## dynarray.h (Page 1 of 2)

```
/*--------------------------------------------------------------------*/
/* dynarray.h                                                         */
/* Author: Bob Dondero                                                */
/*--------------------------------------------------------------------*/

#ifndef DYNARRAY_INCLUDED
#define DYNARRAY_INCLUDED

#include <stddef.h>

/* A DynArray_T object is an array whose length can expand
   dynamically. */

typedef struct DynArray *DynArray_T;

/*--------------------------------------------------------------------*/

/* Return a new DynArray_T object whose length is uLength, or
   NULL if insufficient memory is available. */

DynArray_T DynArray_new(size_t uLength);

/*--------------------------------------------------------------------*/

/* Free oDynArray. */

void DynArray_free(DynArray_T oDynArray);

/*--------------------------------------------------------------------*/

/* Return the length of oDynArray. */

size_t DynArray_getLength(DynArray_T oDynArray);

/*--------------------------------------------------------------------*/

/* Return the uIndex'th element of oDynArray. */

void *DynArray_get(DynArray_T oDynArray, size_t uIndex);

/*--------------------------------------------------------------------*/

/* Assign pvElement to the uIndex'th element of oDynArray.  Return the
   old element. */

void *DynArray_set(DynArray_T oDynArray, size_t uIndex,
                   const void *pvElement);

/*--------------------------------------------------------------------*/

/* Add pvElement to the end of oDynArray, thus incrementing its length.
   Return 1 (TRUE) if successful, or 0 (FALSE) if insufficient memory
   is available. */

int DynArray_add(DynArray_T oDynArray, const void *pvElement);

/*--------------------------------------------------------------------*/

/* Add pvElement to oDynArray such that it is the uIndex'th element.
   Return 1 (TRUE) if successful, or 0 (FALSE) if insufficient memory
   is available. */

int DynArray_addAt(DynArray_T oDynArray, size_t uIndex,
                   const void *pvElement);

/*--------------------------------------------------------------------*/
```

## dynarray.h (Page 2 of 2)

```

/* Remove and return the uIndex'th element of oDynArray. */

void *DynArray_removeAt(DynArray_T oDynArray, size_t uIndex);

/*--------------------------------------------------------------------*/

/* Fill ppvArray with the elements of oDynArray.  ppvArray must point
   to an area of memory that is large enough to hold all elements of
   oDynArray. */

void DynArray_toArray(DynArray_T oDynArray, void **ppvArray);

/*--------------------------------------------------------------------*/

/* Apply function *pfApply to each element of oDynArray, passing
   pvExtra as an extra argument.  That is, for each element pvElement of
   oDynArray, call (*pfApply)(pvElement, pvExtra). */

void DynArray_map(DynArray_T oDynArray,
                  void (*pfApply)(void *pvElement, void *pvExtra),
                  const void *pvExtra);

/*--------------------------------------------------------------------*/

/* Sort oDynArray in the order determined by *pfCompare.
   *pfCompare must return <0, 0, or >0 depending upon whether
   *pvElement1 is less than, equal to, or greater than *pvElement2,
   respectively. */

void DynArray_sort(DynArray_T oDynArray,
                   int (*pfCompare)(const void *pvElement1,
                                    const void *pvElement2));

/*--------------------------------------------------------------------*/

/* Linear search oDynArray for *pvSoughtElement using *pfCompare to
   determine equality.  If the element is found, then assign its
   index to *puIndex and return 1.  If the element is not found, then
   assign nothing to *puIndex and return 0.
   *pfCompare must return 0 if *pvElement1 is equal to pvElement2,
   and non-0 otherwise. */

int DynArray_search(DynArray_T oDynArray,
                    void *pvSoughtElement,
                    size_t *puIndex,
                    int (*pfCompare)(const void *pvElement1,
                                     const void *pvElement2));

/*--------------------------------------------------------------------*/

/* Binary search oDynArray for *pvSoughtElement using *pfCompare to
   determine equality.  If the element is found, then assign its
   index to *puIndex and return 1.  If the element is not found, then
   assign the index where it would belong to *puIndex and return 0.
   *pfCompare must return <0, 0, or >0 if *pvElement1 is less than,
   equal to, or greater than *pvElement2.
   oDynArray must be sorted as determined by *pfCompare. */

int DynArray_bsearch(DynArray_T oDynArray,
                     void *pvSoughtElement,
                     size_t *puIndex,
                     int (*pfCompare)(const void *pvElement1,
                                      const void *pvElement2));

#endif
```

Precept 16
Week 9, Wed/Thu

## dynarray.c (Page 1 of 7)

```
/*--------------------------------------------------------------------*/
/* dynarray.c                                                         */
/* Author: Bob Dondero                                                */
/*--------------------------------------------------------------------*/

#include "dynarray.h"
#include <assert.h>
#include <stdlib.h>

/*--------------------------------------------------------------------*/

/* The minimum physical length of a DynArray object. */

static const size_t MIN_PHYS_LENGTH = 2;

/*--------------------------------------------------------------------*/

/* A DynArray consists of an array, along with its logical and
   physical lengths. */

struct DynArray
{
   /* The number of elements in the DynArray from the client's
      point of view. */
   size_t uLength;

   /* The number of elements in the array that underlies the
      DynArray. */
   size_t uPhysLength;

   /* The array that underlies the DynArray. */
   const void **ppvArray;
};

/*--------------------------------------------------------------------*/

#ifndef NDEBUG

/* Check the invariants of oDynArray.  Return 1 (TRUE) iff oDynArray
   is in a valid state. */

static int DynArray_isValid(DynArray_T oDynArray)
{
   if (oDynArray->uPhysLength < MIN_PHYS_LENGTH) return 0;
   if (oDynArray->uLength > oDynArray->uPhysLength) return 0;
   if (oDynArray->ppvArray == NULL) return 0;
   return 1;
}

#endif

/*--------------------------------------------------------------------*/

/* Increase the physical length of oDynArray.  Return 1 (TRUE) if
   successful and 0 (FALSE) if insufficient memory is available. */

static int DynArray_grow(DynArray_T oDynArray)
{
   const size_t GROWTH_FACTOR = 2;

   size_t uNewLength;
   const void **ppvNewArray;

```

## dynarray.c (Page 2 of 7)

```
   assert(oDynArray != NULL);

   uNewLength = GROWTH_FACTOR * oDynArray->uPhysLength;

   ppvNewArray = (const void**)
      realloc(oDynArray->ppvArray, sizeof(void*) * uNewLength);
   if (ppvNewArray == NULL)
      return 0;

   oDynArray->uPhysLength = uNewLength;
   oDynArray->ppvArray = ppvNewArray;
   return 1;
}

/*--------------------------------------------------------------------*/

DynArray_T DynArray_new(size_t uLength)
{
   DynArray_T oDynArray;

   oDynArray = (struct DynArray*)malloc(sizeof(struct DynArray));
   if (oDynArray == NULL)
      return NULL;

   oDynArray->uLength = uLength;
   if (uLength > MIN_PHYS_LENGTH)
      oDynArray->uPhysLength = uLength;
   else
      oDynArray->uPhysLength = MIN_PHYS_LENGTH;

   oDynArray->ppvArray =
      (const void**)calloc(oDynArray->uPhysLength, sizeof(void*));
   if (oDynArray->ppvArray == NULL)
   {
      free(oDynArray);
      return NULL;
   }

   return oDynArray;
}

/*--------------------------------------------------------------------*/

void DynArray_free(DynArray_T oDynArray)
{
   assert(oDynArray != NULL);
   assert(DynArray_isValid(oDynArray));

   free(oDynArray->ppvArray);
   free(oDynArray);
}

/*--------------------------------------------------------------------*/

size_t DynArray_getLength(DynArray_T oDynArray)
{
   assert(oDynArray != NULL);
   assert(DynArray_isValid(oDynArray));

   return oDynArray->uLength;
}

/*--------------------------------------------------------------------*/
```

## dynarray.c (Page 3 of 7)

```
127:
128: void *DynArray_get(DynArray_T oDynArray, size_t uIndex)
129: {
130:    assert(oDynArray != NULL);
131:    assert(uIndex < oDynArray->uLength);
132:    assert(DynArray_isValid(oDynArray));
133:
134:    return (void*)(oDynArray->ppvArray)[uIndex];
135: }
136:
137: /*--------------------------------------------------------------------*/
138:
139: void *DynArray_set(DynArray_T oDynArray, size_t uIndex,
140:                    const void *pvElement)
141: {
142:    const void *pvOldElement;
143:
144:    assert(oDynArray != NULL);
145:    assert(uIndex < oDynArray->uLength);
146:    assert(DynArray_isValid(oDynArray));
147:
148:    pvOldElement = oDynArray->ppvArray[uIndex];
149:    oDynArray->ppvArray[uIndex] = pvElement;
150:
151:    assert(DynArray_isValid(oDynArray));
152:
153:    return (void*)pvOldElement;
154: }
155:
156: /*--------------------------------------------------------------------*/
157:
158: int DynArray_add(DynArray_T oDynArray, const void *pvElement)
159: {
160:    assert(oDynArray != NULL);
161:    assert(DynArray_isValid(oDynArray));
162:
163:    if (oDynArray->uLength == oDynArray->uPhysLength)
164:       if (! DynArray_grow(oDynArray))
165:          return 0;
166:
167:    oDynArray->ppvArray[oDynArray->uLength] = pvElement;
168:    oDynArray->uLength++;
169:
170:    assert(DynArray_isValid(oDynArray));
171:
172:    return 1;
173: }
174:
175: /*--------------------------------------------------------------------*/
176:
177: int DynArray_addAt(DynArray_T oDynArray, size_t uIndex,
178:                    const void *pvElement)
179: {
180:    size_t u;
181:
182:    assert(oDynArray != NULL);
183:    assert(uIndex <= oDynArray->uLength);
184:    assert(DynArray_isValid(oDynArray));
185:
186:    if (oDynArray->uLength == oDynArray->uPhysLength)
187:       if (! DynArray_grow(oDynArray))
188:          return 0;
189:
```

## dynarray.c (Page 4 of 7)

```
190:    for (u = oDynArray->uLength; u > uIndex; u--)
191:       oDynArray->ppvArray[u] = oDynArray->ppvArray[u-1];
192:
193:    oDynArray->ppvArray[uIndex] = pvElement;
194:    oDynArray->uLength++;
195:
196:    assert(DynArray_isValid(oDynArray));
197:
198:    return 1;
199: }
200:
201: /*--------------------------------------------------------------------*/
202:
203: void *DynArray_removeAt(DynArray_T oDynArray, size_t uIndex)
204: {
205:    const void *pvOldElement;
206:    size_t u;
207:
208:    assert(oDynArray != NULL);
209:    assert(uIndex < oDynArray->uLength);
210:    assert(DynArray_isValid(oDynArray));
211:
212:    pvOldElement = oDynArray->ppvArray[uIndex];
213:
214:    oDynArray->uLength--;
215:
216:    for (u = uIndex; u < oDynArray->uLength; u++)
217:       oDynArray->ppvArray[u] = oDynArray->ppvArray[u+1];
218:
219:    assert(DynArray_isValid(oDynArray));
220:
221:    return (void*)pvOldElement;
222: }
223:
224: /*--------------------------------------------------------------------*/
225:
226: void DynArray_toArray(DynArray_T oDynArray, void **ppvArray)
227: {
228:    size_t u;
229:
230:    assert(oDynArray != NULL);
231:    assert(ppvArray != NULL);
232:    assert(DynArray_isValid(oDynArray));
233:
234:    for (u = 0; u < oDynArray->uLength; u++)
235:       ppvArray[u] = (void*)oDynArray->ppvArray[u];
236: }
237:
238: /*--------------------------------------------------------------------*/
239:
240: void DynArray_map(DynArray_T oDynArray,
241:                   void (*pfApply)(void *pvElement, void *pvExtra),
242:                   const void *pvExtra)
243: {
244:    size_t u;
245:
246:    assert(oDynArray != NULL);
247:    assert(pfApply != NULL);
248:    assert(DynArray_isValid(oDynArray));
249:
250:    for (u = 0; u < oDynArray->uLength; u++)
251:       (*pfApply)((void*)oDynArray->ppvArray[u], (void*)pvExtra);
252: }
```

```
253:
254: /*--------------------------------------------------------------------*/
255:
256: /* Sort the array of elements that resides in memory at
257:    addresses ppvLo...ppvHi in ascending order, as determined
258:    by *pfCompare.
259:    *pfCompare must return <0, 0, or >0 depending upon whether
260:    *pvElement1 is less than, equal to, or greater than *pvElement2,
261:    respectively. */
262:
263: static void DynArray_qsort(
264:    const void **ppvLo,
265:    const void **ppvHi,
266:    int (*pfCompare) (const void *pvElement1, const void *pvElement2))
267: {
268:    /* This function implements a variation of the quicksort algorithm
269:       shown in the book "Algorithms + Data Structures = Programs" by
270:       Niklaus Wirth. */
271:
272:    /* This function uses pointers instead of indices to avoid
273:       complications with using unsigned integers as array indices. */
274:
275:    const void **ppvRight;
276:    const void **ppvLeft;
277:    const void *pvPivot;
278:    const void *pvTemp;
279:
280:    assert(ppvLo != NULL);
281:    assert(ppvHi != NULL);
282:    assert(pfCompare != NULL);
283:
284:    ppvRight = ppvLo;
285:    ppvLeft = ppvHi;
286:    pvPivot = *(ppvLo + ((ppvHi - ppvLo) / 2));
287:
288:    while (ppvRight <= ppvLeft)
289:    {
290:       while ((*pfCompare)(*ppvRight, pvPivot) < 0)
291:          ppvRight++;
292:       while ((*pfCompare)(pvPivot, *ppvLeft) < 0)
293:          ppvLeft--;
294:       if (ppvRight <= ppvLeft)
295:       {
296:          /* Swap *ppvRight and *ppvLeft. */
297:          pvTemp = *ppvRight;
298:          *ppvRight = *ppvLeft;
299:          *ppvLeft = pvTemp;
300:
301:          ppvRight++;
302:          ppvLeft--;
303:       }
304:    }
305:
306:    if (ppvLo < ppvLeft)
307:       DynArray_qsort(ppvLo, ppvLeft, pfCompare);
308:    if (ppvRight < ppvHi)
309:       DynArray_qsort(ppvRight, ppvHi, pfCompare);
310: }
311:
312: /*--------------------------------------------------------------------*/
313:
314: void DynArray_sort(DynArray_T oDynArray,
315:                    int (*pfCompare)(const void *pvElement1,
```



```
316:                                     const void *pvElement2))
317: {
318:    assert(oDynArray != NULL);
319:    assert(pfCompare != NULL);
320:    assert(DynArray_isValid(oDynArray));
321:
322:    if (oDynArray->uLength < 2)
323:       return;
324:
325:    DynArray_qsort(
326:       &oDynArray->ppvArray[0],
327:       &oDynArray->ppvArray[oDynArray->uLength-1],
328:       pfCompare);
329:
330:    assert(DynArray_isValid(oDynArray));
331: }
332:
333: /*--------------------------------------------------------------------*/
334:
335: int DynArray_search(DynArray_T oDynArray,
336:                     void *pvSoughtElement,
337:                     size_t *puIndex,
338:                     int (*pfCompare)(const void *pvElement1,
339:                                      const void *pvElement2))
340: {
341:    size_t u;
342:
343:    assert(oDynArray != NULL);
344:    assert(puIndex != NULL);
345:    assert(pfCompare != NULL);
346:    assert(DynArray_isValid(oDynArray));
347:
348:    for (u = 0; u < oDynArray->uLength; u++)
349:       if ((*pfCompare)(oDynArray->ppvArray[u], pvSoughtElement) == 0)
350:       {
351:          *puIndex = u;
352:          return 1;
353:       }
354:    return 0;
355: }
356:
357:
358:
359: /*--------------------------------------------------------------------*/
360:
361: /* Binary search the array of elements that resides in memory at
362:    addresses ppvLo...ppvHi for pvSoughtElement.
363:    *pfCompare must return <0, 0, or >0 depending upon whether
364:    *pvElement1 is less than, equal to, or greater than *pvElement2,
365:    respectively. */
366:
367: static const void **DynArray_bsearchHelp(
368:    void *pvSoughtElement,
369:    const void **ppvLo,
370:    const void **ppvHi,
371:    int (*pfCompare)(const void *pvElement1, const void *pvElement2),
372:    const void*** pppvInsert)
373: {
374:    /* This function uses pointers instead of indices to avoid
375:       complications with using unsigned integers as array indices. */
376:
377:    const void **ppvMid;
378:    int iCompare;
```

```
379:
380:    assert(ppvLo != NULL);
381:    assert(ppvHi != NULL);
382:    assert(pfCompare != NULL);
383:
384:    while (ppvLo <= ppvHi)
385:    {
386:       ppvMid = ppvLo + ((ppvHi - ppvLo) / 2);
387:       iCompare = (*pfCompare)(pvSoughtElement, *ppvMid);
388:       if (iCompare < 0)
389:          ppvHi = ppvMid - 1;
390:       else if (iCompare > 0)
391:          ppvLo = ppvMid + 1;
392:       else
393:          return ppvMid;
394:    }
395:    *pppvInsert = ppvLo;
396:    return NULL;
397: }
398:
399: /*--------------------------------------------------------------------*/
400:
401: int DynArray_bsearch(DynArray_T oDynArray,
402:                      void *pvSoughtElement,
403:                      size_t *puIndex,
404:                      int (*pfCompare)(const void *pvElement1,
405:                                       const void *pvElement2))
406: {
407:    const void **ppvElement;
408:    const void **ppvInsert;
409:
410:    assert(oDynArray != NULL);
411:    assert(puIndex != NULL);
412:    assert(pfCompare != NULL);
413:    assert(DynArray_isValid(oDynArray));
414:
415:    if (oDynArray->uLength == 0) {
416:       *puIndex = 0;
417:       return 0;
418:    }
419:
420:    ppvElement = DynArray_bsearchHelp(
421:       pvSoughtElement,
422:       &oDynArray->ppvArray[0],
423:       &oDynArray->ppvArray[oDynArray->uLength-1],
424:       pfCompare,
425:       &ppvInsert);
426:
427:    if (ppvElement == NULL) {
428:       *puIndex = (size_t)(ppvInsert - &oDynArray->ppvArray[0]);
429:       return 0;
430:    }
431:
432:    *puIndex = (size_t)(ppvElement - &oDynArray->ppvArray[0]);
433:    return 1;
434: }
```

## testdynarray.c (Page 1 of 3)

```
/*--------------------------------------------------------------------*/
/* testdynarray.c                                                     */
/* Author: Bob Dondero                                                */
/*--------------------------------------------------------------------*/

#include "dynarray.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

/*--------------------------------------------------------------------*/

/* Print string *pvItem using format pvFormat. */

static void printString(void *pvItem, void *pvFormat)
{
   assert(pvItem != NULL);
   assert(pvFormat != NULL);
   printf((char*)pvFormat, (char*)pvItem);
}

/*--------------------------------------------------------------------*/

/* Return -1, 0, or 1 depending upon whether *pvOne is
   less than, equal to, or greater than *pvTwo, respectively. */

static int compareString(const void *pvOne, const void *pvTwo)
{
   assert(pvOne != NULL);
   assert(pvTwo != NULL);
   return strcmp((char*)pvOne, (char*)pvTwo);
}

/*--------------------------------------------------------------------*/

/* Demonstrate the DynArray ADT. Return 0. */

int main(void)
{
   DynArray_T oDynArray;
   size_t uLength;
   char **ppcArray;
   size_t u;
   char *pcElement;
   size_t uIndex = 0;
   int iFound;

   /* Demonstrate DynArray_new. */

   oDynArray = DynArray_new(0);
   if (oDynArray == NULL) exit(EXIT_FAILURE);

   /* Demonstrate DynArray_add. */

   if (! DynArray_add(oDynArray, "Ruth")) exit(EXIT_FAILURE);
   if (! DynArray_add(oDynArray, "Gehrig")) exit(EXIT_FAILURE);
   if (! DynArray_add(oDynArray, "Mantle")) exit(EXIT_FAILURE);
   if (! DynArray_add(oDynArray, "Jeter")) exit(EXIT_FAILURE);

   /* Demonstrate DynArray_getLength. */

   printf("------------------------------------------------------\n");
```

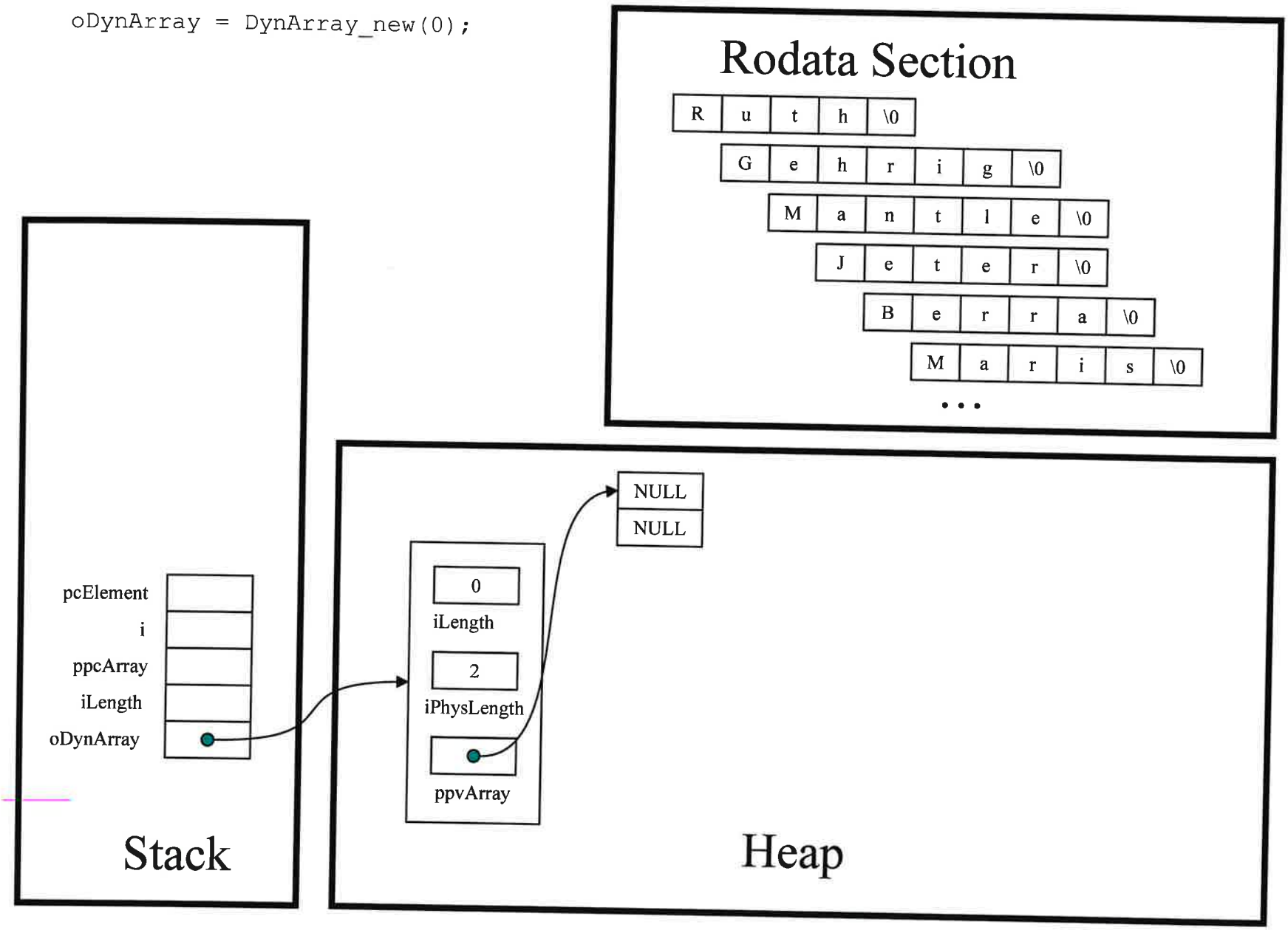
## testdynarray.c (Page 2 of 3)

```
   uLength = DynArray_getLength(oDynArray);
   printf("DynArray length:  %lu\n", uLength);

   /* Demonstrate DynArray_get. */

   printf("------------------------------------------------------\n");
   printf("This output should list 4 elements\n");
   printf("------------------------------------------------------\n");
   uLength = DynArray_getLength(oDynArray);
   for (u = 0; u < uLength; u++)
      printf("%s\n", (char*)DynArray_get(oDynArray, u));

   /* Demonstrate DynArray_set. */

   printf("------------------------------------------------------\n");
   printf("This output should list 4 elements\n");
   printf("------------------------------------------------------\n");
   (void)DynArray_set(oDynArray, 2, "Berra");
   uLength = DynArray_getLength(oDynArray);
   for (u = 0; u < uLength; u++)
      printf("%s\n", (char*)DynArray_get(oDynArray, u));

   /* Demonstrate DynArray_addAt. */

   printf("------------------------------------------------------\n");
   printf("This output should list 5 elements\n");
   printf("------------------------------------------------------\n");
   if (! DynArray_addAt(oDynArray, 1, "Maris")) exit(EXIT_FAILURE);
   uLength = DynArray_getLength(oDynArray);
   for (u = 0; u < uLength; u++)
      printf("%s\n", (char*)DynArray_get(oDynArray, u));

   /* Demonstrate DynArray_removeAt. */

   printf("------------------------------------------------------\n");
   printf("This output should list 4 elements\n");
   printf("------------------------------------------------------\n");
   pcElement = (char*)DynArray_removeAt(oDynArray, 1);
   uLength = DynArray_getLength(oDynArray);
   for (u = 0; u < uLength; u++)
      printf("%s\n", (char*)DynArray_get(oDynArray, u));
   printf("Removed element: %s\n", pcElement);

   /* Demonstrate DynArray_toArray. */

   printf("------------------------------------------------------\n");
   printf("This output should list 4 elements\n");
   printf("------------------------------------------------------\n");
   uLength = DynArray_getLength(oDynArray);
   ppcArray = (char**)calloc((size_t)uLength, sizeof(char*));
   if (ppcArray == NULL) exit(EXIT_FAILURE);
   DynArray_toArray(oDynArray, (void**)ppcArray);
   for (u = 0; u < uLength; u++)
      printf("%s\n", ppcArray[u]);
   free(ppcArray);

   /* Demonstrate DynArray_map. */

   printf("------------------------------------------------------\n");
   printf("This output should list 4 elements\n");
   printf("------------------------------------------------------\n");
   DynArray_map(oDynArray, printString, "%s\n");
```

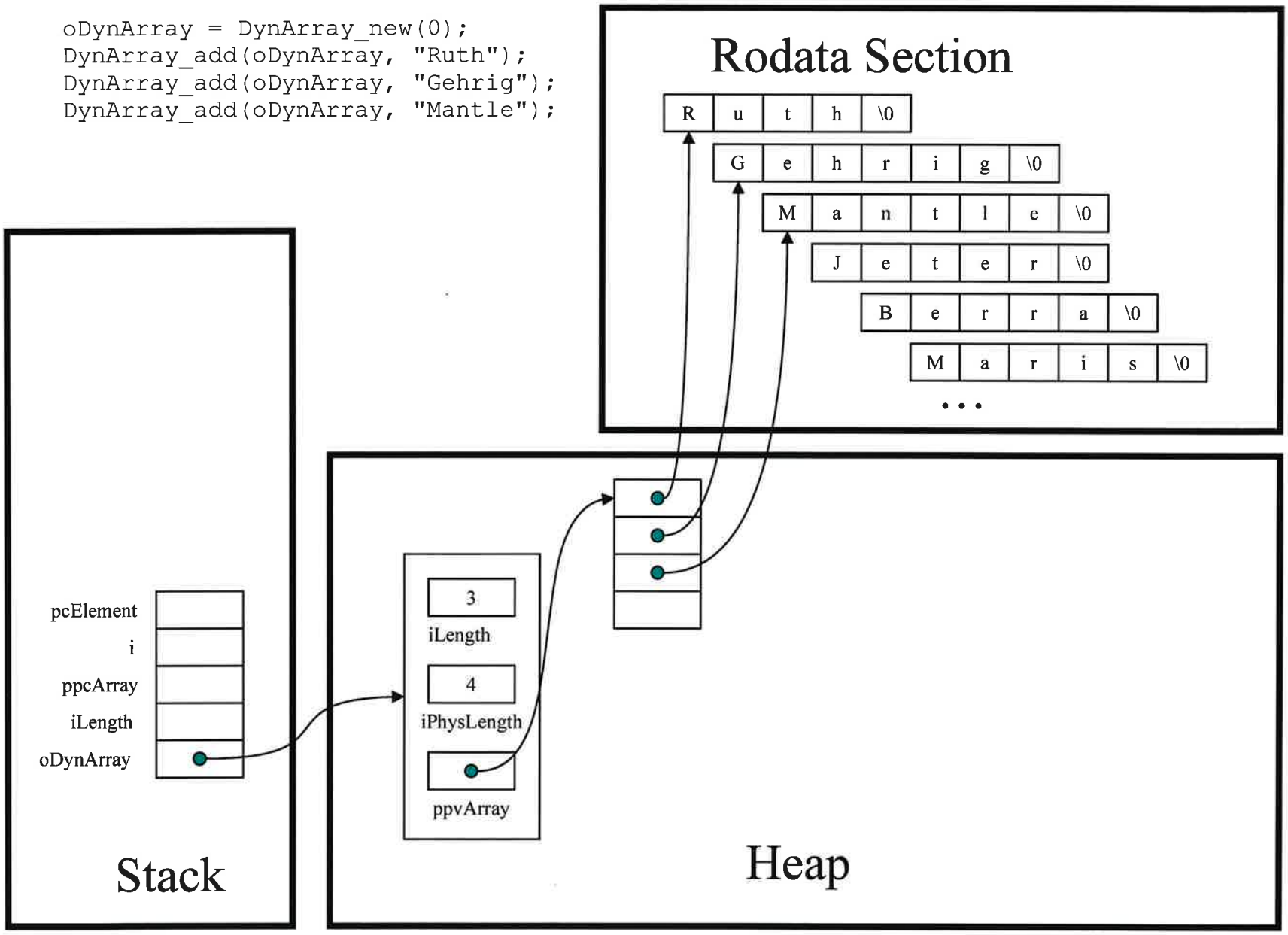
**testdynarray.c (Page 3 of 3)**

```
127:    /* Demonstrate DynArray_sort. */
128:
129:    printf("------------------------------------------------------------\n");
130:    printf("This output should list 4 elements in ascending order\n");
131:    printf("------------------------------------------------------------\n");
132:    DynArray_sort(oDynArray, compareString);
133:    DynArray_map(oDynArray, printString, "%s\n");
134:
135:    /* Demonstrate DynArray_search. */
136:
137:    printf("------------------------------------------------------------\n");
138:    printf("This output should list 1 element\n");
139:    printf("------------------------------------------------------------\n");
140:    iFound = DynArray_search(oDynArray, "Ruth", &uIndex, compareString);
141:    if (iFound)
142:       printf("%s\n", (char*)DynArray_get(oDynArray, uIndex));
143:
144:    /* Demonstrate DynArray_bsearch. */
145:
146:    printf("------------------------------------------------------------\n");
147:    printf("This output should list 1 element\n");
148:    printf("------------------------------------------------------------\n");
149:    iFound = DynArray_bsearch(oDynArray, "Ruth", &uIndex, compareString);
150:    if (iFound)
151:       printf("%s\n", (char*)DynArray_get(oDynArray, uIndex));
152:
153:    /* Demonstrate DynArray_free. */
154:
155:    DynArray_free(oDynArray);
156:
157:    return 0;
158: }
```

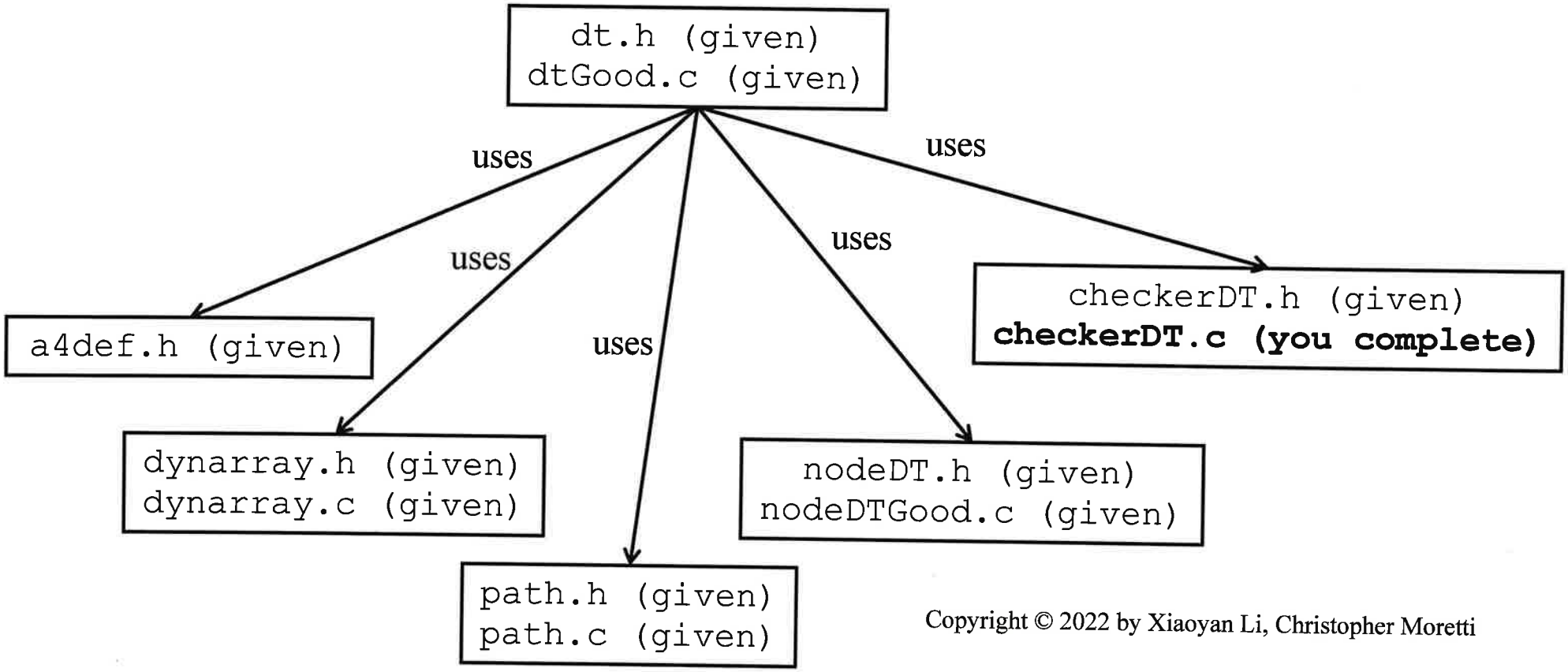
```
oDynArray = DynArray_new(0);
```



```
oDynArray = DynArray_new(0);
DynArray_add(oDynArray, "Ruth");
DynArray_add(oDynArray, "Gehrig");
DynArray_add(oDynArray, "Mantle");
```



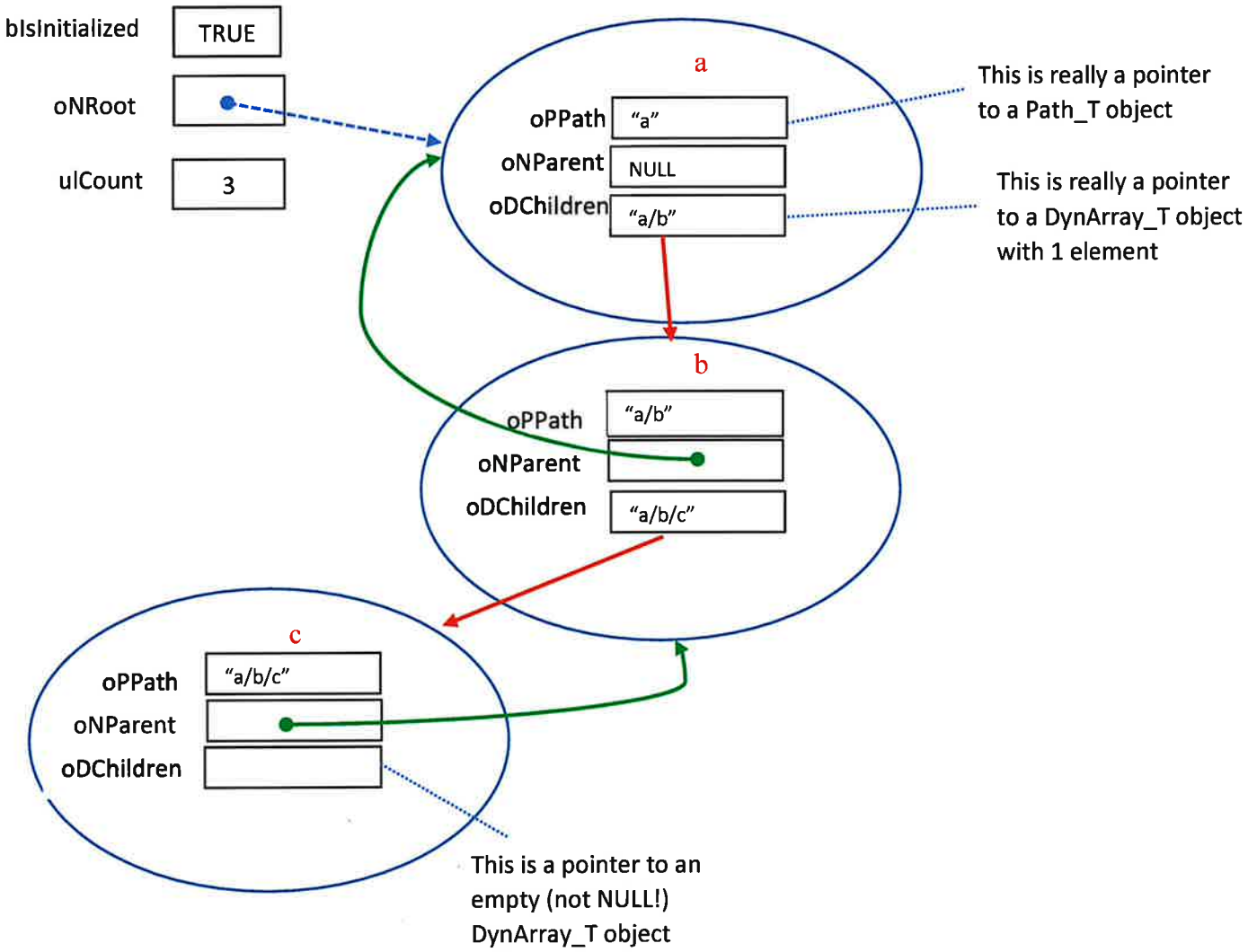
# Princeton University
# COS 217: Introduction to Programming Systems
# DT Code



**Princeton University**

**COS 217: Introduction to Programming Systems**

**DT Data Structures**



**a4def.h (Page 1 of 1)**

```
 1: /*-------------------------------------------------------------------------*/
 2: /* a4def.h                                                                 */
 3: /* Author: Christopher Moretti                                             */
 4: /*-------------------------------------------------------------------------*/
 5:
 6: #ifndef A4DEF_INCLUDED
 7: #define A4DEF_INCLUDED
 8:
 9: /* Return statuses */
10: enum { SUCCESS,
11:        INITIALIZATION_ERROR,
12:        ALREADY_IN_TREE,
13:        NO_SUCH_PATH, CONFLICTING_PATH, BAD_PATH,
14:        NOT_A_DIRECTORY, NOT_A_FILE,
15:        MEMORY_ERROR
16: };
17:
18: /* In lieu of a proper boolean datatype */
19: enum bool { FALSE, TRUE };
20: /* Make enumeration "feel" more like a builtin type */
21: typedef enum bool boolean;
22:
23: #endif
```

## dt.h (Page 1 of 2)

```
/*--------------------------------------------------------------------*/
/* dt.h                                                               */
/* Author: Christopher Moretti and Vikash Modi '23                    */
/*--------------------------------------------------------------------*/

#ifndef DT_INCLUDED
#define DT_INCLUDED

#include <stddef.h>
#include "a4def.h"

/*
  A Directory Tree is a representation of a hierarchy of directories.
*/

/*
   Inserts a new directory into the DT with absolute path pcPath.
   Returns SUCCESS if the new directory is inserted successfully.
   Otherwise, returns:
   * INITIALIZATION_ERROR if the DT is not in an initialized state
   * BAD_PATH if pcPath does not represent a well-formatted path
   * CONFLICTING_PATH if the root exists but is not a prefix of pcPath
   * ALREADY_IN_TREE if pcPath is already in the DT
   * MEMORY_ERROR if memory could not be allocated to complete request
*/
int DT_insert(const char *pcPath);

/*
  Returns TRUE if the DT contains a directory with absolute path
  pcPath and FALSE if not or if there is an error while checking.
*/
boolean DT_contains(const char *pcPath);


/*
  Removes the DT hierarchy (subtree) from the directory with absolute
  path path pcPath. Returns SUCCESS if found and removed.
  Otherwise, returns:
  * INITIALIZATION_ERROR if the DT is not in an initialized state
  * BAD_PATH if pcPath does not represent a well-formatted path
  * CONFLICTING_PATH if the root exists but is not a prefix of pcPath
  * NO_SUCH_PATH if absolute path pcPath does not exist in the DT
  * MEMORY_ERROR if memory could not be allocated to complete request
*/
int DT_rm(const char *pcPath);

/*
  Sets the DT data structure to an initialized state.
  The data structure is initially empty.
  Returns INITIALIZATION_ERROR if already initialized,
  and SUCCESS otherwise.
*/
int DT_init(void);

/*
  Removes all contents of the data structure and
  returns it to an uninitialized state.
  Returns INITIALIZATION_ERROR if not already initialized,
  and SUCCESS otherwise.
*/
int DT_destroy(void);

/*
```

## dt.h (Page 2 of 2)

```
  Returns a string representation of the
  data structure, or NULL if the structure is
  not initialized or there is an allocation error.

  The representation is depth-first, with nodes
  at any given level ordered lexicographically.

  Allocates memory for the returned string,
  which is then owned by client!
*/
char *DT_toString(void);

#endif
```

## dtGood.c (Page 1 of 7)

```
1: /*--------------------------------------------------------------------*/
2: /* dt.c                                                               */
3: /* Author: Christopher Moretti                                        */
4: /*--------------------------------------------------------------------*/
5:
6: #include <stddef.h>
7: #include <assert.h>
8: #include <string.h>
9: #include <stdio.h>
10: #include <stdlib.h>
11:
12: #include "dynarray.h"
13: #include "path.h"
14: #include "nodeDT.h"
15: #include "checkerDT.h"
16: #include "dt.h"
17:
18:
19: /*
20:   A Directory Tree is a representation of a hierarchy of directories,
21:   represented as an AO with 3 state variables:
22: */
23:
24: /* 1. a flag for being in an initialized state (TRUE) or not (FALSE) */
25: static boolean bIsInitialized;
26: /* 2. a pointer to the root node in the hierarchy */
27: static Node_T oNRoot;
28: /* 3. a counter of the number of nodes in the hierarchy */
29: static size_t ulCount;
30:
31:
32:
33: /* ------------------------------------------------------------------------
34:
35:   The DT_traversePath and DT_findNode functions modularize the common
36:   functionality of going as far as possible down an DT towards a path
37:   and returning either the node of however far was reached or the
38:   node if the full path was reached, respectively.
39: */
40:
41: /*
42:   Traverses the DT starting at the root as far as possible towards
43:   absolute path oPPath. If able to traverse, returns an int SUCCESS
44:   status and sets *poNFurthest to the furthest node reached (which may
45:   be only a prefix of oPPath, or even NULL if the root is NULL).
46:   Otherwise, sets *poNFurthest to NULL and returns with status:
47:   * CONFLICTING_PATH if the root's path is not a prefix of oPPath
48:   * MEMORY_ERROR if memory could not be allocated to complete request
49: */
50: static int DT_traversePath(Path_T oPPath, Node_T *poNFurthest) {
51:    int iStatus;
52:    Path_T oPPrefix = NULL;
53:    Node_T oNCurr;
54:    Node_T oNChild = NULL;
55:    size_t ulDepth;
56:    size_t i;
57:    size_t ulChildID;
58:
59:    assert(oPPath != NULL);
60:    assert(poNFurthest != NULL);
61:
62:    /* root is NULL -> won't find anything */
63:    if(oNRoot == NULL) {
```

## dtGood.c (Page 2 of 7)

```
64:       *poNFurthest = NULL;
65:       return SUCCESS;
66:    }
67:
68:    iStatus = Path_prefix(oPPath, 1, &oPPrefix);
69:    if(iStatus != SUCCESS) {
70:       *poNFurthest = NULL;
71:       return iStatus;
72:    }
73:
74:    if(Path_comparePath(Node_getPath(oNRoot), oPPrefix)) {
75:       Path_free(oPPrefix);
76:       *poNFurthest = NULL;
77:       return CONFLICTING_PATH;
78:    }
79:    Path_free(oPPrefix);
80:    oPPrefix = NULL;
81:
82:    oNCurr = oNRoot;
83:    ulDepth = Path_getDepth(oPPath);
84:    for(i = 2; i <= ulDepth; i++) {
85:       iStatus = Path_prefix(oPPath, i, &oPPrefix);
86:       if(iStatus != SUCCESS) {
87:          *poNFurthest = NULL;
88:          return iStatus;
89:       }
90:       if(Node_hasChild(oNCurr, oPPrefix, &ulChildID)) {
91:          /* go to that child and continue with next prefix */
92:          Path_free(oPPrefix);
93:          oPPrefix = NULL;
94:          iStatus = Node_getChild(oNCurr, ulChildID, &oNChild);
95:          if(iStatus != SUCCESS) {
96:             *poNFurthest = NULL;
97:             return iStatus;
98:          }
99:          oNCurr = oNChild;
100:      }
101:      else {
102:         /* oNCurr doesn't have child with path oPPrefix:
103:            this is as far as we can go */
104:         break;
105:      }
106:   }
107:
108:   Path_free(oPPrefix);
109:   *poNFurthest = oNCurr;
110:   return SUCCESS;
111: }
112:
113: /*
114:   Traverses the DT to find a node with absolute path pcPath. Returns a
115:   int SUCCESS status and sets *poNResult to be the node, if found.
116:   Otherwise, sets *poNResult to NULL and returns with status:
117:   * INITIALIZATION_ERROR if the DT is not in an initialized state
118:   * BAD_PATH if pcPath does not represent a well-formatted path
119:   * CONFLICTING_PATH if the root's path is not a prefix of pcPath
120:   * NO_SUCH_PATH if no node with pcPath exists in the hierarchy
121:   * MEMORY_ERROR if memory could not be allocated to complete request
122:  */
123: static int DT_findNode(const char *pcPath, Node_T *poNResult) {
124:    Path_T oPPath = NULL;
125:    Node_T oNFound = NULL;
126:    int iStatus;
```

## dtGood.c (Page 3 of 7)

```
127:
128:    assert(pcPath != NULL);
129:    assert(poNResult != NULL);
130:
131:    if(!bIsInitialized) {
132:       *poNResult = NULL;
133:       return INITIALIZATION_ERROR;
134:    }
135:
136:    iStatus = Path_new(pcPath, &oPPath);
137:    if(iStatus != SUCCESS) {
138:       *poNResult = NULL;
139:       return iStatus;
140:    }
141:
142:    iStatus = DT_traversePath(oPPath, &oNFound);
143:    if(iStatus != SUCCESS)
144:    {
145:       Path_free(oPPath);
146:       *poNResult = NULL;
147:       return iStatus;
148:    }
149:
150:    if(oNFound == NULL) {
151:       Path_free(oPPath);
152:       *poNResult = NULL;
153:       return NO_SUCH_PATH;
154:    }
155:
156:    if(Path_comparePath(Node_getPath(oNFound), oPPath) != 0) {
157:       Path_free(oPPath);
158:       *poNResult = NULL;
159:       return NO_SUCH_PATH;
160:    }
161:
162:    Path_free(oPPath);
163:    *poNResult = oNFound;
164:    return SUCCESS;
165: }
166: /*--------------------------------------------------------------------*/
167:
168:
169: int DT_insert(const char *pcPath) {
170:    int iStatus;
171:    Path_T oPPath = NULL;
172:    Node_T oNFirstNew = NULL;
173:    Node_T oNCurr = NULL;
174:    size_t ulDepth, ulIndex;
175:    size_t ulNewNodes = 0;
176:
177:    assert(pcPath != NULL);
178:    assert(CheckerDT_isValid(bIsInitialized, oNRoot, ulCount));
179:
180:    /* validate pcPath and generate a Path_T for it */
181:    if(!bIsInitialized)
182:       return INITIALIZATION_ERROR;
183:
184:    iStatus = Path_new(pcPath, &oPPath);
185:    if(iStatus != SUCCESS)
186:       return iStatus;
187:
188:    /* find the closest ancestor of oPPath already in the tree */
189:    iStatus= DT_traversePath(oPPath, &oNCurr);
```

## dtGood.c (Page 4 of 7)

```
190:    if(iStatus != SUCCESS)
191:    {
192:       Path_free(oPPath);
193:       return iStatus;
194:    }
195:
196:    /* no ancestor node found, so if root is not NULL,
197:       pcPath isn't underneath root. */
198:    if(oNCurr == NULL && oNRoot != NULL) {
199:       Path_free(oPPath);
200:       return CONFLICTING_PATH;
201:    }
202:
203:    ulDepth = Path_getDepth(oPPath);
204:    if(oNCurr == NULL) /* new root! */
205:       ulIndex = 1;
206:    else {
207:       ulIndex = Path_getDepth(Node_getPath(oNCurr))+1;
208:
209:       /* oNCurr is the node we're trying to insert */
210:       if(ulIndex == ulDepth+1 && !Path_comparePath(oPPath,
211:                                    Node_getPath(oNCurr))) {
212:          Path_free(oPPath);
213:          return ALREADY_IN_TREE;
214:       }
215:    }
216:
217:    /* starting at oNCurr, build rest of the path one level at a time */
218:    while(ulIndex <= ulDepth) {
219:       Path_T oPPrefix = NULL;
220:       Node_T oNNewNode = NULL;
221:
222:       /* generate a Path_T for this level */
223:       iStatus = Path_prefix(oPPath, ulIndex, &oPPrefix);
224:       if(iStatus != SUCCESS) {
225:          Path_free(oPPath);
226:          if(oNFirstNew != NULL)
227:             (void) Node_free(oNFirstNew);
228:          assert(CheckerDT_isValid(bIsInitialized, oNRoot, ulCount));
229:          return iStatus;
230:       }
231:
232:       /* insert the new node for this level */
233:       iStatus = Node_new(oPPrefix, oNCurr, &oNNewNode);
234:       if(iStatus != SUCCESS) {
235:          Path_free(oPPath);
236:          Path_free(oPPrefix);
237:          if(oNFirstNew != NULL)
238:             (void) Node_free(oNFirstNew);
239:          assert(CheckerDT_isValid(bIsInitialized, oNRoot, ulCount));
240:          return iStatus;
241:       }
242:
243:       /* set up for next level */
244:       Path_free(oPPrefix);
245:       oNCurr = oNNewNode;
246:       ulNewNodes++;
247:       if(oNFirstNew == NULL)
248:          oNFirstNew = oNCurr;
249:       ulIndex++;
250:    }
251:
252:    Path_free(oPPath);
```

## dtGood.c (Page 5 of 7)

```
253:     /* update DT state variables to reflect insertion */
254:     if(oNRoot == NULL)
255:        oNRoot = oNFirstNew;
256:     ulCount += ulNewNodes;
257:
258:     assert(CheckerDT_isValid(bIsInitialized, oNRoot, ulCount));
259:     return SUCCESS;
260: }
261:
262: boolean DT_contains(const char *pcPath) {
263:     int iStatus;
264:     Node_T oNFound = NULL;
265:
266:     assert(pcPath != NULL);
267:
268:     iStatus = DT_findNode(pcPath, &oNFound);
269:     return (boolean) (iStatus == SUCCESS);
270: }
271:
272:
273: int DT_rm(const char *pcPath) {
274:     int iStatus;
275:     Node_T oNFound = NULL;
276:
277:     assert(pcPath != NULL);
278:     assert(CheckerDT_isValid(bIsInitialized, oNRoot, ulCount));
279:
280:     iStatus = DT_findNode(pcPath, &oNFound);
281:
282:     if(iStatus != SUCCESS)
283:         return iStatus;
284:
285:     ulCount -= Node_free(oNFound);
286:     if(ulCount == 0)
287:        oNRoot = NULL;
288:
289:     assert(CheckerDT_isValid(bIsInitialized, oNRoot, ulCount));
290:     return SUCCESS;
291: }
292:
293: int DT_init(void) {
294:     assert(CheckerDT_isValid(bIsInitialized, oNRoot, ulCount));
295:
296:     if(bIsInitialized)
297:        return INITIALIZATION_ERROR;
298:
299:     bIsInitialized = TRUE;
300:     oNRoot = NULL;
301:     ulCount = 0;
302:
303:     assert(CheckerDT_isValid(bIsInitialized, oNRoot, ulCount));
304:     return SUCCESS;
305: }
306:
307: int DT_destroy(void) {
308:     assert(CheckerDT_isValid(bIsInitialized, oNRoot, ulCount));
309:
310:     if(!bIsInitialized)
311:        return INITIALIZATION_ERROR;
312:
313:     if(oNRoot) {
314:        ulCount -= Node_free(oNRoot);
315:        oNRoot = NULL;
```

## dtGood.c (Page 6 of 7)

```
316:     }
317:
318:     bIsInitialized = FALSE;
319:
320:     assert(CheckerDT_isValid(bIsInitialized, oNRoot, ulCount));
321:     return SUCCESS;
322: }
323:
324:
325: /* ------------------------------------------------------------------------
326:
327:    The following auxiliary functions are used for generating the
328:    string representation of the DT.
329: */
330:
331: /*
332:    Performs a pre-order traversal of the tree rooted at n,
333:    inserting each payload to DynArray_T d beginning at index i.
334:    Returns the next unused index in d after the insertion(s).
335: */
336: static size_t DT_preOrderTraversal(Node_T n, DynArray_T d, size_t i) {
337:     size_t c;
338:
339:     assert(d != NULL);
340:
341:     if(n != NULL) {
342:        (void) DynArray_set(d, i, n);
343:        i++;
344:        for(c = 0; c < Node_getNumChildren(n); c++) {
345:           int iStatus;
346:           Node_T oNChild = NULL;
347:           iStatus = Node_getChild(n,c, &oNChild);
348:           assert(iStatus == SUCCESS);
349:           i = DT_preOrderTraversal(oNChild, d, i);
350:        }
351:     }
352:     return i;
353: }
354:
355: /*
356:    Alternate version of strlen that uses pulAcc as an in-out parameter
357:    to accumulate a string length, rather than returning the length of
358:    oNNode's path, and also always adds one addition byte to the sum.
359: */
360: static void DT_strlenAccumulate(Node_T oNNode, size_t *pulAcc) {
361:     assert(pulAcc != NULL);
362:
363:     if(oNNode != NULL)
364:        *pulAcc += (Path_getStrLength(Node_getPath(oNNode)) + 1);
365: }
366:
367: /*
368:    Alternate version of strcat that inverts the typical argument
369:    order, appending oNNode's path onto pcAcc, and also always adds one
370:    newline at the end of the concatenated string.
371: */
372: static void DT_strcatAccumulate(Node_T oNNode, char *pcAcc) {
373:     assert(pcAcc != NULL);
374:
375:     if(oNNode != NULL) {
376:        strcat(pcAcc, Path_getPathname(Node_getPath(oNNode)));
377:        strcat(pcAcc, "\n");
378:     }
```

```
379: }
380: /*--------------------------------------------------------------------*/
381:
382: char *DT_toString(void) {
383:    DynArray_T nodes;
384:    size_t totalStrlen = 1;
385:    char *result = NULL;
386:
387:    if(!bIsInitialized)
388:       return NULL;
389:
390:    nodes = DynArray_new(ulCount);
391:    (void) DT_preOrderTraversal(oNRoot, nodes, 0);
392:
393:    DynArray_map(nodes, (void (*)(void *, void*)) DT_strlenAccumulate,
394:                 (void*) &totalStrlen);
395:
396:    result = malloc(totalStrlen);
397:    if(result == NULL) {
398:       DynArray_free(nodes);
399:       return NULL;
400:    }
401:    *result = '\0';
402:
403:    DynArray_map(nodes, (void (*)(void *, void*)) DT_strcatAccumulate,
404:                 (void *) result);
405:
406:    DynArray_free(nodes);
407:
408:    return result;
409: }
```

## dt_client.c (Page 1 of 3)

```
/*-------------------------------------------------------------------------*/
/* dt_client.c                                                             */
/* Author: Christopher Moretti                                             */
/*-------------------------------------------------------------------------*/

#include <assert.h>
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "dt.h"

/* Tests the DT implementation with an assortment of checks.
   Prints the status of the data structure along the way to stderr.
   Returns 0. */
int main(void) {
  char* temp;

  /* Before the data structure is initialized:
     * insert, rm, and destroy should each return INITIALIZATION_ERROR
     * contains should return FALSE
     * toString should return NULL
  */
  assert(DT_insert("1root/2child/3grandchild") == INITIALIZATION_ERROR);
  assert(DT_contains("1root/2child/3grandchild") == FALSE);
  assert(DT_rm("1root/2child/3grandchild") == INITIALIZATION_ERROR);
  assert((temp = DT_toString()) == NULL);
  assert(DT_destroy() == INITIALIZATION_ERROR);

  /* After initialization, the data structure is empty, so
     contains should still return FALSE for any non-NULL string,
     and toString should return the empty string.
  */
  assert(DT_init() == SUCCESS);
  assert(DT_contains("") == FALSE);
  assert(DT_contains("1root") == FALSE);
  assert((temp = DT_toString()) != NULL);
  assert(!strcmp(temp,""));
  free(temp);

  /* A valid path must not:
     * be the empty string
     * start with a '/'
     * end with a '/'
     * have consecutive '/' delimiters.
  */
  assert(DT_insert("") == BAD_PATH);
  assert(DT_insert("/1root/2child") == BAD_PATH);
  assert(DT_insert("1root/2child/") == BAD_PATH);
  assert(DT_insert("1root//2child") == BAD_PATH);

  /* After insertion, the data structure should contain every prefix
     of the inserted path, toString should return a string with these
     prefixes, trying to insert it again should return
     ALREADY_IN_TREE, and trying to insert some other root should
     return CONFLICTING_PATH.
  */
  assert(DT_insert("1root") == SUCCESS);
  assert(DT_insert("1root/2child/3grandchild") == SUCCESS);
  assert(DT_contains("1root") == TRUE);
  assert(DT_contains("1root/2child") == TRUE);
  assert(DT_contains("1root/2child/3grandchild") == TRUE);
  assert(DT_contains("anotherRoot") == FALSE);
```

## dt_client.c (Page 2 of 3)

```
  assert(DT_insert("anotherRoot") == CONFLICTING_PATH);
  assert(DT_contains("anotherRoot") == FALSE);
  assert(DT_contains("1root/2second") == FALSE);
  assert(DT_insert("1root/2child/3grandchild") == ALREADY_IN_TREE);
  assert(DT_insert("anotherRoot/2nope/3noteven") == CONFLICTING_PATH);

  /* Trying to insert a third child should succeed, unlike in BDT */
  assert(DT_insert("1root/2second") == SUCCESS);
  assert(DT_insert("1root/2third") == SUCCESS);
  assert(DT_insert("1root/2ok/3yes/4indeed") == SUCCESS);
  assert(DT_contains("1root") == TRUE);
  assert(DT_contains("1root/2child") == TRUE);
  assert(DT_contains("1root/2second") == TRUE);
  assert(DT_contains("1root/2third") == TRUE);
  assert(DT_contains("1root/2ok") == TRUE);
  assert(DT_contains("1root/2ok/3yes") == TRUE);
  assert(DT_contains("1root/2ok/3yes/4indeed") == TRUE);
  assert((temp = DT_toString()) != NULL);
  fprintf(stderr, "Checkpoint 1:\n%s\n", temp);
  free(temp);

  /* Children of any path must be unique, but individual directories
     in different paths needn't be
  */
  assert(DT_insert("1root/2child/3grandchild") == ALREADY_IN_TREE);
  assert(DT_contains("1root/2second/3grandchild") == FALSE);
  assert(DT_insert("1root/2second/3grandchild") == SUCCESS);
  assert(DT_contains("1root/2child/3grandchild") == TRUE);
  assert(DT_contains("1root/2second/3grandchild") == TRUE);
  assert(DT_insert("1root/2second/3grandchild") == ALREADY_IN_TREE);
  assert(DT_insert("1root/2second/3grandchild/1root") == SUCCESS);
  assert(DT_contains("1root/2second/3grandchild/1root") == TRUE);
  assert((temp = DT_toString()) != NULL);
  fprintf(stderr, "Checkpoint 2:\n%s\n", temp);
  free(temp);

  /* calling rm on a path that doesn't exist should return
     NO_SUCH_PATH, but on a path that does exist should return
     SUCCESS and remove entire subtree rooted at that path
  */
  assert(DT_contains("1root/2second/3grandchild/1root") == TRUE);
  assert(DT_contains("1root/2second/3second") == FALSE);
  assert(DT_rm("1root/2second/3second") == NO_SUCH_PATH);
  assert(DT_contains("1root/2second/3second") == FALSE);
  assert(DT_rm("1root/2second") == SUCCESS);
  assert(DT_contains("1root") == TRUE);
  assert(DT_contains("1root/2child") == TRUE);
  assert(DT_contains("1root/2second") == FALSE);
  assert(DT_contains("1root/2second/3grandchild") == FALSE);
  assert(DT_contains("1root/2second/3grandchild/1root") == FALSE);
  assert((temp = DT_toString()) != NULL);
  fprintf(stderr, "Checkpoint 3:\n%s\n", temp);
  free(temp);

  /* removing the root doesn't uninitialize the structure */
  assert(DT_rm("1anotherroot") == CONFLICTING_PATH);
  assert(DT_rm("1root") == SUCCESS);
  assert(DT_contains("1root/2child") == FALSE);
  assert(DT_contains("1root") == FALSE);
  assert(DT_rm("1root") == NO_SUCH_PATH);
  assert(DT_rm("1anotherroot") == NO_SUCH_PATH);
  assert((temp = DT_toString()) != NULL);
  assert(!strcmp(temp,""));
```

## dt_client.c (Page 3 of 3)

```
127:    free(temp);
128:
129:    /* children should be printed in lexicographic order, depth first */
130:
131:    /* Debugging: you may want to add this line before any failing
132:       assert(!strcmp(...)) line in the code below:
133:       fprintf(stderr, "Checkpoint Promotion:\n%s\n", temp);
134:    */
135:    assert(DT_insert("a/y") == SUCCESS);
136:    assert((temp = DT_toString()) != NULL);
137:    assert(!strcmp(temp,"a\na/y\n"));
138:    free(temp);
139:    assert(DT_insert("a/x") == SUCCESS);
140:    assert((temp = DT_toString()) != NULL);
141:    assert(!strcmp(temp,"a\na/x\na/y\n"));
142:    free(temp);
143:    assert(DT_rm("a/y") == SUCCESS);
144:    assert((temp = DT_toString()) != NULL);
145:    assert(!strcmp(temp,"a\na/x\n"));
146:    free(temp);
147:    assert(DT_insert("a/y2") == SUCCESS);
148:    assert((temp = DT_toString()) != NULL);
149:    assert(!strcmp(temp,"a\na/x\na/y2\n"));
150:    free(temp);
151:    assert(DT_insert("a/y2/GRAND1") == SUCCESS);
152:    assert((temp = DT_toString()) != NULL);
153:    assert(!strcmp(temp,"a\na/x\na/y2\na/y2/GRAND1\n"));
154:    free(temp);
155:    assert(DT_insert("a/y/Grand0") == SUCCESS);
156:    assert(DT_insert("a/y/Grand2") == SUCCESS);
157:    assert(DT_insert("a/y/Grand1/Great_Grand") == SUCCESS);
158:    assert(DT_insert("a/x/Grandx/Great_GrandX") == SUCCESS);
159:    assert((temp = DT_toString()) != NULL);
160:    fprintf(stderr, "Checkpoint 4:\n%s\n", temp);
161:    free(temp);
162:
163:    assert(DT_destroy() == SUCCESS);
164:    assert(DT_destroy() == INITIALIZATION_ERROR);
165:    assert(DT_contains("a") == FALSE);
166:    assert((temp = DT_toString()) == NULL);
167:
168:    return 0;
169: }
```

## nodeDT.h (Page 1 of 2)

```
/*--------------------------------------------------------------------*/
/* nodeDT.h                                                           */
/* Author: Christopher Moretti                                        */
/*--------------------------------------------------------------------*/

#ifndef NODE_INCLUDED
#define NODE_INCLUDED

#include <stddef.h>
#include "a4def.h"
#include "path.h"


/* A Node_T is a node in a Directory Tree */
typedef struct node *Node_T;

/*
  Creates a new node in the Directory Tree, with path oPPath and
  parent oNParent. Returns an int SUCCESS status and sets *poNResult
  to be the new node if successful. Otherwise, sets *poNResult to NULL
  and returns status:
  * MEMORY_ERROR if memory could not be allocated to complete request
  * CONFLICTING_PATH if oNParent's path is not an ancestor of oPPath
  * NO_SUCH_PATH if oPPath is of depth 0
                 or oNParent's path is not oPPath's direct parent
                 or oNParent is NULL but oPPath is not of depth 1
  * ALREADY_IN_TREE if oNParent already has a child with this path
*/
int Node_new(Path_T oPPath, Node_T oNParent, Node_T *poNResult);

/*
  Destroys and frees all memory allocated for the subtree rooted at
  oNNode, i.e., deletes this node and all its descendents. Returns the
  number of nodes deleted.
*/
size_t Node_free(Node_T oNNode);

/* Returns the path object representing oNNode's absolute path. */
Path_T Node_getPath(Node_T oNNode);

/*
  Returns TRUE if oNParent has a child with path oPPath. Returns
  FALSE if it does not.

  If oNParent has such a child, stores in *pulChildID the child's
  identifier (as used in Node_getChild). If oNParent does not have
  such a child, stores in *pulChildID the identifier that such a
  child _would_ have if inserted.
*/
boolean Node_hasChild(Node_T oNParent, Path_T oPPath,
                      size_t *pulChildID);

/* Returns the number of children that oNParent has. */
size_t Node_getNumChildren(Node_T oNParent);

/*
  Returns an int SUCCESS status and sets *poNResult to be the child
  node of oNParent with identifier ulChildID, if one exists.
  Otherwise, sets *poNResult to NULL and returns status:
  * NO_SUCH_PATH if ulChildID is not a valid child for oNParent
*/
int Node_getChild(Node_T oNParent, size_t ulChildID,
                  Node_T *poNResult);
```

## nodeDT.h (Page 2 of 2)

```

/*
  Returns a the parent node of oNNode.
  Returns NULL if oNNode is the root and thus has no parent.
*/
Node_T Node_getParent(Node_T oNNode);

/*
  Compares oNFirst and oNSecond lexicographically based on their paths.
  Returns <0, 0, or >0 if onFirst is "less than", "equal to", or
  "greater than" oNSecond, respectively.
*/
int Node_compare(Node_T oNFirst, Node_T oNSecond);

/*
  Returns a string representation for oNNode, or NULL if
  there is an allocation error.

  Allocates memory for the returned string, which is then owned by
  the caller!
*/
char *Node_toString(Node_T oNNode);

#endif
```

## nodeDTGood.c (Page 1 of 4)

```
1: /*--------------------------------------------------------------------*/
2: /* nodeDT.c                                                           */
3: /* Author: Christopher Moretti                                        */
4: /*--------------------------------------------------------------------*/
5:
6: #include <stdlib.h>
7: #include <assert.h>
8: #include <string.h>
9: #include "dynarray.h"
10: #include "nodeDT.h"
11: #include "checkerDT.h"
12:
13: /* A node in a DT */
14: struct node {
15:    /* the object corresponding to the node's absolute path */
16:    Path_T oPPath;
17:    /* this node's parent */
18:    Node_T oNParent;
19:    /* the object containing links to this node's children */
20:    DynArray_T oDChildren;
21: };
22:
23:
24: /*
25:   Links new child oNChild into oNParent's children array at index
26:   ulIndex. Returns SUCCESS if the new child was added successfully,
27:   or  MEMORY_ERROR if allocation fails adding oNChild to the array.
28: */
29: static int Node_addChild(Node_T oNParent, Node_T oNChild,
30:                          size_t ulIndex) {
31:    assert(oNParent != NULL);
32:    assert(oNChild != NULL);
33:
34:    if(DynArray_addAt(oNParent->oDChildren, ulIndex, oNChild))
35:       return SUCCESS;
36:    else
37:       return MEMORY_ERROR;
38: }
39:
40: /*
41:   Compares the string representation of oNfirst with a string
42:   pcSecond representing a node's path.
43:   Returns <0, 0, or >0 if oNFirst is "less than", "equal to", or
44:   "greater than" pcSecond, respectively.
45: */
46: static int Node_compareString(const Node_T oNFirst,
47:                               const char *pcSecond) {
48:    assert(oNFirst != NULL);
49:    assert(pcSecond != NULL);
50:
51:    return Path_compareString(oNFirst->oPPath, pcSecond);
52: }
53:
54:
55: /*
56:   Creates a new node with path oPPath and parent oNParent.  Returns an
57:   int SUCCESS status and sets *poNResult to be the new node if
58:   successful. Otherwise, sets *poNResult to NULL and returns status:
59:   * MEMORY_ERROR if memory could not be allocated to complete request
60:   * CONFLICTING_PATH if oNParent's path is not an ancestor of oPPath
61:   * NO_SUCH_PATH if oPPath is of depth 0
62:                  or oNParent's path is not oPPath's direct parent
63:                  or oNParent is NULL but oPPath is not of depth 1
64:   * ALREADY_IN_TREE if oNParent already has a child with this path
65: */
66: int Node_new(Path_T oPPath, Node_T oNParent, Node_T *poNResult) {
```

## nodeDTGood.c (Page 2 of 4)

```
67:    struct node *psNew;
68:    Path_T oPParentPath = NULL;
69:    Path_T oPNewPath = NULL;
70:    size_t ulParentDepth;
71:    size_t ulIndex;
72:    int iStatus;
73:
74:    assert(oPPath != NULL);
75:    assert(oNParent == NULL || CheckerDT_Node_isValid(oNParent));
76:
77:    /* allocate space for a new node */
78:    psNew = malloc(sizeof(struct node));
79:    if(psNew == NULL) {
80:       *poNResult = NULL;
81:       return MEMORY_ERROR;
82:    }
83:
84:    /* set the new node's path */
85:    iStatus = Path_dup(oPPath, &oPNewPath);
86:    if(iStatus != SUCCESS) {
87:       free(psNew);
88:       *poNResult = NULL;
89:       return iStatus;
90:    }
91:    psNew->oPPath = oPNewPath;
92:
93:    /* validate and set the new node's parent */
94:    if(oNParent != NULL) {
95:       size_t ulSharedDepth;
96:
97:       oPParentPath = oNParent->oPPath;
98:       ulParentDepth = Path_getDepth(oPParentPath);
99:       ulSharedDepth = Path_getSharedPrefixDepth(psNew->oPPath,
100:                                                 oPParentPath);
101:       /* parent must be an ancestor of child */
102:       if(ulSharedDepth < ulParentDepth) {
103:          Path_free(psNew->oPPath);
104:          free(psNew);
105:          *poNResult = NULL;
106:          return CONFLICTING_PATH;
107:       }
108:
109:       /* parent must be exactly one level up from child */
110:       if(Path_getDepth(psNew->oPPath) != ulParentDepth + 1) {
111:          Path_free(psNew->oPPath);
112:          free(psNew);
113:          *poNResult = NULL;
114:          return NO_SUCH_PATH;
115:       }
116:
117:       /* parent must not already have child with this path */
118:       if(Node_hasChild(oNParent, oPPath, &ulIndex)) {
119:          Path_free(psNew->oPPath);
120:          free(psNew);
121:          *poNResult = NULL;
122:          return ALREADY_IN_TREE;
123:       }
124:    }
125:    else {
126:       /* new node must be root */
127:       /* can only create one "level" at a time */
128:       if(Path_getDepth(psNew->oPPath) != 1) {
129:          Path_free(psNew->oPPath);
130:          free(psNew);
131:          *poNResult = NULL;
132:          return NO_SUCH_PATH;
```

## nodeDTGood.c (Page 3 of 4)

```
133:       }
134:    }
135:    psNew->oNParent = oNParent;
136:
137:    /* initialize the new node */
138:    psNew->oDChildren = DynArray_new(0);
139:    if(psNew->oDChildren == NULL) {
140:       Path_free(psNew->oPPath);
141:       free(psNew);
142:       *poNResult = NULL;
143:       return MEMORY_ERROR;
144:    }
145:
146:    /* Link into parent's children list */
147:    if(oNParent != NULL) {
148:       iStatus = Node_addChild(oNParent, psNew, ulIndex);
149:       if(iStatus != SUCCESS) {
150:          Path_free(psNew->oPPath);
151:          free(psNew);
152:          *poNResult = NULL;
153:          return iStatus;
154:       }
155:    }
156:
157:    *poNResult = psNew;
158:
159:    assert(oNParent == NULL || CheckerDT_Node_isValid(oNParent));
160:    assert(CheckerDT_Node_isValid(*poNResult));
161:
162:    return SUCCESS;
163: }
164:
165: size_t Node_free(Node_T oNNode) {
166:    size_t ulIndex;
167:    size_t ulCount = 0;
168:
169:    assert(oNNode != NULL);
170:    assert(CheckerDT_Node_isValid(oNNode));
171:
172:    /* remove from parent's list */
173:    if(oNNode->oNParent != NULL) {
174:       if(DynArray_bsearch(
175:             oNNode->oNParent->oDChildren,
176:             oNNode, &ulIndex,
177:             (int (*)(const void *, const void *)) Node_compare)
178:          )
179:          (void) DynArray_removeAt(oNNode->oNParent->oDChildren,
180:                                   ulIndex);
181:    }
182:
183:    /* recursively remove children */
184:    while(DynArray_getLength(oNNode->oDChildren) != 0) {
185:       ulCount += Node_free(DynArray_removeAt(oNNode->oDChildren, 0));
186:    }
187:    DynArray_free(oNNode->oDChildren);
188:
189:    /* remove path */
190:    Path_free(oNNode->oPPath);
191:
192:    /* finally, free the struct node */
193:    free(oNNode);
194:    ulCount++;
195:    return ulCount;
196: }
197:
198: Path_T Node_getPath(Node_T oNNode) {
```

## nodeDTGood.c (Page 4 of 4)

```
199:    assert(oNNode != NULL);
200:
201:    return oNNode->oPPath;
202: }
203:
204: boolean Node_hasChild(Node_T oNParent, Path_T oPPath,
205:                       size_t *pulChildID) {
206:    assert(oNParent != NULL);
207:    assert(oPPath != NULL);
208:    assert(pulChildID != NULL);
209:
210:    /* *pulChildID is the index into oNParent->oDChildren */
211:    return DynArray_bsearch(oNParent->oDChildren,
212:             (char*) Path_getPathname(oPPath), pulChildID,
213:             (int (*)(const void*,const void*)) Node_compareString);
214: }
215:
216: size_t Node_getNumChildren(Node_T oNParent) {
217:    assert(oNParent != NULL);
218:
219:    return DynArray_getLength(oNParent->oDChildren);
220: }
221:
222: int  Node_getChild(Node_T oNParent, size_t ulChildID,
223:                    Node_T *poNResult) {
224:
225:    assert(oNParent != NULL);
226:    assert(poNResult != NULL);
227:
228:    /* ulChildID is the index into oNParent->oDChildren */
229:    if(ulChildID >= Node_getNumChildren(oNParent)) {
230:       *poNResult = NULL;
231:       return NO_SUCH_PATH;
232:    }
233:    else {
234:       *poNResult = DynArray_get(oNParent->oDChildren, ulChildID);
235:       return SUCCESS;
236:    }
237: }
238:
239: Node_T Node_getParent(Node_T oNNode) {
240:    assert(oNNode != NULL);
241:
242:    return oNNode->oNParent;
243: }
244:
245: int Node_compare(Node_T oNFirst, Node_T oNSecond) {
246:    assert(oNFirst != NULL);
247:    assert(oNSecond != NULL);
248:
249:    return Path_comparePath(oNFirst->oPPath, oNSecond->oPPath);
250: }
251:
252: char *Node_toString(Node_T oNNode) {
253:    char *copyPath;
254:
255:    assert(oNNode != NULL);
256:
257:    copyPath = malloc(Path_getStrLength(Node_getPath(oNNode))+1);
258:    if(copyPath == NULL)
259:       return NULL;
260:    else
261:       return strcpy(copyPath, Path_getPathname(Node_getPath(oNNode)));
262: }
```

## path.h (Page 1 of 2)

```
/*--------------------------------------------------------------------*/
/* path.h                                                             */
/* Author: Christopher Moretti                                        */
/*--------------------------------------------------------------------*/

#ifndef PATH_INCLUDED
#define PATH_INCLUDED

#include <stddef.h>
#include "a4def.h"

/* An object representing an absolute path in a tree */
typedef const struct path * Path_T;

/*
  Creates a new path object representing the absolute path in pcPath.
  Returns an int SUCCESS status and sets *poPResult to be the new path
  if successful. Otherwise, sets *poPResult to NULL and returns status:
  * MEMORY_ERROR if memory could not be allocated to complete request
  * BAD_PATH if the string argument is the empty string
             or begins with or ends with a '/'
             or contains consecutive '/' delimiters
*/
int Path_new(const char *pcPath, Path_T *poPResult);

/*
  Creates a "deep copy" of oPPath, duplicating all its contents.
  Returns an int SUCCESS status and sets *poPResult to be the new path
  if successful. Otherwise, sets *poPResult to NULL and returns status:
  * MEMORY_ERROR if memory could not be allocated to complete request
  * NO_SUCH_PATH if oPPath's depth is 0
*/
int Path_dup(Path_T oPPath, Path_T *poPResult);

/*
  Creates a new path object representing a prefix (i.e., ancestor) of
  oPPath with depth ulDepth. In the case that ulDepth is the same as
  oPPath's depth, this is equivalent to Path_dup.
  Returns an int SUCCESS status and sets *poPResult to be the new path
  if successful. Otherwise, sets *poPResult to NULL and returns status:
  * MEMORY_ERROR if memory could not be allocated to complete request
  * NO_SUCH_PATH if ulDepth is 0 or is greater than oPPath's depth
*/
int Path_prefix(Path_T oPPath, size_t ulDepth, Path_T *poPResult);

/* Destroys and frees all memory allocated for oPPath. */
void Path_free(Path_T oPPath);

/* Returns the string representation of the absolute path oPPath. */
const char *Path_getPathname(Path_T oPPath);

/*
  Returns the length (not including trailing '\0') of the string
  representation of the absolute path oPPath.
*/
size_t Path_getStrLength(Path_T oPPath);

/*
  Compares oPPath1 and oPPath2 lexicographically based on pathname.
  Returns <0, 0, or >0 if oPPath1 is "less than", "equal to", or
  "greater than" oPPath2, respectively.
*/
int Path_comparePath(Path_T oPPath1, Path_T oPPath2);
```

## path.h (Page 2 of 2)

```

/*
  Compares oPPath's pathname with pcStr lexicographically.
  Returns <0, 0, or >0 if oPPath is "less than", "equal to", or
  "greater than" pcStr, respectively.
*/
int Path_compareString(Path_T oPPath, const char *pcStr);

/*
  Returns the number of separate levels (components) in oPPath.
  For example, the absolute path "someRoot" has depth 1, and
  "someRoot/aChild/aGrandChild/aGreatGrandChild" has depth 4.
*/
size_t Path_getDepth(Path_T oPPath);

/*
  Returns the length, in components, of the longest prefix shared by
  oPPath1 and oPPath2. For example the absolute paths
  "Charles/William/George" and "Charles/Harry/Archie" have a shared
  prefix depth of 1 (just Charles), whereas "Charles/William/George"
  and "Charles/William/Charlotte" have a shared prefix depth of 2.
*/
size_t Path_getSharedPrefixDepth(Path_T oPPath1, Path_T oPPath2);

/*
  Returns the string version of the component of oPPath at level
  ulLevel. This count is from 0, so with level 0 the root of oPPath
  would be returned.
  Returns NULL if If ulLevel is greater than oPPath's maxium level.
*/
const char *Path_getComponent(Path_T oPPath, size_t ulLevel);

#endif
```

## path.c (Page 1 of 6)

```
/*--------------------------------------------------------------------*/
/* path.c                                                             */
/* Author: Christopher Moretti                                        */
/*--------------------------------------------------------------------*/

#include <assert.h>
#include <stdlib.h>
#include <string.h>

#include "dynarray.h"
#include "path.h"

/* An absolute path */
struct path {
   /* The string representation of the path,
      which uses '/' as the component delimiter */
   const char *pcPath;
   /* The string length of pcPath */
   size_t ulLength;
   /* The ordered collection of component strings in the path */
   DynArray_T oDComponents;
};

/*
  Frees pcStr. This wrapper is used to match the requirements of the
  callback function pointer passed to DynArray_map. pvExtra is unused.
*/
static void Path_freeString(char *pcStr, void *pvExtra) {
   /* pcStr may be NULL, as this is a no-op to free.
      pvExtra may be NULL, as it is unused. */
   free(pcStr);
}

/*
  Sets *poDComponents to be an ordered collection of component strings
  in pcPath, or NULL if an error occurs.
  Returns one of the following statuses:
  * SUCCESS if no error occurrs
  * BAD_PATH if pcPath is the empty string,
              or begins or ends with a '/',
              or contains consecutive '/' delimiters
  * MEMORY_ERROR if memory could not be allocated to complete request
*/
static int Path_split(const char *pcPath, DynArray_T *poDComponents) {
   const char *pcStart = pcPath;
   const char *pcEnd = pcPath;
   char *pcCopy;
   DynArray_T oDSubstrings;

   assert(pcPath != NULL);
   assert(poDComponents != NULL);

   /* path cannot be empty string */
   if(*pcPath == '\0') {
      *poDComponents = NULL;
      return BAD_PATH;
   }

   oDSubstrings = DynArray_new(0);
   if(oDSubstrings == NULL) {
      *poDComponents = NULL;
      return MEMORY_ERROR;
   }
```

## path.c (Page 2 of 6)

```

   /* validate and split pcPath */
   while(*pcEnd != '\0') {
      pcEnd = pcStart;
      /* component can't start with delimiter */
      if(*pcEnd == '/') {
         DynArray_map(oDSubstrings,
                      (void (*)(void*, void*)) Path_freeString, NULL);
         DynArray_free(oDSubstrings);
         *poDComponents = NULL;
         return BAD_PATH;
      }

      /* advance pcEnd to end of next token */
      while(*pcEnd != '/' && *pcEnd != '\0')
         pcEnd++;

      /* final component can't end with slash */
      if(*pcEnd == '\0' && *(pcEnd-1) == '/') {
         DynArray_map(oDSubstrings,
                      (void (*)(void*, void*)) Path_freeString, NULL);
         DynArray_free(oDSubstrings);
         *poDComponents = NULL;
         return BAD_PATH;
      }

      pcCopy = calloc((size_t)(pcEnd-pcStart+1), sizeof(char));
      if(pcCopy == NULL) {
         DynArray_map(oDSubstrings,
                      (void (*)(void*, void*)) Path_freeString, NULL);
         DynArray_free(oDSubstrings);
         *poDComponents = NULL;
         return MEMORY_ERROR;
      }

      if( DynArray_add(oDSubstrings, pcCopy) == 0) {
         DynArray_map(oDSubstrings,
                      (void (*)(void*, void*)) Path_freeString, NULL);
         DynArray_free(oDSubstrings);
         *poDComponents = NULL;
         return MEMORY_ERROR;
      }

      while(pcStart != pcEnd) {
         *pcCopy = *pcStart;
         pcCopy++;
         pcStart++;
      }

      pcStart++;
   }

   *poDComponents = oDSubstrings;
   return SUCCESS;
}


int Path_new(const char *pcPath, Path_T *poPResult) {
   struct path *psNew;
   int iSplitResult;

   assert(pcPath != NULL);
   assert(poPResult != NULL);
```

## path.c (Page 3 of 6)

```
127:
128:    psNew = calloc(1, sizeof(struct path));
129:    if(psNew == NULL) {
130:       *poPResult = NULL;
131:       return MEMORY_ERROR;
132:    }
133:
134:    /* instantiate and fill list of components */
135:    iSplitResult = Path_split(pcPath, &psNew->oDComponents);
136:    if(iSplitResult != SUCCESS) {
137:       Path_free(psNew);
138:       *poPResult = NULL;
139:       return iSplitResult;
140:    }
141:
142:    psNew->ulLength = strlen(pcPath);
143:    psNew->pcPath = malloc(psNew->ulLength+1);
144:    if(psNew->pcPath == NULL) {
145:       Path_free(psNew);
146:       *poPResult = NULL;
147:       return MEMORY_ERROR;
148:    }
149:    strcpy((char *)psNew->pcPath, pcPath);
150:
151:    *poPResult = psNew;
152:    return SUCCESS;
153: }
154:
155: int Path_prefix(Path_T oPPath, size_t ulDepth, Path_T *poPResult) {
156:    struct path *psNew;
157:    size_t ulIndex, ulLength, ulSum;
158:    const char *pcComponent;
159:    char *pcCopy;
160:    char *pcBuild;
161:    char *pcInsert;
162:
163:    assert(oPPath != NULL);
164:    assert(poPResult != NULL);
165:
166:    /* cannot build empty path */
167:    if(ulDepth == 0) {
168:       *poPResult = NULL;
169:       return NO_SUCH_PATH;
170:    }
171:
172:    /* cannot have a prefix longer than oPPath */
173:    if(Path_getDepth(oPPath) < ulDepth) {
174:       *poPResult = NULL;
175:       return NO_SUCH_PATH;
176:    }
177:
178:    psNew = calloc(1, sizeof(struct path));
179:    if(psNew == NULL) {
180:       *poPResult = NULL;
181:       return MEMORY_ERROR;
182:    }
183:
184:    psNew->oDComponents = DynArray_new(ulDepth);
185:    if(psNew->oDComponents == NULL) {
186:       Path_free(psNew);
187:       *poPResult = NULL;
188:       return MEMORY_ERROR;
189:    }
```

## path.c (Page 4 of 6)

```
190:
191:    pcBuild = calloc(Path_getStrLength(oPPath)+1, sizeof(char));
192:    if(pcBuild == NULL) {
193:       Path_free(psNew);
194:       *poPResult = NULL;
195:       return MEMORY_ERROR;
196:    }
197:
198:    pcInsert = pcBuild;
199:    ulSum = 0;
200:
201:    for(ulIndex = 0; ulIndex < ulDepth; ulIndex++) {
202:       /* deep copy each component to new DynArray */
203:       pcComponent = Path_getComponent(oPPath, ulIndex);
204:       ulLength = strlen(pcComponent);
205:       pcCopy = calloc(ulLength + 1, sizeof(char));
206:       if(pcCopy == NULL) {
207:          free(pcBuild);
208:          Path_free(psNew);
209:          *poPResult = NULL;
210:          return MEMORY_ERROR;
211:       }
212:       strcpy(pcCopy, pcComponent);
213:       (void) DynArray_set(psNew->oDComponents, ulIndex, pcCopy);
214:       /* construct prefix's pathname string */
215:       strcpy(pcInsert, pcComponent);
216:       pcInsert[ulLength] = '/';
217:       ulSum += ulLength + 1;
218:       pcInsert += ulLength + 1;
219:    }
220:    pcBuild[ulSum-1] = '\0';
221:
222:    /* shrink allocation to fit prefix's pathname string if needed */
223:    pcInsert = realloc(pcBuild, ulSum);
224:    if(pcInsert == NULL) {
225:       free(pcBuild);
226:       Path_free(psNew);
227:       *poPResult = NULL;
228:       return MEMORY_ERROR;
229:    }
230:    psNew->ulLength = ulSum-1;
231:    psNew->pcPath = pcInsert;
232:
233:    *poPResult = psNew;
234:    return SUCCESS;
235: }
236:
237: int Path_dup(Path_T oPPath, Path_T *poPResult) {
238:    assert(oPPath != NULL);
239:    assert(poPResult != NULL);
240:
241:    return Path_prefix(oPPath, Path_getDepth(oPPath), poPResult);
242: }
243:
244: void Path_free(Path_T oPPath) {
245:    if(oPPath != NULL) {
246:       free((char *)oPPath->pcPath);
247:
248:       if(oPPath->oDComponents != NULL) {
249:          DynArray_map(oPPath->oDComponents,
250:                       (void (*)(void*, void*)) Path_freeString, NULL);
251:          DynArray_free(oPPath->oDComponents);
252:       }
```

## path.c (Page 5 of 6)

```
253:    }
254:    free((struct path*) oPPath);
255: }
256:
257: const char *Path_getPathname(Path_T oPPath) {
258:    assert(oPPath != NULL);
259:
260:    return oPPath->pcPath;
261: }
262:
263: size_t Path_getStrLength(Path_T oPPath) {
264:    assert(oPPath != NULL);
265:
266:    return oPPath->ulLength;
267: }
268:
269: int Path_comparePath(Path_T oPPath1, Path_T oPPath2) {
270:    assert(oPPath1 != NULL);
271:    assert(oPPath2 != NULL);
272:
273:    return strcmp(oPPath1->pcPath, oPPath2->pcPath);
274: }
275:
276: int Path_compareString(Path_T oPPath, const char *pcStr) {
277:    assert(oPPath != NULL);
278:    assert(pcStr != NULL);
279:
280:    return strcmp(oPPath->pcPath, pcStr);
281: }
282:
283: size_t Path_getDepth(Path_T oPPath) {
284:    assert(oPPath != NULL);
285:
286:    return DynArray_getLength(oPPath->oDComponents);
287: }
288:
289: size_t Path_getSharedPrefixDepth(Path_T oPPath1, Path_T oPPath2) {
290:    size_t ulDepth1, ulDepth2, ulMin, i;
291:
292:    assert(oPPath1 != NULL);
293:    assert(oPPath2 != NULL);
294:
295:    ulDepth1 = Path_getDepth(oPPath1);
296:    ulDepth2 = Path_getDepth(oPPath2);
297:    if(ulDepth1 < ulDepth2)
298:       ulMin = ulDepth1;
299:    else
300:       ulMin = ulDepth2;
301:    for(i = 0; i < ulMin; i++) {
302:       if(strcmp(Path_getComponent(oPPath1, i),
303:                 Path_getComponent(oPPath2, i)))
304:          return i;
305:    }
306:    return ulMin;
307: }
308:
309: const char *Path_getComponent(Path_T oPPath, size_t ulLevel) {
310:    assert(oPPath != NULL);
311:
312:    if(ulLevel >= Path_getDepth(oPPath))
313:       return NULL;
314:
315:    return DynArray_get(oPPath->oDComponents, ulLevel);
```

## path.c (Page 6 of 6)

```
316: }
```

**checkerDT.h (Page 1 of 1)**

```
/*--------------------------------------------------------------------*/
/* checkerDT.h                                                        */
/* Author: Christopher Moretti                                        */
/*--------------------------------------------------------------------*/

#ifndef CHECKER_INCLUDED
#define CHECKER_INCLUDED

#include "nodeDT.h"


/*
   Returns TRUE if oNNode represents a directory entry
   in a valid state, or FALSE otherwise.
*/
boolean CheckerDT_Node_isValid(Node_T oNNode);

/*
   Returns TRUE if the hierarchy is in a valid state or FALSE
   otherwise.  The data structure's validity is based on a boolean
   bIsInitialized indicating whether the DT is in an initialized
   state, a Node_T oNRoot representing the root of the hierarchy, and
   a size_t ulCount representing the total number of directories in
   the hierarchy.
*/
boolean CheckerDT_isValid(boolean bIsInitialized,
                          Node_T oNRoot,
                          size_t ulCount);

#endif
```

## checkerDT.c (Page 1 of 2)

```
/*--------------------------------------------------------------------------*/
/* checkerDT.c                                                              */
/* Author:                                                                  */
/*--------------------------------------------------------------------------*/

#include <assert.h>
#include <stdio.h>
#include <string.h>
#include "checkerDT.h"
#include "dynarray.h"
#include "path.h"



/* see checkerDT.h for specification */
boolean CheckerDT_Node_isValid(Node_T oNNode) {
   Node_T oNParent;
   Path_T oPNPath;
   Path_T oPPPath;

   /* Sample check: a NULL pointer is not a valid node */
   if(oNNode == NULL) {
      fprintf(stderr, "A node is a NULL pointer\n");
      return FALSE;
   }

   /* Sample check: parent's path must be the longest possible
      proper prefix of the node's path */
   oNParent = Node_getParent(oNNode);
   if(oNParent != NULL) {
      oPNPath = Node_getPath(oNNode);
      oPPPath = Node_getPath(oNParent);

      if(Path_getSharedPrefixDepth(oPNPath, oPPPath) !=
         Path_getDepth(oPNPath) - 1) {
         fprintf(stderr, "P-C nodes don't have P-C paths: (%s) (%s)\n",
                 Path_getPathname(oPPPath), Path_getPathname(oPNPath));
         return FALSE;
      }
   }

   return TRUE;
}

/*
   Performs a pre-order traversal of the tree rooted at oNNode.
   Returns FALSE if a broken invariant is found and
   returns TRUE otherwise.

   You may want to change this function's return type or
   parameter list to facilitate constructing your checks.
   If you do, you should update this function comment.
*/
static boolean CheckerDT_treeCheck(Node_T oNNode) {
   size_t ulIndex;

   if(oNNode!= NULL) {

      /* Sample check on each node: node must be valid */
      /* If not, pass that failure back up immediately */
      if(!CheckerDT_Node_isValid(oNNode))
         return FALSE;

```

## checkerDT.c (Page 2 of 2)

```
      /* Recur on every child of oNNode */
      for(ulIndex = 0; ulIndex < Node_getNumChildren(oNNode); ulIndex++)
      {
         Node_T oNChild = NULL;
         int iStatus = Node_getChild(oNNode, ulIndex, &oNChild);

         if(iStatus != SUCCESS) {
            fprintf(stderr, "getNumChildren claims more children than ge
tChild returns\n");
            return FALSE;
         }

         /* if recurring down one subtree results in a failed check
            farther down, passes the failure back up immediately */
         if(!CheckerDT_treeCheck(oNChild))
            return FALSE;
      }
   }
   return TRUE;
}

/* see checkerDT.h for specification */
boolean CheckerDT_isValid(boolean bIsInitialized, Node_T oNRoot,
                          size_t ulCount) {

   /* Sample check on a top-level data structure invariant:
      if the DT is not initialized, its count should be 0. */
   if(!bIsInitialized)
      if(ulCount != 0) {
         fprintf(stderr, "Not initialized, but count is not 0\n");
         return FALSE;
      }

   /* Now checks invariants recursively at each node from the root. */
   return CheckerDT_treeCheck(oNRoot);
}
```

**Princeton University**

**COS 217: Introduction to Programming Systems**

**DT Algorithms**

1. int DT_insert(const char* pcPath);

Step 1). If the DT is not initialized, return INITIALIZATION_ERROR.

Step 2) Build a Path object for the path that we want to insert. Find farthest Node reachable from the root following the given path (static function DT_traversePath).

If traversal fails, free the path and return the failure status back up to the caller.

If traversal result (Node_curr) is NULL, but the root is not NULL, then return CONFLICTING_PATH.

Calculate the "depth" of Node_curr. (Implicitly 0 if Node_curr is NULL)

If Node_curr is the node with the path we want, return ALREADY_IN_TREE.

Otherwise, Node_curr is going to be Parent of the next node to be added

Step 3) Starting at Node's depth + 1 (or at 1 if Node is NULL), for each depth until we reach the depth of the final path that we want to insert:

Create a new Path object that is a prefix at that depth of the final path

Create a new node at that depth (Node_new) with the new Path as its path and Node_curr as its parent. If Node_new fails, free the paths, free any nodes we've already made here in step 3, and return the failure status back up to the caller.

Otherwise, the new node is now Node_curr. Keep track of the new nodes we've added here in step 3 (in case one eventually fails and we have to delete them all)

Continue with the next iteration of Step 3 at the next depth.

Step 4) Once we have added all the new nodes to reach our final path that we wanted:

if the root is NULL, set the first new node we made to be the root.

Add the number of new nodes to the data structure's count state variable

Return SUCCESS.

(30)

2. int DT_rm(const char *pcPath);

Step 1). Get a pointer to the Node with the path we want to remove. (Use helper function DT_findNode, which traverses path as far as it can, then returns SUCCESS and sets a pointer to the Node found in the traversal only if "as far is it can" is "all the way"). Otherwise, return the error status returned by findNode:

If the DT is not initialized, findNode returns INITIALIZATION_ERROR.

If findNode can't create a Path object with the path, it returns the error status.

If findNode doesn't get all the way to path we want, it returns NO_SUCH_PATH

Step 2). Call Node_free on the Node found by findNode in order to delete the entire hierarchy rooted at that node. Node_free will return the number of nodes removed

Step 3). Node_free removes its parameter Node from that Node's parent's list of children. This will disconnect the Node from the DT.

If the parameter Node has no parent (i.e., it's the root), this step is not done.

Step 4) For every element in Node's list of children, call Node_free recursively on that Child (i.e., goto step 2, but on a child instead of the Node returned by findNode) to remove that sub-hierarchy.

Accumulate the return values from all recursive calls to count total number of nodes removed.

Step 5) Once all Node's children have been recursively destroyed:

free the now-empty DynArray

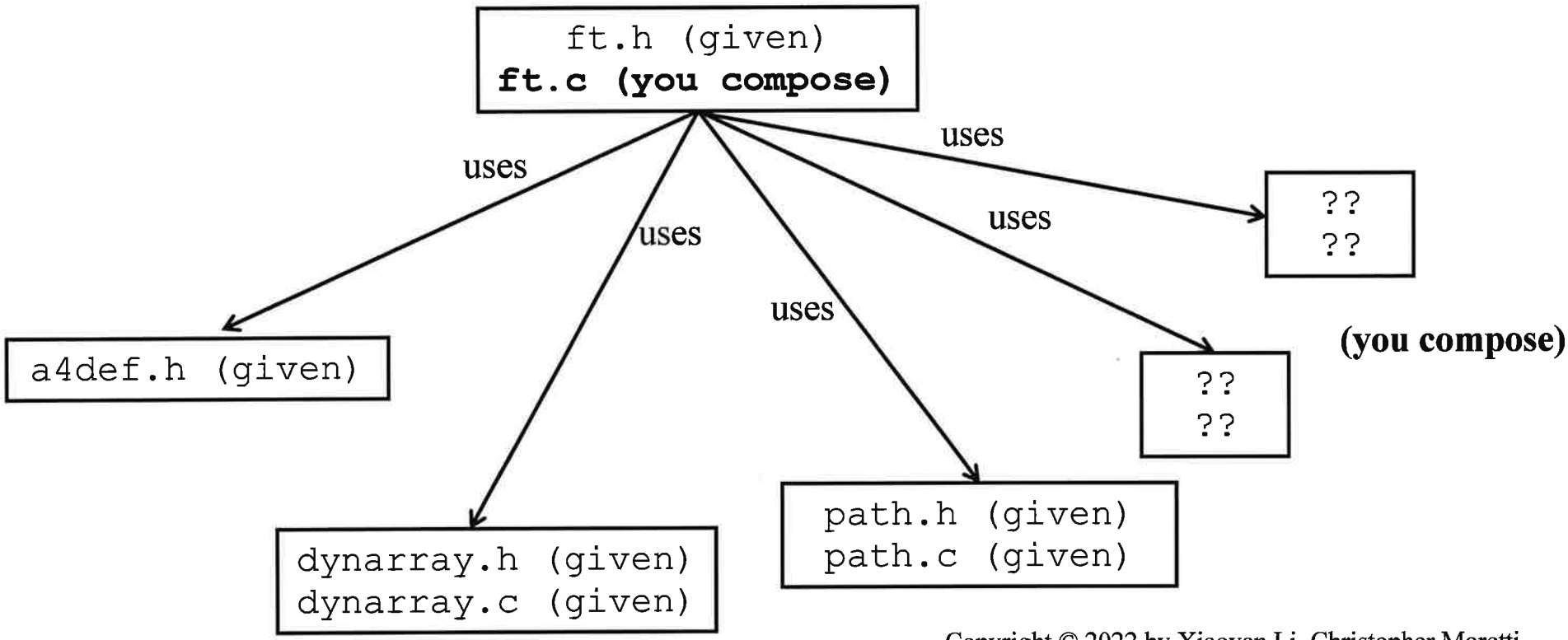
free Node's path object

free the Node itself

Return the total count of nodes removed (including this Node itself).

Step 6) Once all the recursion finishes and the Node_free call on the original pointer found by findNode returns (finally!) back to DT_rm:
decrement count of the DT by the number of nodes removed
If the count is now 0 (i.e., we removed the root), set the DT's root to NULL
Return SUCCESS

# Princeton University
# COS 217: Introduction to Programming Systems
# FT Code



## ft.h (Page 1 of 3)

```
 1: /*--------------------------------------------------------------------*/
 2: /* ft.h                                                               */
 3: /* Author: Christopher Moretti and Vikash Modi '23                    */
 4: /*--------------------------------------------------------------------*/
 5:
 6: #ifndef FT_INCLUDED
 7: #define FT_INCLUDED
 8:
 9: /*
10:   A File Tree is a representation of a hierarchy of directories and
11:   files: the File Tree is rooted at a directory, directories
12:   may be internal nodes or leaves, and files are always leaves.
13: */
14:
15: #include <stddef.h>
16: #include "a4def.h"
17:
18: /*
19:    Inserts a new directory into the FT with absolute path pcPath.
20:    Returns SUCCESS if the new directory is inserted successfully.
21:    Otherwise, returns:
22:    * INITIALIZATION_ERROR if the FT is not in an initialized state
23:    * BAD_PATH if pcPath does not represent a well-formatted path
24:    * CONFLICTING_PATH if the root exists but is not a prefix of pcPath
25:    * NOT_A_DIRECTORY if a proper prefix of pcPath exists as a file
26:    * ALREADY_IN_TREE if pcPath is already in the FT (as dir or file)
27:    * MEMORY_ERROR if memory could not be allocated to complete request
28: */
29: int FT_insertDir(const char *pcPath);
30:
31: /*
32:   Returns TRUE if the FT contains a directory with absolute path
33:   pcPath and FALSE if not or if there is an error while checking.
34: */
35: boolean FT_containsDir(const char *pcPath);
36:
37: /*
38:   Removes the FT hierarchy (subtree) at the directory with absolute
39:   path pcPath. Returns SUCCESS if found and removed.
40:   Otherwise, returns:
41:   * INITIALIZATION_ERROR if the FT is not in an initialized state
42:   * BAD_PATH if pcPath does not represent a well-formatted path
43:   * CONFLICTING_PATH if the root exists but is not a prefix of pcPath
44:   * NO_SUCH_PATH if absolute path pcPath does not exist in the FT
45:   * NOT_A_DIRECTORY if pcPath is in the FT as a file not a directory
46:   * MEMORY_ERROR if memory could not be allocated to complete request
47: */
48: int FT_rmDir(const char *pcPath);
49:
50:
51: /*
52:    Inserts a new file into the FT with absolute path pcPath, with
53:    file contents pvContents of size ulLength bytes.
54:    Returns SUCCESS if the new file is inserted successfully.
55:    Otherwise, returns:
56:    * INITIALIZATION_ERROR if the FT is not in an initialized state
57:    * BAD_PATH if pcPath does not represent a well-formatted path
58:    * CONFLICTING_PATH if the root exists but is not a prefix of pcPath,
59:                        or if the new file would be the FT root
60:    * NOT_A_DIRECTORY if a proper prefix of pcPath exists as a file
61:    * ALREADY_IN_TREE if pcPath is already in the FT (as dir or file)
62:    * MEMORY_ERROR if memory could not be allocated to complete request
63: */
```

## ft.h (Page 2 of 3)

```
64: int FT_insertFile(const char *pcPath, void *pvContents,
65:                   size_t ulLength);
66:
67: /*
68:   Returns TRUE if the FT contains a file with absolute path
69:   pcPath and FALSE if not or if there is an error while checking.
70: */
71: boolean FT_containsFile(const char *pcPath);
72:
73: /*
74:   Removes the FT file with absolute path pcPath.
75:   Returns SUCCESS if found and removed.
76:   Otherwise, returns:
77:   * INITIALIZATION_ERROR if the FT is not in an initialized state
78:   * BAD_PATH if pcPath does not represent a well-formatted path
79:   * CONFLICTING_PATH if the root exists but is not a prefix of pcPath
80:   * NO_SUCH_PATH if absolute path pcPath does not exist in the FT
81:   * NOT_A_FILE if pcPath is in the FT as a directory not a file
82:   * MEMORY_ERROR if memory could not be allocated to complete request
83: */
84: int FT_rmFile(const char *pcPath);
85:
86: /*
87:   Returns the contents of the file with absolute path pcPath.
88:   Returns NULL if unable to complete the request for any reason.
89:
90:   Note: checking for a non-NULL return is not an appropriate
91:   contains check, because the contents of a file may be NULL.
92: */
93: void *FT_getFileContents(const char *pcPath);
94:
95: /*
96:   Replaces current contents of the file with absolute path pcPath with
97:   the parameter pvNewContents of size ulNewLength bytes.
98:   Returns the old contents if successful. (Note: contents may be NULL.)
99:   Returns NULL if unable to complete the request for any reason.
100: */
101: void *FT_replaceFileContents(const char *pcPath, void *pvNewContents,
102:                              size_t ulNewLength);
103:
104: /*
105:   Returns SUCCESS if pcPath exists in the hierarchy,
106:   Otherwise, returns:
107:   * INITIALIZATION_ERROR if the FT is not in an initialized state
108:   * BAD_PATH if pcPath does not represent a well-formatted path
109:   * CONFLICTING_PATH if the root's path is not a prefix of pcPath
110:   * NO_SUCH_PATH if absolute path pcPath does not exist in the FT
111:   * MEMORY_ERROR if memory could not be allocated to complete request
112:
113:   When returning SUCCESS,
114:   if path is a directory: sets *pbIsFile to FALSE, *pulSize unchanged
115:   if path is a file: sets *pbIsFile to TRUE, and
116:                      sets *pulSize to the length of file's contents
117:
118:   When returning another status, *pbIsFile and *pulSize are unchanged.
119: */
120: int FT_stat(const char *pcPath, boolean *pbIsFile, size_t *pulSize);
121:
122: /*
123:   Sets the FT data structure to an initialized state.
124:   The data structure is initially empty.
125:   Returns INITIALIZATION_ERROR if already initialized,
126:   and SUCCESS otherwise.
```

## ft.h (Page 3 of 3)

```
127: */
128: int FT_init(void);
129:
130: /*
131:   Removes all contents of the data structure and
132:   returns it to an uninitialized state.
133:   Returns INITIALIZATION_ERROR if not already initialized,
134:   and SUCCESS otherwise.
135: */
136: int FT_destroy(void);
137:
138: /*
139:   Returns a string representation of the
140:   data structure, or NULL if the structure is
141:   not initialized or there is an allocation error.
142:
143:   The representation is depth-first with files
144:   before directories at any given level, and nodes
145:   of the same type ordered lexicographically.
146:
147:   Allocates memory for the returned string,
148:   which is then owned by client!
149: */
150: char *FT_toString(void);
151:
152: #endif
```

# COS217 Exercise Based on Precept 16

**Before you start A4 Part 2 (internal tests to validate the Directory Tree), do this exercise (with your partner) to get better prepared.**

**Understanding the Directory Tree** CHECKLIST:

( ) Review Lecture 15 and Precepts 15 and 16

( ) Open a browser window to the Precept 15 Handouts and an armlab window in the A4 2DT directory.

( ) Make and run dtGood. Display the A4_DTstructure.pdf and the output of dtGood side-by-side. Make sure you understand the relationship between the tree diagram and the first paragraph of output.

( ) Annotate A4_DTstructure.pdf or draw your own trees to reflect the other paragraphs of output.

( ) Change the browser window to dt_client.c. Discuss each paragraph of code. Try to correlate paragraphs of code with the corresponding paragraph of output. (Note: Some paragraphs of code do not generate any output unless something goes wrong. Some paragraphs of code generate more than one paragraph of output.)

( ) Write a small client program to generate a tree of your own design. Compile it with dtGood.c and nodeGood.c and run the executable. Does the output match your tree?

**Invariants for the Directory Tree** CHECKLIST:

( ) Open a browser window and display A4_DTstructure.pdf. (You might also want to look at the relevant slides form Lecture 14)

( ) Look at the global state variables. How many are there? What are they?

( ) If the DT is in an uninitialized state, is there any value or combination of values of root or count that would indicate a problem? If so, what?

( ) If the DT is in an initialized state, is there any value or combination of values of root or count that would indicate a problem? If so, what?

( ) Look at the nodes. How many fields does a node have? What are they?

( ) Are there any values of each node field that would indicate a problem?

( ) Are there any relationships between nodes that must be true?

( ) Are there any relationships between the nodes and the global state variables that must be true?

( ) Study the incomplete checkerDT.c code.

( ) Which of the invariants you listed above would you check by adding code to CheckerDT_isValid()?

( ) Which of the invariants would you check by adding code to CheckerDT_Node_isValid()?

( ) Which check might require you to edit CheckerDT_treeCheck()? (If / when you edit CheckerDT_treeCheck() be very careful. It is recursive code that traverses the tree structure.)