパイラ標準ではスタック域に）
0570:       INTEGER       :: sp, cnt                     ! Stack Pointer & CouNTer
0571:
0572:       IF ( .NOT. ASSOCIATED(this%root) ) RETURN
0573:       ! 帰りがけ順(後順)による横断(post-order traversal)
0574:       sp = 1                                       ! 帰りがけの場合，全部 stack(:) に喰わせて
0575:       stack(sp)%ptr => this%root                   ! しまってからでないと，吐き出せない.
0576:       DO WHILE ( sp > 0 )
0577:          IF      ( ASSOCIATED( stack(sp)%ptr%L) ) THEN
0578:             stack(sp+1)%ptr => stack(sp)%ptr%L     ! push
0579:             stack(sp)%ptr%L => NULL()              ! 帰りがけを判別するために必要なフラグ立て
0580:             sp = sp +1
0581:          ELSE IF ( ASSOCIATED( stack(sp)%ptr%R) ) THEN
0582:             stack(sp+1)%ptr => stack(sp)%ptr%R     ! push
0583:             stack(sp)%ptr%R => NULL()              ! 帰りがけを判別するために必要なフラグ立て
0584:             sp = sp +1
0585:          ELSE                                      ! 帰りがけ
0586:             DEALLOCATE( stack(sp)%ptr )
0587:             sp = sp -1                             ! pop
0588:          END IF
0589:       END DO
0590:
0591:       this%nvtx = 0
0592:
0593:    END SUBROUTINE  dealloc_NR
0594:
0595: !
```

```
0596:
0597: !*************************************************************************************************
0598:    SUBROUTINE renumber_R ( this, o2n )
0599:       ! データの番号付け替え（学習用の再帰版）
0600:       ! ～ 現データorg番号から新データnew番号への変換テーブル o2n(:) に基づく ～
0601:       TYPE(tree_c), INTENT(inout) :: this
0602:       INTEGER,      INTENT(in   ) :: o2n(:)
0603:       TYPE(vtx), POINTER :: org_root
0604:       IF ( .NOT. ASSOCIATED(this%root) ) RETURN
0605:       org_root  => this%root     ! 根を org_root に繋ぎかえたうえで
0606:       this%root => NULL()        ! this%root に新しく木を茂らせるために.
0607:       ! 帰りがけ順(後順)による横断 (post-order traversal)
0608:       CALL trvsl_post( org_root )
0609:    CONTAINS
0610:       RECURSIVE SUBROUTINE trvsl_post( ptr )
0611:          TYPE(vtx), POINTER, INTENT(inout) :: ptr ! ポインタ結合状態へのINTENT指定は F2003
0612:          IF ( .NOT. ASSOCIATED(ptr) ) RETURN
0613:          CALL trvsl_post( ptr%L )
0614:          CALL trvsl_post( ptr%R )
0615:          ! ようやくここまで到達した時には，もはや帰り道
0616:          ptr%c = o2n(ptr%c)      ! 番号を付け替えるとともに，this%rootに追記するため，
0617:          ptr%L => NULL()         ! 左右の子への両枝を切り落として一度葉に戻しておく
0618:          ptr%R => NULL()         ! 以上，ptrの身辺整理を終え，this%root以下への繋ぎかえ準備完了
0619:          ! 付け替えによる追記（再帰の中に再帰が入る）
0620:          CALL insert_kernel( this%root, ptr )
0621:       END SUBROUTINE trvsl_post
0622:       RECURSIVE SUBROUTINE insert_kernel( ptr, p_vtx )
0623:          TYPE(vtx), POINTER, INTENT(inout) :: ptr
0624:          TYPE(vtx), POINTER, INTENT(inout) :: p_vtx
0625:          IF ( ASSOCIATED(ptr) ) THEN
0626:             IF      ( p_vtx%c == ptr%c ) THEN     ! 普通は有り得ないが，
0627:                this%nvtx = this%nvtx -1           ! o2n(:) が 1対1 の対応でない場合の処理
0628:                DEALLOCATE( p_vtx )
0629:                RETURN
0630:             ELSE IF ( p_vtx%c <  ptr%c ) THEN
0631:                CALL insert_kernel( ptr%L, p_vtx )
0632:             ELSE
0633:                CALL insert_kernel( ptr%R, p_vtx )
0634:             END IF
0635:          ELSE
0636:             ptr => p_vtx
0637:             RETURN
0638:          END IF
0639:       END SUBROUTINE insert_kernel
0640:    END SUBROUTINE renumber_R
0641:
0642:
0643:    SUBROUTINE renumber_NR ( this, o2n )
0644:
0645:       ! データの番号付け替え
0646:       ! ～ 現データold番号から新データnew番号への変換テーブル o2n(:) に基づく ～
0647:
0648:       TYPE(tree_c), INTENT(inout) :: this
0649:       INTEGER,      INTENT(in   ) :: o2n(:)
0650:
0651:       TYPE(tree_c), POINTER :: org_root
0652:
0653:       TYPE(vtx_ptr)         :: stack(this%nvtx+1) ! 自動配列（標準ではスタック領域に積まれる）
0654:       INTEGER               :: sp, cnt            ! Stack Pointer & CouNTer
0655:
0656:       IF ( .NOT. ASSOCIATED(this%root) ) RETURN
0657:
0658:       ! 帰りがけ順(後順)による横断(post-order traversal)
0659:       sp = 1
0660:       stack(sp)%ptr => this%root
0661:       this%root     => NULL()                    ! this%root に新しく木を茂らせるために
0662:       DO WHILE ( sp > 0 )
0663:          IF      ( ASSOCIATED( stack(sp)%ptr%L) ) THEN
0664:             stack(sp+1)%ptr => stack(sp)%ptr%L    ! push
0665:             stack(sp)%ptr%L => NULL()             ! 帰りがけを判別するために必要なフラグ立て
0666:             sp = sp +1
0667:          ELSE IF ( ASSOCIATED( stack(sp)%ptr%R) ) THEN
0668:             stack(sp+1)%ptr => stack(sp)%ptr%R    ! push
0669:             stack(sp)%ptr%R => NULL()             ! 帰りがけを判別するために必要なフラグ立て
0670:             sp = sp +1
0671:          ELSE                                     ! 帰りがけ
```

```
0672:             ! 当該データstack(sp)%ptr%c の番号を付け替えて, this%root 以下に追記
0673:             stack(sp)%ptr%c = o2n( stack(sp)%ptr%c )
0674:             CALL insert_vtx_NR( this%root, stack(sp)%ptr )
0675:             sp = sp -1                                  ! pop
0676:          END IF
0677:       END DO
0678:
0679:    CONTAINS
0680:
0681:       SUBROUTINE insert_vtx_NR ( root, p_vtx )
0682:
0683:          TYPE(vtx), POINTER, INTENT(inout) :: root  ! 新しい木の根
0684:          TYPE(vtx), POINTER, INTENT(inout) :: p_vtx ! 追記したい頂点
0685:
0686:          TYPE(vtx), POINTER :: ptr
0687:
0688:          IF ( .NOT. ASSOCIATED( root ) ) THEN
0689:             root => p_vtx
0690:             RETURN
0691:          END IF
0692:
0693:          ptr => root
0694:          BS_loop: DO
0695:             IF      ( p_vtx%c == ptr%c ) THEN      ! 普通は有り得ないが,
0696:                this%nvtx = this%nvtx -1            ! o2n(:) が 1対1 の対応でない場合の処理
0697:                DEALLOCATE( p_vtx )
0698:                EXIT BS_loop
0699:             ELSE IF ( p_vtx%c <  ptr%c ) THEN
0700:                IF ( ASSOCIATED(ptr%L) ) THEN
0701:                   ptr   => ptr%L
0702:                ELSE
0703:                   ptr%L => p_vtx
0704:                   EXIT BS_loop
0705:                END IF
0706:             ELSE  ! ( p_vtx%c >  ptr%c )
0707:                IF ( ASSOCIATED(ptr%R) ) THEN
0708:                   ptr   => ptr%R
0709:                ELSE
0710:                   ptr%R => p_vtx
0711:                   EXIT BS_loop
0712:                END IF
0713:             END IF
0714:          END DO BS_loop
0715:
0716:       END SUBROUTINE insert_vtx_NR
0717:
0718:    END SUBROUTINE renumber_NR
0719:
0720: END MODULE BS_tree_class
```