C Programming:
Data Structures and Algorithms

An introduction to elementary
programming concepts in C

Jack Straub, Instructor
Version 2.07 DRAFT

C Programming: Data Structures and Algorithms
Version 2.07 DRAFT

Table of Contents

Course Overview

C Programming: Data Structures and Algorithms is a ten week course, consisting of three hours per week lecture, plus assigned reading, weekly quizzes and five homework projects. This is primarily a class in the C programming language, and introduces the student to data structure design and implementation.

Objectives

Upon successful completion of this course, you will have demonstrated the following skills:

- The ability to write C-language code according to a project specification;
- The ability to develop C-language modules following standard industry practices and conventions; and
- The ability to properly design data structures and the algorithms to transform them.

In order to receive credit for this course, you must meet the following criteria:

- Achieve 80% attendance in class;
- Achieve a total of 70% on the final examination; and
- Satisfactorily complete all projects.

Instructor

Jack Straub
425 888 9119 (9:00 a.m. to 3:00 p.m. Monday through Friday)
jstraub@centurytel.net
http://faculty.washington.edu/jstraub/

Text Books

Required

No text book is required. However it is *strongly* recommended that you acquire one of the data structures text books listed below; at least one of your projects will require you to do your own research on a data structure not covered in class.

Recommended
C A Reference Manual, Fifth Edition by Samuel P. Harbison, and Guy L. Steele Jr., Prentice Hall, 2002

C Primer Plus, Fifth Edition by Stephen Prata, Sams Publishing, 2006

Recommended Data Structures Textbooks
Data Structures and Program Design in C, Second Edition by Robert Kruse et al.; Prentice Hall, 1997

Fundamentals of Data Structures in C by Ellis Horowitz, Sartaj Sahni and Susan Anderson-Freed; W. H. Freeman, 1992

Algorithms in C, Third Edition Parts 1 - 4 by Robert Sedgewick; Addison-Wesley, 1998

Course Outline

Week	Topics	Assigned Reading	Work Due
1	*Basic Skills, Core Module*	Kruse Chapters 1 and 2 Horowitz Chapter 1 Sedgewick Chapters 1 and 2	
2	*Doubly Linked Lists*	Kruse Chapter 5, through 5.2 Horowitz Chapter 4 Sedgewick Chapter 3, through 3.5	Quiz 1, Project 1
3	*Sorting*	Kruse Chapter 7, through 7.7 Horowitz Chapter 7, through 7.6 Sedgewick Chapter 6 through 6.5, Chapter 8	Quiz 2
4	*Modules, Abstract Data Types*	Kruse Section 4.8 Horowitz Section 1.3 Sedgewick Section 4.1	Quiz 3, Project 2
5	*Stacks*	Kruse Sections 3.1 and 3.2 Horowitz Section 3.1 Sedgewick Sections 4.2 through 4.5	Quiz 4
6	*Priority Queues*	Kruse Sections 7.8 through 7.10 Horowitz Section 3.2 Sedgewick Section 4.6, Chapter 9 through 9.1	Quiz 5, Project 3
7	*System Life Cycle*	Kruse Chapter 9, through 9.4 Horowitz Section 1.1, Chapter 5, through 5.3 Sedgewick Chapter 5, through 5.4	Quiz 6
8	*Binary/N-ary Trees*	Kruse Chapter 10, through 10.2 Horowitz Chapter 5, remainder Sedgewick Chapter 5, remainder	Quiz 7, Project 4
9	*Final Exam*	Kruse Chapter 10, remainder Horowitz Chapter 10, through 10.5 Sedgewick Chapter 16	Quiz 8, Project 5
10	*Wrap up*		

Recommended Reading

	Topics	Assigned Reading
Week 1	*Basic Skills, Core Module*	H&S Sections 5.10, 5.4.1, 5.8 Prata pp. 578 – 589, 480 - 486
Week 2	*Doubly Linked Lists*	H&S Section 5.6, through 5.6.4 Prata pp. 684 - 692
Week 3	*Sorting*	H&S Sections 20.5; Chapter 19 Prata pp. 665 - 670
Week 4	*Modules, Abstract Data Types*	Prata pp. 692 - 718
Week 5	*Stacks*	H&S Sections 7.4.4, 7.5.8
Week 6	*Priority Queues*	Kruse Chapter 11 Prata pp. 708 - 730
Week 7	*System Life Cycle*	Kruse Chapter 12
Week 8	*Binary/N-ary Trees*	Prata 730 - 756
Week 9	*Final Exam*	H&S Chapter 4
Week 10	*Wrap up*	

1. Basics

We will begin this section by reviewing C skills that are particularly important to the study of data structures and algorithms, including *typedefs, pointers and arrays,* and *dynamic memory allocation*. Next we will define a set of utilities that will be useful in implementing the data structures that we will discuss in the remainder of the course. By the end of this section you will be ready to complete your first project.

1.1 Objectives

At the conclusion of this section, and with the successful completion of your first project, you will have demonstrated the ability to:

- Use *typedef* to declare the basic types used to represent a data structure;
- Use *dynamic memory allocation* to create the components of a data structure; and
- Implement core utilities to serve as the foundation of a software development project.

1.2 Typedef

In most C projects, the *typedef statement* is used to create equivalence names for other C types, particularly for structures and pointers, but potentially for any type. Using typedef equivalence names is a good way to hide implementation details. It also makes your code more readable, and improves the overall portability of your product.

1.2.1 Typedef and Portability

Typedef is frequently used to improve code portability. To cite a simple example, suppose you had a need to declare a data structure that was guaranteed to occupy the same amount of memory on every platform. If this structure had integer fields you might be tempted to declare them to be either *short* or *long*, believing that these types would always translate to two- or four-byte integers, respectively. Unfortunately ANSI C makes no such guarantee. Specifically, if you declare a field to be type *long*, it will break when you try to port your code to an Alpha running Digital UNIX, where an *int* is four bytes, and a *long* is eight bytes. To avoid this problem you can use typedef in conjunction with conditional compilation directives to declare integer equivalence types; on most platforms the equivalence type for a four-byte field will be *long*, but under Alpha/Digital UNIX it will be *int*:

```
#if defined( ALPHA ) && defined ( DIGITAL_UNIX )
    typedef int BYTE4_t;
#else
    typedef long BYTE4_t;
#endif
```

For the record, a variable or field that needed to be exactly four bytes will then be declared like this:

```
BYTE4_t field_name;
```

1.2.2 Typedef and Structures

To create a structure variable suitable for describing a street address, and to pass the variable to a routine that could print it, you might use code like that shown in **Figure 1-1**, which explicitly declares all structure components using the *struct* keyword. Normally, however, you would declare equivalence names for the address structure using *typedef*, and then declare structure variables and parameters using the equivalence names, as shown in **Figure 1-2**. Note that, in this example, one typedef statement was used to create two equivalence names: *ADDRESS_t*, which is equivalent to *struct address_s*, and *ADDRESS_p_t*, which is equivalent to *struct address_s**. This is a very common technique in modern C usage.

```
struct address_s
{
    char *street;
    char *city;
    char *region;
    char *country;
    char *postal_code;
};

static void print_address(
    struct address_s *address_info
);

static void print_an_address( void )
{
    struct address_s address;

    address.street = “1823 23rd Ave NE”;
    address.city = “Seattle”;
    address.region = “WA”;
    address.postal_code = “98023”;
    print_address( &address );
}
```

Figure 1-1: Traditional Structure Declarations

1.2.3 Typedef and Functions

Recall that the type of a function is a combination of the function’s *return* type, and the number and type of each of the function’s parameters. Typedef can be used to declare an equivalence name for a function type, and, subsequently, the equivalence name can be used to declare a prototype for a function of that type. One use of this is to reduce repetition when declaring many functions of the same type. For example, if you are implementing a standard sort algorithm, you will need to define a prototype for a *compare function*; specifically, a function used to determine whether one object is greater than another. A prototype for such a function would traditionally be declared like this:

```
/* Returns 1 if item1 is greater than *
 * or equal to item2; 0 otherwise.    */
static int is_greater_equal(
    const void *item1,
    const void *item2
);
```

```
typedef struct address_s
{
    char *street;
    char *city;
    char *region;
    char *country;
    char *postal_code;
} ADDRESS_t, *ADDRESS_p_t;

static void print_address(
    ADDRESS_p_t address_info
);

static void print_an_address( void )
{
    ADDRESS_t address;

    address.street = "1823 23rd Ave NE";
    address.city = "Seattle";
    address.region = "WA";
    address.postal_code = "98023";
    print_address( &address );
}
```

Figure 1-2: Structure Declarations using *typedef*

Instead, however, we might consider declaring an equivalence type for a compare function, then declaring each compare function prototype using the equivalence type. We could do that this way:

```
typedef int COMPARE_PROC_t( const void *, const void * );

static COMPARE_PROC_t is_greater_equal;
```

This technique can also be used to drastically simplify compound declarations. For example, what if we had to declare a field in a structure to be a pointer to a compare function? This is a frequent occurrence, and traditionally would be done this way:

```
typedef struct sort_data_s
{
    int *sort_array;
    int (*test_proc)( const void *item1, const void *item2 );
} SORT_DATA_t, *SORT_DATA_p_t;
```

This structure declaration becomes much easier to write (and to read) if you use the compare function equivalence type from the previous example, and extend it to embrace type *pointer to compare function*, as shown in **Figure 1-3.**

Possibly the ultimate example of using typedef to simplify function-related declarations is to rewrite the prototype for the ANSI function *signal*. Signal is a function with two

parameters; the first is type *int*, and the second is *pointer to function with one parameter of type int that returns void*; the return value of signal is *pointer to function with one parameter of type int that returns void*. The prototype for signal is usually declared like this:

```
void (*signal(
      int sig,
      void (*func)(int)))(int);
```

This prototype is much easier to decipher if we just declare and use an equivalence for type *pointer to function with one parameter of type int that returns void*:

```
typedef void SIG_PROC_t( int );
typedef SIG_PROC_t *SIG_PROC_p_t;

SIG_PROC_p_t signal(
    int          sig,
    SIG_PROC_p_t func
);
typedef int COMPARE_PROC_t( const void *, const void * );
typedef COMPARE_PROC_t *COMPARE_PROC_p_t;

typedef struct sort_data_s
{
    int              *sort_array;
    COMPARE_PROC_p_t test_proc;
} SORT_DATA_t, *SORT_DATA_p_t;
```

```
typedef int COMPARE_PROC_t( const void *, const void * );
typedef COMPARE_PROC_t *COMPARE_PROC_p_t;

typedef struct sort_data_s
{
    int              *sort_array;
    COMPARE_PROC_p_t test_proc;
} SORT_DATA_t, *SORT_DATA_p_t;
```

Figure 1-3: Typedef and Function Types

1.3 Pointers and Arrays

In C, pointers and arrays are very closely related. With only two exceptions, the name of an array is equivalent to a pointer to the first element of the array. In **Figure 1-4**, the first parameter of the sort_dates function is declared to be *pointer to date structure*; the actual call to sort_dates passes the name of an *array of date_structures* as the corresponding argument, and the C compiler obligingly converts the name to a pointer. Since a C subroutine can't determine the length, or *cardinality*, of an array passed as an argument, the second argument to sort_dates specifies the length of the array.

The two exceptions are that an array name cannot be an *lvalue* (it cannot appear on the left side of the equal sign in an assignment); and that C does not treat the name as a pointer when used as the argument of the sizeof operator. If you use the name of an array as an argument of sizeof, the size of the entire array is computed. This allows us to create a very convenient macro which I have called *CARD* (short for *cardinality*); this takes the size of the array divided by the size of the first element of the array, which yields the total

number of elements in the array. As illustrated in **Figure 1-5**, this macro allows us to dynamically determine the size of an array.

1.4 Dynamic Memory Allocation

Dynamic memory allocation in C is performed using one of the standard library functions *malloc, calloc* or *realloc* (or a cover for one of these routines). Dynamically allocated memory must eventually be freed by calling *free*. If allocated memory is not properly freed a *memory leak* results, which could result in a program malfunction.

```
typedef struct date_s
{
    short year;
    char  month;
    char  day;
} DATE_t, *DATE_p_t;

static void sort_dates( DATE_p_t dates, int num_dates );

    DATE_t dates[4] = { {1066,  3, 27},
                        {1941, 12,  1},
                        {1492, 10, 12},
                        {1815, 10, 14}
                      };
    . . .
    sort_dates( dates, 4 );
```

Figure 1-4: Pointers and Arrays

The ability to dynamically allocate memory accounts for much of the power of C. It also accounts for much of the complexity of many C programs, and, subsequently, is the source of many of their problems. One of the biggest problems associated with dynamically allocated memory comes from trying to deal with allocation failure. Many organizations effectively short-circuit this problem by writing cover routines for the dynamic memory allocation routines that *do not return* when an error occurs; instead, they abort the program. In **Figure 1-6** we see a cover for *malloc*; writing cover routines for *calloc* and *realloc* is left as an exercise for the student.

```
#define CARD( arr )    (sizeof((arr))/sizeof(*(arr)))
    . . .
    sort_dates( dates, CARD( dates ) );
```

Figure 1-5: Arrays and *sizeof*

1.5 The Core Module

I am going to leave a formal definition of the term *module* for later. For now, let's just say that a module is a collection of related facilities, including functions, macros, and

data type declarations. A module consists of a minimum of three files, a *private header file*, a *public header file* and a *principal source file*. In this section we will develop the *core module*, which will contain a collection of facilities to assist us in writing code throughout the remainder of the course. The name of the module will be *CDA* (short for *C: Data Structures and Algorithms*) and will consist of the following three files:

- cdap.h (the private header file),
- cda.h (the public header file) and
- cda.c (the principal source file)

```
#include <stdlib.h>

void *PROJ_malloc( size_t size )
{
    void *mem = malloc( size );

    if ( mem == NULL && size > 0 )
        abort();

    return mem;
}
```

Figure 1-6: Cover routine for *malloc*

1.5.1 The Private Header File

The concept of a *private header file* will occupy much of our time later in the course. For our first module it must be present (in this course every module must have a private header file), but otherwise has little significance. It consists entirely of an *include sandwich* (recall the *every* header file must have an include sandwich), and a statement to include the public header file. It is shown in its entirety in **Figure 1-7**.

```
#ifndef CDAP_H    /* begin include sandwich          */
#define CDAP_H    /* second line of include sandwich */

#include <cda.h>  /* include public header file      */

#endif            /* end include sandwich            */
```

Figure 1-7: CDA Module Private Header File *cdap.h*

1.5.2 The Principal Source File

The *principal source file* for the CDA module will contain cover routines for *malloc, calloc*, *realloc* and *free*. The cover routine for malloc resembles closely the cover that we saw in **Section 1.4**, and is shown below:

```
void *CDA_malloc( size_t size )
{
    void *mem = malloc( size );

    if ( mem == NULL && size > 0 )
```

```
            abort();

        return mem;
    }
```

The cover routines for *calloc* and *realloc* will be called *CDA_calloc* and *CDA_realloc*, respectively, and are left as an exercise for the student. The cover routine for *free* is called *CDA_free*, and is shown in **Figure 1-8**. Outside of the cover routines in cda.c, none of the code that we write in this class will *ever* make direct use of the standard memory allocation routines. Instead, they will make use of the CDA covers.

1.5.3 The Public Header File

The CDA module public header file consists of three parts, as discussed below.

```
void CDA_free( void *mem )
{
    if ( mem != NULL )
        free( mem );
}
```

Figure 1-8: CDA_free

Part 1: Common or Convenient Macros

This part of the public header file consists of the declaration of macros for *true* and *false* to help make our code more readable, and macros to encapsulate frequent operations to help make our code more reliable. The declarations are shown in **Figure 1-9**, and discussed in the following paragraphs.

```
#define CDA_TRUE             (1)
#define CDA_FALSE            (0)

#define CDA_ASSERT( exp )    (assert( (exp) ))
#define CDA_CARD( arr )      (sizeof((arr))/sizeof(*(arr)))
#define CDA_NEW( type )      ((type *)CDA_malloc( sizeof(type) ))
#define CDA_NEW_STR( str )   \
    (strcpy( (char *)CDA_malloc( strlen( (str) ) + 1 ), (str) ))
#define CDA_NEW_STR_IF( str ) \
    ((str) == NULL ? NULL : CDA_NEW_STR( (str) ))
```

Figure 1-9: CDA Module Public Macros

- *CDA_TRUE* and *CDA_FALSE* merely help to make our code more readable by giving us symbolic names for Boolean values.
- *CDA_ASSERT*, for now, is a simple cover for the standard library *assert* macro. On many projects it is common to use alternative assert macros that provide more detailed feedback then the standard assert macro. We are not going to do that now; we do, however, want to leave open the possibility of doing it later. So we will write a macro to use in place of the usual assert and use it in all of our projects; later, if we decide to write a new version, all we will have to do is change cda.h and recompile our code in order to take advantage of it. Here is an example of its use:

Instead of:

```
assert( size <= MAXIMUM_SIZE );
```

Use:

```
CDA_ASSERT( size <= MAXIMUM_SIZE );
```

- *CDA_CARD* is simply a generic implementation of the *CARD* macro that we saw in **Section 0**. For example:

```
int inx      = 0;
int iarr[10];
for ( inx = 0 ; inx < CDA_CARD( iarr ) ; ++inx )
    iarr[inx] = -1;
```

- *CDA_NEW* is a macro to encapsulate what we call a *new* operation; that is, the allocation of memory to serve as a data structure instance, or a component of a data structure instance. For example:

  Instead of:

```
ADDRESS_p_t address = CDA_malloc( sizeof(ADDRESS_t) );
```

  Use:

```
ADDRESS_p_t address = CDA_NEW( ADDRESS_t );
```

- *CDA_NEW_STR* and *CDA_NEW_STR_IF* encapsulate the operations needed to make a copy of a string. This is an important activity, because what we call a string in C is merely the address of an array of type *char*. If you try to store such an address in a data structure you leave yourself open to problems because that memory typically belongs to someone else, and it can be modified or deallocated at any time; so, before storing a string in a data structure, you must be certain that it occupies memory that you control. *CDA_NEW_STR* unconditionally makes a copy of a string; *CDA_NEW_STR_IF* evaluates to NULL if the target string is NULL, and to *CDA_NEW_STR* if the string is non-NULL. Here is an example of their use:

  Instead of:

```
if ( string == NULL )
    copy = NULL;
else
{
    copy = CDA_malloc( strlen( string ) + 1 );
    strcpy( copy, string );
}
```

  Use:

```
copy = CDA_NEW_STR_IF( string );
```

Part 2: Common or Convenient Type Declarations

This part of the public header file consists of the declaration of a *Boolean* type to help make our code more readable, plus declarations of types to represent integers of a specific size to help make our code more portable. The representative integer types will be:

- signed and unsigned eight-bit integer: *CDA_INT8_t* and *CDA_UINT8_t*
- signed and unsigned sixteen-bit integer: *CDA_INT16_t* and *CDA_UINT16_t*
- signed and unsigned thirty two-bit integer: *CDA_INT32_t* and *CDA_UINT32_t*

The first three such type declarations are shown in ; the remaining declarations are left as an exercise.

```
typedef int             CDA_BOOL_t;
typedef signed char     CDA_INT8_t;
typedef unsigned char   CDA_UINT8_t;
```

Figure 1-10: CDA Module Public Type Declarations

Part 3: Prototypes for the Public Functions

This part of the public header file consists of the prototypes for the functions in cda.c.

1.6 Activity

Interactively develop a module to encapsulate a timer.

2. Doubly Linked Lists

In this section we will examine one of the most common and versatile data structures in data processing: the doubly linked list. In designing the VAX, Digital Equipment Corporation engineers felt that this type of list was so important that they designed machine-level instructions for manipulating them.

In addition to learning about doubly linked lists, in this lesson you will begin to learn how to formally define data structures, and to encapsulate data structure functionality in modules.

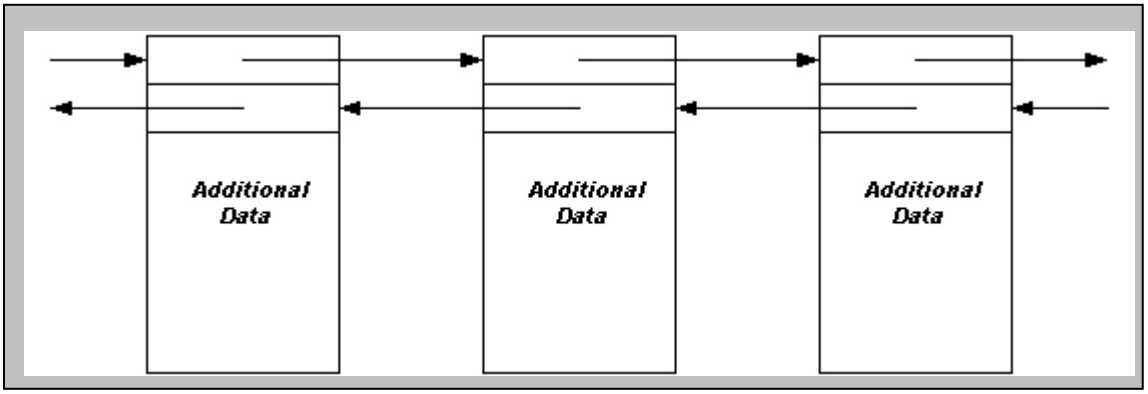
2.1 Objectives

At the conclusion of this section, and with the successful completion of your second project, you will have demonstrated the ability to:

- Formally define the *enqueuable item* and *doubly linked list* structures; and
- Implement a circular, doubly linked list data structure.

2.2 Overview

There are several ways to implement a doubly linked list. The common feature of all such implementations is that a doubly linked list consists of zero or more *enqueuable items*. An enqueuable item consists of, at a minimum, two pointers; when an enqueuable item is enqueued, one pointer provides a *forward link* to the *next* item in the list (if any) and the other a *backward link* to the *previous* item in the list (if any). **Figure 2-1** portrays three



enqueuable items in a list.

Figure 2-1: Enqueuable Items

The specific implementation that we will consider in this course is a doubly linked list that is *anchored* by a queue head, and that is *circular*. The anchor consists of, at a minimum, a forward/backward link pair. The forward link points to the first item in the list, and the backward link points to the last item in the list. The list is circular because the forward link of the last item in the list, and the backward link of the first item in the list point to the anchor. This arrangement is illustrated in **Figure 2-2**.

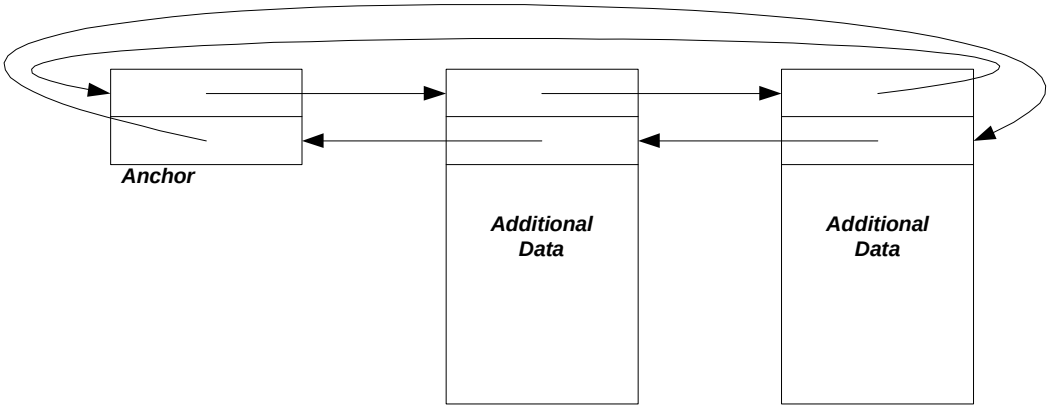


Figure 2-2 Anchored, Circular List

Before we go on, a quick word about the picture shown in **Figure 2-2**. This is the standard way of representing a doubly linked list, but it sometimes leads to confusion among students. In particular, it is easy to interpret the drawing as if the backward link of an item held the address of the backward link of the previous item. This is not so; the forward and backward links of an item always contain the base addresses of the structures it links to. A slightly more realistic representation of a linked list can be seen in **Figure 2-3**; however, this representation is harder to draw and harder to read, so the style of representation used in **Figure 2-2** is the one that we will use for the remainder of this course.

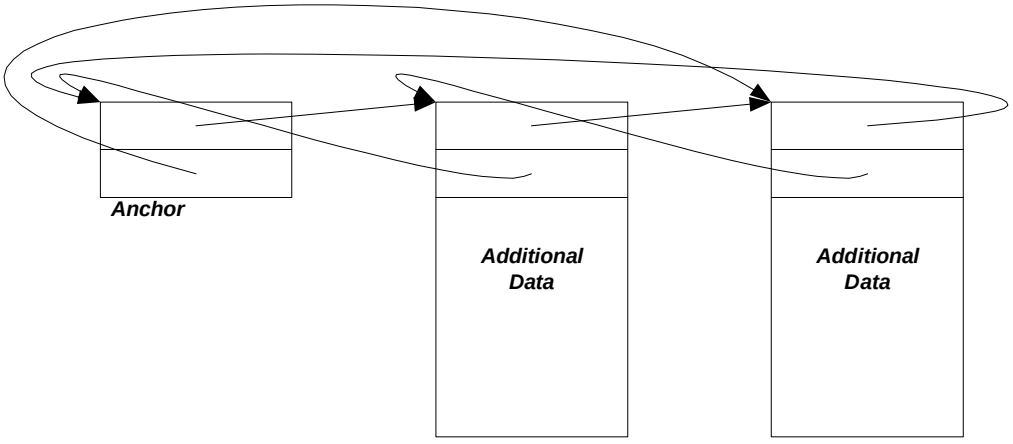


Figure 2-3 Another List Representation

2.3 Definitions

In this section we will discuss formal definitions for the elements of our implementation of a linked list. Specifically, we need to define exactly what we mean by a doubly linked list, an anchor and an enqueuable item; we also have to strictly define the states that these items may occupy.

2.3.1 Enqueuable Item

In our implementation, an enqueuable item is a struct consisting of two pointers of type *pointer to enqueuable item*, followed by a *name* and zero or more bytes of application data. The two pointers, called *flink* for forward link and *blink* for backward link, must occupy the first two fields of the struct, and the name (type char*) must occupy the third. These first three fields are the *static part* of the item. In C, we can't declare a data type representing a variable amount of data, but we can create a data type to represent the static portion as follows:

```
typedef struct enq_item_s
{
    struct enq_item_s *flink;
    struct enq_item_s *blink;
    char              *name;
} ENQ_ITEM_t, *ENQ_ITEM_p_t;
```

Enqueuable items that contain additional data can be declared by creating a type whose first member is a field of type ENQ_ITEM_t. An example is shown in **Figure 2-4**.

```
typedef struct address_s
{
  ENQ_ITEM_t item;
  char       name[31];
  char       address1[31];
  char       address2[31];
  char       city[31];
  char       state[31];
  char       postal_code[21];
} ADDRESS_t, *ADDRESS_p_t;
```

Figure 2-4 Enqueuable Item with Application Data

Now that we know what an enqueuable item looks like, we have to define the *states* that such an item can occupy. In this case there are only two: an enqueuable item may be *enqueued* or *unenqueued*. We define an enqueued item as one whose flink and blink point to other enqueuable items; when an item is unenqueued, its flink and blink point to itself. This is illustrated in **Figure 2-5**.

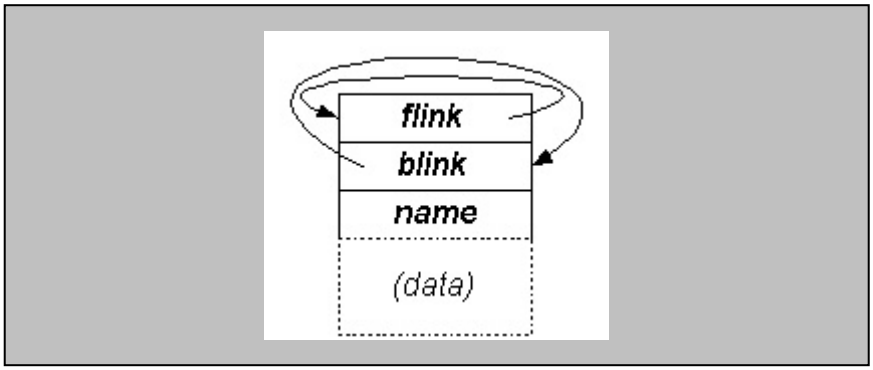


Figure 2-5 Enqueuable Item in the Unenqueued State

2.3.2 Anchor

An *anchor* is a struct consisting of a *flink*, a *blink* and a *name;* in other words, it is an enqueuable item with no application data. The declaration of the anchor type is shown below:

```
typedef ENQ_ITEM_t ENQ_ANCHOR_t, *ENQ_ANCHOR_p_t;
```

2.3.3 Doubly Linked List

We define a doubly linked list as an anchor plus zero or more enqueuable items; a list is always identified by the address of its anchor.

> Note: For the remainder of this course, unqualified use of the word *list* will be synonymous with *doubly linked list*. Unqualified reference to the anchor of a list typically will be interchangeable with a reference to the list itself; for example, "pass the list to subroutine X" is interchangeable with "pass the address of the anchor of the list to subroutine X."

A doubly linked list may occupy two states: *empty* and *nonempty*. A nonempty list contains at least one item; an empty list contains none. When a list is in a nonempty state, the flink and blink of the anchor point to the first and last items in the list, respectively; the blink of the first item and the flink of the last item each point to the anchor. When the list is empty, the flink and blink of the anchor each point to the anchor. In a nonempty list, the first item in the list is referred to as the *list head* or just *head*; the last item in the list is referred to as the *list tail*, or just *tail.* This is illustrated in **Figure 2-6**.

Now that we know how the structure of a doubly linked list is defined, we need to define the operations that may be performed on a list, and the protocols required to perform those operations. Operations, together with the protocols for performing them are called *methods*, and will be discussed in the next section.

2.3.4 Methods

In this class we will define the following operations that may be performed on a doubly linked list, and the elements that belong to a doubly linked list:

- Create a new doubly linked list
- Create a new enqueuable item
- Test whether an item is enqueued
- Test whether a list is empty
- Add an item to the head of a list
- Add an item to the tail of a list
- Add an item after a previously enqueued item
- Add an item before a previously enqueued item
- Dequeue an item from a list
- Dequeue the item at the head of a list

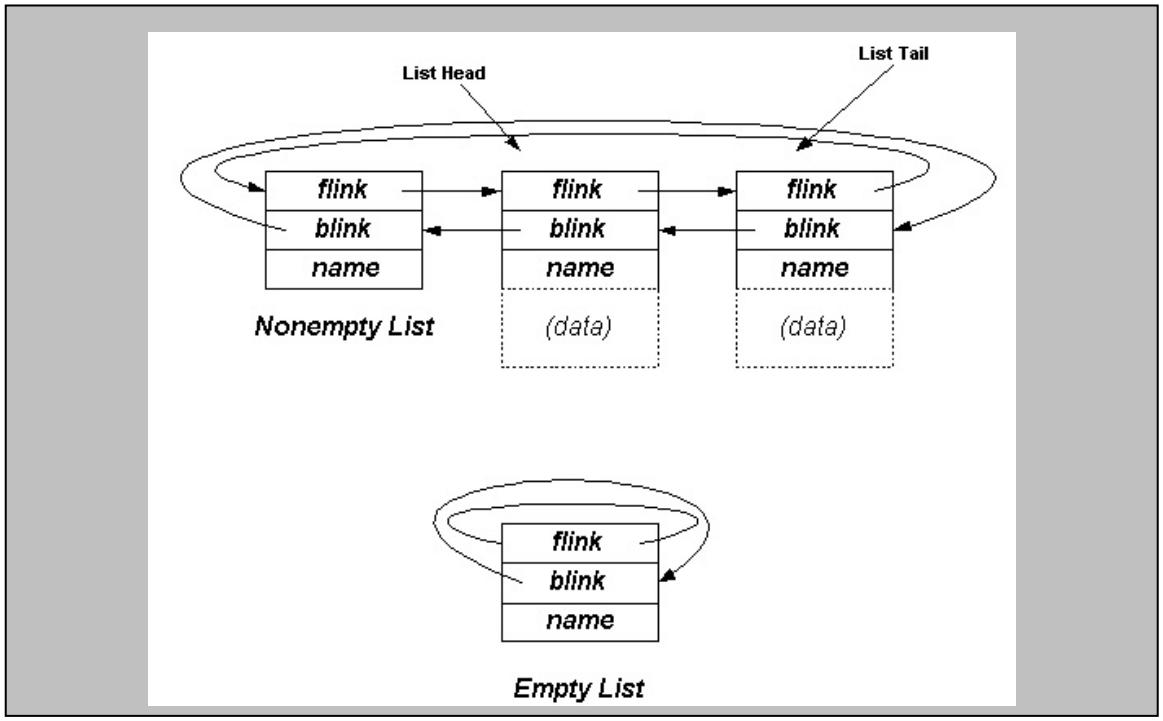


Figure 2-6 List States

- Dequeue the item at the tail of a list
- Get the item at the head of a list (without dequeing it)
- Get the item at the tail of a list
- Given an item, get the following item
- Given an item, get the previous item
- Get the name of a list
- Get the name of an item
- Destroy an item
- Empty a list
- Destroy a list

In the following sections, we will discuss some the details of the above methods. Determining the remaining details, and actually implementing the methods will constitute your second project.

2.3.5 ENQ_create_list: Create a New Doubly linked List

This method will dynamically allocate the space for a new list anchor, and return its address to the caller. The caller is responsible for ultimately freeing the memory by calling ENQ_destroy_list when the list is no longer needed.

```
Synopsis:
    ENQ_ANCHOR_p_t ENQ_create_list( const char *name );
Where:
    name -> the name of the list
```

```
Returns:
    The address of the list
Notes:
   1. The caller is responsible for freeing the memory occupied
      by the list by calling ENQ_destroy_list.
   2. The list name is copied into a private buffer which is
      freed when the list is destroyed.
```

The implementation of this method is straightforward; the only trick is to remember to initialize the queue to an empty state after creating it:

```
ENQ_ANCHOR_p_t ENQ_create_list( const char *name )
{
    ENQ_ANCHOR_p_t list = CDA_NEW( ENQ_ANCHOR_t );

    list->flink = list;
    list->blink = list;
    list->name = CDA_NEW_STR_IF( name );
    return list;
}
```

2.3.6 ENQ_create_item: Create a New Enqueuable Item

Creating a new item is essentially the same as creating a new list; allocate space for the item, then initialize it to the unenqueued state. The only difference is that allowance must be made for the extra application data space. To this end an extra argument must be passed to indicate the entire amount of space to be allocated; **note that this includes the space required for the static portion of the item**. We're also going to add an assertion to verify that the requested size is at least as great as required. When the item is no longer needed, the caller is responsible for freeing the memory it occupies by calling ENQ_destroy_item.

```
Synopsis:
    ENQ_ITEM_p_t ENQ_create_item( const char *name,
                                  size_t     size
                                );
Where:
    name -> the name of the item
    size == size of item required
Returns:
    The address of the item
Notes:
   1. The caller is responsible for freeing the memory occupied
      by the item by calling ENQ_destroy_item.
   2. The item name is copied into a private buffer which is
      freed when the item is destroyed.
```

Here is the implementation of this method:

```
ENQ_ITEM_p_t ENQ_create_item( const char *name, size_t size )
{
    ENQ_ITEM_p_t item = (ENQ_ITEM_p_t)CDA_malloc( size );

    CDA_ASSERT( size >= sizeof(ENQ_ITEM_t) );
    item->flink = item;
```

```
        item->blink = item;
        item->name = CDA_NEW_STR_IF( name );
        return item;
    }
```

2.3.7 ENQ_is_item_enqed: Test Whether an Item is Enqueued

Determining whether an item is enqueued merely requires knowing the definition of the possible states for an item, as discussed above.

```
Synopsis:
    CDA_BOOL_t ENQ_is_item_enqed( ENQ_ITEM_p_t item );
Where:
    item -> item to test
Returns:
    CDA_TRUE if the item is enqueued, CDA_FALSE otherwise
Notes:
    None
```

Here's the method implementation:

```
CDA_BOOL_t ENQ_is_item_enqed( ENQ_ITEM_p_t item )
{
    CDA_BOOL_t rcode =
        (item->flink == item ? CDA_FALSE : CDA_TRUE);

    return rcode;
}
```

2.3.8 ENQ_is_list_empty: Test Whether a List is Empty

As in the previous section, determining whether a list is empty merely requires knowing the definition of the possible states for a list.

```
Synopsis:
    CDA_BOOL_t ENQ_is_list_empty( ENQ_ANCHOR_p_t list );
Where:
    list -> list to test
Returns:
    CDA_TRUE if the list is empty, CDA_FALSE otherwise
Notes:
    None
```

The implementation of this method is left to the student.

2.3.9 ENQ_add_head: Add an Item to the Head of a List

This method inserts an item at the front of a list.

```
Synopsis:
    ENQ_ITEM_p_t ENQ_add_head( ENQ_ANCHOR_p_t list,
                               ENQ_ITEM_p_t   item
                             );
Where:
    list -> list in which to enqueue
    item -> item to enqueue
```

```
Returns:
    address of enqueued item
Notes:
    None
```

The implementation of this method is left to the student. You should use an assertion to verify that the item is not already enqueued before performing the operation:

```
CDA_ASSERT( !ENQ_is_item_enqed( item ) );
```

2.3.10 ENQ_add_tail: Add an Item to the Tail of a List

This operation is nearly identical to ENQ_add_head.

```
Synopsis:
    ENQ_ITEM_p_t ENQ_add_tail( ENQ_ANCHOR_p_t list,
                               ENQ_ITEM_p_t   item
                             );
Where:
    list -> list in which to enqueue
    item -> item to enqueue
Returns:
    address of enqueued item
Notes:
    None
```

The implementation of this method is left to the student. You should use an assertion to verify that the item is not already enqueued before performing the operation.

2.3.11 ENQ_add_after: Add an Item After a Previously Enqueued Item

```
Synopsis:
    ENQ_ITEM_p_t ENQ_add_after( ENQ_ITEM_p_t item,
                                ENQ_ITEM_p_t after
                              );
Where:
    item  -> item to enqueue
    after -> item after which to enqueue
Returns:
    address of enqueued item
Notes:
    None
```

The implementation of this method is left to the student. You should use an assertion to verify that the item to enqueue is not already enqueued.

2.3.12 ENQ_add_before: Add an Item Before a Previously Enqueued Item

```
Synopsis:
    ENQ_ITEM_p_t ENQ_add_before( ENQ_ITEM_p_t item,
                                 ENQ_ITEM_p_t before
                               );
Where:
    item   -> item to enqueue
    before -> item before which to enqueue
```

```
Returns:
    address of enqueued item
Notes:
    None
```

The implementation of this method is left to the student. You should use an assertion to verify that the item to enqueue is not already enqueued.

2.3.13 ENQ_deq_item: Dequeue an Item from a List

This method will remove an item from a list and return its address. Note that, because of our careful definition of an item in an unenqueued state, it is not necessary to make sure that the item is enqueued before dequeing it; the chosen algorithm works equally well on both enqueued and unenqueued items.

```
Synopsis:
    ENQ_ITEM_p_t ENQ_deq_item( ENQ_ITEM_p_t item );
Where:
    item -> item to dequeue
Returns:
    address of dequeued item
Notes:
    None
```

The only trick to performing this operation is to make sure that, once dequeued, the item is set to an unenqueued state.

```
ENQ_ITEM_p_t ENQ_deq_item( ENQ_ITEM_p_t item )
{
    item->blink->flink = item->flink;
    item->flink->blink = item->blink;
    item->flink = item;
    item->blink = item;
    return item;
}
```

2.3.14 ENQ_deq_head: Dequeue the Item at the Head of a List

This method removes the item from the head of a list and returns its address.

```
Synopsis:
    ENQ_ITEM_p_t ENQ_deq_head( ENQ_ANCHOR_p_t list );
Where:
    list -> list from which to dequeue
Returns:
    If queue is nonempty, the address of the dequeued item;
    Otherwise the address of the list
Notes:
    None
```

The implementation of this method is left to the student. Take careful note of the documentation for the return value, and remember to initialize the dequeued item after dequeing it.

2.3.15 ENQ_deq_tail: Dequeue the Item at the Tail of a List

This method removes the item from the tail of a list and returns its address.

```
Synopsis:
    ENQ_ITEM_p_t ENQ_deq_tail( ENQ_ANCHOR_p_t list );
Where:
    list -> list from which to dequeue
Returns:
    If queue is nonempty, the address of the dequeued item;
    Otherwise the address of the list
Notes:
    None
```

The implementation of this method is left to the student. Take careful note of the documentation of the return value, and remember to initialize the dequeued item after dequeing it.

2.3.16 ENQ_GET_HEAD: Get the Address of the Item at the Head of a List

This method returns the address of the item at the head of a list *without* dequeing it. It is so straightforward that many list implementations don't bother with it. In this class, however, we will concentrate on providing all operations on a data structure from within the module that owns the data structure. We will, in this case, follow a middle ground; rather than implement the method as a procedure, we will make it a macro.

> **Note:** Some students are confused by the *synopsis*, below, thinking that it shows the prototype of a *function*, contradicting the above statement that this will be implemented as a *macro*. This is not true. This is merely a standard why of summarizing how a method is used, whether its implementation is as a function, or a function-like macro. See, for example, the documentation for **getc** macro in Chapter 15 of Harbison & Steele.

```
Synopsis:
    ENQ_ITEM_p_t ENQ_GET_HEAD( ENQ_ANCHOR_p_t list );
Where:
    list -> list to interrogate
Returns:
    If queue is nonempty, the address of the first list item;
    Otherwise the address of the list
Notes:
    None
```

Here's the implementation of the method.

```
#define ENQ_GET_HEAD( list ) ((list)->flink)
```

2.3.17 ENQ_GET_TAIL: Get the Address of the Item at the Tail of a List

This method, also a macro, is nearly identical to ENQ_GET_HEAD.

```
Synopsis:
    ENQ_ITEM_p_t ENQ_GET_TAIL( ENQ_ANCHOR_p_t list );
Where:
    list -> list to interrogate
```

```
Returns:
    If queue is nonempty, the address of the last list item;
    Otherwise the address of the list
Notes:
    None
```

The implementation of this method is left to the student. Like ENQ_GET_HEAD it should be implemented as a macro.

2.3.18 ENQ_GET_NEXT: Get the Address of the Item After a Given Item

Given an item, this method, implemented as a macro, returns the address of the next item in the list without dequeing it.

```
Synopsis:
    ENQ_ITEM_p_t ENQ_GET_NEXT( ENQ_ITEM_p_t item );
Where:
    item -> item to interrogate
Returns:
    If there is a next item, the address of the next item;
    Otherwise the address of the list that item belongs to
Notes:
    None
```

The implementation of this method is left to the student. It should be implemented as a macro.

2.3.19 ENQ_GET_PREV: Get the Address of the Item Before a Given Item

Given an item, this macro returns the address of the previous item in the list without dequeing it.

```
Synopsis:
    ENQ_ITEM_p_t ENQ_GET_PREV( ENQ_ITEM_p_t item );
Where:
    item -> item to interrogate
Returns:
    If there is a previous item, the address of the previous item;
    Otherwise the address of the list that item belongs to
Notes:
    None
```

The implementation of this method is left to the student. It should be implemented as a macro.

2.3.20 ENQ_GET_LIST_NAME: Get the Name of a List

This may seem like a trivial operation, but later on if we decide to change the implementation we'll be glad we implemented this as a method. We'll make it a macro. Notice that the return type is a pointer to a constant string; the caller may NOT modify it.

```
Synopsis:
    const char *ENQ_GET_LIST_NAME( ENQ_ANCHOR_p_t list );
Where:
    list -> list to interrogate
```

```
Returns:
    The name of the list
Notes:
    The string representing the list name belongs to the
    implementation; the caller may not modify it.
```

Here's the implementation:

```
#define ENQ_GET_LIST_NAME( list ) \
  ((const char *)((list)->name))
```

2.3.21 ENQ_GET_ITEM_NAME: Get the Name of an Item

This method is nearly identical to ENQ_GET_LIST_NAME. It will be a macro that returns the name of an item as type (const char *).

```
Synopsis:
    const char *ENQ_GET_ITEM_NAME( ENQ_ITEM_p_t item );
Where:
    item -> item to interrogate
Returns:
    The name of the item
Notes:
    The string representing the item name belongs to the
    implementation; the caller may not modify it.
```

The implementation of this method is left to the student. It should be implemented as a macro.

2.3.22 ENQ_destroy_item: Destroy an Item

This method will free the memory associated with an item. If the item is enqueued, it will be dequeued before freeing. Note that the method explicitly return **NULL**.

```
Synopsis:
    ENQ_ITEM_p_t ENQ_destroy_item( ENQ_ITEM_p_t item );
Where:
    item -> item to destroy
Returns:
    NULL
Notes:
    The item to destroy may be enqueued or unenqueued. If
    enqueued, it will be dequeued prior to destruction.
```

There are two things to remember as part of the implementation. First, don't forget to free the item name. Second, thanks to the way that we chose to define an enqueuable item, there is no need to test whether an item is enqueued before we dequeue it.

```
ENQ_ITEM_p_t ENQ_destroy_item( ENQ_ITEM_p_t item )
{
    ENQ_deq_item( item );
    CDA_free( item->name );
    CDA_free( item );

    return NULL;
}
```

2.3.23 ENQ_empty_list: Empty a List

This method will remove all items from a list, and destroy them, leaving the list empty.

```
Synopsis:
    ENQ_ANCHOR_p_t ENQ_empty_list( ENQ_ANCHOR_p_t list );
Where:
    list -> list to empty
Returns:
    The address of the list
Notes:
    All items enqueued in the list will be destroyed.
```

Here is the implementation:

```
ENQ_ANCHOR_p_t ENQ_empty_list( ENQ_ANCHOR_p_t list )
{
    while ( !ENQ_is_list_empty( list ) )
        ENQ_destroy_item( list->flink );

  return list;
}
```

2.3.24 ENQ_destroy_list: Destroy a List

This method will empty a list, then free the memory occupied by the list anchor. Note that it explicitly returns **NULL**.

```
Synopsis:
    ENQ_ANCHOR_p_t ENQ_destroy_list( ENQ_ANCHOR_p_t list );
Where:
    list -> list to destroy
Returns:
    NULL
Notes:
    All items enqueued in the list will be destroyed.
```

The implementation of this method is left to the student. Remember to empty the list and free the list name before freeing the anchor.

2.4 Case Study

Alice has joined a team of programmers working on an accounting system for a restaurant. She has been given the job of writing a module that will temporarily accumulate the total receipts and tips for the restaurant's waiters and waitresses, and then print out the results. The major requirements she was given are listed below:

1. The module must have an initialization method which will be called once, and which should initialize the module's data structures to an empty state.
2. The module must have a shutdown method which will be called once, and which will free any resources allocated for the module.
3. The module must have a method that will accept the name of an employee, the amount of a single check, and the amount of the tip associated with the check. For any employee there are likely to be many checks, hence many calls to this method.

4. The module must have a print method that will print, alphabetically, the name of each employee, their total receipts and tips, and the average amount of each check and tip.

Here is an example of the output of the print method:

```
Brenda
Total receipts: 328.99 (Average: 82.25)
Total tips: 62.00 (Average: 15.50)
Tom
Total receipts: 321.23 (Average: 160.62)
Total tips: 64.00 (Average: 32.00)
Terry
Total receipts: 138.15 (Average: 46.05)
Total tips: 46.00 (Average: 15.34)
```

Alice has decided that she will implement the accumulation method by allocating a bucket for each employee as the employee is "discovered," that is, when the accumulation method has been passed the name of an employee for whom a bucket has not already been allocated. Each time a new receipt is received for a known employee, the amount of the receipt will be accumulated in the existing bucket. She has also decided that she will store each bucket in a list of some kind, in alphabetical order. Here is the pseudocode for the accumulation method, which Alice has decided to call *addReceipt*:

```
addReceipt( name, check, tip )
    for each bucket in list
        if bucket-name == name
            add check to bucket
            add tip to bucket
            increment # receipts in bucket
        else if bucket-name > name
            allocate new-bucket
            add check to new-bucket
            add tip to new-bucket
            set # receipts in new-bucket = 1
            add new-bucket to list before bucket
        else
            next bucket
    if no bucket in list satisfies the above:
        allocate new-bucket
        add check to new-bucket
        add tip to new-bucket
        set # receipts in new-bucket = 1
        add new-bucket to end of list
```

Alice has decided to create and maintain her list using the ENQ module. That means that her remaining three methods will be implemented rather simply, as follows:

1. The initialization method will create the list.
2. The print method will traverse the list from front to back, printing out the employee information found in each bucket.
3. The shutdown method will destroy the list.

In keeping with local project standards, Alice now has to select a name for her module, and create public and private header files for it. She has chosen the name TIPS. Her public and private header files (with comments removed to save space) follow:

```
/* TIPS Private Header File */
#ifndef TIPSP_H
#define TIPSP_H

#include <tips.h>

#endif /* #ifndef TIPSP_H */
```

```
/* TIPS Public Header File */
#ifndef TIPS_H
#define TIPS_H

void TIPS_addReceipt(
    const char *waitress,
    double     check,
    double     tip
);

void TIPS_close(
    void
);

void TIPS_init(
    void
);

void TIPS_printReceipts(
    void
);

#endif /* #ifndef TIPS_H */
```

Alice has only one decision left: how to implement her "bucket." She knows that this will be a data structure with fields for accumulating checks, tips, and check count. Since she has decided to implement her list via the ENQ module, she knows that her bucket will have to be an enqueuable item, as defined by the ENQ module; that is, it will have to have as its first member an ENQ_ITEM_t structure. Alice's implementation of the TIPS source file is shown below.

```
#include <cda.h>
#include <enq.h>
#include <tipsp.h>

#define NOT_FOUND      (0)
#define FOUND_EXACT    (1)
#define FOUND_GREATER  (2)

typedef struct receipts_s
{
    ENQ_ITEM_t item;
    double     checkTotal;
    double     tipTotal;
    int        numReceipts;
} RECEIPTS_t, *RECEIPTS_p_t;

static const char *tipListName = "Tip Queue";

static ENQ_ANCHOR_p_t anchor = NULL;
```

```
void TIPS_init( void )
{
    CDA_ASSERT( anchor == NULL );
    anchor = ENQ_create_list( tipListName );
}

void TIPS_addReceipt( const char *waitperson, double check, double tip )
{
    RECEIPTS_p_t receipts = NULL;
    RECEIPTS_p_t bucket   = NULL;
    int          result   = NOT_FOUND;
    int          compare  = 0;

    CDA_ASSERT( anchor != NULL );

    receipts = (RECEIPTS_p_t)ENQ_GET_HEAD( anchor );
    while ( (result == NOT_FOUND) && ((ENQ_ANCHOR_p_t)receipts != anchor) )
    {
        compare = strcmp( waitperson, ENQ_GET_ITEM_NAME( (ENQ_ITEM_p_t)receipts ));
        if ( compare == 0 )
            result = FOUND_EXACT;
        else if ( compare < 0 )
            result = FOUND_GREATER;
        else
            receipts = (RECEIPTS_p_t)ENQ_GET_NEXT( (ENQ_ITEM_p_t)receipts );
    }

    switch ( result )
    {
        case FOUND_EXACT:
            receipts->checkTotal += check;
            receipts->tipTotal += tip;
            ++receipts->numReceipts;
            break;

        case FOUND_GREATER:
            bucket = (RECEIPTS_p_t)ENQ_create_item( waitperson, sizeof(RECEIPTS_t) );
            bucket->checkTotal = check;
            bucket->tipTotal = tip;
            bucket->numReceipts = 1;
            ENQ_add_before( (ENQ_ITEM_p_t)bucket, (ENQ_ITEM_p_t)receipts );
            break;

        case NOT_FOUND:
            bucket = (RECEIPTS_p_t)ENQ_create_item( waitperson, sizeof(RECEIPTS_t) );
            bucket->checkTotal = check;
            bucket->tipTotal = tip;
            bucket->numReceipts = 1;
            ENQ_add_tail( anchor, (ENQ_ITEM_p_t)bucket );
            break;

        default:
            CDA_ASSERT( CDA_FALSE );
            break;
    }
}

void TIPS_printReceipts( void )
{
    RECEIPTS_p_t receipts = NULL;

    CDA_ASSERT( anchor != NULL );

    receipts = (RECEIPTS_p_t)ENQ_GET_HEAD( anchor );
    while ( receipts != (RECEIPTS_p_t)anchor )
    {
        printf( "%s\n", ENQ_GET_ITEM_NAME( (ENQ_ITEM_p_t)receipts ) );
        printf( "Total receipts: %.2f (Average: %.2f)\n",
                receipts->checkTotal,
                receipts->checkTotal / receipts->numReceipts
```

```
              );
        printf( "Total tips: %.2f (Average: %.2f)\n",
                receipts->tipTotal,
                receipts->tipTotal / receipts->numReceipts
              );
        printf( "\n" );
        receipts = (RECEIPTS_p_t)ENQ_GET_NEXT( (ENQ_ITEM_p_t)receipts );
    }
}

void TIPS_close( void )
{
    CDA_ASSERT( anchor != NULL );
    ENQ_destroy_list( anchor );
    anchor = NULL;
}
```

2.5 Activity

Discuss the concept of *subclassing*, and ways of implementing it in C.

3. Sorting

In this section we will discuss sorting algorithms in general, and three sorting algorithms in detail: *selection sort, bubble sort* and *mergesort*.

3.1 Objectives

At the conclusion of this section, and with the successful completion of your third project, you will have demonstrated the ability to:

- Define the differences between the *selection sort, bubble sort* and *mergesort* sorting algorithms; and
- Implement a traditional mergesort algorithm.

3.2 Overview

The effort to bring order to the vast amounts of data that computers collect is reflective of humankind's eons long effort to organize and catalog information. The two main reasons to sort data are:

- To prepare organized reports; and
- To pre-process data to reduce the time and/or complexity of a *second pass* analysis process.

The main problem with sorts is that they tend to be time consuming. There have been many clever optimized sorting algorithms developed, but they tend to introduce so much coding complexity that they are hard to understand and implement. It would be nice if these optimized algorithms could be packaged as general utilities and used by developers without having to understand their details, and sometimes they are; but there are two reasons why this isn't always practical:

- The most efficient optimizations usually take into account detailed knowledge of the data being sorted. For example, sorting the results of a chemical analysis might take into account expectations about the distribution of data based on previous experience.
- Some knowledge of the format of the data structures being sorted is required by the implementation. For example, the excellent implementation of quick sort in the C Standard Library function *qsort* requires that data be organized in an array, therefore it cannot be used to sort a linked list.

We will now discuss the details of several common sorting techniques. As a project you will implement the mergesort algorithm as a mechanism to sort generalized arrays.

3.3 Bubble Sort

The bubble sort algorithm moves sequentially through a list of data structures dividing it into a sorted part, and an unsorted part. A bubble sort starts at end of the list, compares adjacent elements, and swaps them if the right-hand element (the element later in the list) is less than the left-hand element (the element with earlier in the list). With successive

passes through the list, each member *percolates* or *bubbles* to its ordered position at the beginning of the list. The pseudocode looks like this:

```
numElements = number of structures to be sorted
for ( inx = 0 ; inx < numElements - 1 ; ++inx )
    for ( jnx = numElements - 1 ; jnx != inx ; --jnx )
        if ( element( jnx ) < element( jnx - 1 ) )
            swap( element( jnx ), element( jnx - 1 ) )
```

3.4 Select Sort

A selection sort looks a lot like a bubble sort. It moves iteratively through a list of data structures, dividing the list into a sorted and an unsorted part. The pseudocode looks like this:

```
numElements = number of structures to be sorted
for ( inx = 0 ; inx < numElements - 1 ; ++inx )
  least = inx
  for ( jnx = inx + 1 ; jnx < numElements ; ++jnx )
    if ( element( least ) > element( jnx ) )
      least = jnx
  swap( element( least ), element( inx ) )
```

The big difference between a select sort and a bubble sort is the efficiency it introduces by reducing the number of swaps that must be performed on each pass through the list. As you can see, instead of performing a swap each time it determines that an element is out of order, it merely keeps track of the position of the smallest element it has found so far; then, at the end of each iteration, it performs exactly one swap, installing the smallest element in the correct position in the list.

3.5 Mergesort

The main idea behind the mergesort algorithm is to recursively divide a data structure in half, sort each half independently, and then merge the results. Since the data structure needs to be split, this algorithm works best with well-organized, contiguous structures such as arrays. To mergesort an array, divide the array in half, and independently sort the two halves. When each half has been sorted, merge the results into a temporary buffer, then copy the contents of the buffer back to the original array. Here is the pseudocode for sorting an array:

```
mergesort( array, numElements )
  if ( numElements > 1 )
    lowHalf = numElements / 2
    highHalf = numElements - lowHalf
    array2 = array + lowHalf
    mergesort( array, lowHalf )
    mergesort( array2, highHalf )

    inx = jnx = knx = 0
    while ( inx < lowHalf && jnx < highHalf )
      if ( array[inx] < array2[jnx] )
        tempArray[knx++] = array[inx++]
      else
        tempArray[knx++] = array2[jnx++]
```

```
    while ( inx < lowHalf )
      tempArray[knx++] = array[inx++]
    while ( jnx < highHalf )
      tempArray[knx++] = array2[jnx++]

    for ( inx = 0 ; inx < numElements ; ++inx )
      array[inx] = tempArray[inx]
```

3.6 A Mergesort Implementation in C

We will now discuss the process of converting the above mergesort pseudocode into an actual implementation in C. Completing the process will be the subject of your next project. The mergesort algorithm will be encapsulated in a single function, and our first job will be to decide on the function's return type, and the type of the function's parameters, then we will have four technical problems to overcome.

3.6.1 The Mergesort Function's Footprint

We're not yet ready to say what all the function's parameters are, but we can say that the return type will be *void*, and the first two parameters will be a pointer to the array, and the number of elements in the array. Like *qsort*, we want to be able to sort an array of any type, so we will want our array pointer to be type *void**; and in keeping with common practice we will make the *number-of-elements* parameter type *size_t*. Here is what we have so far:

```
void mergesort( void   *array,
                size_t num_elements,
                (other parameters)
              );
```

3.6.2 The Pointer Arithmetic Problem

This line of pseudocode illustrates the first we have to overcome:

```
array2 = array + lowHalf
```

The problem with the above code is that it requires the compiler to do pointer arithmetic; and, in order to do pointer arithmetic the compiler must know the size of the element referenced by the pointer. And since the input to our mergesort function can be an array of any data type, there is no way for the compiler to intrinsically determine the size of an element in the array. Specifically, recall from your past classes that you cannot do pointer arithmetic with a void pointer. To solve this problem we will have to do the pointer arithmetic ourselves; this requires us to know the size of an element in the array, and the only way to know that is if the caller tells us, so we will have to add another parameter to the mergesort function's parameter list:

```
void mergesort( void   *array,
                size_t num_elements,
                size_t element_size,
                (other parameters)
              );
```

Now that we know the size of an element in the array, consider this:

- The C data type used to represent a single byte is *char*; and

- If we have a pointer to the first byte of one element in the array, and we increase the value of the pointer by the size of an element, we will have a new pointer to the next element of the array.

Now we can solve pointer arithmetic problems such as *array2 = array + lowHalf* with code like this:

```
typedef unsigned char BYTE_t;
void mergesort( void   *array,
                size_t num_elements,
                size_t element_size,
                (other parameters)
              )
{
    if( num_elements > 1 )
    {
        size_t lowHalf     = num_elements / 2;
        size_t highHalf    = num_elements - lowHalf;
        BYTE_t *array1     = array;
        BYTE_t *array2     = array1 + lowHalf * element_size;
        . . .
```

3.6.3 The Assignment Problem

This line of pseudocode illustrates the second problem to be overcome:

```
tempArray[knx++] = array[inx++]
```

First, recall that indexing into an array requires pointer arithmetic, which, as we discussed above, is a problem for the compiler because it doesn't intrinsically know the size of an element in one of our arrays. Second, note that performing an assignment like that suggested by the above pseudocode requires copying the all the bytes from one element of one array into another array; and, since the compiler doesn't know how many bytes that is, it can't perform the copy for us. The solution is to use the *elementSize* parameter to do the following:

- Locate the address of the element in the source array that we want to copy;
- Locate the address of the element in the target array that we want to copy to; and
- Use *memcpy* to perform the copy.

Here, for example, is one *possible* way to translate the second while loop in the mergesort pseudocode into C code:

```
while ( inx < (int)lowHalf )
{
    memcpy( tempArray + knx * element_size,
            array1 + inx * element_size,
            element_size
          );
    ++inx;
    ++knx;
}
```

3.6.4 The Comparison Problem

The next problem we want to talk about concerns the comparison portion of the algorithm as represented by this line of pseudocode:

```
if ( array[inx] < array2[jnx] )
```

You should already recognize that we have yet another pointer arithmetic problem here. But, in addition, we have a problem of *interpretation*. Specifically: how can our implementation interpret the data in the arrays in a way that allows us to determine which element is less than the other? The answer is: we can't. The only person who can tell us how to interpret the data is the programmer who called our mergesort function to begin with. We solve the problem the same way the ANSI library function qsort does: we ask the caller to pass the address of a function that knows how to interpret the data in the array, then we call the function to perform the comparison. The comparison function that the user writes will take as arguments the addresses of two elements which mergesort will pass as type *void**. The comparison function will cast the pointer to a type appropriate to the type of the array, perform the comparison, and return one of the following values:

- -1 if the first element is less than the second;
- 1 if the first element is greater than the second; and
- 0 if the two elements are equal.

Here is what the final function declaration looks like after adding the comparison function parameter:

```
typedef int CMP_PROC_t( const void*, const void * );
typedef CMP_PROC_t *CMP_PROC_p_t;
typedef unsigned char BYTE_t;
void mergesort( void         *array,
                size_t       num_elements,
                size_t       element_size,
                CMP_PROC_p_t cmp_proc
              );
```

And here is one *possible* way to implement the first while loop from the mergesort pseudocode:

```
while ( inx < (int)lowHalf && jnx < (int)highHalf )
{
    if ( cmp_proc( array1 + inx * element_size,
                   array2 + jnx * element_size
                 ) < 0
       )
    {
        memcpy( tempArray + knx * element_size,
                array1 + inx * element_size,
                element_size
              );
        inx++;
        knx++;
    }
    else
    {
        /* to be completed by the student */
    }
}
```

3.6.5 The Temporary Array

The last problem we need to solve is the presence of the temporary array: where did it come from? We'll have to allocate it dynamically; and, of course, free it before exiting. And we'll use the cover routines from our CDA module:

```
tempArray = CDA_malloc( num_elements * element_size );
. . .
CDA_free( temp_array );
```

4. Modules

In this section we will discuss how projects are modularized. We will concentrate on the way in which individual programs are organized into *source code modules*, including public, private and local data structures and methods. You will be introduced to common industry conventions for naming and controlling the modules and their constituent parts.

4.1 Objectives

At the conclusion of this section, and with the successful completion of your third project, you will have demonstrated the ability to:

- Organize a program into modules;
- Apply naming and access control conventions to modules; and
- Identify the difference between public, private and local data structures and methods.

4.2 Overview

The successful completion of a large, complex project is almost always accomplished by breaking the project into smaller, more easily digested pieces. This process, which Douglas Hoffstadter so aptly described as "chunking," is formally called *modularization*, and some of the pieces that result from the process are often referred to as *modules*.

The process of modularization begins at a very high level. **Figure 4-1** shows how a theoretical General Administration System is broken into three executable processes for encapsulating general ledger, accounts payable/receivable and inventory control functionality. The general ledger process is broken into chunks of functionality representing general ledger utilities (GL), user interface (UI), database (DB) and error (ERR) processing. The error chunk is further broken into sets of source files for performing signal processing, stack dumping and error reporting. The chunking process continues from there, as the source files declare data structures, subroutines, sub-subroutines, etc.

For our purposes, we will define a *C Source Module* as beginning at the next to last level of the tree, with GL, UI, DB and ERR each representing a separate module. In the next section we will discuss the components of the C source module.

4.3 C Source Module Components

A C source module (from now on we'll just say *module* most of the time) consists of three classes of data: *public data* for use outside the module, *private data* for use inside the module and *local data* for using within a source file component of a module. These three classes of data are discussed below.

4.3.1 Public Data

A module is typically intended as a building block for an application. This means that there must be some way for other modules to pass data in, and get results out. To accomplish this, the module publishes a set of *public* data declarations and methods; that

is, declarations and methods that can be used or called from outside the module. These public declarations and methods are often called an *application programmer's interface*, or *API* for short.

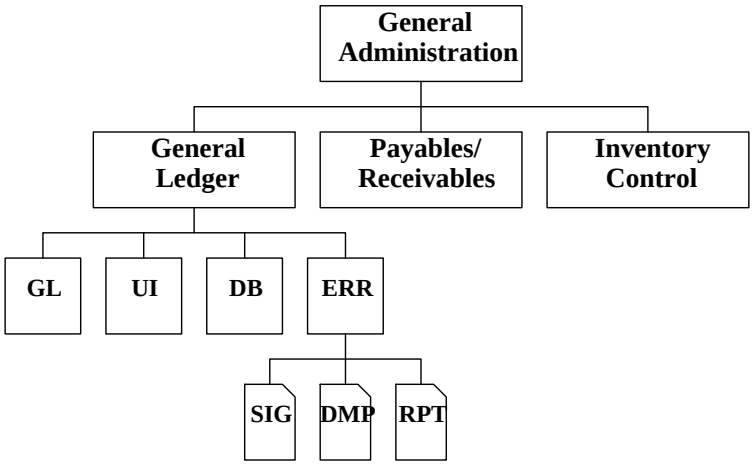


Figure 4-1 Chunking

The mechanism for publishing public data is the module's *public header file*. A module typically only has one public header file, which contains prototypes for all public methods, and declarations of data structures, data types and macros required to interact with those methods. To use an example from the C Standard Library, we might say that the *string handling module* is represented by the public header file *string.h*.

It is very important to do a thorough job of defining a public API before beginning implementation. Other programmers, sometimes hundreds of them, working outside your module are going to be using your API, and if you decide to change it in mid-development you will have an impact on each of them.

4.3.2 Private Data

Not all modules have private data. Those that do are constructed from multiple source files, or are prepared for the eventuality of adding additional source files. When this is the case, there is often a need to share data declarations among the module's source files, declarations that you don't wish to share with the rest of the project. Likewise, there may be a need for one source file to make a call to a function in another source file, which should not be available to the project at large. Declarations and methods shared within, but not without a module's source files are called *private*.

Private declarations and methods are desirable; they represent a form of *data hiding*. When data is hidden from users of your API you can change it without affecting anyone but the other programmers working on your module. So, for example, if you discover a clever way to optimize a private data structure after beginning development you can deploy it with relatively little impact on the overall project.

Private data declarations and methods are published via the module's *private header file*. On a properly managed project, it is understood that no source file in a module ever includes another module's private header file (except for *friends* of the module, but that's a topic we won't be covering).

4.3.3 Local Data

Within a single source file there are almost always subroutines that exist for use only by other functions in the file. Likewise, a source file will often declare data types and global state variables solely for local use. These data and subroutines are referred to as *local* or *static*. To force functions and global variables to be local, you must explicitly declare them to be static; other declarations, including typedefs and macros, are local by default.

Local declarations should never be placed in a header file; local declarations, including prototypes of local functions, should always be placed at the top of the source file that uses them.

4.4 Review: Scope

This is a good time to stop and review the concept of *scope* for an *identifier*. The scope of an identifier represents the extent of its visibility. A function or global variable with external scope can be seen throughout the entire program; functions and global variables have external scope by default. Functions and global variables declared static have their scope restricted to a single source file. Macros, typedef names and other declarations, such as *enum* and *struct*, are always restricted in scope to a single source file. Variables and prototypes declared inside a compound statement are limited in scope to that compound statement.

4.5 A Bit about Header Files

Before we go one we should also have a brief review of header files.

Header files are often referred to as *include* files, but this is misleading. There are files that may be included that are not header files; other .c files, for example.

By convention, the name of a header file always ends in ".h," and a header file never includes *defining declarations* (that is, declarations that result in memory allocations). These declarations are valid in a header file because they require no memory allocation:

```
typedef int CDA_BOOL_t;
int abs( int );
extern int errNum;
```

But these declarations *do* require memory allocation, and are not allowed in a header file:

```
int errNum = 0;

int abs( int val )
{
    return val < 0 ? -val : val;
}
```

Finally, every header file should contain an *include sandwich*.

4.6 Module Conventions

Clearly established conventions for writing modules can help to better organize the data on a project, and to prevent conflicts and bugs. These are some of the problems that a simple set of conventions can help to avoid:

- Which module does printFractalData belong to?
- What's the name of the header file that declares ADDRESS_DATA_t?
- I need to look at the source file for dumpStack; what file is it in?
- Is dumpCurrRegisters a private or public method?
- I named one of my public functions *processErr* and so did George over in the data base group; which one of us has to change?
- I declared a prototype for a function in the UI module with two arguments, and last week they changed it to need three arguments, and I was up all might figuring that out!

The following conventions are similar to those adopted by many large, successful projects. You are expected to follow them in the course of preparing your project source code.

1. Every module is assigned a short, unique name. For example, ENQ for the doubly linked list module, and SRT for the general sorting module.
2. Every module will name its public header file using the module name, followed by .h. For example, enq.h and srt.h will be the public header files for the ENQ and SRT modules, respectively.
3. Every module will name its private header file using the module name followed by p.h. For example, enqp.h and srtp.h.
4. Every module will name its principal source file using the module name. If the module requires additional source files, they will be named using the module name followed by an underscore as a prefix. For example, the ERR module may contain source files err.c and err_dump.c.
5. The name of a public identifier declared by a module will always begin with the module name in upper case followed by a single underscore. For example, ENQ_ITEM_t and ENQ_add_head.
6. The name of a private identifier declared by a module will always begin with the module name in upper case followed by two underscores. For example, ERR__DATA_p_t and ERR__default_abt_handler.
7. The name of a local identifier will always begin with a lower-case character, and will NOT identify the module to which it belongs.
8. A source file will never explicitly declare public or private data; the only valid means to obtain such a declaration is to include the public or private header file that publishes it.
9. Local data declarations will never appear in a header file; they always appear in the source file that requires them.
10. A source file within a module will always include the module's public and private header files. Specifically in this class, the module's private header file will include the public header file, and a source file within the module will include just the module's private header file. This requires all modules to have a private header file even if one isn't strictly needed.

5. Abstract Data Types

In this section we will define the term *abstract data type* or *ADT*, and show you how it relates to our concept of a *module*. We will close the section with a definition of the *list ADT*.

5.1 Objectives

At the conclusion of this section, and with the successful completion of your third project, you will have demonstrated the ability to:

- Define the term *abstract data type*;
- Describe the *list ADT;* and
- Create a module that implements an abstract data type.

5.2 Overview

An *abstract data type* is a set of values and associated operations that may be performed on those values. The classic example of an abstract data type is the set of integers, and associated operations that may be performed on integers, such as addition, subtraction, multiplication, etc. Another example of the abstract data type is the *array*, in conjunction with the *array operator*, []. In C, operations may be implemented using *built-in operators* or *methods*. **Table 1** shows, in part, how the operations for the abstract data type *int* are implemented.

Operation	Built-in Operator	Method
Addition	`+`	
Subtraction	`-`	
Multiplication	`*`	
Division	`/`	
Modulus	`%`	
Increment	`++`	
Decrement	`--`	
Absolute Value		`abs()`
nth Power		`pow()`

nth Root		`pow()`

Table 1 The Integer ADT

An operation performed on one or more valid members of the set of values must yield another member of the set; in the set *int*, for example, 3 + 5 yields 8. When an operation is attempted on a member that is not a member of the set; or when the result of an operation falls outside of the set; an *exception* is said to occur. For example, in the context of the *int* ADT, the attempt to compute the square root of -1 leads to an exception. When an exception occurs, the abstract data type implementation should *handle the exception*. Exception handling is discussed in **Section 5.3**, below.

To implement an abstract data type in C that is not provided by the compiler or standard library, the programmer:

1. Carefully defines the set of values that are to be associated with the ADT;
2. Fully defines the operations that may be performed within the context of the ADT;
3. Declares a set of data structures to represent the ADT's legal values; and
4. Implements individual methods to perform the defined operations.

As we have defined it in this class, a module is an excellent vehicle for representing an ADT; two examples are our implementation of stacks and hash tables.

5.3 Exception Handling

An exception occurs when an abstract data type's method is handed invalid input, or when the operation performed by the method yields invalid data. When an exception occurs, the method must *handle the exception.* The three most common ways of handling an exception are:

- Ignore the exception.
  This is what C (usually) does when an attempt is made to access an element outside an array, or when two large integers are multiplied together yielding an *integer overflow.*
- Return an error value.
  This is what the C standard library function **fgets** does when you try to read past the end of a file.
- Throw the exception.
  This is what C does when you attempt to divide by zero; it's also what our cover routines for the **malloc** family of functions do when an attempt to allocate memory fails.

Ignoring an exception can be dangerous. However it's not always a bad strategy, particularly when a method is buried deep in an implementation, and relies on code that you wrote to pass it only valid data. Performing validation tests in this case can be inefficient, and can increase the complexity of your code, leading to the possibility that the validation will actually introduce a flaw. A good compromise is often to employ *assertions,* which can be used to validate input during testing, and deactivated in

production. This is how the ENQ module's add-head method deals with an attempt to add an item that is already enqueued.

Returning an error value is often the best way to handle an exception, but has its drawbacks. Many times an error return is so rare, and so devastating that it is hard to test; the classic example of this is an error return from **malloc**. Often an inexperienced (or careless) programmer has neglected to check for an error return. In this case, the program failure, when it finally occurs, may be many statements after the actual flaw, and can be very difficult to debug. In cases such as these, the method that detects the exception may be better off *throwing* it.

Throwing the exception involves raising a signal. One way to do this is to call the standard library routine *abort*, which, in most implementations, raises SIGABRT. Any signal can be raised by calling the standard library function *raise*. In many ways throwing an exception in a method is the best way to handle rare errors; if a user of the method really needs to know that an error occurred, he or she can define a *signal handler* to trap the signal. **Figure 5-1** shows an example of a routine that locks a system resource, and then calls a method that may throw an exception; if the exception is thrown, the user traps it, unlocks the system resource, and then re-raises the signal. This is also called *catching the exception.*

```
#include <signal.h>
typedef void SIG_PROC_t( int sig );
typedef SIG_PROC_t *SIG_PROC_p_t;

static SIG_PROC_t trap_sigint;

static SIG_PROC_p_t old_sigint = SIG_DFL;

    . . .
    if ( (old_sigint = signal( SIGINT, trap_sigint )) == SIG_ERR )
        abort();
    SYS_lock_res( SYS_RESOURCE_OPEN );
    queue_id = SYS_open_queue( "PROCESS_STREAM" );
    SYS_unlock_res( SYS_RESOURCE_OPEN );
    if ( signal( SIGINT, old_sigint ) == SIG_ERR )
        abort();
    . . .

static void trap_sigint( int sig )
{
    SYS_unlock_res( SYS_RESOURCE_OPEN );
    if ( signal( sig, old_sigint ) == SIG_ERR )
        abort();
    raise( sig );
}
```

Figure 5-1 Catching an Exception

A more sophisticated routine might attempt to recover from the thrown exception by saving the system state using **setjmp**, then executing a **longjmp** to restore the system state. This mechanism should only be used when recovery from an exception is absolutely crucial, and the programmer knows exactly how to use it.

5.4 Classes of ADTs

Your textbook divides abstract data types into three classes:

1. Atomic Data Types
   These are data types that are usually considered indivisible, for example the type *int*. In C, most atomic data types are represented by the built-in data types.
2. Fixed-Aggregate Data Types
   These are data types composed of components of a fixed size. The classic example of a fixed-aggregate data type in C is the *complex number* data type (see below).
3. Variable-Aggregate Data Types
   These data types are composed of components of varying size. The most common example in C is the *array* data type. Our doubly linked lists and hash tables are also variable-aggregate data types.

5.4.1 The Complex Number ADT

As an example of a fixed-aggregate data type, consider the complex numbers. In mathematics, a complex number can be expressed as the sum of its real and imaginary parts. A typical representation of a complex number may be defined as:

$a + ib$

where *i* is the square root of -1. The addition of two complex numbers, $(a_1 + ib_1) + (a_2 + ib_2)$ is defined as:

$(a_1 + a_2) + i(b_1 + b_2)$

To implement this functionality in C, we might declare the data type and method shown in **Figure 5-2**.

```
typedef struct cpx_num_s
{
    double real;
    double imaginary;
} CPX_NUM_t, *CPX_NUM_p_t;

CPX_NUM_p_t CPX_compute_sum( CPX_NUM_p_t cpx1, CPX_NUM_p_t cpx2 )
{
    CPX_NUM_p_t cpx_sum = CDA_NEW( CPX_NUM_t );

    cpx_sum->real = cpx1->real + cpx2->real;
    cpx_sum->imaginary = cpx1->imaginary + cpx2->imaginary;
    return cpx_sum;
}
```

Figure 5-2 Complex Number ADT Addition

Taken in conjunction with the declaration of CPX_NUM_t and CPX_NUM_p_t, above, the complex number abstract data type could be implemented using the set of methods shown in **Figure 5-3**.

```
CPX_NUM_p_t
CPX_compute_diff( CPX_NUM_p_t cpx1, CPX_NUM_p_t cpx2 );
/* returns (cpx1 - cpx 2) */

CPX_NUM_p_t
CPX_compute_neg( CPX_NUM_p_t cpx )
/* returns -cpx */

CPX_NUM_p_t
CPX_compute_prod( CPX_NUM_p_t cpx1, CPX_NUM_p_t cpx2 )
/* returns (cpx1 * cpx 2) */

CPX_NUM_p_t
CPX_compute_sum( CPX_NUM_p_t cpx1, CPX_NUM_p_t cpx2 );
/* returns (cpx1 + cpx2) */

CPX_NUM_p_t
CPX_compute_quot( CPX_NUM_p_t cpx1, CPX_NUM_p_t cpx2 );
/* returns (cpx1 / cpx 2) */
```

Figure 5-3 Complex Number ADT Methods

5.4.2 The List ADT

As an example of an ADT consider the definition of the list ADT provided by your textbook. To paraphrase:

- A list is a sequence of some type *T*.

Kruse defines the following operations that may be performed on a list:

- Create the list;
- Determine whether the list is empty;
- Determine whether the list is full;
- Find the size of the list;
- Add a new entry to the end of the list;
- Traverse the list (performing some operation at each node); and
- Clear the list

To the list of operations, I am going to add the following:

- Destroy the list.

Note that the definition of the ADT says *nothing* about how the list is to be implemented. I can imagine implementing the list as an array, which would be relatively efficient, or as a linked list utilizing our ENQ module, which would be less efficient but more flexible. Let's go ahead and design the public interface for a list ADT, then give a couple of examples of alternative implementation strategies.

First, in accordance with our module naming standards, we need a name for the module; I have chosen *LIST*. That means our module will consist (at a minimum) of *list.c*, *list.h* and *listp.h*. When the user creates a list, she will tell us the size of an entry in the list (note how different this is from our ENQ module, where entries can be different sizes; in a *list*,

the entries are all the same type, hence the same size). She will also give us a *hint* about the maximum size of the list. Why is this called a "hint"? Because the implementation may choose to force the maximum size to a higher, more convenient number, or to ignore it altogether. In this specific example, if we implement the list as an array we will need the maximum size, but if we use the ENQ module we will ignore it. Once the list is successfully created and initialized we will return to the user an *ID* that she can subsequently use to access the list. We want this ID to be an *opaque type*, that is, something that identifies the list, but does not allow the user access to the internals. The ID will most likely be a pointer to a control structure that is declared in our private header file. Often in C we use the type *void** for such opaque data types; doing so, however, results in *weak typing*; that's because your compiler will allow you to silently assign to a void pointer almost any other pointer type. In order to achieve *strong typing* we are going to assume that the control structure is type *struct list__control_s*, and then we can declare the public list ID as follows:

```
typedef struct list__control_s *LIST_ID_t;
```

Note that, from the perspective of the public API this is what, in C, we refer to as an *incomplete declaration*. From examining the public header file we can conclude that a list ID is a pointer to a particular kind of struct, but we have no idea what members are contained in the struct. Since we don't know what the members are we can't access them. To be truly opaque the user shouldn't even assume that a list ID is a pointer type; and, in fact, we should be free to change our minds next week and make the ID an int. So to be really flexible, we should provide the user with a value that she can use to initialize a list ID that isn't yet assigned to a list. We will call this value LIST_NULL_ID, and declare it like this:

```
#define LIST_NULL_ID    (NULL)
```

With these two public declarations, a user can declare and initialize a list ID variable this way:

```
LIST_ID_t list = LIST_NULL_ID;
```

What if we do change our minds next week and decide to make a list ID an int? Then we will just change LIST_NULL_ID to something appropriate, such as –1. The user will have to recompile her code, but will not have to change any of it.

Traversal and destruction of a list pose special problems because they require manipulation or disposal of data owned by the user. To state the problem a little differently:

- When we traverse the list we will touch each entry and do something with the data; *but do what?*
- When we destroy the list, the data in each entry may need to be disposed of; *but how?*

Only the user can answer these questions. So here's what we are going to do: when the user calls the traversal method she will pass the address of a function, called a *traversal proc*, which knows how to manipulate the data at each entry. And when she calls the destroy method she will call pass the address of a function, called a *destroy proc* that knows how to dispose of the data in an entry. Such functions are called *callbacks* because

the implementation uses them to *call back* to the user's code in order accomplish some part of its operation. We will declare the types of these functions as follows:

```
typedef void LIST_DESTROY_PROC_t( void * );
typedef LIST_DESTROY_PROC_t *LIST_DESTROY_PROC_p_t;

typedef void LIST_TRAVERSAL_PROC_t( void * );
typedef LIST_TRAVERSAL_PROC_t *LIST_TRAVERSAL_PROC_p_t;
```

Keeping the above declarations in mind, we can define the public interface as follows.

5.4.2.1 LIST_create_list

This method will create an empty list and return to the user a value that identifies the list.

```
Synopsis:
   LIST_ID_t LIST_create_list( size_t     max_list_size,
                               size_t     entry_size,
                               const char *name
                             )
Where:
   max_list_size == a hint about the maximum size of the list
   entry_size    == the size of an entry in the list
   name          -> the name of the list
Returns:
   The list ID
Exceptions:
   Throws SIGABRT if the list can't be created.
Notes:
   3. The caller is responsible for freeing the memory occupied
      by the list by calling LIST_destroy_list.
   4. Following creation, the list is guaranteed to hold at least
      max_list_size entries; it may be able to hold more. See
      also LIST_add_entry.
```

5.4.2.2 LIST_add_entry

This method will add an entry to the end of the list.

```
Synopsis:
   LIST_ID_t LIST_add_entry( LIST_ID_t list, const void *data )
Where:
   list  == ID of a previously created list
   data  -> data to be appended to list tail
Returns:
   data
Exceptions:
   Throws SIGABRT if the new entry can't be created.
Notes:
   1. The data argument must point to a block of memory equal in
      size to the entry size as specified in LIST_create_list. A
      new entry is created for the list and the data is COPIED
      INTO IT.
```

5.4.2.3 LIST_traverse_list

This method will traverse the list in order, calling the user's traversal proc at each node.

```
Synopsis:
    LIST_ID_t LIST_traverse_list(
        LIST_ID_t               list,
        LIST_TRAVERSAL_PROC_p_t traversal_proc
                                )
Where:
    list           == ID of a previously created list
    traversal_proc -> function to call for each node
Returns:
    list
Exceptions:
    None
Notes:
   1. For consistency with other modules and methods, the
      traversal proc may be NULL.
```

5.4.2.4 LIST_is_list_empty

This method returns a Boolean value indicating whether a list is empty.

```
Synopsis:
    CDA_BOOL_t LIST_is_list_empty( LIST_ID_t list )
Where:
    list == ID of a previously created list
Returns:
    CDA_TRUE if list is empty, CDA_FALSE otherwise
Exceptions:
    None
Notes:
    None
```

5.4.2.5 LIST_is_list_full

This method returns a Boolean value indicating whether a list is full.

```
Synopsis:
    CDA_BOOL_t LIST_is_list_full( LIST_ID_t list )
Where:
    list == ID of a previously created list
Returns:
    CDA_TRUE if list is full, CDA_FALSE otherwise
Exceptions:
    None
Notes:
    None
```

5.4.2.6 LIST_get_list_size

This method returns the number of elements in the list.

```
Synopsis:
    size_t LIST_is_list_full( LIST_ID_t list )
Where:
    list == ID of a previously created list
Returns:
    The size of the list
Exceptions:
    None
Notes:
    None
```

5.4.2.7 LIST_clear_list

This method will return a list to its initial, empty state, destroying each node in the process. If the user specifies a destroy proc, it will be called for each node in the list prior to destroying the node.

```
Synopsis:
    LIST_ID_t LIST_clear_list( LIST_ID_t             list,
                               LIST_DESTROY_PROC_p_t destroy_proc
                             )
Where:
    list         == ID of a previously created list
    destroy_proc -> function to call for each node
Returns:
    list
Exceptions:
    None
Notes:
   1. If not needed, the destroy proc may be NULL
```

5.4.2.8 LIST_destroy_list

This method will first clear the list (see *LIST_clear_list*) and then destroy the list itself. If the user specifies a destroy proc, it will be called for each node in the list prior to destroying the node.

```
Synopsis:
    LIST_ID_t LIST_destroy_list(
        LIST_ID_t             list,
        LIST_DESTROY_PROC_p_t destroy_proc
                               )
Where:
    list         == ID of a previously created list
    destroy_proc -> function to call for each node
Returns:
    LIST_NULL_ID
Exceptions:
    None
```

```
Notes:
    If not needed, the destroy proc may be NULL
```

5.4.3 Implementation Choices

When preparing to implement an ADT you are typically faced with a number of choices. A comparison of choices will typically reveal some common tradeoffs, such as:

- Efficiency vs. flexibility. Generally (though not always) increased flexibility comes at the cost of decreased efficiency; and
- Simplicity vs. complexity. Generally (though not always) if you strive to increase the efficiency and/or flexibility of your code, the code will become more complex. As code becomes more complex it is more likely to contain flaws and be more difficult to test and maintain. As a rule you want to keep your code as simple as possible while providing the flexibility demanded by the requirements of your current project, and anticipating future requirements. Efficiency is *infrequently* a concern; you should only sacrifice simplicity in favor of efficiency in a few, clearly defined and well-thought-out circumstances.

Hiding the details of your implementation; that is, keeping as much of the module *private* as you absolutely can; is the key to successful implementations. Because you can change the private details of a module without requiring adaptation on the part of the user, you can easily evolve an implementation to meet changing flexibility and efficiency issues; you can even provide alternative implementations that provide different levels of tradeoff between efficiency and flexibility.

Let's examine two alternative implementations for the list ADT discussed above. One will implement the list as an array (being relatively efficient) and the other will implement the list using our ENQ module (being relatively flexible). Keep in mind that although the two implementations are wildly different, they meet identical functional requirements.

5.4.3.1 Private Declarations for the Array Implementation

To implement a list as an array we are going to allocate a block of memory suitable for storage of an array of some maximum number of elements of a fixed size. These two values correspond to the *max_list_size* and *entry_size* arguments passed by the user to the list create method. When an entry is added to the list we will add it to the next sequentially available entry of the array. This arrangement is illustrated in **Figure 5-4**.

In order to facilitate this we are going to need a control structure that can hold a pointer to the array, the array size, the element size and the *next element* index. We should also store a copy of the list name. Thinking ahead a little bit, I know that we are going to want to do pointer arithmetic to move between entries in the array, so let's also declare the type of an entry pointer (we'll make this equivalent to a char* for reasons that will soon become clear). Then, since the address of an array is the same as a pointer to the first element of the array, the member of the control structure that points to the array is type "pointer to entry." The complete private header file for our list module when implemented as an array is shown in **Figure 5-5.**

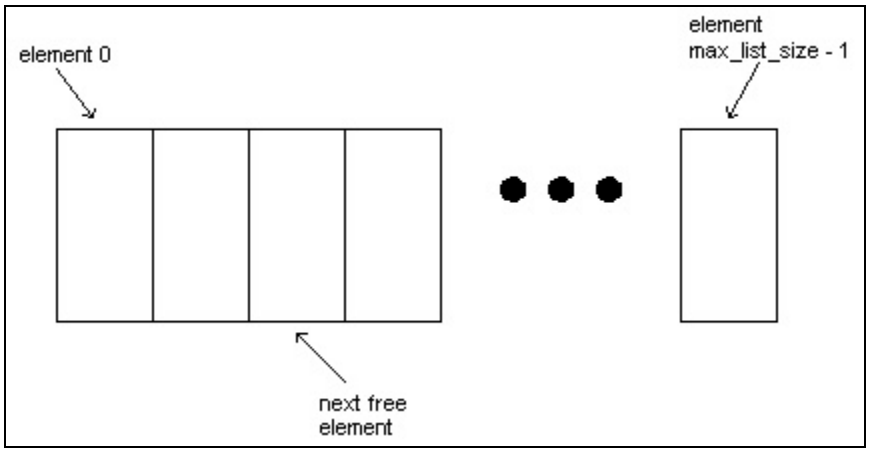


Figure 5-4 Implementing a List as an Array

```
#ifndef LISTP_H
#define LISTP_H

#include <list.h>
#include <stddef.h>

typedef char *LIST__ENTRY_p_t;

typedef struct list__control_s
{
    LIST__ENTRY_p_t array;
    size_t          max_size;
    size_t          entry_size;
    int             next;
    char            *name;
} LIST__CONTROL_t, *LIST__CONTROL_p_t;

#endif
```

Figure 5-5 Private Declarations for Array List Implementation

Implementing the create method for a list implemented as an array requires us to do the following:

1. Allocate memory for the control structure;
2. Allocate memory for the array, and store a pointer to it in the control structure;
3. Initialize the remaining members of the control structure; and
4. Return a pointer to the control structure to the caller. Remember that, since the caller doesn't include listp.h, there is no way for the caller to use the pointer to access the control structure.

The complete create method for implementing a list as an array is shown in **Figure 5-6**.

```
LIST_ID_t LIST_create_list( size_t     max_list_size,
                            size_t     entry_size,
                            const char *name
                          )
{
    LIST__CONTROL_p_t list  = CDA_NEW( LIST__CONTROL_t );
```

```
    list->array = CDA_calloc( max_list_size, entry_size );
    list->max_size = max_list_size;
    list->entry_size = entry_size;
    list->next = 0;
    list->name = CDA_NEW_STR_IF( name );

    return list;
}
```

Figure 5-6 Create Method for Array List Implementation

5.4.3.2 Private Declarations for the ENQ Implementation

To implement a list using the ENQ module our control structure will simply contain the address of an ENQ module anchor. To add an entry to the list we will simply create a new enqueuable item and add it at the tail of the ENQ list. The enqueuable item will have as its application data a single member that points to a block of memory to hold the user's data. This strategy is illustrated in **Figure 5-7**.

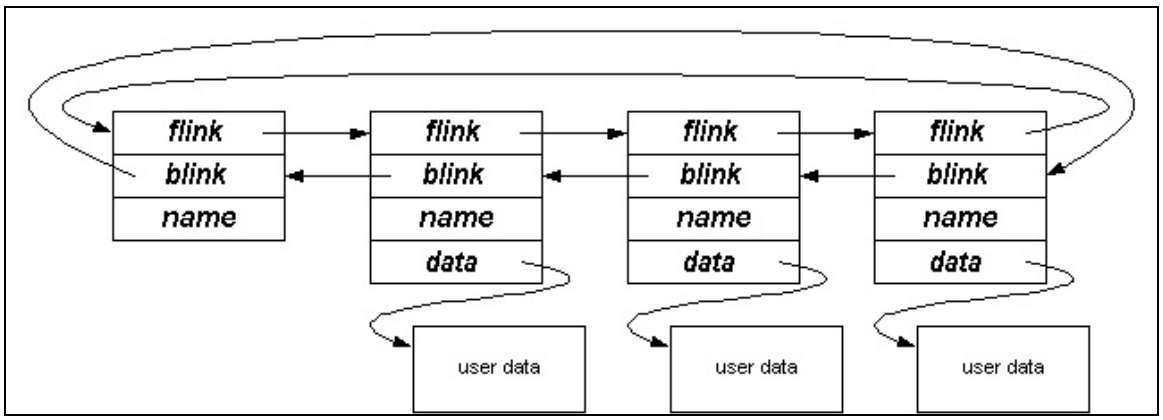


Figure 5-7 Implementing a List using the ENQ Module

Our private declarations will consist of a control structure that contains the address of a list anchor, plus the size of a user entry. We will also keep a copy of the list name, and the maximum list size specified by the user (we won't be using this value, but it doesn't hurt to hang onto it, and we may find a use for it in the future). Also, we will need to declare the type of an item to store in our ENQ list; remember that this will be an enqueuable item as defined by the ENQ module, and so the type of its first member must be ENQ_ITEM_t. The complete private header file for implementing a list via the ENQ module is shown in **Figure 5-8**.

```
#ifndef LISTP_H
#define LISTP_H

#include <list.h>
#include <enq.h>
#include <cda.h>
#include <stddef.h>

typedef struct list__control_s
{
    ENQ_ANCHOR_p_t  anchor;
    size_t          max_size;
    size_t          entry_size;
    char            *name;
} LIST__CONTROL_t, *LIST__CONTROL_p_t;

typedef struct list__entry_s
{
    ENQ_ITEM_t  item;
    void        *data;
} LIST__ENTRY_t, *LIST__ENTRY_p_t;

#endif
```

Figure 5-8 Private Declarations for ENQ List Implementation

Implementing the create method for a list implemented via the ENQ module requires us to do the following:

1. Allocate memory for the control structure;
2. Create an ENQ list, and store the address of its anchor in the control structure; and
3. Return a pointer to the control structure to the caller. As before, since the caller doesn't include listp.h, there is no way for her to use the pointer to access the control structure.

The complete create method for this implementation option is shown in **Figure 5-9**.

```
LIST_ID_t LIST_create_list( size_t     max_list_size,
                            size_t     entry_size,
                            const char *name
                          )
{
    LIST__CONTROL_p_t list  = CDA_NEW( LIST__CONTROL_t );

    list->anchor = ENQ_create_list( name );
    list->max_size = max_list_size;
    list->entry_size = entry_size;
    list->name = CDA_NEW_STR_IF( name );

    return list;
}
```

Figure 5-9 Create Method for ENQ List Implementation

5.4.3.3 Efficiency vs. Flexibility: The Add and Get-Size Methods

To demonstrate the issue of efficiency vs. flexibility, lets take a look at the *add* and *get-size* methods for the two different implementations. They are shown in **Figure 5-10** and **Figure 5-11**.

```
/* Add method for the array-based implementation */
const void *LIST_add_entry( LIST_ID_t list, const void *data )
{
    LIST__ENTRY_p_t nextEntry =
        list->array + list->next * list->entry_size;
    if ( list->next >= (int)list->max_size )
        abort();

    memmove( nextEntry, data, list->entry_size );
    ++list->next;

    return data;
}

/* Add method for the ENQ module-based implementation */
const void *LIST_add_entry( LIST_ID_t list, const void *data )
{
    LIST__ENTRY_p_t entry =
        (LIST__ENTRY_p_t)ENQ_create_item( NULL,
                                          sizeof(LIST__ENTRY_t)
                                        );

    entry->data = CDA_malloc( list->entry_size );
    memmove( entry->data, data, list->entry_size );
    ENQ_add_tail( list->anchor, (ENQ_ITEM_p_t)entry );

    return data;
}
```

Figure 5-10 Alternative List Add Method Implementations

As you can see, the array-based add method is fairly efficient. Using simple pointer arithmetic we locate the address of the next unused element in the array and copy the user's data into it; however it is inflexible because the size of the list cannot exceed the maximum length of the array, which the user must somehow calculate prior to creating the list. The ENQ module-based implementation is more flexible because it can dynamically grow to virtually any length, relieving the user of the need to determine a maximum length. This flexibility, however, comes at the cost of two extra dynamic memory allocations, which drastically reduces its efficiency.

The difference in efficiency can be seen even more clearly by examining the get-size method. The array-based implementation is extremely efficient, simply returning the *next* value out of the control structure. The ENQ module-based implementation, however, must traverse the list counting the elements as it goes.

Note that the array-based implementation can be made more flexible by using realloc to eliminate its dependence on a maximum size parameter. And the ENQ-module based implementation can be made more efficient in a couple of ways, for example by redesigning the LIST__ENTRY_t type to allow the user's data to reside directly in a

LIST_ENTRY_t variable, thereby eliminating a dynamic memory allocation. However each of these design changes comes at the cost of greatly increased complexity in the code.

```
/* Get-size method for the array-based implementation */
size_t LIST_get_list_size( LIST_ID_t list )
{
    size_t       rcode = (size_t)list->next;

    return rcode;
}

/* Get-size method for the ENQ module-based implementation */
size_t LIST_get_list_size( LIST_ID_t list )
{
    size_t       rcode = 0;
    ENQ_ITEM_p_t item  = ENQ_GET_HEAD( list->anchor );

    while ( item != list->anchor )
    {
        ++rcode;
        item = ENQ_GET_NEXT( item );
    }

    return rcode;
}
```

Figure 5-11 Alternative List Get-Size Methods

5.4.3.4 The Traverse and Clear Methods

Earlier when we talked about the traverse and empty methods, we introduced the concept of the *callback function*. So for the sake of completeness let's take a quick look at these two methods.

The traverse method must sequentially visit each entry in the list and give the user the opportunity to do something with the data there. For the ENQ module-based implementation all we do is find the first item, then follow the items' flinks till we reach the end. In the array-based implementation, we set a LIST__ENTRY_p_t variable to the start of the array, and use pointer arithmetic to sequentially find subsequent entries in the array (this is why made LIST__ENTRY_p_t equivalent to char*; the logic wouldn't work with a void*). In either case, each time we walk to a new entry in the list, we take a pointer to the user's data and pass it to the user's traversal proc. The code for both implementations is shown in **Figure 5-12**.

To clear a list in the array-based implementation we merely have to set the *next* indicator back to 0. To clear an ENQ module-based list we have to destroy each item in the list. In each case, however, we must first walk to each entry in the list, find the address of the user's data, and pass the address to the user's destroy proc for final disposition. The code for doing this is shown in **Figure 5-13**.

```
/* Traverse method for the array-based implementation */
LIST_ID_t LIST_traverse_list(
    LIST_ID_t               list,
    LIST_TRAVERSAL_PROC_p_t traversal_proc
                            )
{
    LIST__ENTRY_p_t entry = list->array;
    int             inx   = 0;

    for ( inx = 0 ; inx < list->next ; ++inx )
    {
        if ( traversal_proc != NULL )
            traversal_proc( entry );
        entry += list->entry_size;
    }

    return list;
}

/* Traverse method for the ENQ module implementation */
LIST_ID_t LIST_traverse_list(
    LIST_ID_t               list,
    LIST_TRAVERSAL_PROC_p_t traversal_proc
                            )
{
    ENQ_ITEM_p_t item = ENQ_GET_HEAD( list->anchor );

    while ( item != list->anchor )
    {
        LIST__ENTRY_p_t entry = (LIST__ENTRY_p_t)item;
        if ( traversal_proc != NULL )
            traversal_proc( entry->data );
        item = ENQ_GET_NEXT( item );
    }

    return list;
}
```

Figure 5-12 Alternative List Traverse Methods

```
/* Clear method for the array implementation */
LIST_ID_t LIST_clear_list( LIST_ID_t             list,
                           LIST_DESTROY_PROC_p_t destroy_proc
                         )
{
    int inx = 0;

    for ( inx = 0 ; inx < list->next ; ++inx )
        if ( destroy_proc != NULL )
            destroy_proc( list->array + inx * list->entry_size );

    list->next = 0;
    return list;
}

/* Clear method for the ENQ module implementation */
LIST_ID_t LIST_clear_list( LIST_ID_t             list,
                           LIST_DESTROY_PROC_p_t destroy_proc
                         )
{
    while ( !ENQ_is_list_empty( list->anchor ) )
    {
        LIST__ENTRY_p_t entry =
            (LIST__ENTRY_p_t)ENQ_deq_head( list->anchor );
        if ( destroy_proc != NULL )
            destroy_proc( entry->data );
        CDA_free( entry->data );
        ENQ_destroy_item( (ENQ_ITEM_p_t)entry );
    }

    return list;
}
```

Figure 5-13 Alternative List Clear Methods

6. Stacks

One of the most basic data structures in data processing is the *stack*. Surprisingly simple in its implementation, it is a highly versatile mechanism, often useful in implementing recursive and multithreaded algorithms. In this section we will examine a complete stack implementation. The material that we cover will be used in conjunction with the sorting algorithms you learned earlier to complete Project 3.

6.1 Objectives

At the conclusion of this section, and with the successful completion of your third project, you will have demonstrated the ability to:

- Compare a stack to a *last-in, first-out (lifo) queue;*
- Perform a complete, modularized stack implementation; and
- Use a stack to implement recursive algorithms.

6.2 Overview

A stack is often referred to as a last-in, first-out (LIFO) queue; that's because a stack grows by adding an item to its tail, and shrinks by removing an item from its tail, so the last item added to (or *pushed onto*) the stack is always the first to be removed (or *popped*); see **Figure 6-1**. Although in some implementations you may access data in the middle of a stack, it is illegal to remove any item but the end-most.

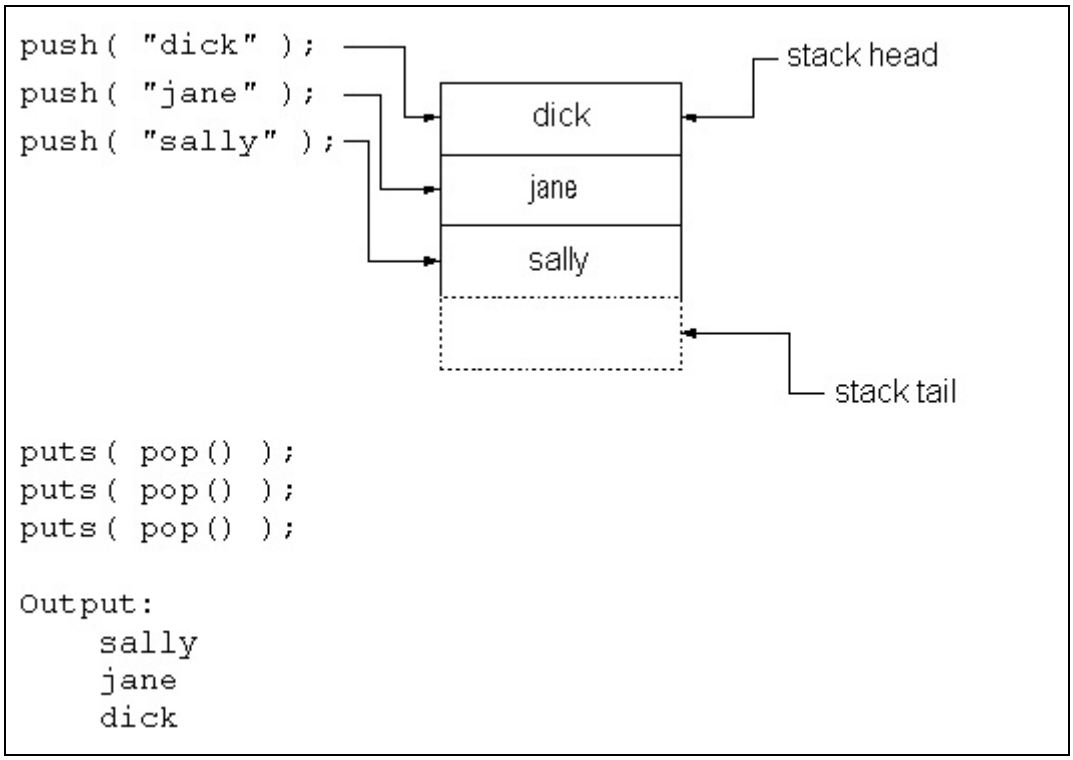


Figure 6-1: Stack Operations: Push and Pop

Stacks are often implemented using a list, with *push* corresponding to *add-tail* and *pop* to *remove-tail*. However the traditional implementation, and the most common view of a stack, is as an array. When a stack is treated as an array there are two alternative implementations: *top-down* and *bottom-up*. As seen in **Figure 6-2**, the beginning of a top-down stack is given as the first address following the last element of the array; the beginning of a bottom-up stack is given as the address of the first element of the array.

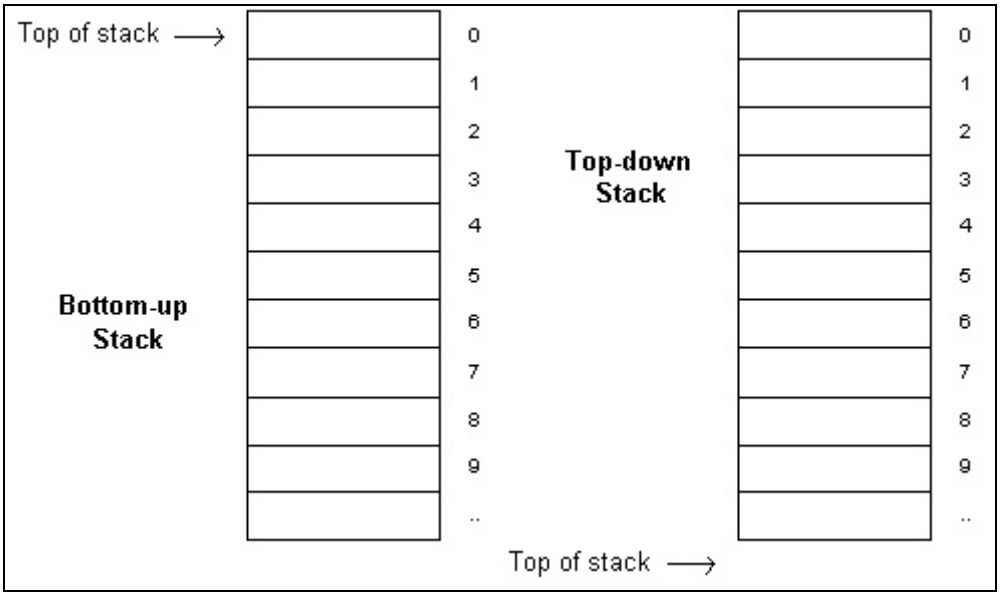


Figure 6-2: Two Stack Implementation Strategies

When you push an item onto the stack, the item that you add *occupies a location* on the stack. A *stack pointer* is used to indicate the position in the array that the pushed item should occupy. In a bottom-up implementation the stack pointer always indicates the *first unoccupied location*; to execute a push operation, first store the new item at the location indicated by the stack pointer, then increment the stack pointer. In a top-down implementation, the stack pointer always indicates the *last occupied location*; to execute a push operation, first decrement the stack pointer, then store the new item at the indicated location. The push operation is illustrated in **Figure 6-3**.

Note: It should be clear that bottom-up and top-down stacks are essentially equivalent. Application programmers tend to prefer bottom-up stacks because they're more intuitive; system programmers often prefer top-down stack for historical, and for obscure (and, for us, irrelevant) technical reasons. For the sake of avoiding confusion the remainder of this discussion will focus on bottom-up stacks.

A stack is empty when no position on the stack is occupied. When a bottom-up stack is empty, the stack pointer will indicate the first element of the array; when full, the stack pointer will indicate the first address after the last element of the array. This is shown in **Figure 6-4**. An attempt to push an item onto a full stack results in a *stack overflow condition*.

As shown in **Figure 6-5**, the last item pushed onto the stack can be removed by popping it off the stack. To pop an item off a bottom-up stack, first decrement the stack pointer, then remove the item at the indicated location. Note that once an item has been popped

off the stack, the location it formerly occupied is now *unoccupied*, and you should not expect to find a predictable value at that location. An application should never try to pop an item off an empty stack; attempting to do so constitutes an egregious malfunction on the part of the application.

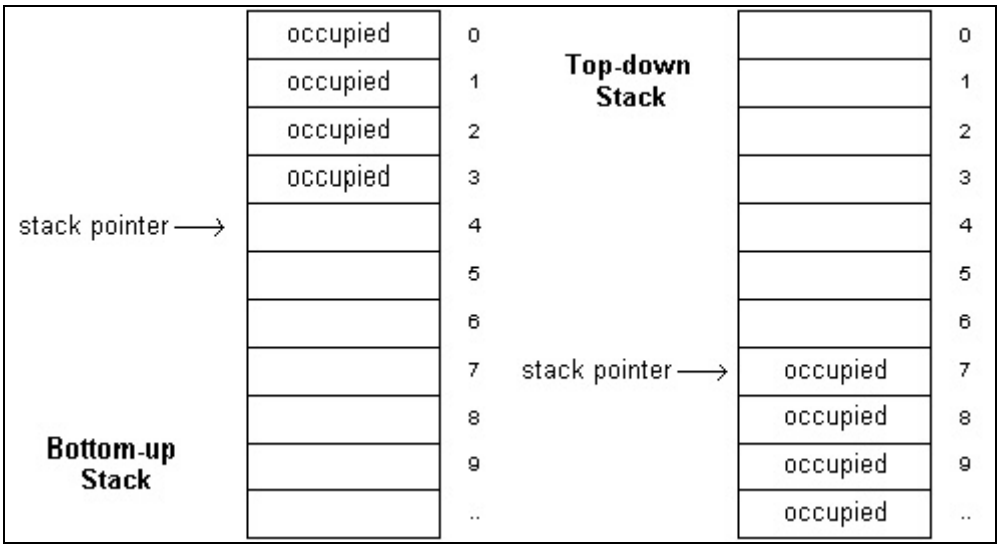


Figure 6-3: Push Operation

A note about popping, and the values of unoccupied locations: If you experiment with stacks using a simple test driver you might conclude that the value popped off the stack continues to occupy the stack at its previous location. However applications that share a stack among asynchronously executed subroutines will find the values of such locations unpredictable. Asynchronous or interrupt-driven environments are where the magic of stacks is most useful; unfortunately they are beyond the scope of this course.

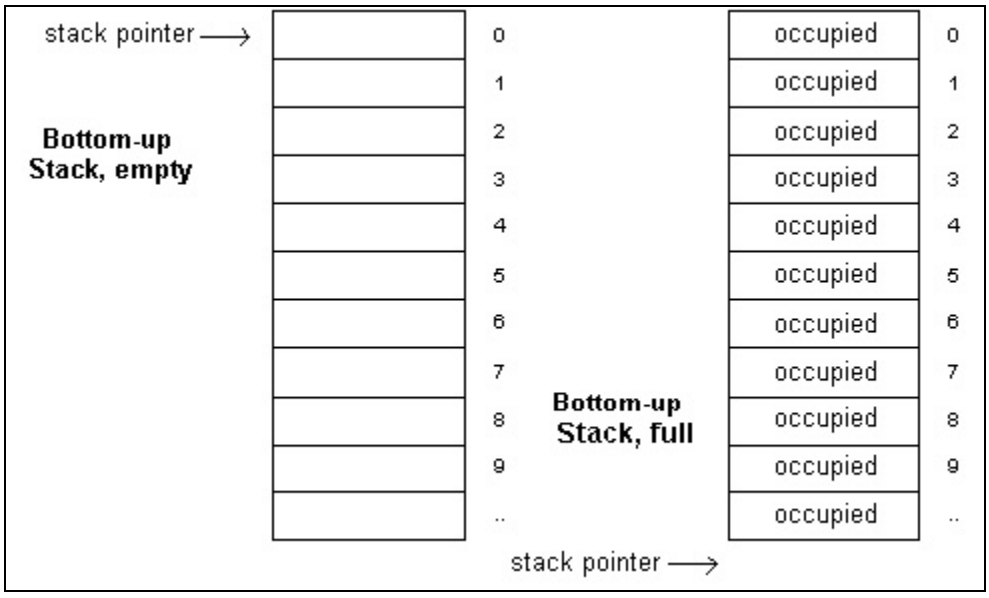


Figure 6-4: Two Stack States

As a first example of using a stack, following is an example of a subroutine to reverse the characters in a string.

```
static char *reverse_chars( char *string )
{
    char *temp  = string;
    char stack[50];
    char *sptr = stack;

    while ( *temp != '\000' )
        *sptr++ = *temp++;  /* push */

    temp = string;
    while ( sptr != stack )
        *temp++ = *--sptr;  /* pop */

    return string;
}
```

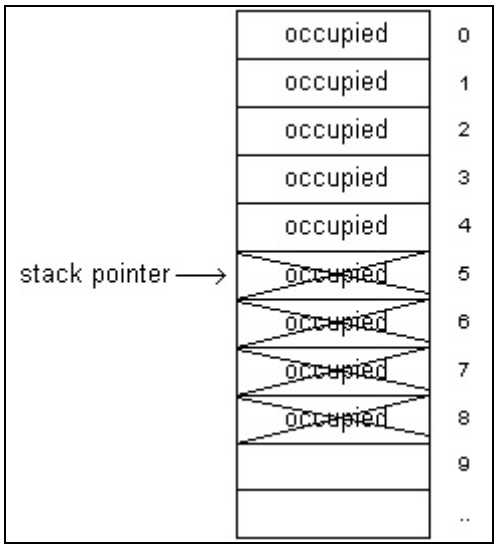


Figure 6-5: Pop Operation

6.3 Stacks And Recursion

A single stack can be shared among many instances of a recursive function. A recursively-invoked function merely begins using the stack at the point that the previous invocation left it. Suppose we were writing an interpreter that could parse and evaluate function-like strings like these:

`add( a, b )`	Evaluate a + b
`mul( a, b )`	Evaluate a * b
`div( a, b )`	Evaluate a / b
`sum( a, b, c, . . . N )`	Compute the sum of the arguments
`and( a, b, c, . . . N )`	Compute the logical AND of the arguments from right to left
`etc.`	etc.

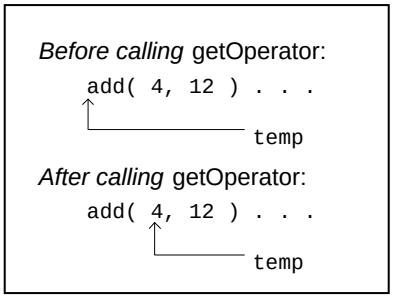
Let's also suppose that our interpreter will also be able to process strings that consist of a embedded instances of the above, such as this one:

```
and( 0xff, sum( mul(10, 50), div(24, 3), add( mul(2, 4), 2 ) ) )
```

Then we could implement our interpreter as a recursive function that called itself each time it had to evaluate a subexpression. Such a function might look like the function *StackCalc* as shown in **Figure 6-6**.

```
long StackCalc( const char *expr, const char **end )
{
    long        rval        = 0;
    long        operator    = 0;
    long        operand     = 0;
    int         count       = 0;
    CDA_BOOL_t  working     = CDA_TRUE;
    const char  *temp       = skipWhite( expr );

    operator = getOperator( &temp );
    while ( working )
    {
        if ( getOperand( &operand, &temp ) )
        {
            ++count;
            push( operand );
        }
        else
            working = CDA_FALSE;
    }

    switch ( operator )
    {
        case ADD:
            rval = xadd( count );
            break;
            . . .
    }

    if ( end != NULL )
        *end = skipWhite( temp );

    return rval;
}
```

Figure 6-6 StackCalc Function Entry Point



StackCalc first obtains an operation identifier by parsing the operator token, such as *add*, *mul*, etc. To do this it calls the function *getOperator*, which takes a input a pointer to the string being parsed; it's return value is the operation identifier, and it updates the string to point to the first token after the operator token. Next iteratively calls *getOperand* to obtain each operand. The return value of getOperand is TRUE if an operand was parsed, and FALSE if the end of the operand list was detected. If getOperand returns TRUE it also return the operand via the parameter *operand*. It always updates the string pointer to

point to the next unparsed token in the string. Each time getOperand returns the value of an operand StackCalc pushes it onto a stack, keeping a count of the number of operands that have been pushed. When getOperand indicates that the last argument has been parsed StackCalc calls a function to pop each operand off the stack and perform the evaluation. The function for performing the *sum* evaluation is shown in **Figure 6-7**.

```
static long xsum( int count )
{
    long    rcode   = 0;
    int     inx     = 0;

    for ( inx = 0 ; inx < count ; ++inx )
        rcode += pop();

    return rcode;
}
```

Figure 6-7 StackCalc Function to Evaluate *sum*

The last thing StackCalc does before returning is to set the *end* parameter to point to the portion of the string following whatever it has just evaluated. For example, if the string it evaluated was "add(2, 4)" it will leave *end* point to the end of the string; but if it had just evaluated the *add* portion of "sum(add(1, 2), mul(2, 4))" it will leave *end* pointing to *mul*.

The recursive logic in StackCalc is contained in the function getOperand, which evaluates the input string as follows:

- Is the next token a right parenthesis? If so the end of the argument list has been reached.
- Does the next token start with a digit? Is so parse the argument by calling *strtol*.
- Does the next token start with an alphabetic character? If so parse the subexpression by recursively calling StackCalc.

Let's consider what's going on with the stack while StackCalc is evaluating this string:

```
and( 0xff, sum( mul(10, 50), div(24, 3), add( mul(2, 4), 2 ) ) )
```

We will refer to various calls to StackCalc as *instances* of the function. When the user first calls StackCalc we enter *instance 0*. Whenever instance 0 makes a recursive call we enter *instance 1*, etc.

1. Instance 0 obtains the identifier for *and*, and pushes 0xff onto the stack, then calls instance 1 to evaluate "sum(. . .)."
2. Instance 1 obtains the identifier for *sum* and immediately calls instance 2 to evaluate "mul(. . .)."
3. Instance 2 obtains the identifier for *mul* and pushes 10 and 50 onto the stack. At this point our stack has the state shown in **Figure 6-8**.
4. Instance 2 pops two operands off the stack, evaluates 10 * 50 and returns the result to instance 1, which pushes it onto the stack. Then instance 1 again calls instance 2 to evaluate "div(. . .)."

5. Instance 2 obtains the identifier for *div* and pushes 24 and 3 onto the stack which now has the state shown in **Figure 6-9**.

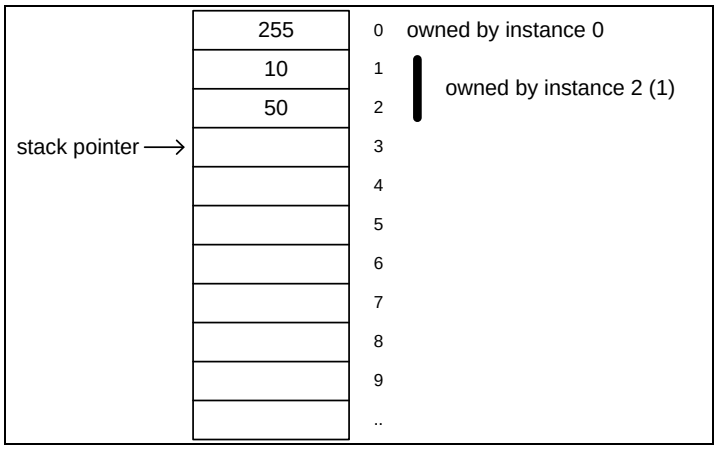


Figure 6-8 Stack State: Processing mul(10, 50)

6. Instance 2 pops two operands off the stack, evaluates 24 / 3 and returns the result to instance 1 which pushes it onto the stack. Instance 1 calls instance 2 once again to evaluate "add(. . .)."
7. Instance 2 obtains the identifier for *add* and immediately calls instance 3 to evaluate "mul(…)."

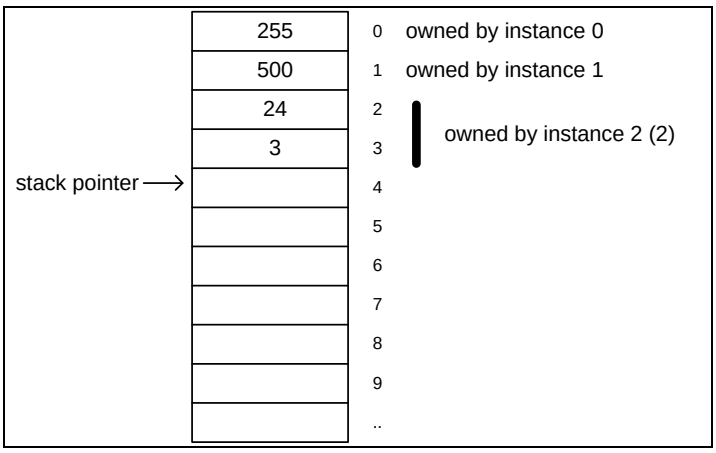


Figure 6-9 Stack State: Processing div(24, 3)

8. Instance 3 obtains the identifier for *mul* and pushes 2 and 4 onto the stack. The current state of the stack is now shown in **Figure 6-10**.
9. Instance 3 pops two operands off the stack, evaluating 2 * 4, and returns the result to instance 2 which pushes it onto the stack. Then instance 2 pushes 2 onto the stack.
10. Instance 2 pops two operands off the stack, computing 8 + 2, and returns the result to instance 1which pushes it onto the stack.
11. Instance 1 pops 3 arguments off the stack, evaluating 10 + 8 + 500, and returns the result to instance 0 which pushes it onto the stack.
12. Instance 0 pops 2 arguments off the stack, evaluating 518 & 0xff, and returns the result to the user.

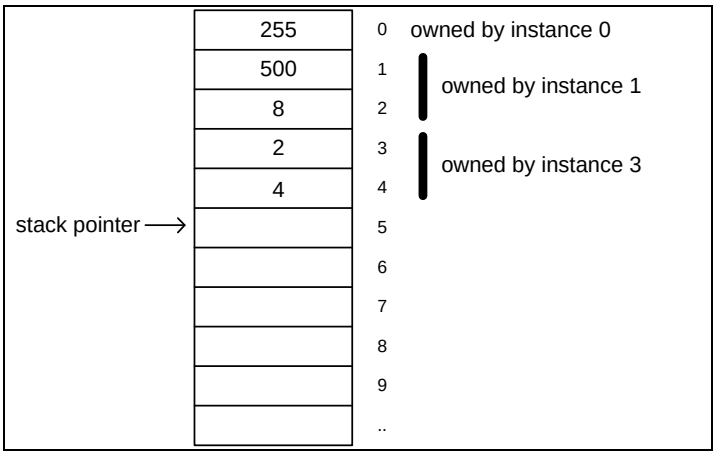


Figure 6-10 Stack State: Processing mul(2, 4)

6.4 A Minimal Stack Module

A minimal stack module will implement the following methods:

- Create a stack;
- Push an item onto a stack;
- Pop an item off of a stack;
- Peek at a stack (return the value of the last item on the stack without removing it);
- Determine whether a stack is empty;
- Determine whether a stack is full;
- Clear a stack, leaving it empty; and
- Destroy a stack.

We'll proceed by examining a stack implementation from a functional point of view, beginning with the public data types and discussing each method in detail; then we'll look at an example, and describe the details of the implementation.

6.4.1 STK Module Public Declarations

We have chosen the name *STK* for our module, so the public declarations will be placed in a header file named *stk.h*. In addition to the prototypes for our public methods, stk.h will contain an incomplete declaration for a stack ID, and a macro declaring a value for a NULL stack ID (compare these declarations to those for the LIST module which we discussed earlier). These declarations are shown in **Figure 6-11**.

```
#define STK_NULL_ID (NULL)
typedef struct stk__control_s *STK_ID_t;
```

Figure 6-11 STK Module Pulic Declarations

6.4.2 STK_create_stack: Create a Stack

This method will create a stack of a given size. The stack will be used for storing items of type (void *).

```
Synopsis:
    STK_ID_t STK_create_stack( size_t size );
Where:
    size == size of the stack in items
Exceptions:
    Throws SIGABRT on create failure
Returns:
    stack ID to be used in all subsequent stack operations
Notes:
    None
```

6.4.3 STK_push_item: Push an Item onto a Stack

This method will push an item of type (void *) onto a stack.

```
Synopsis:
    void STK_push_item( STK_ID_t stack, void *item );
Where:
    stack == stack id returned by STK_create_stack
    item  == item to push
Exceptions:
    Throws SIGABRT on stack overflow
Returns:
    void
Notes:
    None
```

6.4.4 STK_pop_item: Pop an Item off a Stack

This method will remove the top item from the stack and return it.

```
Synopsis:
    void *STK_pop_item( STK_ID_t stack );
Where:
    stack == stack id returned by STK_create_stack
Exceptions:
    See notes
Returns:
    Top item of stack
Notes:
    This method contains an assertion which will throw SIGABRT if
    the user attempts to illegally remove an item from an empty
    stack.  The assertion is disabled in production code.
```

6.4.5 STK_peek_item: Get the Top Item of a Stack

This method returns the top item of a stack without removing it.

```
Synopsis:
    void *STK_peek_item( STK_ID_t stack );
Where:
    stack == stack id returned by STK_create_stack
Exceptions:
    See notes
```

```
Returns:
    The value at the top of the stack
Notes:
    This method contains and assertion which will throw SIGABRT if
    the user attempts to illegally get an item from an empty
    stack.  The assertion is disabled in production code.
```

6.4.6 STK_is_stack_empty: Determine If a Stack is Empty

This method indicates whether or not a stack is empty.

```
Synopsis:
    CDA_BOOL_t STK_is_stack_empty( STK_ID_t stack );
Where:
    stack == stack id returned by STK_create_stack
Exceptions:
    None
Returns:
    CDA_TRUE if stack is empty,
    CDA_FALSE otherwise
Notes:
    None
```

6.4.7 STK_is_stack_full: Determine If a Stack is Full

This indicates whether or not a stack is full.

```
Synopsis:
    CDA_BOOL_t STK_is_stack_full( STK_ID_t stack );
Where:
    stack == stack id returned by STK_create_stack
Exceptions:
    None
Returns:
    CDA_TRUE if stack is full,
    CDA_FALSE otherwise
Notes:
    None
```

6.4.8 STK_clear_stack

This method removes all items from a stack, leaving the stack in an empty state.

```
Synopsis:
    void STK_clear_stack( STK_ID_t stack );
Where:
    stack == stack id returned by STK_create_stack
Exceptions:
    None
Returns:
    void
Notes:
    None
```

6.4.9 STK_destroy_stack: Destroy a Stack

This method destroys a stack, freeing all resources associated with it.

```
Synopsis:
    STK_ID_t STK_destroy_stack( STK_ID_t stack );
Where:
    stack == stack id returned by STK_create_stack
Exceptions:
    None
Returns:
    STK_NULL_ID
Notes:
    None
```

6.4.10 Simple Stack Example

In this example we will reexamine our mergesort implementation. You will recall that two of the main problems with our original implementation were its heavy reliance on dynamic memory allocation, and dealing with pointer arithmetic. This new implementation will substitute a single stack for the many dynamic memory allocations, and eliminate the pointer arithmetic problem by dealing solely with arrays of pointers.

The pseudocode for the new mergesort algorithm is found in **Figure 6-12**. The first difference you will notice between this implementation and our earlier one is that it is sorting an *array of void pointers*, rather than merely using a void pointer to indicate the start of a generic array. Since the implementation can now assume that the size of each element of the array is *sizeof(void *)* there is no longer any reason for the user to pass an *element_size* argument. It also means that the user can *only* sort arrays of pointer, but the pointers may be of almost any type (specifically, any type except pointer-to-function). **Figure 6-13** shows one example of an array that a user may *not* sort with the modified sort routine and two examples that she *may*.

Another advantage of the new strategy is, since the type of an element in the array is fully understood by the compiler, we no longer have problems with pointer arithmetic; the compiler can do the arithmetic quite easily. For example, dividing the array into two parts is reduced to the following three lines of code:

```
lowHalf = numElements / 2;
highHalf = numElements – lowHalf;
array2 = array + lowHalf;
```

The next obvious difference is the logic to make sure the stack is created the first time mergesort is called. Note that we never destroy the stack; we do this quite intentionally. In a modern operator system allocating memory exactly once and then hanging on to it till the program expires is *not* a memory leak. A drawback of this strategy is that we have to predetermine the maximum size of an array that we are likely to want sort prior to writing our code. There are ways to deal with this shortcoming, but to keep the problem simple for now we will just use a macro, *MAX_STACK_SIZE*, which defines this size as a constant.

```
STK_ID_t stack = STK_NULL_ID;
mergesort( void **array, size_t numElements )
```

```
    if ( stack == STK_NULL_ID )
        stack = STK_create_stack( MAX_STACK_SIZE )

    if ( numElements > 1 )
        lowHalf = numElements / 2
        highHalf = numElements - lowHalf
        array2 = array + lowHalf
        mergesort( array, lowHalf )
        mergesort( array2, highHalf )

        inx = jnx = 0
        while ( inx < lowHalf && jnx < highHalf )
            if ( array[inx] < array2[jnx] )
                STK_push_item( stack, array[inx++] )
            else
                STK_push_item( stack, array2[jnx++] )

        while ( inx < lowHalf )
            STK_push_item( stack, array[inx++] )
        while ( jnx < highHalf )
            STK_push_item( stack, array2[jnx++] )

        inx = numElements;
        while ( inx > 0 )
            array[--inx] = STK_pop_item( stack )
```

Figure 6-12 Stack-Based Implementation of Mergesort

```
/* An array that may not be sorted with the new mergesort */
int iarr[] = { 5, 1, 7, 9 };

/* Two arrays that may be sorted with the new mergesort */
char *sarr[]  = { “dick”, “sally”, “jane”, “spot” };
int  *piarr[] = { &iarr[0], &iarr[1], &iarr[2], &iarr[3] };

mergesort( (void**)sarr, CDA_CARD(sarr) ):
mergesort( (void**)piarr, CDA_CARD(piarr) );
```

Figure 6-13 Calling the New Mergesort Routine

6.4.11 Implementation Details

As illustrated in **Figure 6-14**, our private header file will consist of a single declaration: a control structure which contains a pointer to an array of type void* to use as the stack, a stack pointer and the size of the stack.

```
#ifndef STKP_H
#define STKP_H

#include <stk.h>
#include <stddef.h>

typedef struct stk__control_s
{
  void   **stack;
  void   **sptr;
  size_t size;
} STK__CONTROL_t, *STK__CONTROL_p_t;

#endif
```

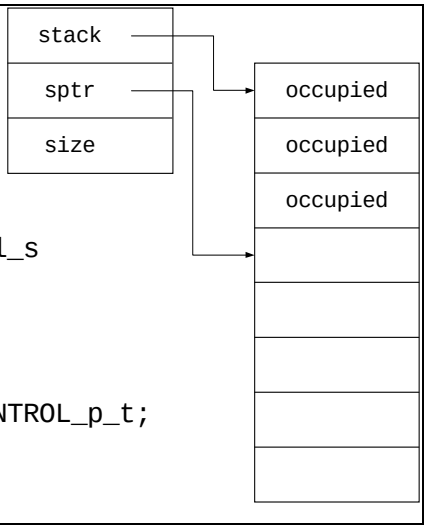


Figure 6-14 STK Module Private Header File

The implementation of all but three of our methods is show below. The implementation of STK_push_item, STK_pop_item and STK_is_stack_full is left as an exercise.

```
#include <stkp.h>
#include <stdlib.h>

void STK_clear_stack( STK_ID_t stack )
{
    stack->sptr = stack->stack;
}

STK_ID_t STK_create_stack( size_t size )
{
    STK__CONTROL_p_t stack = CDA_NEW( STK__CONTROL_t );

    stack->stack = CDA_calloc( size, sizeof(void *) );
    stack->sptr = stack->stack;
    stack->size = size;

    return (STK_ID_t)stack;
}

STK_ID_t STK_destroy_stack( STK_ID_t stack )
{
    CDA_free( stack->stack );
    CDA_free( stack );

    return STK_NULL_ID;
}


void *STK_peek_item( STK_ID_t stack )
{
    CDA_ASSERT( stack->sptr != stack->stack );
    return *(stack->sptr - 1);
}

void *STK_pop_item( STK_ID_t stack )
```

```
{
}

void STK_push_item( STK_ID_t stack, void *item )
{
}

CDA_BOOL_t STK_is_stack_empty( STK_ID_t stack )
{
    CDA_BOOL_t rcode =
        stack->sptr == stack->stack ? CDA_TRUE : CDA_FALSE;

    return rcode;
}

CDA_BOOL_t STK_is_stack_full( STK_ID_t stack )
{
}
```

6.5 A More Robust Stack Module

In addition to the basic functionality described above, additional features are often available in stack implementations. The most popular ones are these:

- Mark the stack;
- Index into the stack from a mark;
- Clear the stack to a mark;
- Grab stack space; and
- Create a segmented stack.

This additional functionality will be discussed below.

6.5.1 Stack Marks

A stack mark identifies some position within the stack, and permits the user full access to the occupied portion of the stack. In order to implement marks, we need a new public declaration: a *stack mark type* typedef. Our purposes, a stack mark will just be an integer index into the stack:

```
typedef int STK_MARK_t, *STK_MARK_p_t;
```

Next, we need to modify the push method so that it returns a mark representing the position of the pushed item.

```
STK_MARK_t STK_push_item( STK_ID_t stack, void *item )
{
    STK_MARK_t mark = stack->sptr - stack->stack;
    . . .
    return mark;
}
```

Now we can define two new methods: one to interrogate the stack at some offset from a mark, and one to change the value of a stack location at some offset from a mark. We

will validate, using assertions, that the indexed value is legal. The implementation of these routines is shown in **Figure 6-15** and **Figure 6-16**.

```
void *STK_get_item( STK_ID_t stack, STK_MARK_t mark, int offset )
{
    CDA_ASSERT( stack->stack + mark >= stack->stack );
    CDA_ASSERT( stack->stack + mark < stack->sptr );
    CDA_ASSERT( stack->stack + mark + offset >= stack->stack );
    CDA_ASSERT( stack->stack + mark + offset < stack->sptr );

    return *(stack->stack + mark + offset);
}
```

Figure 6-15 STK_get_item

```
void STK_change_item( STK_ID_t   stack,
                      STK_MARK_t mark,
                      int        offset,
                      void       *val
                    )
{
    CDA_ASSERT( stack->stack + mark >= stack->stack );
    CDA_ASSERT( stack->stack + mark < stack->sptr );
    CDA_ASSERT( stack->stack + mark + offset >= stack->stack );
    CDA_ASSERT( stack->stack + mark + offset < stack->sptr );

    *(stack->stack + mark + offset) = val;
}
```

Figure 6-16 STK_change_item

To access an item at a particular location on the stack, the user passes a previously obtained mark plus an offset. An offset of 0 indicates the marked location itself; an offset of –1 indicates the item that was pushed immediately before the marked item, and an offset of 1 indicates the item immediately after the marked item, etc.

The methods to clear a stack and to grab stack space are shown in **Figure 6-17** and **Figure 6-18**. To clear a stack we simply reset the stack pointer to a marked location; to grab stack space we simply advance the stack pointer; although we have not explicitly written any values to the stack locations that we skipped, those locations are now considered occupied, and may be modified using STK_change_item.

```
void STK_clear_to_mark( STK_ID_t stack, STK_MARK_t mark )
{
    CDA_ASSERT( stack->stack + mark >= stack->stack );
    CDA_ASSERT( stack->stack + mark < stack->sptr );

    stack->sptr = stack + mark;
}
```

Figure 6-17 STK_clear_to_mark

```
STK_MARK_t STK_grab_space( STK_ID_t     stack,
                           int          num_slots,
                           STK_MARK_p_t bottom_mark
                         )
{
    if ( stack->sptr + num_slots >= stack->stack + stack->size )
        abort();
```

```
    if ( bottom_mark != NULL )
        *bottom_mark = stack->sptr - stack->stack;
    stack->sptr += num_slots;

    return stack->sptr - stack->stack - 1;
}
```

Figure 6-18 STK_grab_space

6.5.2 Segmented Stacks

A *segmented stack* is implemented via a *noncontiguous array*. When an application tries to push an item onto a segmented stack that is full, instead of throwing an exception the implementation allocates a new stack segment and silently performs the operation. In order to implement a segmented stack, we'll need to make a few changes to our data structure.

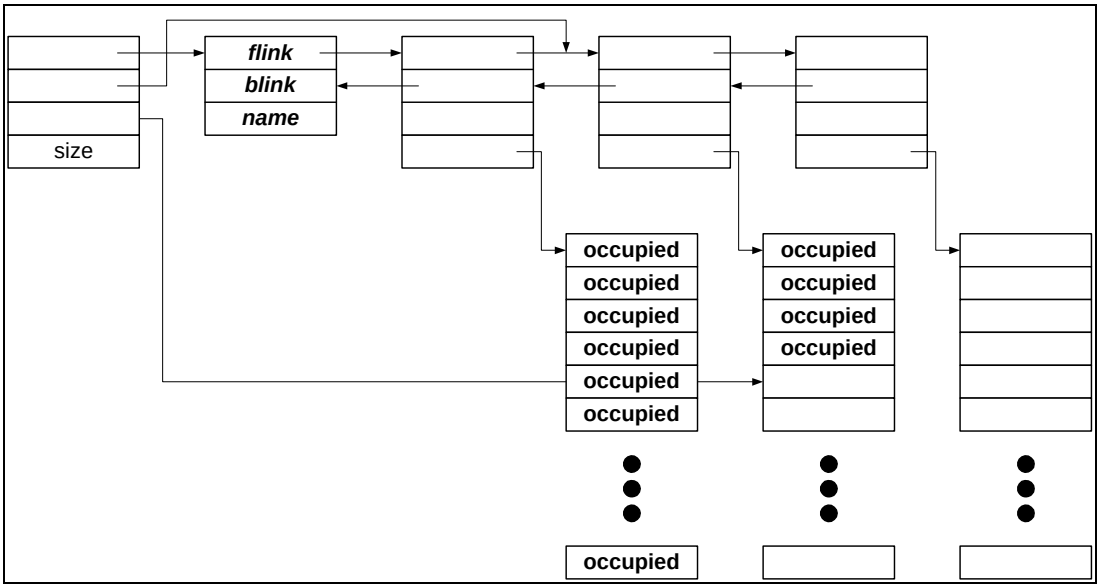


Figure 6-19 Segmented Stack Layout

First, with a segmented stack, it will be easier to make our stack pointer an integer index rather than a pointer; that will make it easier to figure out which segment a stack slot falls in. Second, as suggested by **Figure 6-19**, each segment in the stack will be an controlled by an enqueuable item; and rather than keeping a pointer to the stack in the stack control structure, we will keep a pointer to a list anchor, plus a pointer to the current stack segment. The new declarations and create method are shown in **Figure 6-20** and **Figure 6-21**. Note that since none of our public API has changed, a segmented implementation could replace a simple implementation with absolutely no impact on the applications that use it.

Pushing and popping data on a segmented stack now requires watching out for segment boundaries, and allocating a new segment when an application tries to push data onto a full stack. The implementation details are left to the imagination of the reader.

```
typedef struct stk__stack_seg_s
{
```

```
    ENQ_ITEM_t item;
    void       **stack;
} STK__stack_seg_t, *STK__stack_seg_p_t;

typedef struct stk__control_s
{
    ENQ_ANCHOR_p_t     seg_list;
    STK__STACK_SEG_p_t curr_seg;
    int                sptr;
    size_t size;
} STK__CONTROL_t, *STK__CONTROL_p_t;
```

Figure 6-20 Segmented Stack Private Declarations

```
STK_ID_t STK_create_stack( size_t size )
{
    STK__CONTROL_p_t stack = CDA_NEW( STK__CONTROL_t );

    stack->seg_list = ENQ_create_list( "Stack" );
    stack->curr_seg = (STK__STACK_SEG_p_t)
        ENQ_create_item( sizeof(STK__STACK_SEG_t) );
    ENQ_add_tail( stack->seg_list,(ENQ_ITEM_p_t)stack->curr_seg);
    stack->curr_seg->stack = CDA_calloc( size, sizeof(void *) );
    stack->sptr = 0;
    stack->size = size;

    return (STK_ID_t)stack;
}
```

Figure 6-21 Segmented Stack Create Method

7. Priority Queues

In this section we will discuss the *queue*, a very simple structure that organizes data on a first-come, first-served basis, and the *priority queue*, which organizes data according to an arbitrary designation of importance. Priority queues are a practical means of implementing *schedulers*, such as real-time operating system schedulers that execute processes, or communications schedulers that transmit messages according to their priority.

In addition, we will continue our discussion of implementation choices that trade-off speed against data size, and issues of algorithmic complexity. And we will examine two priority queue implementations: a *simple implementation* that offers basic functionality with relatively low complexity and slow execution, and a *robust implementation* that offers enhanced functionality and fast execution at the cost of additional complexity and data size requirements.

7.1 Objectives

At the conclusion of this section, and with the successful completion of your fourth project, you will have demonstrated the ability to:

- Discuss the usefulness and application of priority queues;
- Choose between algorithms that trade-off speed and complexity against data size; and
- Perform a complete, modularized priority queue implementation.

7.2 Overview

A *queue* may be visualized as an array to which elements may be added or removed. A new element is always *added* to the end of the queue, and elements are always *removed* from the front of the queue. Therefore the first element added to the queue is always the first element removed, so a queue is often referred to a *first-in, first-out queue* or *FIFO*. This arrangement is illustrated in **Figure 7-1**. A typical application of a queue is an order-fulfillment system. An online bookseller, for example, takes an order from a customer over the Internet; orders are added to the end of a queue as they are received. In the shipping department orders are removed from the front of the queue and fulfilled in the order that they were received.

A *priority queue* is a queue in which each element is assigned a *priority*. A priority is typically an integer, and higher integers represent higher priorities. All elements having the same priority are said to belong to the same *priority class*. As seen in **Figure 7-2**, a priority queue may be visualized as an array in which all elements of the same priority class are grouped together, and higher priority classes are grouped closer to the front of the array. A new element is *added* to the tail of its priority class, and elements are always *removed* from the front of the queue. A remove operation, therefore, always addresses the element of the highest priority class that has been enqueued the longest. Imagine modifying our order fulfillment system to take into account preferred customers. Let's say that customers with ten years standing get served before customers with five years

standing, and that customers with five years standing get served before new customers. Then when a customer with five years standing places an order, the order goes in the queue ahead of orders from new customers, but behind older orders from customers with five years standing or more.

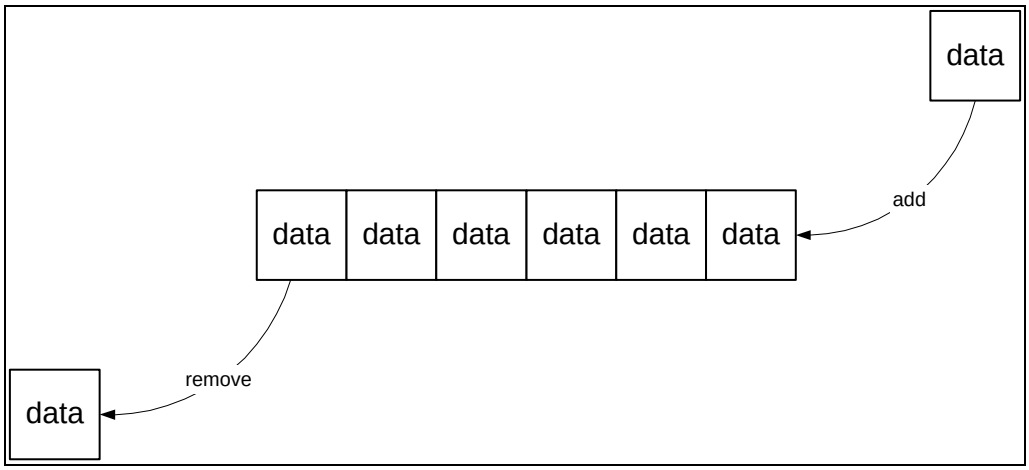


Figure 7-1 Visualizing a Queue as an Array

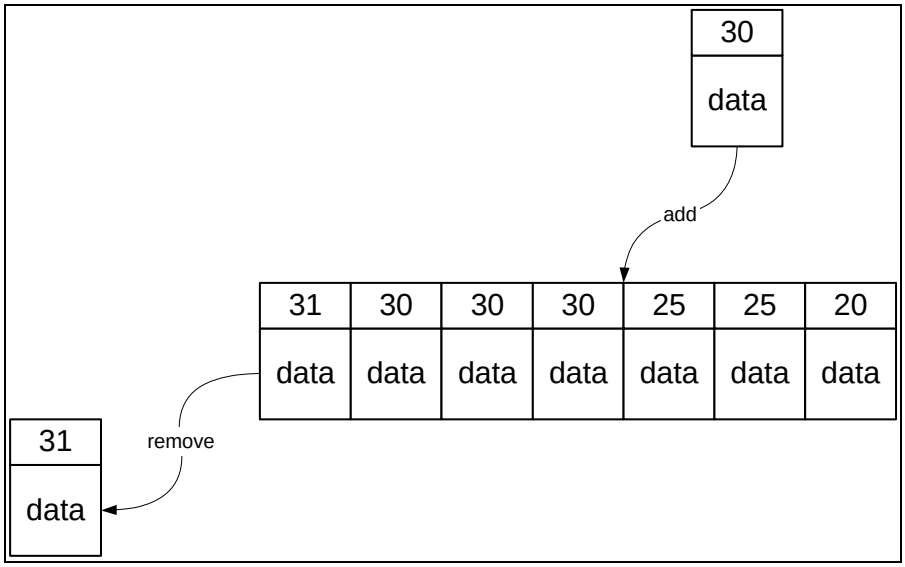


Figure 7-2 Visualizing a Priority Queue as an Array

Let's discuss *queues* and *priority queues* a little more detail. We'll start with queue.

7.3 Queues

As discussed in your textbook, a *queue* is an ordered sequence of elements on which we may perform the following operations (note: your textbook defines only nine operations; I have taken the liberty of adding the *destroy* operation):

1. Create a queue;
2. Determine if a queue is empty;
3. Determine if a queue is full;
4. Determine the size of a queue;

5. Append a new entry to a queue;
6. Retrieve (without removing) the entry at the front of a queue;
7. Remove the entry at the front of a queue;
8. Clear a queue;
9. Traverse a queue; and
10. Destroy a queue.

As with the *list* module that we saw earlier, we have a couple of choices when it comes to the actual implementation of a queue, but we need not make any of those choices in order to fully specify the behavior of a queue; which is another way of saying that we should be able to define the public interface of a *queue module* without specifying the private details. And to define the public interface first we must choose a module name, then we must declare the data types to represent:

- The ID of a queue;
- Any callback functions associated with a queue operation; and
- The data that will be used as input and output values for methods such as *remove* and *append*.

The name of our sample queue module will be QUE. Once again employing an incomplete declaration to achieve both *encapsulation* and *strong typing*, a queue ID will be a pointer to a control structure tag, and we will declare a value to be used as a NULL queue ID. There will be two callback function types associated with our module, one for traversing a queue, and one for destroying it. In each case a callback will have a single argument representing a pointer to the user data associated with an element in the queue and returning void. That gives us the public declarations shown in **Figure 7-3**.

```
#define QUE_NULL_ID     (NULL)
typedef struct que__control_s *QUE_ID_t;

typedef void QUE_DESTROY_PROC_t( void *data );
typedef QUE_DESTROY_PROC_t *QUE_DESTROY_PROC_p_t;
typedef void QUE_TRAVERSE_PROC_t( void *data );
typedef QUE_TRAVERSE_PROC_t *QUE_TRAVERSE_PROC_p_t;
```

Figure 7-3 QUE Module Public Declarations

For input and output values for such functions as *append* and *remove* we could keep it simple and just use a void pointer to represent the user's data. However, experience suggests that applications often entwine the use of queues and other kinds of lists. For example, in a communications application it is not unusual for a process to undergo state changes such as this one:

1. Wait in a work-in-progress queue while transmission parameters are established;
2. Switch to a priority queue awaiting the availability of resources to perform the transmission;
3. Switch to a response queue awaiting a response to the transmission;
4. Switch to a garbage collection queue to await destruction or reuse.

To facilitate such state changes it makes sense for a queue element to be able to move quickly and easily from one queue to another. To that end we will make the design decision that a value appended to or removed from a queue will always be represented by an *enqueuable item* (as defined by our ENQ module), and that such an item will contain a pointer to the user's data. To accomplish this, we will need one additional declaration for a *queue item*, which is shown in **Figure 7-4**; we will also define two additional queue operations:

- Create and return a queue item; and
- Destroy a queue item.

```
typedef struct que_item_s
{
    ENQ_ITEM_t  item;
    void        *data;
} QUE_ITEM_t, *QUE_ITEM_p_t;
```

Figure 7-4 Another QUE Module Public Declaration

Our decision to make our queue a collection of enqueuable items has pretty much determined that the heart of our implementation will be a linked list as defined by our ENQ module, so the private header file for our queue module will consist of the declaration of a control structure containing a pointer to an anchor. The complete private header file is shown in **Figure 7-5**. Next let's discuss the details of just a couple of the QUE module methods.

```
#ifndef QUEP_H
#define QUEP_H

#include <que.h>
#include <enq.h>

typedef struct que__control_s
{
    ENQ_ANCHOR_p_t  anchor;
} QUE__CONTROL_t, *QUE__CONTROL_p_t;

#endif
```

Figure 7-5 QUE Module Private Header File

7.3.1 QUE_create_queue

This method will create a new, empty queue, and return to the user an ID to use in future operations.

```
Synopsis:
    QUE_ID_t QUE_create_queue( const char  *name );
Where:
    name == queue name; may be NULL
Exceptions:
    Throws SIGABRT if the queue cannot be created
Returns:
    Queue ID
Notes:
    None
```

Here is the implementation of this method:

```
QUE_ID_t QUE_create_queue( const char *name )
{
    QUE__CONTROL_p_t    qid = CDA_NEW( QUE__CONTROL_t );

    qid->anchor = ENQ_create_list( name );
    return qid;
}
```

7.3.2 QUE_create_item

This method will create a queue item containing the user's data.

```
Synopsis:
    QUE_ITEM_p_t QUE_create_item( const char *name, void *data );
Where:
    name == item name; may be NULL
    data == user's data
Exceptions:
    Throws SIGABRT if the item cannot be created
Returns:
    Address of queue item
Notes:
    None
```

Here is the implementation of this method:

```
QUE_ITEM_p_t QUE_create_item( const char *name, void *data )
{
    QUE_ITEM_p_t    item    =
       (QUE_ITEM_p_t)ENQ_create_item( name, sizeof(QUE_ITEM_t) );

    item->data = data;
    return item;
}
```

7.3.3 QUE_clear_queue

This method will destroy all the items in a queue, leaving the queue empty.

```
Synopsis:
    QUE_ID_t QUE_clear_queue( QUE_ID_t             qid,
                              QUE_DESTROY_PROC_p_t destroyProc
                            );
Where:
    qid         == ID of queue to clear
    destroyProc == address of destroyProc; may be NULL
Exceptions:
    None
Returns:
    Queue ID
Notes:
    If the data contained in the queue items requires cleanup, the
    user should pass the address of a clean up function as the
    destroy proc argument. If non-NULL, the clean up function will
```

```
        be called once for each item in the queue, passing the data
        from the item.
```

Here is the implementation of this method:

```
QUE_ID_t QUE_clear_queue( QUE_ID_t             qid,
                          QUE_DESTROY_PROC_p_t destroyProc
                        )
{
    QUE_ITEM_p_t    item    = NULL;

    while ( !ENQ_is_list_empty( qid->anchor ) )
    {
        item = (QUE_ITEM_p_t)ENQ_GET_HEAD( qid->anchor );
        if ( destroyProc != NULL )
            destroyProc( item->data );
        QUE_destroy_item( item );
    }

    return qid;
}
```

7.3.4 Other QUE Module Methods

Here is a quick synopsis of the other QUE module methods; the implementation is left to the reader.

- `QUE_ITEM_p_t QUE_append( QUE_ID_t queue, QUE_ITEM_p_t item )`
  appends *item* to *queue->anchor* using *ENQ_add_tail*; *item* is returned.
- `CDA_BOOL_t QUE_is_queue_empty( QUE_ID_t queue )`
  tests *queue->anchor* using *ENQ_is_list_empty* and returns the result.
- `CDA_BOOL_t QUE_is_queue_full( QUE_ID_t queue )`
  always returns *CDA_FALSE*.
- `QUE_ITEM_p_t QUE_destroy_item( QUE_ITEM_p_t item )`
  uses *ENQ_destroy_item* to destroy *item;* returns NULL.
- `QUE_ID_t QUE_destroy_queue( QUE_ID_t qid, QUE_DESTROY_PROC_p_t destroyProc )`
  calls *QUE_clear_queue* to empty the queue, then destroys *qid->anchor* using *ENQ_destroy_list* and frees *qid* using *CDA_free;* returns QUE_NULL_ID.
- `QUE_ITEM_p_t QUE_remove( QUE_ID_t queue )`
  removes and returns the first item in the queue using *ENQ_deq_head*; returns NULL if the queue is empty.
- `QUE_ITEM_p_t QUE_retrieve( QUE_ID_t queue )`
  returns, without removing, the first item in the queue using *ENQ_GET_HEAD*; returns NULL if the queue is empty.
- `QUE_ID_t QUE_traverse_queue( QUE_ID_t queue, QUE_TRAVERSE_PROC_p_t traverse_proc )`
  traverses the queue using *ENQ_GET_HEAD* and *ENQ_GET_NEXT;* for each *item* in the queue, calls *traverse_proc* passing *item->data*.

7.3.5 QUE Module Sample Program

Figure 7-6 shows a sample program that uses the QUE module. It doesn't do anything particularly interesting; it merely demonstrates how the QUE module methods are used.

7.4 Simple Priority Queues

Next we want to define a *simple priority queue*, and design a module to encapsulate one. For our purposes, a simple priority queue implementation consists of the following operations:

1. Create a queue;
2. Create a queue item;
3. Determine if a queue is empty;
4. Add a new item to a queue;
5. Remove the item at the front of a queue;
6. Empty a queue;
7. Destroy a queue item; and
8. Destroy a queue.

In the interest of encapsulation, we will also consider two additional methods, called *accessor methods* because they access a field in a structure:

9. Get the priority associated with an item; and
10. Get the data associated with an item.

As with queues, we now have to choose a name for our priority queue module, decide how to encapsulate the ID of a priority queue, declare the types of any callback functions, and determine the type of an element that will populate a priority queue. Our module name will be PRQ, and an ID will once again be an incomplete declaration of a pointer to a control structure; we will also need to declare a value to use as a NULL ID. We will have one callback function for use with the *empty* and *destroy* methods.

```
#include <que.h>
#include <stdio.h>
#include <stdlib.h>

static QUE_TRAVERSE_PROC_t  traverse;
static QUE_DESTROY_PROC_t   destroy;

static const char *itemNames_[] =
{ "hydrogen", "helium",   "lithium",  "beryllium",  "boron",
  "carbon",   "nitrogen", "oxygen",   "fluorine",   "neon"
};

int main( int argc, char **argv )
{
    int             *data   = NULL;
    QUE_ID_t        qid     = QUE_NULL_ID;
    QUE_ITEM_p_t    item    = NULL;
    int             inx     = 0;

    qid     = QUE_create_queue( "test" );
    for ( inx = 0 ; inx < CDA_CARD( itemNames_ ) ; ++inx )
    {
        data = CDA_malloc( sizeof(int) );
        *data = inx;
        item = QUE_create_item( itemNames_[inx], data );
        QUE_append( qid, item );
    }

    QUE_traverse_queue( qid, traverse );
    for ( inx = 0 ; inx < CDA_CARD( itemNames_ ) / 2 ; ++inx )
    {
        item = QUE_remove( qid );
        printf( "Removing %s\n", item->item.name );
        CDA_free( item->data );
        QUE_destroy_item( item );
    }

    QUE_destroy_queue( qid, destroy );
    return EXIT_SUCCESS;
}

static void traverse( void *data )
{
    int *iData    = data;
    printf( "Traversing %d\n", *iData );
}

static void destroy( void *data )
{
    int *iData    = data;
    printf( "Destroying %d\n", *iData );
    CDA_free( iData );
}
```

Figure 7-6 Sample QUE Module Usage

As with our queue module, a member of the queue will be an enqueuable item within which user data will be stored as a void*; plus we will need to store an integer value representing the priority of the queue item. The public declarations for our PRQ module are shown in **Figure 7-7**.

```
#define PRQ_NULL_ID (NULL)

typedef void PRQ_DESTROY_PROC_t( void *data );
typedef PRQ_DESTROY_PROC_t *PRQ_DESTROY_PROC_p_t;

typedef struct prq__control_s *PRQ_ID_t;
typedef struct prq_item_s
{
    ENQ_ITEM_t   enq_item;
    void         *data;
    CDA_UINT32_t priority;
} PRQ_ITEM_t, *PRQ_ITEM_p_t;
```

Figure 7-7 PRQ Module Public Declarations

Like the queue module, choosing an enqueuable item as type of element to populate the priority queue strongly suggests that there will be an ENQ-style list at the heart of our implementation, and our private declarations are going to strongly resemble the QUE module private declarations. However, for reasons that will become clear later, we also want to introduce the idea of a *maximum priority*. This will be an integer value defining an upper limit on the priority of an item in a given priority queue. Therefore the control structure for our PRQ implementation will contain both a pointer to an ENQ anchor, plus an integer field for storing the maximum priority. The complete private header file for our simple priority queue implementation is shown in **Figure 7-8**.

```
#ifndef PRQP_H
#define PRQP_H

#include <prq.h>
#include <enq.h>

typedef struct prq__control_s
{
      ENQ_ANCHOR_p_t anchor;
      CDA_UINT32_t   max_priority;
} PRQ__CONTROL_t, *PRQ__CONTROL_p_t;

#endif
```

Figure 7-8 Simple PRQ Module Private Header File

Now let's look at the method specification for our simple priority queue. After that we'll see an example of using a priority queue, then examine the simple priority queue implementation.

7.4.1 PRQ_create_priority_queue

This function will create a priority queue, and return the ID of the queue to be used in all subsequent priority queue operations. The caller names the priority queue, and provides the maximum priority for the queue as an unsigned integer. The maximum priority is

saved and checked on subsequent operations, but is not otherwise used in the current implementation (it is for possible future use).

```
Synopsis:
    PRQ_ID_t PRQ_create_queue( const char   *name,
                               CDA_UINT32_t max_priority
                             );
Where:
    name         -> queue name
    max_priority == maximum priority supported by queue
Returns:
    queue id
Exceptions:
    Throws SIGABRT if queue can't be created
Notes:
    max_priority is stored and checked on subsequent operations,
    but is not otherwise used at this time.
```

7.4.2 PRQ_create_item

This method will create an item that is a subclass of ENQ_ITEM_t, and that can be added to a priority queue. The item is created in an unenqueued state.

```
Synopsis:
    PRQ_ITEM_p_t PRQ_create_item( void         *value,
                                 CDA_UINT32_t priority
                               );
Where:
    value    -> value to be stored in the item
    priority == the priority of the item
Returns:
    Address of created item
Exceptions:
    Throws SIGABRT if item cannot be created
Notes:
    None
```

7.4.3 PRQ_is_queue_empty

This method will determine whether a priority queue is empty.

```
Synopsis:
    CDA_BOOL_t PRQ_is_queue_empty( PRQ_ID_t queue );
Where:
     queue == ID of the queue to test
Returns:
    CDA_TRUE if the queue is empty, CDA_FALSE otherwise
Exceptions:
    None
Notes:
    None
```

7.4.4 PRQ_add_item

This method will add an item to a priority queue.

```
Synopsis:
    PRQ_ITEM_p_t PRQ_add_item( PRQ_ID_t     queue,
                               PRQ_ITEM_p_t item
                             );
Where:
    queue == id of priority queue
    item  -> item to add
Returns:
    Address of enqueued item
Exceptions:
    Throws SIGABRT if the item's priority is higher than the
    maximum priority allowed for the queue.
Notes:
    None
```

7.4.5 PRQ_remove_item

This method removes and returns the highest priority item from a priority queue.

```
Synopsis:
    PRQ_ITEM_p_t PRQ_remove_item( PRQ_ID_t queue );
Where:
    queue == id of target priority queue
Returns:
    If priority queue is non-empty:
        The address of the highest priority item in the queue
    Otherwise:
        NULL
Exceptions:
    None
Notes:
    None
```

7.4.6 PRQ_GET_DATA

This is a macro that will return the user data contained in a priority item.

```
Synopsis:
    void *PRQ_GET_DATA( PRQ_ITEM_p_t item );
Where:
    item -> item from which to retrieve value
Returns:
    The value of item
Exceptions:
    None
Notes:
    None
```

7.4.7 PRQ_GET_PRIORITY

This is a macro that will return the priority of a priority item.

```
Synopsis:
    CDA_UINT32_t PRQ_GET_PRIORITY( PRQ_ITEM_p_t item );
Where:
    item -> item from which to retrieve priority
Returns:
    The priority of item
Exceptions:
    None
Notes:
    None
```

7.4.8 PRQ_destroy_item

This method will destroy a priority queue item.

```
Synopsis:
    PRQ_ITEM_p_t PRQ_destroy_item( PRQ_ITEM_p_t item );
Where:
    item -> item to destroy;
Returns:
    NULL
Exceptions:
    None
Notes:
    None
```

7.4.9 PRQ_empty_queue

This method will remove and destroy all items in a priority queue. The caller may optionally pass the address of a procedure to call prior to destroying the item; if specified, the value member of each destroyed item will be passed to this procedure.

```
Synopsis:
    PRQ_ID_t PRQ_empty_queue( PRQ_ID_t             queue,
                              PRQ_DESTROY_PROC_p_t destroy_proc
                            );
Where:
    queue        == id of queue to empty
    destroy_proc -> optional destroy callback procedure
Returns
    Queue ID
Exceptions:
    None
Notes:
    The caller may pass NULL for the destroy_proc parameter.
```

7.4.10 PRQ_destroy_queue

This method will destroy all items in a priority queue, and then destroy the queue. The caller may optionally pass the address of a procedure to call prior to destroying each item in the queue; if specified, the value of each destroyed item will be passed to this procedure.

```
Synopsis:
    PRQ_ID_t PRQ_destroy_queue( PRQ_ID_t             queue,
                               PRQ_DESTROY_PROC_p_t destroy_proc
                             );
Where:
    queue        == id of queue to destroy
    destroy_proc -> optional destroy callback procedure
Returns
    PRQ_NULL_ID
Exceptions:
    None
Notes:
    The caller may pass NULL for the destroy_proc parameter.
```

7.4.11 Priority Queue Example

This example depicts a module that maintains a queue of transactions awaiting execution. The principal functionality is contained in two independent methods, *TRANS_enq_transaction* and *TRANS_get_transaction*. In addition the module has an initialization method and a shutdown method. These four methods are discussed below, followed by the code for module.

TRANS_enq_transaction:

This method accepts data representing a transaction to execute, and a string representing the operation. It must enqueue the transaction for later execution, presumably when the appropriate system resources become available. The key idea here is that different types of operations are assigned different priorities; in particular, *add* transactions have the highest priority, *modify* transactions have a priority just below add transactions and *delete* transactions have the lowest priority. The method dynamically allocates memory in which to store the transaction data, and an enumerated value specifying the operation. *PRQ_create_item* is called specifying the dynamically allocated memory as the data, and a priority appropriate for the type of operation, and the item it creates is added to the queue.

TRANS_get_transaction:

This method removes from the queue an item containing a transaction to be executed. Assuming there is such an item, the method removes from the item the transaction data and the operation indicator; the transaction data will be the method's return data, and the operation is translated into a string which is returned via the *operation* parameter. Prior to returning, the PRQ item is destroyed. If the queue is empty, meaning that there are no pending transactions to execute, the method returns NULL.

TRANS_init:

This method is to be called once by the application prior to any calls to TRANS_enq_transaction or TRANS_get_transaction. It performs module initialization, which in this case simply means creating the transaction queue.

TRANS_shutdown:

This method is to be called by the application to terminate transaction processing. The queue is destroyed. Note that many items contain memory that was dynamically allocated by TRANS_enq_transaction, and now requires freeing. This is accomplished by passing to *PRQ_destroy_queue* the address of a destroy proc to perform the cleanup. (Note that the destroy proc in the example code contains a call to *printf* that you wouldn't normally see in production code; the logic is there to help make the example a little more meaningful to the reader.)

```
#include <stdio.h>
#include <cda.h>
#include <prq.h>
#include <trans.h>

#define DEL_PRI     (0)
#define MOD_PRI     (1)
#define ADD_PRI     (2)
#define MAX_PRI     (ADD_PRI)

typedef enum operation_e
{
    del,
    mod,
    add
} OPERATION_e_t;

typedef struct queue_data_s
{
    OPERATION_e_t   operation;
    void            *data;
} QUEUE_DATA_t, *QUEUE_DATA_p_t;

static PRQ_DESTROY_PROC_t destroyProc;

static PRQ_ID_t queue_  = PRQ_NULL_ID;

void TRANS_init( void )
{
    CDA_ASSERT( queue_ == PRQ_NULL_ID );
    queue_ = PRQ_create_queue( "TRANSACTION QUEUE", MAX_PRI );
}

void TRANS_shutdown( void )
{
    if ( queue_ != PRQ_NULL_ID )
        queue_ = PRQ_destroy_queue( queue_, destroyProc );
}

void TRANS_enq_transaction( void *data, const char *operation )
{
    CDA_UINT32_t    pri     = 0;
    QUEUE_DATA_p_t  qdata   = CDA_NEW( QUEUE_DATA_t );
    PRQ_ITEM_p_t    item    = NULL;
```

```
    qdata->data = data;
    if ( strcmp( operation, "delete" ) == 0 )
    {
        pri = DEL_PRI;
        qdata->operation = del;
    }
    else if ( strcmp( operation, "modify" ) == 0 )
    {
        pri = MOD_PRI;
        qdata->operation = mod;
    }
    else if ( strcmp( operation, "add" ) == 0 )
    {
        pri = ADD_PRI;
        qdata->operation = add;
    }
    else
        abort();

    item = PRQ_create_item( qdata, pri );
    PRQ_add_item( queue_, item );
}

void *TRANS_get_transaction( const char **operation )
{
    QUEUE_DATA_p_t  qdata   = NULL;
    void            *data   = NULL;
    PRQ_ITEM_p_t    item    = PRQ_remove_item( queue_ );

    if ( item != NULL )
    {
        qdata = PRQ_GET_DATA( item );
        data = qdata->data;
        switch ( qdata->operation )
        {
            case del:
                *operation = "delete";
                break;
            case mod:
                *operation = "modify";
                break;
            case add:
                *operation = "add";
                break;
            default:
                assert( CDA_FALSE );
                break;
        }

        CDA_free( qdata );
        PRQ_destroy_item( item );
    }

    return data;
}

static void destroyProc( void *data )
```

```
{
    QUEUE_DATA_p_t qdata    = data;
    printf( "Destroying transaction type %d\n", qdata->operation
);
    CDA_free( qdata );
}
```

7.4.12 Simple Priority Queue Module Implementation

The code for the simple implementation is shown below, except for the details of the create method, which will be an exercise for the reader. Note that, thanks to careful selection of the mechanism for implementing the enabling doubly linked module, the code is quite straightforward.

```
#include <stdlib.h>
#include <cda.h>
#include <enq.h>
#include <prqp.h>

PRQ_ID_t PRQ_create_queue( const char   *name,
                          CDA_UINT32_t max_priority
                        )
{
    . . .
}

PRQ_ITEM_p_t PRQ_create_item( void *data, CDA_UINT32_t priority )
{
    PRQ_ITEM_p_t item = NULL;

    item = (PRQ_ITEM_p_t)ENQ_create_item( NULL,
                                          sizeof(PRQ_ITEM_t)
                                        );
    item->data = data;
    item->priority = priority;

    return item;
}

PRQ_ITEM_p_t PRQ_add_item( PRQ_ID_t queue, PRQ_ITEM_p_t item )
{
    CDA_BOOL_t     found  = CDA_FALSE;
    PRQ_ITEM_p_t   temp   = NULL;
    ENQ_ANCHOR_p_t anchor = queue->anchor;

    if ( item->priority > queue->max_priority )
        abort();

    temp = (PRQ_ITEM_p_t)ENQ_GET_HEAD( anchor );
    while ( !found && temp != NULL )
       if ( temp == (PRQ_ITEM_p_t)anchor )
           temp = NULL;
       else if ( temp->priority < item->priority )
           found = CDA_TRUE;
       else
          temp = (PRQ_ITEM_p_t)ENQ_GET_NEXT( (ENQ_ITEM_p_t)temp );
```

```
    if ( temp != NULL )
        ENQ_add_before( (ENQ_ITEM_p_t)item, (ENQ_ITEM_p_t)temp );
    else
        ENQ_add_tail( anchor, (ENQ_ITEM_p_t)item );

    return item;
}

PRQ_ITEM_p_t PRQ_remove_item( PRQ_ID_t queue )
{
    ENQ_ITEM_p_t    item    = ENQ_deq_head( queue->anchor );

    if ( item == queue->anchor )
        item = NULL;

    return (PRQ_ITEM_p_t)item;
}

PRQ_ITEM_p_t PRQ_destroy_item( PRQ_ITEM_p_t item )
{
    ENQ_destroy_item( (ENQ_ITEM_p_t)item );
    return NULL;
}

CDA_BOOL_t PRQ_is_queue_empty( PRQ_ID_t queue )
{
    CDA_BOOL_t  rcode   = ENQ_is_list_empty( queue->anchor );

    return rcode;
}

PRQ_ID_t PRQ_empty_queue( PRQ_ID_t             queue,
                          PRQ_DESTROY_PROC_p_t destroy_proc
                        )
{
    ENQ_ANCHOR_p_t anchor = queue->anchor;
    PRQ_ITEM_p_t   item   = NULL;

    while ( !ENQ_is_list_empty( anchor ) )
    {
        item = (PRQ_ITEM_p_t)ENQ_GET_HEAD( anchor );
        if ( destroy_proc != NULL )
            destroy_proc( item->data );
        ENQ_destroy_item( (ENQ_ITEM_p_t)item );
    }

    return queue;
}

PRQ_ID_t PRQ_destroy_queue( PRQ_ID_t             queue,
                            PRQ_DESTROY_PROC_p_t destroy_proc
                          )
{
    PRQ_empty_queue( queue, destroy_proc );
    ENQ_destroy_list( queue->anchor );
    CDA_free( queue );
```

```
    return PRQ_NULL_ID;
}
```

7.5 A More Robust Priority Queue Implementation

The simple priority queue implementation discussed above is suitable for many applications. But many other applications require additional functionality. Consider the following three scenarios, in which an operating system utilizes a priority queue to execute processes (that is, programs) of various priorities:

> **Scenario 1:** A priority 25 process has control of the CPU when it is *preempted* by a priority 30 process; this means that the priority 25 process must give up control of the CPU so that the priority 30 process can be run in its place. The priority 25 process will be returned to the priority queue to wait *at the head* of its priority class until such time as the priority 30 process is finished. Our simple priority queue implementation is unsuitable for this because it can only add items *to the tail* of a priority class.
>
> **Scenario 2:** A few priority 15 processes are dominating control of the CPU, meanwhile several priority 14 processes are languishing in the priority queue. To circumvent this possibly damaging situation the operating system will occasionally ignore the priority 15 processes, and grant control to the first waiting priority 14 process; to do this, it must be able to remove the item *at the head* of priority class 14.
>
> **Scenario 3:** The operating system allows an operator to cancel a process waiting in the priority queue. To do this it must be able to remove an item from the priority queue *regardless of its position*.

A more robust priority queue implementation might offer these additional public methods:

- Dequeue an arbitrary item from the queue;
- Enqueue an item at the head of its priority class; and
- Dequeue an item from the head of its priority class.

Implementing these additional methods require no changes to the public API that we defined for the simple implementation presented above. Some new methods have to be added to the API, and some changes need to be made to the module internals; they are the subjects of your next project. In addition, we want to consider an optimization of the implementation.

The scenarios cited above require the intervention of a part of an operating system called the *scheduler* because it schedules processes for execution. The scheduler is a critical component that runs virtually every time a process is executed; a slow scheduler will have a negative impact on nearly every task that your computer performs. It is therefore in our interest to make sure that the scheduler runs as quickly as possible. Consider the scheduler's interaction with the priority queue; from the point of view of the simple implementation that we examined in the last section, what is the most time consuming operation that it has to perform? That would be adding an item to the queue, because it

has to search the entire linked list to locate the end of the target priority class. If we keep this same architecture for the robust implementation we will have the same problem with locating the head of a priority class in order to execute the *enqueue-priority-class-head* and *dequeue-priority-class-head* operations. So to optimize the operation of our robust implementation we will borrow from the design of the VAX/VMS scheduler, originally designed by Digital Equipment Corporation, which eliminates searching for the head and tail of a priority class.

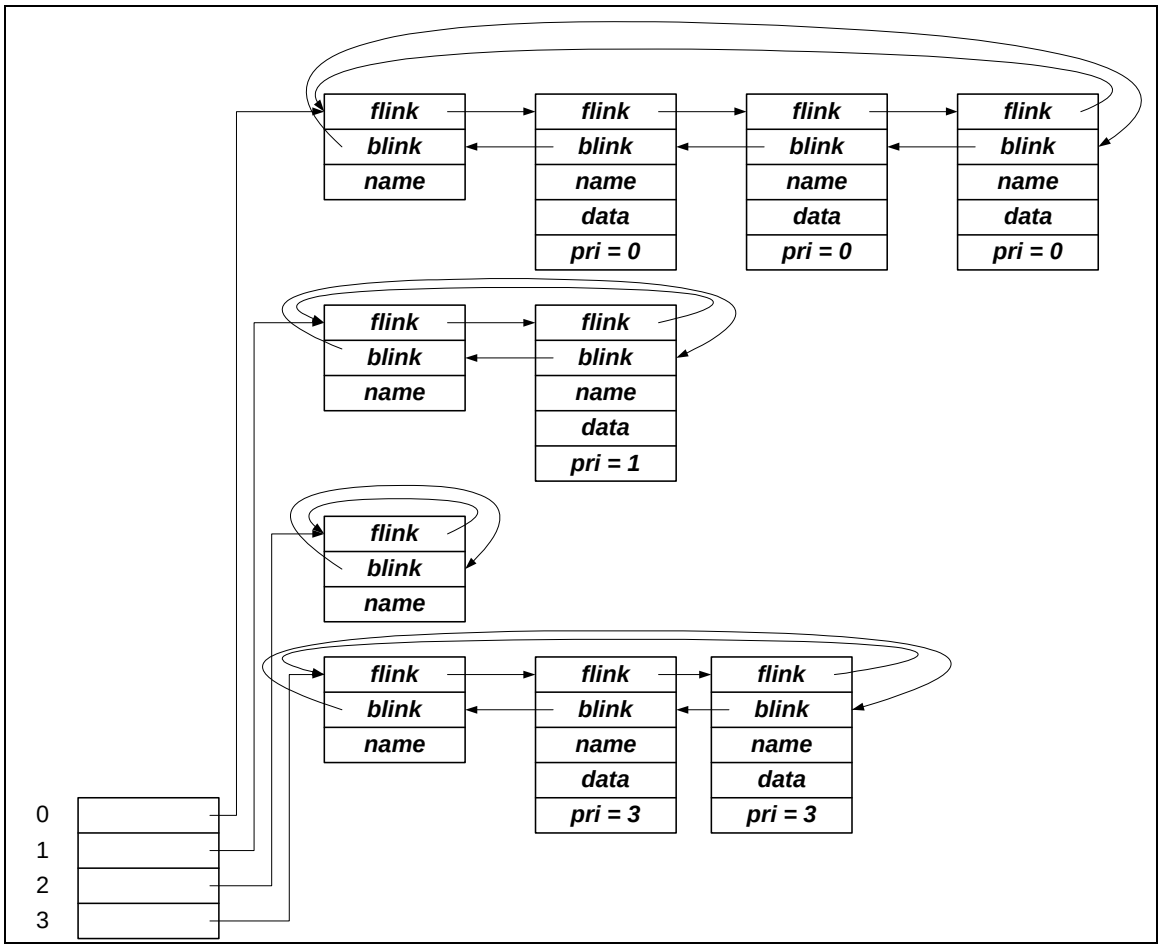


Figure 7-9 PRQ Optimized Structure

Refer to **Figure 7-9**. This depicts an implementation strategy that uses not one linked list, but an array of linked lists. The array contains one pointer-to-anchor for every priority class, and each list contains only items from a single priority class. Now to add an item to the head or tail of a priority class, we can use the priority as an index into the array to find the target list; once we have the target list, we merely add the item using ENQ_add_tail (for the normal *add* method) or ENQ_add_head (for the new *add-priority-class-head* method).

Note that our new implementation now runs faster, but at the expense of occupying additional space, and some added complexity of implementation. In the simple implementation we had to create one control structure, plus one list. In the robust implementation we will have to allocate a control structure, then allocate an array of pointers-to-anchors, then create a list for each element of the array. And to remove an

item from the queue, we must first locate the non-empty queue associated with the highest priority class and then remove its head.

8. The System Life Cycle

In this section we will examine the process of system development, called *the system life cycle*. This describes how a system is conceived, designed, constructed and maintained. We will also take a close look at the testing activities that are needed at each step of the system life cycle in order to ensure the success of the project.

8.1 Objectives

At the conclusion of this section you will be able to:

- Describe the components of the *system life cycle*; and
- Describe how *testing* fits into the system life cycle.

8.2 Overview

The steps in which a data processing system should be constructed are called the *system life cycle*. There are many different ways of viewing this process, and many different disciplines that define it, but they all contain these five essential activities:

1. Specification
2. Design
3. Implementation
4. Acceptance Testing
5. Maintenance

The process is said to be *iterative*. A problem found during the design step could lead to a reevaluation of the specification. A problem found during implementation could lead to a change in the system design, which could in turn lead to changes in the system specification.

Note that step four is dedicated to *testing*. This does not refer to the kind of testing that you, as a developer, perform in the course of writing your code. This is a special kind of activity that concentrates on proving the requirements of the system as established in the *specification* phase. In fact, as we'll see, testing activities are not confined to any one phase of the process, but take place at every step along the way.

The sections below will present an overview of each phase of the system life cycle. Then we'll go back and examine each phase a second time, concentrating on the testing activities that take place for that phase.

8.2.1 Specification Phase

This phase of the system life cycle will define and bound the problem to be solved, or process to be executed. No coding is done at this time. Only the functional requirements of the system are discussed: What is the system trying to accomplish? What data are to be captured? Once the data are captured, what questions will they answer, and what decisions will they help us to make?

During this step, a great deal of attention is paid to the *external specification* (in some disciplines, this is broken into a separate step). The external specification defines the characteristics of human interaction with the system. It defines the format of the screens (or graphical user interface) that will be used for data entry, and the reports that will be used to summarize data. It also defines the criteria that will determine whether or not the final implementation meets the requirements of the specification.

It is crucial that the system specification clearly delineates the bounds and characteristics of the system, and that it represents a concise agreement or *contract* between the system developers and system users. An incomplete or poorly documented specification invariably leads to a frustrating and endless implementation.

8.2.2 Design Phase

System design determines what data processing tools will be used or created to meet the system specification criteria. For example, hardware, programming languages and data base packages are selected. The project is broken into programs, the interaction between programs is defined, and the programs are broken into high-level modules. Rarely (and in poorly managed projects) is any coding performed during this step. In object-oriented systems, the major system objects are modeled; that is, a definition of the data contained in an object, and the interface to the object is determined.

Also during this step, project management documents the coding, documentation and implementation standards that system construction must follow. A code management system and quality assurance tools are selected, and guidelines for their use are documented in the implementation standard. Project personnel are designated to be responsible for enforcing conformance to the standards.

8.2.3 Implementation Phase

It is during implementation that actual construction of the system begins. The high level modules identified during system design are broken into submodules, and utilities modules are defined. Following the established coding standard, functions within modules are coded according to system design and assembled into individual programs.

8.2.4 Acceptance Testing Phase

As we'll see shortly, testing can be viewed from a variety of perspectives. In this context, we refer to the functional testing that determines whether the requirements of the system are met. Tests are tied directly to that portion of the system specification that documents the requirements that the system is intended to satisfy. When problems are uncovered, they are reported to the development organization for resolution.

It is important during this phase of the process that testing is carefully conducted and problems are carefully monitored. On all projects, test cases must be carefully documented, and agreed to by both the developers and the users. A test case management system can be extremely helpful in documenting test cases, and establishing dependencies between test cases. On all projects but the smallest, a defect tracking system should be used to track flaws and flaw resolution.

It is best if the implementation organization is not responsible for this phase of the system life cycle. If the end user is not sufficiently technical, or otherwise unable to do the testing, a third party should be employed.

8.2.5 Maintenance

This part of the system life cycle entails fixing flaws that are discovered in the system after deployment. Normally, fixes to the software aren't distributed before the next scheduled release of the system. Occasionally, fixes, or *field releases*, are dispatched immediately. Field releases are expensive and dangerous; if a thorough job is done during the specification and testing phases of the system life cycle, they will not be necessary.

Note that maintenance activity often takes place in parallel with development of a new system release. In this case, changes made by maintenance must also be integrated with the new system code.

8.3 Testing

The concept of *testing* goes far beyond the acceptance testing described in **Section 8.2.4**. Testing begins with the start of the system specification, and embraces every stage of the system life cycle, as discussed below.

8.3.1 Testing at the System Specification Level

Testing at the specification level mainly involves high-level documentation of the acceptance testing procedure, and the needs of the test team at the time of acceptance testing. A member of the test team should participate in all requirements definition activities. At this stage, the test team is responsible for pointing out when requirements conflict, or when requirements are essentially untestable. For example, a requirement that a system run without failing for five years is unrealistic, because it would take a minimum of five years to determine whether the requirement is met.

Immediately upon completion of the system specification, the test team should begin documenting specific test cases to be executed during acceptance testing.

8.3.2 Testing at the Design Level

In many ways, the design phase is to the implementation team what the specification phase is to the design team. Testing during this phase mainly entails documenting the acceptance criteria for the programs and modules that are its output. As a requirement of a program or module is tentatively defined, the question should be asked, Is the requirement testable? If the answer is no, the requirement should be modified. Occasionally at this time a prototype representing some small portion of the system may be developed in order to prove that a part of the design is feasible. Such a prototype is sometimes called a *proof of concept prototype*.

Upon completion of the design phase, the designers should have a high-level plan for determining whether the implementation conforms to the design, and should begin developing specific test cases for proving this.

Also during this time the test team should develop both high-level and detailed plans for conducting acceptance testing. Often acceptance testing will require specialized tools that must be either purchased or developed. If possible, the testers should coordinate the acquisition and use of such tools with the developers. The developers may find that these same tools are helpful in their own testing efforts, and the testers may request that the developers design their code in such a way as to make deployment of the tools more effective.

8.3.3 Testing at the Implementation Level

Testing during this phase is often called *unit testing*. This is a very broad term that can apply to any of the following types of testing.

Algorithm Testing
: This kind of testing entails proving that an algorithm is sound prior to coding it in a function or module. The proof may be mathematical in nature, or it may involve coding the algorithm into a test driver that will exercise it in detail.

Function Testing
: This kind of testing will prove that an individual function correctly validates its input, and produces the correct output. Sometimes a function is tested at the same time a module is tested; a symbolic debugger can be a valuable aid, if this is the case. Sometimes the function will first be coded into a test driver, which will exercise it independently of the module.

Module Testing
: This kind of testing proves that a module can correctly do its job. Exercising the program in which the module resides may test the module, but it can be more valuable to code the module into a test driver, first.

Program Testing
: This kind of testing proves that a program correctly validates its input, and produces the expected output.

In three items out of the above list, we discussed using *test drivers* to perform testing. This is also called *bench testing*. One of the advantages of bench testing is that it isolates a coding unit before testing it. To use an automotive example, suppose your car won't start, and you suspect the starter motor. It would be hard to test your theory while the starter was still in the car, because the alternator, battery, cables, or a variety of other things might be the source of the problem. So you take the starter out of the car, put in on a bench, and hook it up to a battery and cables that you know are working correctly. The battery and cables on the bench are directly analogous to a test driver.

Another advantage of bench testing is that it can more thoroughly be used to test a system component. Say you want to test a function's response to all possible illegal input. This would be difficult to do within the module and program that owns the function, because they are designed to always provide the function with valid data. A test driver, on the other hand, can be specially designed to feed the function any possible data, including invalid input.

In order to be able to do bench testing, the component that you want to test has to be easily removed from the program, module or function within which it resides. It is always a good idea to keep bench testing in mind as you design your code. In fact, it can help to keep this in mind during the system design phase, and sometimes even during the system specification phase.

Finally, an important part of the developer's job is to design and carefully document test cases that can be used to prove the proper execution of each module, and to assemble these test cases into a *module test plan*. The module test plan can be used throughout development to verify the module; later, the maintenance group will use it to verify changes that they make to your module.

8.3.4 Testing at the Acceptance Testing Level

The activities that we discussed in **Section 8.2.4** can be broken down into three types of testing:

> *Functional testing* is the execution specific test cases to prove a system requirement.
>
> *Retesting* is functional testing performed to prove a system requirement that was previously found to be unmet.
>
> *Regression testing* takes place after any change is made to the system. It is functional testing that proves that requirements that were met prior to the change continue to be met.

Regression testing may be the most difficult of the three activities. A *regression* is a flaw that is introduced when another flaw is corrected. The regression might occur in the neighborhood of the flaw that was fixed, but it often occurs in a separate module, or even a separate program (the possibility of introducing a regression outside of the module is greatly reduced by doing a good job designing modules with hidden implementations, and well defined APIs). When the development organization fixes a flaw uncovered by the test team, the test team must re-execute the test cases associated with the flaw. In addition, they must subject large portions of the system to *regression testing*. Ideally, every test case that had formerly passed will be re-executed. Since this is not always be feasible, the test team will often choose not to re-execute test cases judged not to be at all related to the flaw; for example, if the flaw that was fixed was a spelling error in a screen associated with the inventory data base, the test team might choose not to re-execute test cases against reports from the receivables data base.

8.3.5 Testing at the Maintenance Level

Testing at the maintenance level is a combination of testing strategies at the implementation and acceptance testing levels. A maintenance programmer who fixes a flaw will typically unit test the solution, then probably regression test the module it resides in. The maintenance group as a whole will (if they're doing their job) regression test the entire system on a weekly or monthly basis.

In order for the maintenance group to be able to do their job well, the development group must do a careful, thorough job of documenting their module test plans during

implementation. Regression testing tools can be an invaluable aid in automating regression tests, and improving the overall quality of the system. If specific regression testing tools are to be used, they should be identified during the design phase; test cases devised during implementation should be designed as much as possible to work with the selected tools.

9. Binary Trees

The next data structure we're going to examine is the *tree*. A tree is a versatile mechanism for solving many problems, including parsing and fast indexing. In this section we will define in detail the implementation of a special kind of tree, the *binary tree*. In the next section, we will learn how to use a binary tree to create a more general kind of structure called an *n-ary tree*.

9.1 Objectives

At the conclusion of this section you will be able to:

- Provide a formal definition of a binary tree;
- Construct a binary tree as an array or as a linked structure; and
- Use a binary tree as an index.

9.2 Overview

Conceptually, a binary tree is a structure that consists of zero or more linked *nodes*. In a minimal implementation, a node consists of exactly three elements: a *data pointer*, a *left pointer* and a *right pointer*. This concept is illustrated in **Figure 9-1**.

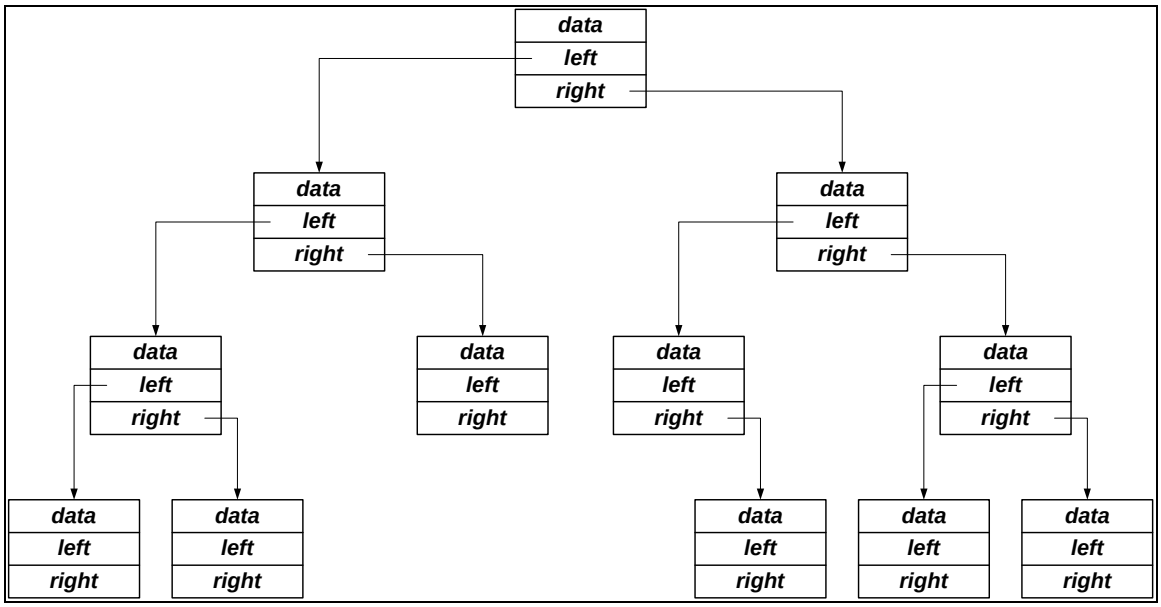


Figure 9-1 A Binary Tree

With respect to a binary tree we can define the following components and concepts (refer to **Figure 9-2** and **Figure 9-3**):

Node:

The basic component of a binary tree, comprising a left pointer, a right pointer and data.

Root Node:

The first node in the tree.

Child:

For a given node, another node that depends from the original node's left or right pointer. The child depending from the left pointer is referred to as the *left child* and the child depending from the right pointer is referred to as the *right child*. Any node in the tree may have zero, one or two children.

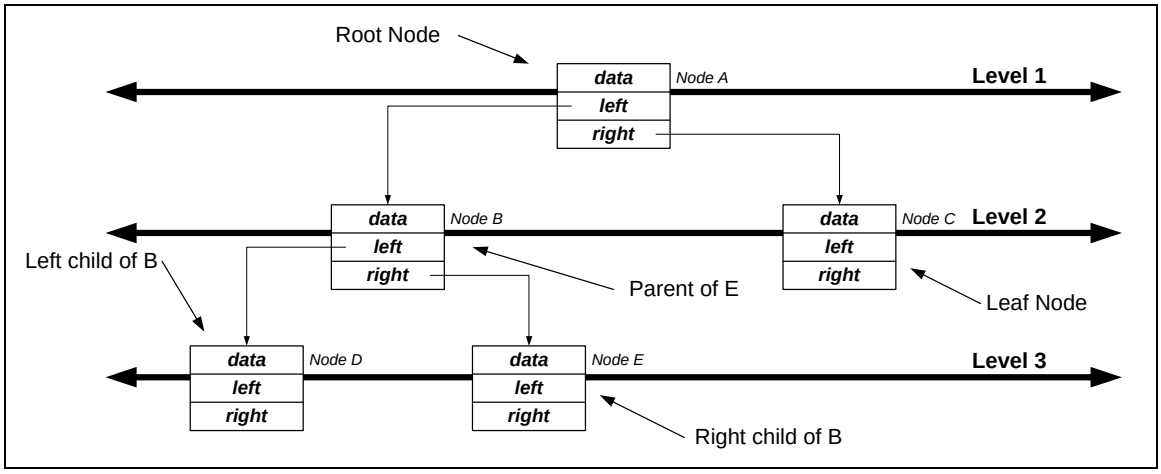


Figure 9-2 Binary Tree Concepts

Leaf Node:

A node with zero children.

Parent:

The node from which a given node depends. The root node of a binary tree has no parent; every other node has exactly one parent.

Empty Tree:

For our purposes, a binary tree will be called *empty* if it has no root node. (Note that in some implementations an *empty tree* might be a binary tree with only a root node.)

Level:

A relative position in the binary tree. The root node occupies *level 1* the children of the root occupy *level 2* and so on. In general, the children of a node that occupies *level n* themselves occupy *level n + 1*. The maximum number of nodes that can occupy *level N* is $2^{(N-1)}$.

Depth:

The maximum number of levels in a binary tree. This value determines the maximum capacity of a binary tree. If a binary tree has a depth of *N*, it can contain a maximum of $2^N - 1$ nodes.

Distance Between Two Nodes:

The distance between two nodes is the number of levels that must be crossed to traverse from one to the other. In **Figure 9-2**, the distance between nodes A and E is 2.

Balanced Binary Tree:

A binary tree in which the distance between the root node and any leaf differs by no more than one.

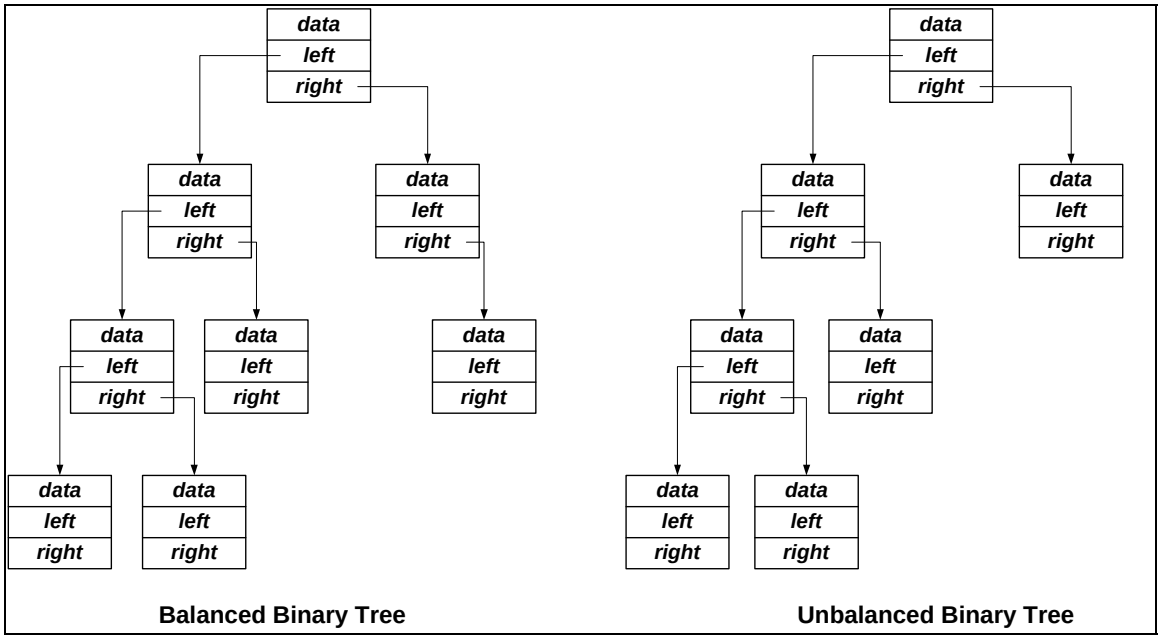


Figure 9-3 Balanced Binary Trees

Subtree:

Within a binary tree, a *subtree* is any node plus all of its descendants. The top node is called the *root* of the subtree. In this way a binary tree is said to be *recursively defined.*

9.3 Binary Tree Representation

A binary tree is most often represented in one of two ways:

- As a contiguous array
- As a dynamically linked structure

These two representations will be examined, below.

9.3.1 Contiguous Array Representation

A binary tree can be represented as an array only if the maximum depth of the tree is known in advance. The size of the array is then the maximum number of nodes in the array. The root node occupies the array element at index 0; the children of the root occupy the elements at indices 1 and 2. In general, the left and right children of the node at index *n* occupy array elements *2 * n + 1* and *2 * n + 2*, respectively. The parent of a node at index *m* occupies the array element at index *(m - 1) / 2*. **Figure 9-4** illustrates how a binary tree of depth 4 can be represented by an array.

9.3.2 Dynamically Linked Representation

Representing a binary tree as an array has some important applications, particularly in the area of persistent storage. However for large binary trees it has two disadvantages:

- If the binary tree is not full, as is usually the case, the array will contain a lot of wasted space.
- Once the tree becomes full, it cannot be extended without allocating a new array.

It is often preferable to dynamically allocate each node as it is required, and to keep pointers to the node's children within the node. This arrangement creates a tree more like that shown above, in **Figure 9-1**.

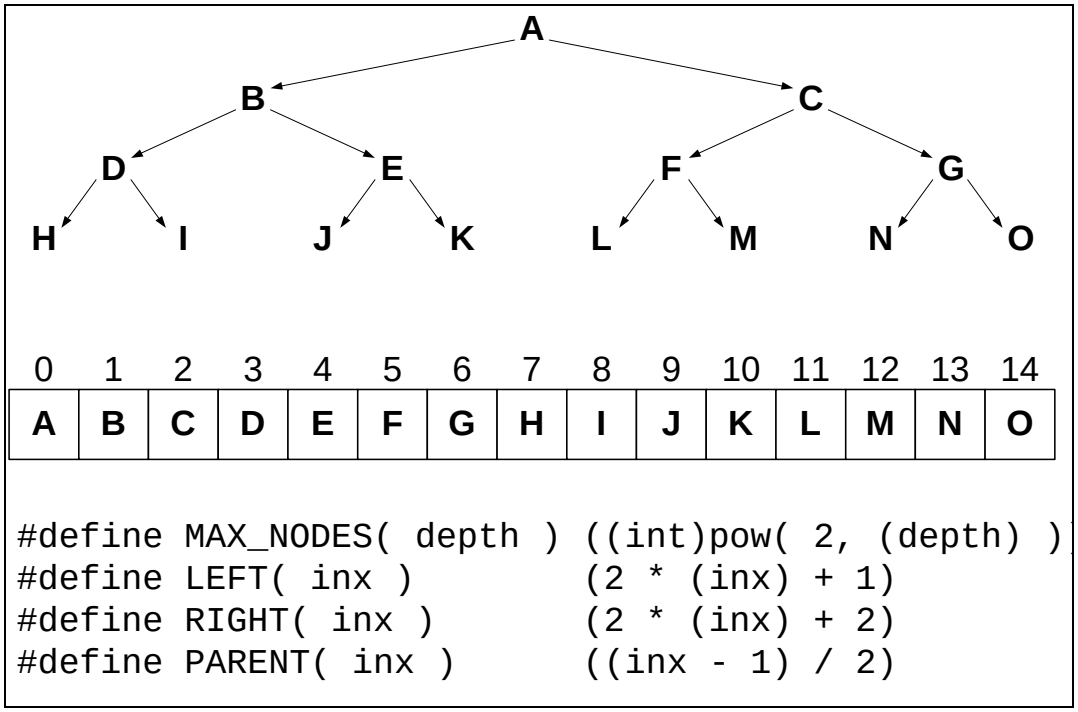


Figure 9-4 Representing a Binary Tree as an Array

9.4 A Minimal Binary Tree Implementation

Next we would like to examine a binary tree module implemented using a linked representation. Each node in the tree will be allocated as needed, and will consist of pointers for *user data*, *left child* and *right child*. The following operations will be defined:

- Create a binary tree
- Add a root node to a tree
- Add a left child to a node
- Add a right child to a node
- Get a tree's root node
- Get the data associated with a node
- Get a node's left child

- Get a node's right child
- Inquire whether a tree is empty
- Inquire whether a node is a leaf
- Traverse a binary tree
- Destroy a subtree
- Destroy a binary tree

9.4.1 Public Declarations

Our implementation will, of course, be modular. The name of our module will be BTREE. The two principal data types declared in our public header file will be incomplete data types that serve to identify a tree, and a node in a tree. There are two macros that can be used to represent a NULL tree ID, and a NULL node ID. And there are two callback proc declarations, one for destroying a tree or subtree, the other for traversing, or *visiting* a tree. There is also a declaration for a *traverse order* type that will be discussed later. All the public data type declarations are shown in **Figure 9-5**.

```
#define BTREE_NULL_ID       (NULL)
#define BTREE_NULL_NODE_ID  (NULL)

typedef void BTREE_DESTROY_PROC_t( void *data );
typedef BTREE_DESTROY_PROC_t *BTREE_DESTROY_PROC_p_t;

typedef void BTREE_VISIT_PROC_t( void *data );
typedef BTREE_VISIT_PROC_t *BTREE_VISIT_PROC_p_t;

typedef struct btree__control_s  *BTREE_ID_t;
typedef struct btree__node_s     *BTREE_NODE_ID_t;

typedef enum btree_traverse_order_e
{
    BTREE_PREORDER,
    BTREE_INORDER,
    BTREE_POSTORDER
} BTREE_TRAVERSE_ORDER_e_t;
```

Figure 9-5 BTREE Module Public Data Types

9.4.2 Private Declarations

As shown in **Figure 9-6**, our implementation will declare two private data structures: a control structure and a node structure. The address of the control structure will be the ID of the tree; the address of a node will be that node's ID. The control structure contains nothing more than a pointer to the root node. A node structure contains pointers for its left and right children, and for the user's data. We have also chosen to add pointers for the node's parent, and a pointer back to control structure in which the node resides; strictly speaking these two extra pointers are not required, however experience shows that they may be helpful in managing the implementation.

```
#ifndef BTREEP_H
#define BTREEP_H
```

```
#include <btree.h>

typedef struct btree__node_s      *BTREE__NODE_p_t;
typedef struct btree__control_s   *BTREE__CONTROL_p_t;

typedef struct btree__node_s
{
    void                *data;
    BTREE__CONTROL_p_t  tree;
    BTREE__NODE_p_t     parent;
    BTREE__NODE_p_t     left;
    BTREE__NODE_p_t     right;
} BTREE__NODE_t;


typedef struct btree__control_s
{
      BTREE__NODE_p_t root;
}  BTREE__CONTROL_t;

#endif
```

Figure 9-6 BTREE Private Header File

9.4.3 BTREE_create_tree

This method will create an empty binary tree and return its ID.

```
Synopsis:
      BTREE_ID_t BTREE_create_tree( void );
Returns:
      Binary tree ID
Exceptions:
      Throws SIGABRT if tree cannot be created
Notes:
      None
```

Here is the code for BTREE_create_tree:

```
BTREE_ID_t BTREE_create_tree( void )
{
    BTREE__CONTROL_p_t tree = CDA_NEW( BTREE__CONTROL_t );
    tree->root = NULL;

    return tree;
}
```

9.4.4 BTREE_add_root

This method will add the root node to a tree; the tree must not already have a root node.

```
Synopsis:
      BTREE_NODE_ID_t
      BTREE_add_root( BTREE_ID_t tree, void *data );
Where:
      tree == tree to which to add
      data -> user data to be stored with the node
```

```
Exceptions:
      Throws SIGABRT if node can't be created
Notes:
      Tree must be empty when this routine is called
```

Here is the code for this method:

```
BTREE_NODE_ID_t BTREE_add_root( BTREE_ID_t tree, void *data )
{
    BTREE__NODE_p_t node = CDA_NEW( BTREE__NODE_t );
    CDA_ASSERT( tree->root == NULL );

    node->data = data;
    node->tree = tree;
    node->parent = NULL;
    node->left = NULL;
    node->right = NULL;
    tree->root = node;

    return node;
}
```

9.4.5 BTREE_add_left

This method will add a left child to a node; the node must not already have a left child.

```
Synopsis:
      BTREE_NODE_ID_t
      BTREE_add_left( BTREE_NODE_ID_t node, void *data );
Where:
      node == id of node to which to add
      data -> user data to be associated with this node
Returns:
      ID of new node
Exceptions:
      Throws SIGABRT if node cannot be created
Notes:
      The node must not already have a left child
```

Here is the code for this method:

```
BTREE_NODE_ID_t BTREE_add_left( BTREE_NODE_ID_t node, void *data )
{
    BTREE__NODE_p_t left = CDA_NEW( BTREE__NODE_t );
    CDA_ASSERT( node->left == NULL );

    left->data = data;
    left->tree = node->tree;
    left->parent = node;
    left->left = NULL;
    left->right = NULL;
    node->left = left;

    return left;
}
```

9.4.6 BTREE_add_right

This method will add a right child to a node; the node must not already have a right child.

```
Synopsis:
    BTREE_NODE_ID_t
    BTREE_add_right( BTREE_NODE_ID_t node, void *data )
Where:
      node == id of node to which to add
      data -> user data to be associated with this node
Returns:
      ID of new node
Exceptions:
      Throws SIGABRT if node cannot be created
Notes:
      The node must not already have a right child
```

The code for this method is left as an exercise to the student.

9.4.7 BTREE_get_root

This method returns the root node of a tree.

```
Synopsis:
      BTREE_NODE_ID_t BTREE_get_root( BTREE_ID_t tree );
Where:
      tree == ID of tree to interrogate
Returns:
      ID of root node; BTREE_NODE_NULL if tree is empty
Exceptions:
      None
Notes:
      None
```

Here is the code for this method:

```
BTREE_NODE_ID_t BTREE_get_root( BTREE_ID_t tree )
{
    return tree->root;
}
```

9.4.8 BTREE_get_data, BTREE_get_left, BTREE_get_right

These three methods obtain the data, and left and right children, respectively, associated with a node.

```
Synopsis:
      void            *BTREE_get_data( BTREE_NODE_ID_t node );
      BTREE_NODE_ID_t BTREE_get_left( BTREE_NODE_ID_t node );
      BTREE_NODE_ID_t BTREE_get_right( BTREE_NODE_ID_t node );
Where:
      node == id of node to interrogate
Returns:
      The data, left or right child associated with the node;
      BTREE_get_left and BTREE_get_right return BTREE_NULL_NODE
      if there is no relevant child.
```

```
Exceptions:
      None
Notes:
      None
```

Here is the code for these methods:

```
void *BTREE_get_data( BTREE_NODE_ID_t node )
{
    return node->data;
}

BTREE_NODE_ID_t BTREE_get_left( BTREE_NODE_ID_t node )
{
    return node->left;
}

BTREE_NODE_ID_t BTREE_get_right( BTREE_NODE_ID_t node )
{
    return node->right;
}
```

9.4.9 BTREE_is_empty

This method returns true if a tree is empty. Recall that, according to our definition, a tree is empty if it has no root node.

```
Synopsis:
      CDA_BOOL_t BTREE_is_empty( BTREE_ID_t tree );
Where:
      tree == id of tree to test
Returns:
      CDA_TRUE if tree is empty,
      CDA_FALSE otherwise
Exceptions:
      None
Notes:
      None
```

The code for this method is left as an exercise to the student.

9.4.10 BTREE_is_leaf

This method returns true if a node is a leaf. Recall that, according to our definition, a node is a leaf if it has no children.

```
Synopsis:
      CDA_BOOL_t BTREE_is_leaf( BTREE_NODE_ID_t node );
Where:
      node == id of node to test
Returns:
      CDA_TRUE if node is a leaf,
      CDA_FALSE otherwise
Exceptions:
      None
```

```
Notes:
      None
```

The code for this method is left as an exercise to the student.

9.4.11 BTREE_traverse_tree

A binary tree is traversed by examining, or *visiting*, every node in the tree in some order. In our implementation, there are three possible orders of traversal:

- Inorder traversal
- Preorder traversal
- Postorder traversal

A fourth possible traversal, *level traversal*, is discussed in your textbook. The details of binary tree traversal will be discussed later. Recall that BTREE_VISIT_PROC_p_t is declared as follows:

```
typedef void BTREE_VISIT_PROC_t( void *data );
typedef BTREE_VISIT_PROC_t *BTREE_VISIT_PROC_p_t;
Synopsis:
     BTREE_ID_t
     BTREE_traverse_tree( BTREE_ID_t               tree,
                          BTREE_TRAVERSE_ORDER_e_t order,
                          BTREE_VISIT_PROC_p_t     visit_proc
                        );
Where:
      tree       == id of tree to traverse
      order      == order in which to traverse tree
      visit_proc -> user proc to call each time a node is visited
Returns:
      tree
Exceptions:
      None
Notes:
1. Order may be BTREE_INORDER, BTREE_PREORDER or
   BTREE_POSTORDER.
2. Each time visit_proc is called the data associated with the
   node is passed.
```

9.4.12 BTREE_destroy_tree, BTREE_destroy_subtree

These methods will destroy a tree or subtree, respectively. Each node in the tree or subtree is visited and freed. Prior to destroying a node, the caller is given the opportunity to free resources associated with a node's data via an optional destroy procedure. Recall that BTREE_DESTROY_PROC_p_t is declared as follows:

```
typedef void BTREE_DESTROY_PROC_t( void *data );
typedef BTREE_DESTROY_PROC_t *BTREE_DESTROY_PROC_p_t;
Synopsis:
  BTREE_NODE_ID_t
  BTREE_destroy_subtree( BTREE_NODE_ID_t        node,
                         BTREE_DESTROY_PROC_p_t destroy_proc
                       );
```

```
    BTREE_ID_t
    BTREE_destroy_tree( BTREE_ID_t             tree,
                        BTREE_DESTROY_PROC_p_t destroy_proc
                      );
Where:
      tree         == id of tree to destroy
      node         == root of subtree to destroy
      destroy_proc -> proc to call prior to node destruction
Returns:
      BTREE_destroy_subtree returns BTREE_NULL_NODE_ID
      BREE_destroy_tree returns BTREE_NULL_ID
Exceptions:
      None
Notes:
      NULL may be passed in place of destroy_proc
```

The method to destroy a subtree is a recursive procedure that destroys all left children, followed by all right children, followed by the node itself. Note that the parent of the node must be updated, and a special check must be made to determine whether the root of the tree has been destroyed. Here is the code for the method:

```
BTREE_NODE_ID_t
BTREE_destroy_subtree( BTREE_NODE_ID_t        node,
                       BTREE_DESTROY_PROC_p_t destroy_proc
                     )
{
    if( node->left != NULL )
        BTREE_destroy_subtree( node->left,  destroy_proc );

    if( node->right != NULL )
        BTREE_destroy_subtree( node->right, destroy_proc );

    if( destroy_proc != NULL )
        destroy_proc( node->data );

    if( node == node->tree->root )
        node->tree->root = NULL;
    else if ( node == node->parent->left )
        node->parent->left = NULL;
    else if ( node == node->parent->right )
        node->parent->right = NULL;
    else
        CDA_ASSERT( CDA_FALSE );

    CDA_free( node );
    return BTREE_NULL_NODE_ID;
}
```

The method to destroy a tree merely destroys the subtree represented by the root, and then frees the control structure:

```
BTREE_ID_t
BTREE_destroy_tree( BTREE_ID_t             tree,
                    BTREE_DESTROY_PROC_p_t destroy_proc
                  )
{
```

```
        BTREE_destroy_subtree( tree->root, destroy_proc );
        CDA_free( tree );
        return BTREE_NULL_ID;
    }
```

9.5 Using a Binary Tree as an Index

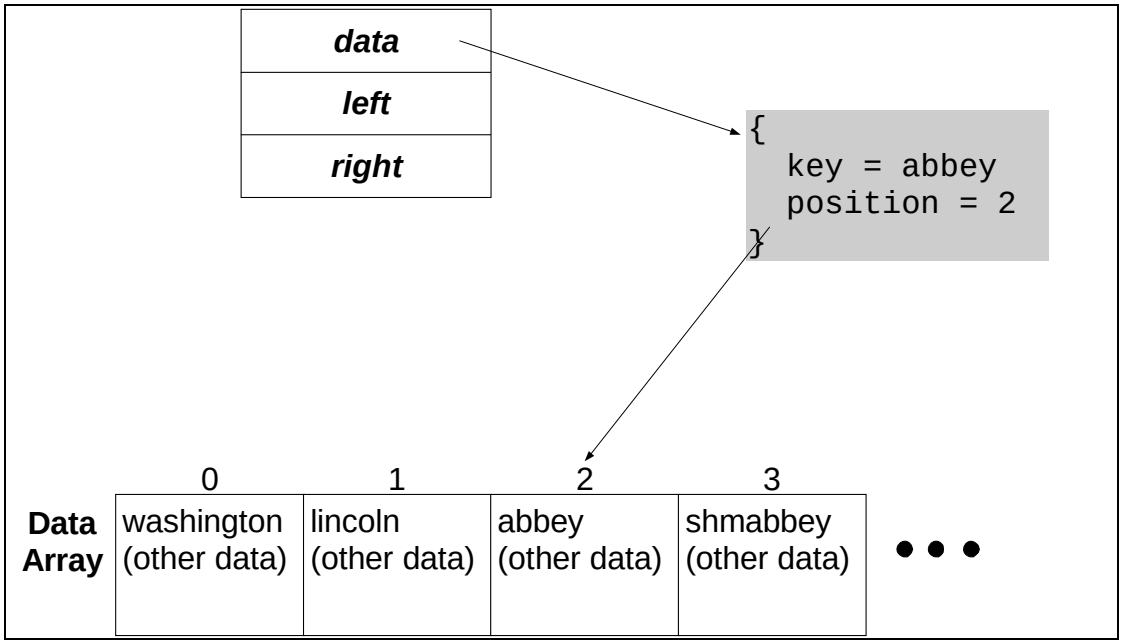


Figure 9-7 Indexing an Array with a Binary Tree

One of the more straightforward uses of a binary tree is as a *sorted index* to a data structure. Suppose we have a list of records stored in an array, and each record is identified by a unique name called a *key*. An example would be an array of student records keyed by student name. Each record can be quickly found by using its array index, but you typically want to access the record for a particular name. To be able to quickly look up a record by name, we could create a *lookup index* for the array in the form of a binary tree. Each node in the binary tree would contain the name of a record in the array, plus the array index of the record. This arrangement is illustrated in **Figure 9-7**.

```
typedef struct data_s
{
    char *key;
    int  position;
} DATA_t, *DATA_p_t;

char *recs[] = { . . . }
if ( num-recs > 0 )
    tree = BTREE_create_tree()
    data = CDA_NEW( DATA_t )
    data->key = CDA_NEW_STR( recs[0] )
    data->position = 0
    root = BTREE_add_root( tree, data )
    for ( inx = 1 ; inx < num-recs ; ++inx )
        data = CDA_NEW( DATA_t )
        data->key = CDA_NEW_STR( recs[inx] )
```

```
        data->position = inx
        add( data, root )
```

Figure 9-8 Pseudocode for Start of Index Creation

Here is how we go about building the lookup index (refer to **Figure 9-8**). First we declare a key record, *DATA_t*, capable of holding a key and an array index. Now for each record in the array, allocate and add a new key record to a binary tree. The key record for array index 0 will be stored in the root of the tree. Additional key records will be added to the tree by descending through the tree comparing the key to be added to the key stored in an existing node. Each time the new key is found to be less than an existing key, descend to the left; if the new key is greater than the existing key descend to the right. When you attempt to descend and find that the node you are descending from does not have the target child you're done descending, and add the key as the target child.

```
add( DATA_p_t data, BTREE_NODE_t node )
    node_data = BTREE_get_data( node )
    strc = strcmp( data->key, node_data->key )
    CDA_ASSERT( strc != 0 )

    if ( strc < 0 )
        next_node = BTREE_get_left( node )
        if ( next_node == BTREE_NULL_NODE )
            BTREE_add_left( node, data )
        else
            add( data, next_node )
    else
        next_node = BTREE_get_right( node )
        if ( next_node == BTREE_NULL_NODE )
            BTREE_add_right( node, data )
        else
            add( data, next_node )
```

Figure 9-9 Index Creation *Add* Function Pseudocode

Figure 9-9 shows the pseudocode for a recursive function to accomplish the add operation. **Figure 9-10** shows the result of using the recursive function to create an index using the following array of keys:

```
static const char *test_data[] =
{ "washington",   "lincoln",      "abbey",        "shmabbey",
  "gettysburg",   "alabama",      "zulu",         "yorkshire",
  "xerxes",       "wonderland",   "tiparary",     "sayville",
  "montana",      "eratosthenes", "pythagoras",   "aristotle",
  "kant"
};
```

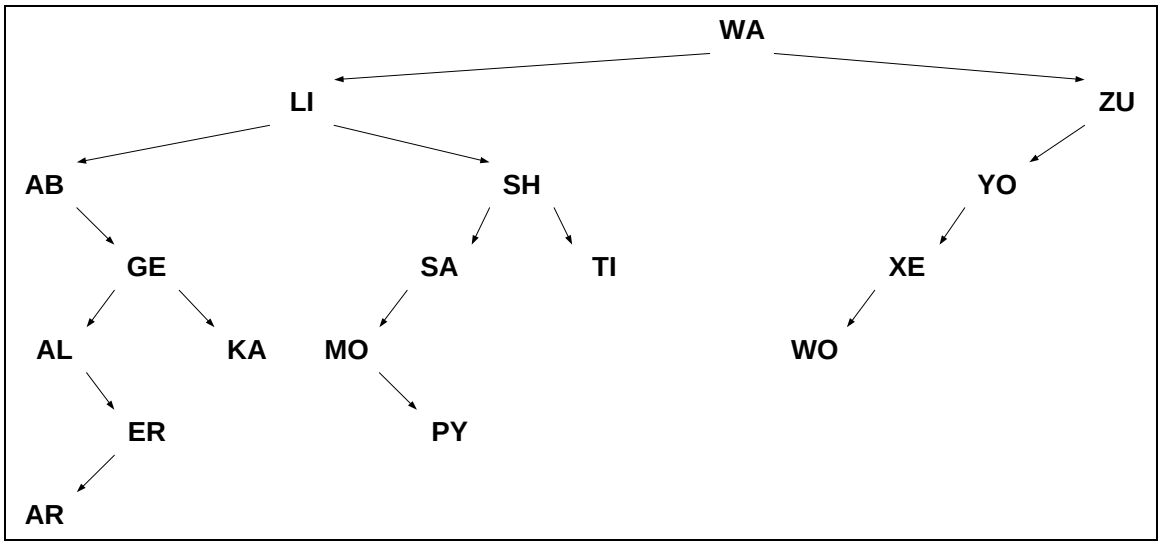


Figure 9-10 A Binary Tree Index

Now suppose you have a key, and you want to search the binary tree for it in order to determine the array position that the associated record occupies. You would follow a procedure similar to the one used to create the tree. Starting with the root, compare the key to the key stored in the node; if they are equal you're done, otherwise recursively branch left or right until you find the key you're looking for, or you reach a leaf; if you reach a leaf without finding the target key, it means the key isn't in the binary tree. The maximum number of comparisons that you have to perform is equal to the depth of the tree. The pseudocode for the search function is shown in **Figure 9-11**. **Section 9.6** contains sample code that demonstrates the creation and validation of an index; the results of the validation are shown in **Figure 9-13**.

Using a binary tree as an index can be a valuable tool, but before doing so you must analyze your data carefully. The index will be most efficient if keys tend to occur randomly; if they tend to occur in order, you will wind up with a very inefficient index like that shown in **Figure 9-12**. In cases where you must make certain your index is efficient you will have to resort to *balancing* your tree; this subject is discussed in your textbook.

```
root = BTREE_get_root( tree )
inx = get_position( key, root )
. . .
/* Return array position of key, or -1 if key not found */
int get_position( char *key, BTREE_NODE_t node )
    data = BTREE_get_data( node )
    strc = strcmp( key, data->key )
    if ( strc == 0 )
        rcode = data->position
    else if ( strc < 0 )
        next_node = BTREE_get_left( node )
        if ( next_node == BTREE_NULL_NODE )
            rcode = -1
        else
            rcode = get_position( key, next_node )
    else if ( strc > 0 )
        next_node = BTREE_get_right( node )
        if ( next_node = BTREE_NULL_NODE )
            rcode = -1
        else
            rcode = get_position( key, next_node )

    return rcode
```

Figure 9-11 Pseudocode for Searching a Binary Tree Index

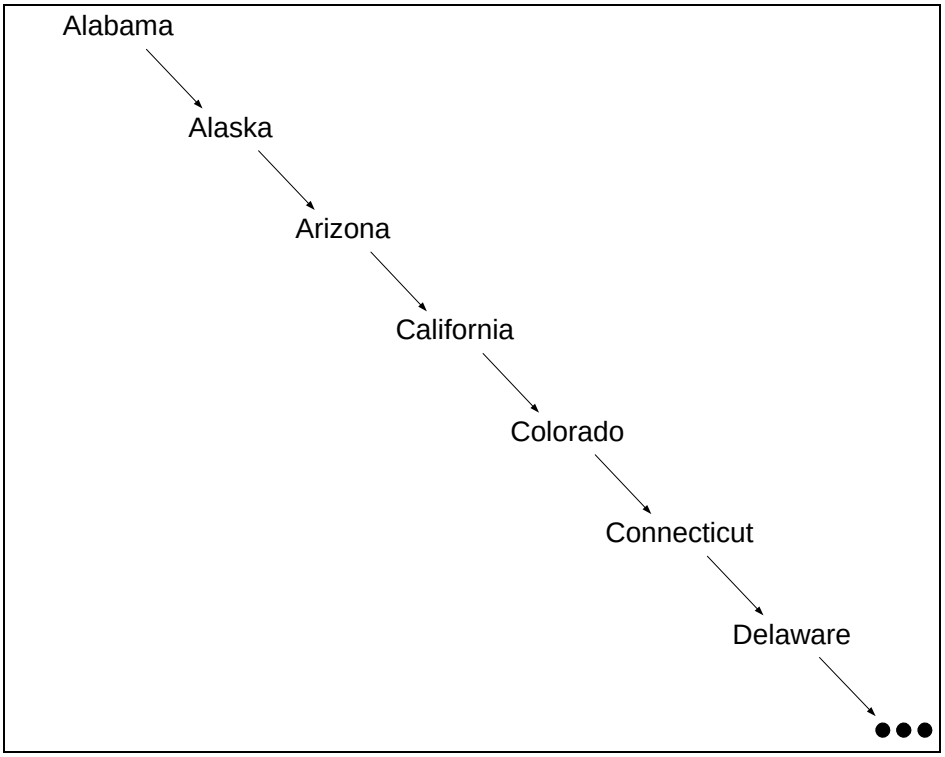


Figure 9-12 An Inefficient Binary Tree Index

9.6 Using a Binary Tree as an Index – Demonstration

```
#include <stdio.h>
#include <string.h>
#include <btree.h>
```

```
typedef struct data_s
{
    char *key;
    int  position;
} DATA_t, *DATA_p_t;

static void create_tree( const char **recs, int num_recs );
static void add( DATA_p_t data, BTREE_NODE_ID_t node );
static int get_from_btree( const char *key );
static int get_pos( const char *key, BTREE_NODE_ID_t node );

static const char *test_data[] =
{ "washington",   "lincoln",      "abbey",        "shmabbey",
  "gettysburg",   "alabama",      "zulu",         "yorkshire",
  "xerxes",       "wonderland",   "tiparary",     "sayville",
  "montana",      "eratosthenes", "pythagoras",   "aristotle",
  "kant"
};

static BTREE_ID_t   tree    = BTREE_NULL_ID;

int main( int argc, char **argv )
{
    int inx  = 0;
    int jnx  = 0;

    create_tree( test_data, CDA_CARD( test_data ) );
    for ( inx = 0 ; inx < CDA_CARD( test_data ) ; ++inx )
    {
        jnx = get_from_btree( test_data[inx] );
        if ( inx != jnx )
            printf( "%3d !!!", jnx );
        printf( "%3d, %s\n", inx, test_data[inx] );
    }

    return EXIT_SUCCESS;
}

static void create_tree( const char **recs, int num_recs )
{
    DATA_p_t        data = CDA_NEW( DATA_t );
    BTREE_NODE_ID_t root = BTREE_NULL_NODE_ID;
    int             inx  = 0;

    if ( num_recs > 0 )
    {
        data->key = CDA_NEW_STR( recs[0] );
        data->position = 0;
        tree = BTREE_create_tree();
        root = BTREE_add_root( tree, data );

        for ( inx = 1 ; inx < num_recs ; ++inx )
        {
            data = CDA_NEW( DATA_t );
            data->key = CDA_NEW_STR( recs[inx] );
            data->position = inx;
```

```
                add( data, root );
            }
        }
    }

    static void add( DATA_p_t data, BTREE_NODE_ID_t node )
    {
        DATA_p_t        node_data = BTREE_get_data( node );
        BTREE_NODE_ID_t next_node = BTREE_NULL_NODE_ID;
        int             strc      = 0;

        strc = strcmp( data->key, node_data->key );
        CDA_ASSERT( strc != 0 );

        if ( strc < 0 )
        {
            next_node = BTREE_get_left( node );
            if ( next_node == BTREE_NULL_NODE_ID )
                BTREE_add_left( node, data );
            else
                add( data, next_node );
        }
        else
        {
            next_node = BTREE_get_right( node );
            if ( next_node == BTREE_NULL_NODE_ID )
                BTREE_add_right( node, data );
            else
                add( data, next_node );
        }
    }

    static int get_from_btree( const char *key )
    {
        BTREE_NODE_ID_t root  = BTREE_get_root( tree );
        int             rcode = 0;

        rcode = get_pos( key, root );
        return rcode;
    }

    static int get_pos( const char *key, BTREE_NODE_ID_t node )
    {
        DATA_p_t        node_data = BTREE_get_data( node );
        BTREE_NODE_ID_t next_node = BTREE_NULL_NODE_ID;
        int             strc      = 0;
        int             rcode     = -1;

        strc = strcmp( key, node_data->key );
        if ( strc == 0 )
            rcode = node_data->position;
        else if ( strc < 0 )
        {
            next_node = BTREE_get_left( node );
            if ( next_node == BTREE_NULL_NODE_ID )
                ;
            else
```

```
                rcode = get_pos( key, next_node );
        }
        else
        {
            next_node = BTREE_get_right( node );
            if ( next_node == BTREE_NULL_NODE_ID )
                ;
            else
                rcode = get_pos( key, next_node );
        }

        return rcode;
}
```

```
  0, washington
  1, lincoln
  2, abbey
  3, shmabbey
  4, gettysburg
  5, alabama
  6, zulu
  7, yorkshire
  8, xerxes
  9, wonderland
 10, tiparary
 11, sayville
 12, montana
 13, eratosthenes
 14, pythagoras
 15, aristotle
 16, kant
```

Figure 9-13 Index Validation Results

9.7 Traversing a Binary Tree

As mentioned earlier, you *traverse* a binary tree by *visiting* each node in the tree in a designated order; in our implementation, the three possible orders are *inorder*, *preorder* and *postorder*. Our implementation has one public method, BTREE_traverse_tree that handles all three cases; it looks like this:

```
BTREE_ID_t
BTREE_traverse_tree( BTREE_ID_t               tree,
                     BTREE_TRAVERSE_ORDER_e_t order,
                     BTREE_VISIT_PROC_p_t     visit_proc
                   )
{
    switch ( order )
    {
        case BTREE_PREORDER:
            traverse_preorder( tree->root, visit_proc );
            break;

        case BTREE_INORDER:
            traverse_inorder( tree->root, visit_proc );
            break;

        case BTREE_POSTORDER:
```

```
                traverse_postorder( tree->root, visit_proc );
                break;

            default:
                CDA_ASSERT( CDA_FALSE );
                break;
        }

        return tree;
    }
```

Each of the three subroutines will be a recursive procedure that traverses the tree in the indicated order, and calls *visit_proc* each time a node is visited, passing the node's data. The cases are discussed individually, below.

9.7.1 Inorder Traversal

Inorder traversal involves following a tree to the left until a null child is reached, at which time the parent is visited. Traversal continues recursively with the left branch of the parent's right child. The pseudocode for this procedure is shown in **Figure 9-14**.

```
void inorder( BTREE__NODE_p_t node )
    if ( node != NULL )
        inorder( node->left )
        visit( node )
        inorder( node->right )
```

Figure 9-14 Inorder Traversal Pseudocode

Here is a code sample that uses inorder traversal to traverse the binary tree index created in **Section 9.6**. The results of the traversal are shown in **Figure 9-15**.

```
static BTREE_VISIT_PROC_t   visit_proc;
static void traverse_index_inorder( BTREE_t tree )
{
    BTREE_traverse_tree( tree, BTREE_INORDER, visit_proc );
}

static void visit_proc( void *visit_data )
{
    DATA_p_t data = visit_data;

    printf( "%3d, %s\n", data->position, data->key );
}
```

9.7.2 Preorder Traversal

In a preorder traversal, a node is visited, and then all nodes along the node's left branch are recursively visited. When a null node is encountered, the nodes along the left branch of the parent's right child are recursively visited. The pseudocode for a preorder traversal is shown in **Figure 9-16**.

```
  2, abbey
  5, alabama
 15, aristotle
 13, eratosthenes
  4, gettysburg
```

```
 16, kant
  1, lincoln
 12, montana
 14, pythagoras
 11, sayville
  3, shmabbey
 10, tiparary
  0, washington
  9, wonderland
  8, xerxes
  7, yorkshire
  6, zulu
```

Figure 9-15 Inorder Traversal Results

```
void preorder( BTREE__NODE_p_t node )
    if ( node != NULL )
        visit( node )
        preorder( node->left )
        preorder( node->right )
```

Figure 9-16 Preorder Traversal Pseudocode

If we substitute a preorder traversal for the inorder traversal in the example in **Section 9.7.1** we will get the results shown in **Figure 9-17**. Of course, this isn't a very good example of the use of a preorder traversal. A better example would be the use of binary trees to parse prefix expressions; this subject is discussed in your textbook.

```
  0, washington
  1, lincoln
  2, abbey
  4, gettysburg
  5, alabama
 13, eratosthenes
 15, aristotle
 16, kant
  3, shmabbey
 11, sayville
 12, montana
 14, pythagoras
 10, tiparary
  6, zulu
  7, yorkshire
  8, xerxes
  9, wonderland
```

Figure 9-17 Preorder Traversal Results

9.7.3 Postorder Traversal

In a postorder traversal, all the children of a node are visited before the node is visited. The pseudocode for the postorder traversal procedure is shown in **Figure 9-19**.

```
void postorder( BTREE__NODE_p_t node )
    if ( node != NULL )
        postorder( node->left )
        postorder( node->right )
        visit( node )
```

Figure 9-18 Postorder Traversal Pseudocode

If we substitute a postorder traversal for the inorder traversal in the example in **Section 9.7.1** we will get the results shown in **Figure 9-17**. Once again, this isn't a very good example of the use of this type of traversal. A better example would be the use of binary trees to parse postfix expressions as discussed in your textbook.

```
15, aristotle
 13, eratosthenes
  5, alabama
 16, kant
  4, gettysburg
  2, abbey
 14, pythagoras
 12, montana
 11, sayville
 10, tiparary
  3, shmabbey
  1, lincoln
  9, wonderland
  8, xerxes
  7, yorkshire
  6, zulu
  0, washington
```

Figure 9-19 Postorder Traversal Results

10. N-ary Trees

In this section we will discuss the *n-ary tree* structure. Like a binary tree, an n-ary tree is a collection of linked nodes beginning with a root; however, where each node in a binary tree can have up to two children, each node in an n-ary tree can have an unlimited number of children. Interestingly this would seem to suggest that an n-ary tree has a more extensive structure than a binary tree, however n-ary trees are typically constructed using binary trees.

10.1 Objectives

At the conclusion of this section you will be able to:

- Define an n-ary tree, and explain the relationships between nodes in an n-ary tree;
- Perform a modularized n-ary tree implementation using a binary tree; and
- Use an n-ary tree to construct a *directory*.

10.2 Overview

Figure 10-1 shows one way to view an n-ary tree, suggesting that each node contains a link to each of zero or more children. Note that not every node has to have the same number of children. One way to build a tree like this would be to keep an array of child pointers in each node. However this approach has one big disadvantage: arrays are static entities. Each node would have a maximum number of children, and no matter how many children a node actually has, it would be forced to allocate an array big enough to hold the maximum.

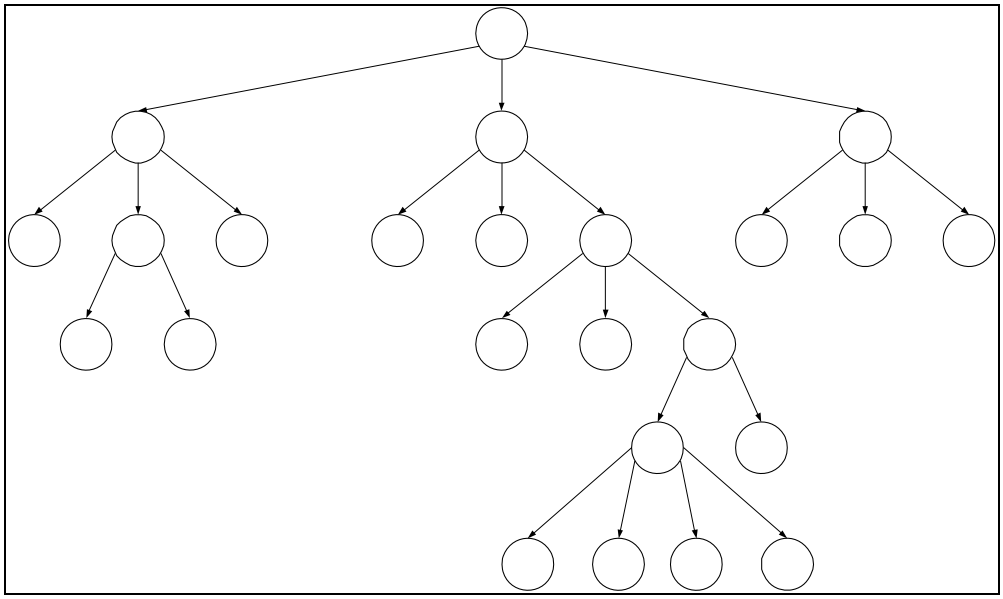


Figure 10-1 N-ary Tree: Multilink View

Another approach to constructing an n-ary tree is to use linked lists to link all the children of a node. The most popular is approach is to store an n-ary tree as a binary tree using the *left child, right sibling* method. This method is illustrated in **Figure 10-2**. It shows that

the right child of node N in the binary tree is treated as a sibling of node N in the n-ary tree. The right child of node N in the binary tree is referred to as node N's *nearest sibling*, and the right child of any sibling of N is node N's sibling. The left child of node M in the binary tree is referred to as node M's *nearest child*, and any sibling of a child of M is also a child of M. In considering the relationships between nodes in an N-ary tree we have one situation that is, at least at first, anti-intuitive. Suppose that, in a binary tree, B is the right child of A. Then in the corresponding n-ary tree, B is a sibling of A, but A is *not* a sibling of B.

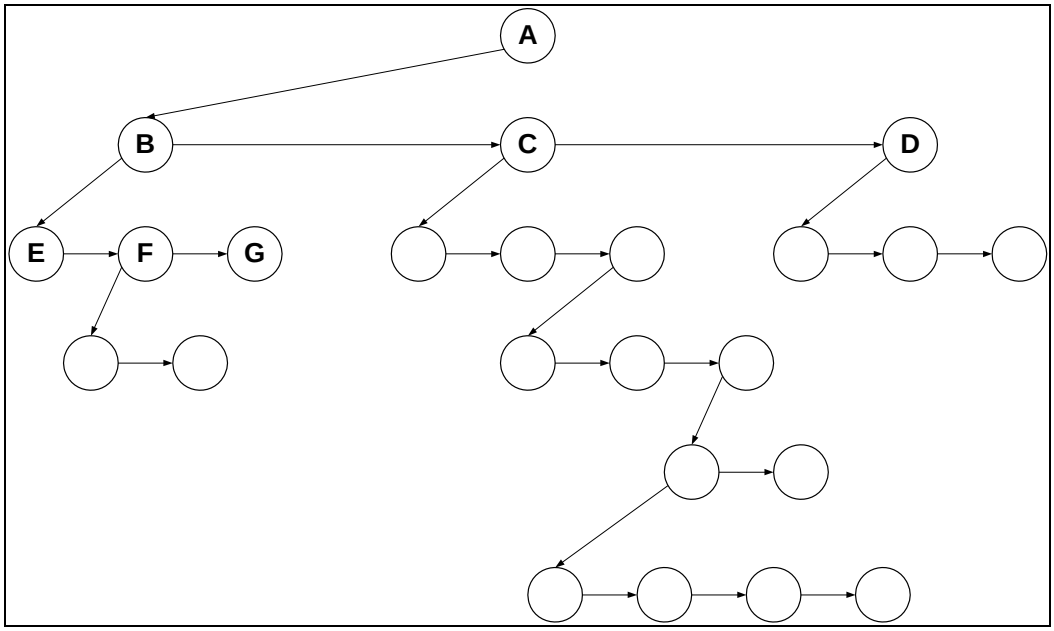


Figure 10-2 N-ary Tree: Binary Tree View

10.3 A Small N-ary Tree Implementation

Let's look at a modular n-ary tree implementation. It will be similar to our binary tree implementation, and will incorporate methods to perform the following operations:

- Create an n-ary tree.
- Add a root node to an n-ary tree.
- Add a child to a node.
- Add a sibling to a node.
- Get the root node of an n-ary tree.
- Determine whether a node has a child.
- Determine whether a node has a sibling.
- Get the data associated with a node.
- Get the nearest child of a node.
- Get the nearest sibling of a node.
- Destroy an n-ary tree.

10.3.1 Public Data Types

Our module name will be NTREE, and our public declarations will rely heavily on the declarations of our BTREE module, which makes sense because our n-ary trees will *be* binary trees; they are shown in **Figure 10-3**. We will declare incomplete types to represent the ID of a tree and a node, and we will use macros to define values to represent a NULL tree and a NULL node. We will also need a destroy callback type.

```
#include <btree.h>

#define NTREE_NULL_ID           (BTREE_NULL_ID)
#define NTREE_NULL_NODE_ID      (BTREE_NULL_NODE_ID)

typedef BTREE_DESTROY_PROC_t    NTREE_DESTROY_PROC_t;
typedef BTREE_DESTROY_PROC_p_t  NTREE_DESTROY_PROC_p_t;

typedef BTREE_ID_t              NTREE_ID_t;
typedef BTREE_NODE_ID_t         NTREE_NODE_ID_t;
```

Figure 10-3 NTREE Module Public Declarations

10.3.2 Private Declarations

Our implementation has no private declarations of consequence; everything is borrowed from the BTREE module. The complete private header file is shown in **Figure 10-4**.

```
#ifndef NTREEP_H
#define NTREEP_H

#include <ntree.h>

#endif
```

Figure 10-4 NTREE Module Private Header File

10.3.3 NTREE_create_tree

The description of NTREE_create_tree is identical to BTREE_create_tree. The code for this method is shown below.

```
NTREE_ID_t NTREE_create_tree( void )
{
    BTREE_ID_t tree = BTREE_create_tree();
    return tree;
}
```

10.3.4 NTREE_add_root

The description of NTREE_add_root is identical to that of BTREE_add_root. The code is shown below.

```
NTREE_NODE_ID_t NTREE_add_root( NTREE_ID_t tree, void *data )
{
    BTREE_NODE_ID_t node = BTREE_NULL_NODE_ID;
    CDA_ASSERT( BTREE_is_empty( tree ) );
    node = BTREE_add_root( tree, data );
    return node;
}
```

10.3.5 NTREE_add_child

This method will add a child to the list of a node's children; it does so by locating the last child in the list, and adding a sibling.

```
Synopsis:
    NTREE_NODE_ID_t
    NTREE_add_child( NTREE_NODE_ID_t node, void *data );
Where:
    node == node to which to add
    data == data to store at new node
Returns:
    ID of new node
Exceptions:
    Throws SIGABRT if node can't be created
Notes:
    None
```

Here is the code for this method:

```
NTREE_NODE_ID_t
NTREE_add_child( NTREE_NODE_ID_t node, void *data )
{
    NTREE_NODE_ID_t child = NTREE_NULL_NODE_ID;

    if( !NTREE_has_child( node ) )
        child = BTREE_add_left( node, data );
    else
        child = NTREE_add_sib( NTREE_get_child( node ), data );

    return child;
}
```

10.3.6 NTREE_add_sib: Add a Sibling to a Node

This method will add a sibling to a node's list of children.

```
Synopsis:
    NTREE_NODE_ID_t
    NTREE_add_sib( NTREE_NODE_ID_t node, void *data );
Where:
    node == node to which to add
    data == data to store at new node
Returns:
    ID of new node
Exceptions:
    Throws SIGABRT if node can't be created
Notes:
    None
```

Here is the code for this method:

```
NTREE_NODE_ID_t NTREE_add_sib( NTREE_NODE_ID_t node, void *data )
{
    NTREE_NODE_ID_t last = node;
    NTREE_NODE_ID_t sib = NTREE_NULL_NODE_ID;
```

```
        while( NTREE_has_sib( last ) )
            last = NTREE_get_sib( last );
        sib = BTREE_add_right( last, data );

        return sib;
    }
```

10.3.7 NTREE_get_root

This method is identical to BTREE_get_root. The code consists of a simple call to BTREE_get_root.

10.3.8 NTREE_has_child

This method will determine whether a node has a child, which is the same as determining if it has a left child in its binary tree.

```
Synopsis:
    CDA_BOOL_t NTREE_has_child( NTREE_NODE_ID_t node );
Where:
    node == node to interrogate
Returns:
    CDA_TRUE if node has a child,
    CDA_FALSE otherwise
Exceptions:
    None
Notes:
    None
```

Here is the code for this method:

```
CDA_BOOL_t NTREE_has_child( NTREE_NODE_ID_t node )
{
    CDA_BOOL_t      rcode = CDA_TRUE;
    BTREE_NODE_ID_t child = BTREE_get_left( node );

    if( child == BTREE_NULL_NODE_ID )
        rcode = CDA_FALSE;

    return rcode;
}
```

10.3.9 NTREE_has_sib

This method will determine whether a node has a sibling, which is the same as determining if it has a right child in its binary tree.

```
Synopsis:
    CDA_BOOL_t NTREE_has_sib( NTREE_NODE_ID_t node );
Where:
    node == node to interrogate
Returns:
    CDA_TRUE if node has a sibling,
    CDA_FALSE otherwise
Exceptions:
    None
```

```
Notes:
    None
```

Here is the code for this method:

```
CDA_BOOL_t NTREE_has_sib( NTREE_NODE_ID_t node )
{
    CDA_BOOL_t      rcode = CDA_TRUE;
    BTREE_NODE_ID_t sib   = BTREE_get_right( node );

    if ( sib == BTREE_NULL_NODE_ID )
        rcode = CDA_FALSE;

    return rcode;
}
```

10.3.10 NTREE_get_data, NTREE_get_child, NTREE_get_sib

These methods return the data, nearest child and nearest sibling of a node. The descriptions of these three routines are identical to the descriptions of BTREE_get_data, BTREE_get_left and BTREE_get_right, respectively. The code for each method consists of a simple call to its analogous BTREE method.

10.3.11 NTREE_destroy_tree

The description of this routine is identical to the description of BTREE_destroy_tree. The code is shown below.

```
NTREE_ID_t
NTREE_destroy_tree( NTREE_ID_t              tree,
                    NTREE_DESTROY_PROC_p_t destroy_proc
                  )
{
    BTREE_destroy_tree( tree, destroy_proc );
    return NTREE_NULL_ID;
}
```

10.4 Directories

N-ary trees are used in a variety of applications, including those for parsing expressions consisting of multiple operands (such as *conditional expressions* in C, which require evaluation of three operands). But one of the most popular uses is the creation of *directories*.

A directory is a hierarchical arrangement of nodes, some or all of which can contain other nodes. One familiar example is the directory structure on your computer's hard drive. **Figure 10-5** shows part of the layout of the directory structure on the C drive on my Windows NT workstation.

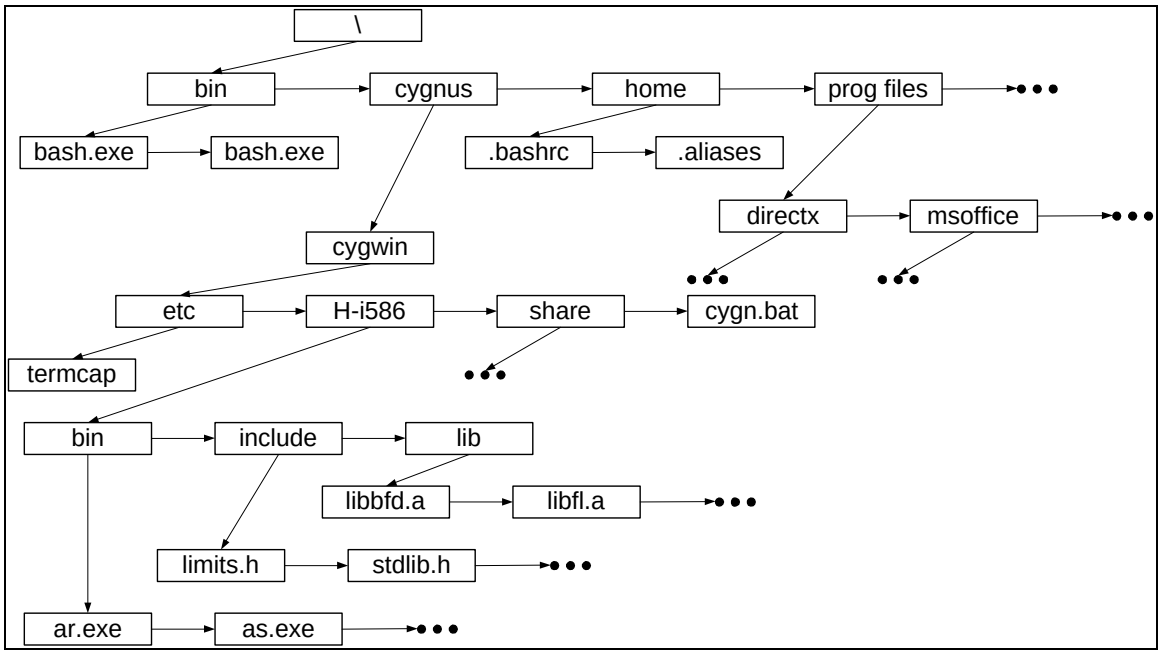


Figure 10-5 A Disk Directory Structure

The *root directory*, represented by the backslash (\) is a container that can hold other directories and "regular" files, among them the *bin, Cygnus, home* and *program files* subdirectories. The *bin* subdirectory contains two non-container files; the *Cygnus* subdirectory contains the *cygwin* subdirectory, which in turn contains three additional subdirectories plus the regular file *cygnus.bat*. As the figure suggests, an n-ary tree is a good data structure for representing this organization. The root node represents the root directory, and children of the root represent all the files and subdirectories contained by the root. Children of the n-ary node *cygwin* represent files and subdirectories contained in the *cygwin* subdirectory, and so forth. As an n-ary tree, the directory structure is extensible to any practical limit.

More generally we can say that a *directory element* is a node that has a *name,* a list of *properties* and may or may not contain other directory elements. A *property* is a distinguishing characteristic of a node; it has a *name* and a *value.* A *directory* is any collection of *directory elements* beginning with a *root node* that must be a container. Within a directory, every node is represented by a *distinguished name*; a distinguished name, or DN, is the concatenation of the node's name with the names of 0 or more of its ancestors. A *fully qualified distinguished name*, or FQDN, is the DN of a node beginning with the root, and every node must have a unique FQDN. Any DN that is not fully qualified has a context *relative* to some other node in the tree. Consider the node *ar.exe* in the above illustration. Its name is *ar.exe,* and some of properties are its size (184,320 bytes) its creation date (December 25, 1999) and its type (application). It has a DN relative to *cygwin* of *H-i586\bin\ar.exe*, and an FQDN of *\cygnus\cygwin\H-i586\bin\ar.exe*.

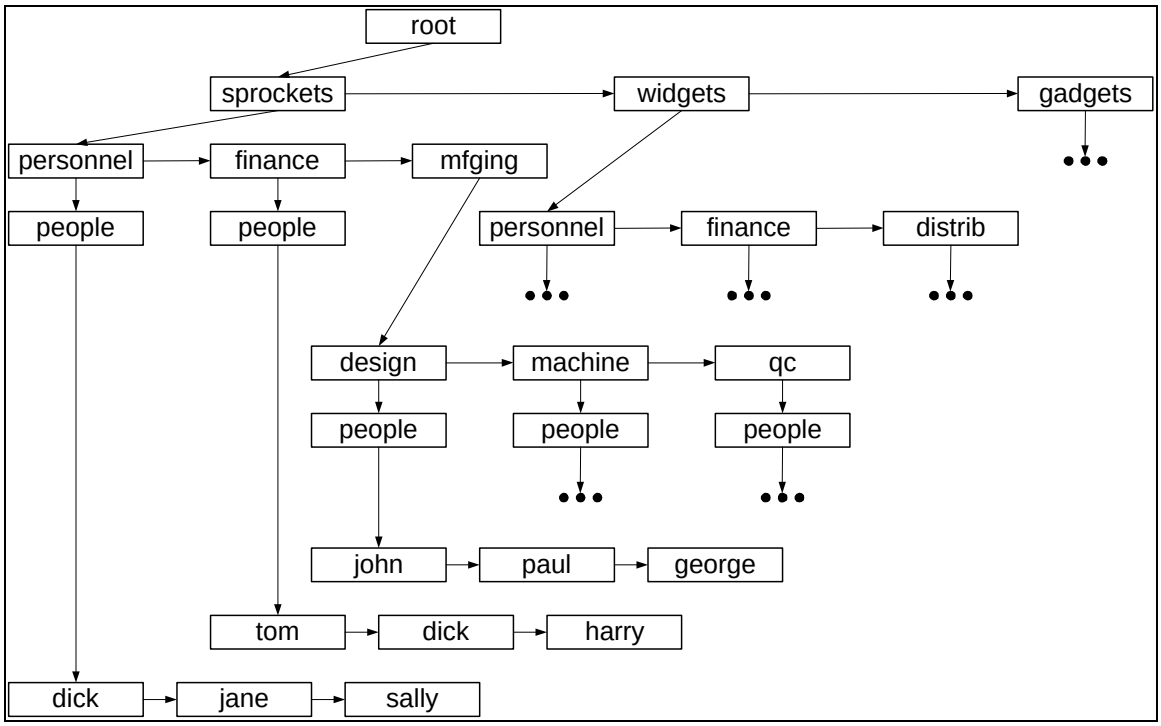


Figure 10-6 Organizational Directory

In **Figure 10-6** we see an example of another popular type of directory, the *organizational directory*. Children of the root represent different *organizations*, in the case of our example possibly different companies. Each organization is broken down into one or more *organizational units*, which in turn are broken down into *groups* that contain either *subgroups* or *members*. Organization *sprockets* has three organizational units, *personnel, finance* and *manufacturing*. *Personnel* has one group named *people* which has three members, *dick, jane* and *sally*. Note that two nodes within the hierarchy are both named *dick*, however their FQDNs are unique: *\sprockets\personnel\people\dick* and *\sprockets\finance\people\dick*. Each node in the directory has a list of properties. For example, *\sprockets* might have a list of properties that include the following:

- `ceo = “George F. Snyde”`
- `address1 = “121 Buena Vista Drive”`
- `city = “San Jose”`

\sprockets\personnel might have a list of properties that include these:

- `director = “A. C. Calhoun”`
- `location = “Downtown”`

And *\sprockets\personnel\people\spot* might have, among other properties, this one:

- `fullname = “Spot D. Barker”`

10.4.1 A Simple Directory Module

Let's take a look at a very simple (and incomplete) module for building and interrogating a directory. Our module name will be CDIR (as in *C Language API For Directory Access*), and we'll consider the following operations:

- Create a new directory
- Add a child to a directory element
- Add a property to a directory element
- Get a directory node
- Interrogate a property in a directory element
- Destroy a directory

To keep our example simple we will assume that every directory element can be a container. We will also define a property to be a name/value pair, where both the name and the value are NULL-terminated character strings.

10.4.2 Public Data Types

We're going to need an incomplete data type to represent a directory ID, and another to represent a node in a directory tree; and we'll need types to represent both a NULL directory ID and a NULL node ID. As you might expect, our implementation will ultimately be based on the NTREE module, so our public types will be based on declarations from the NTREE module. They are shown in **Figure 10-7**.

```
#define CDIR_NULL_ID        (NTREE_NULL_ID)
#define CDIR_NULL_NODE_ID   (NTREE_NULL_NODE_ID)

typedef NTREE_ID_t          CDIR_ID_t;
typedef NTREE_NODE_ID_t     CDIR_NODE_ID_t;
```

Figure 10-7 CDIR Module Public Data Types

10.4.3 CDIR_create_dir

This method will create a new directory with an empty root node.

```
Synopsis:
    CDIR_ID_t CDIR_create_dir( const char *name );
Where:
    name == name of the directory
Returns:
    ID of directory
Exceptions:
    Throws SIGABRT if directory can't be created
Notes:
    When the directory is no longer needed, the user must destroy
    it by calling CDIR_destroy_dir.
```

10.4.4 CDIR_add_child

This method will add a child to some node in the directory. The name of the new node may be specified as a relative DN or as an FQDN.

```
Synopsis:
    CDIR_NODE_ID_t CDIR_add_child( CDIR_ID_t       cdir,
                                   CDIR_NODE_ID_t  base,
                                   const char      **names,
                                   size_t          num_names
                                 );
Where:
    cdir      == the (possibly NULL) ID of the target directory
    base      == the (possibly NULL) ID of a node in the directory
    names     == an array of strings representing the
                 distinguished name of the child to create
    num_names == size of the names array
Returns:
    CDIR_NULL_NODE_ID if names is an invalid DN, otherwise
    the ID of the new node
Exceptions:
    Throws SIGABRT if node can't be created due to resource
    allocation problems (e.g. dynamic memory failure)
Notes:
   1. The distinguished name of the target node is the
      concatenation of the strings in the names array.
   2. The parent of the new node must already exist; that is, the
      node whose DN is formed by concatenating the first
      num_names – 1 strings from the names array. If the parent
      node does not already exist, the operation will fail and
      CDIR_NULL_NODE_ID will be returned.
   3. If base is CDIR_NULL_NODE_ID then cdir must be non-NULL and
      the names array is understood to define an FQDN. Otherwise
      cdir is ignored, and the names array is understood to
      define a DN relative to base.
```

10.4.5 CDIR_add_property

This method will add a property to a node.

```
Synopsis:
    const char *CDIR_add_property( CDIR_NODE_ID_t  node,
                                   const char      *name,
                                   const char      *value
                                 );
Where:
    node   == node to which to add property
    name   == name of the property to add
    value  == value of the property to add
Returns:
    value
Exceptions:
    Throws SIGABRT if property can't be created
Notes:
    None
```

10.4.6 CDIR_get_node

This method will return the ID of a node in the directory. The name of the target node may be a relative DN, or an FQDN

```
Synopsis:
    CDIR_NODE_ID_t CDIR_get_node( CDIR_ID_t       cdir,
                                  CDIR_NODE_ID_t  base,
                                  const char      **names,
                                  size_t          num_names
                                );
Where:
    cdir      == the (possibly NULL) ID of the target directory
    base      == the (possibly NULL) ID of a node in the directory
    names     == an array of strings representing the
                 distinguished name of the child to create
    num_names == size of the names array
Returns:
    CDIR_NULL_NODE_ID if the target node could not be located,
    otherwise the ID of the target node
Exceptions:
    None
Notes:
   1. The distinguished name of the target node is the
      concatenation of the strings in the names array.
   2. If base is CDIR_NULL_NODE_ID then cdir must be non-NULL and
      the names array is understood to define an FQDN. Otherwise
      cdir is ignored, and the names array is understood to
      define a DN relative to base.
```

10.4.7 CDIR_get_property

This method will return the value of a property in a node.

```
Synopsis:
    CDA_BOOL_t CDIR_get_property( CDIR_NODE_ID_t  node,
                                  const char      *name,
                                  const char      **value
                                );
Where:
    node   == node in which to find property
    name   == name of the property to find
    value  -> variable to which to return property value
Returns:
    CDA_TRUE if property could be found, CDA_FALSE otherwise
Exceptions:
    None
Notes:
    None
```

10.4.8 CDIR_destroy_dir

This method will destroy a directory.

```
Synopsis:
    CDIR_ID_t CDIR_destroy_dir( CDIR_ID_t cdir );
Where:
    cdir == directory ID
Returns:
    CDIR_NULL_ID
```

```
Exceptions:
    None
Notes:
    None
```

10.4.9 Implementation Structure

Our implementation will be based on an n-ary tree as defined by our NTREE module. A directory node will simply be an NTREE node. The list of properties associated with a node will be maintained in the data portion of the NTREE node; specifically, the data portion of a node will be an ENQ anchor, where the name of the anchor will be the name of the node. Each time we add a property we will add an item to the tail of the list; the name of the item will be the name of the property, and the value of the property will be stored in the data portion of the item itself. This strategy is illustrated in **Figure 10-8**.

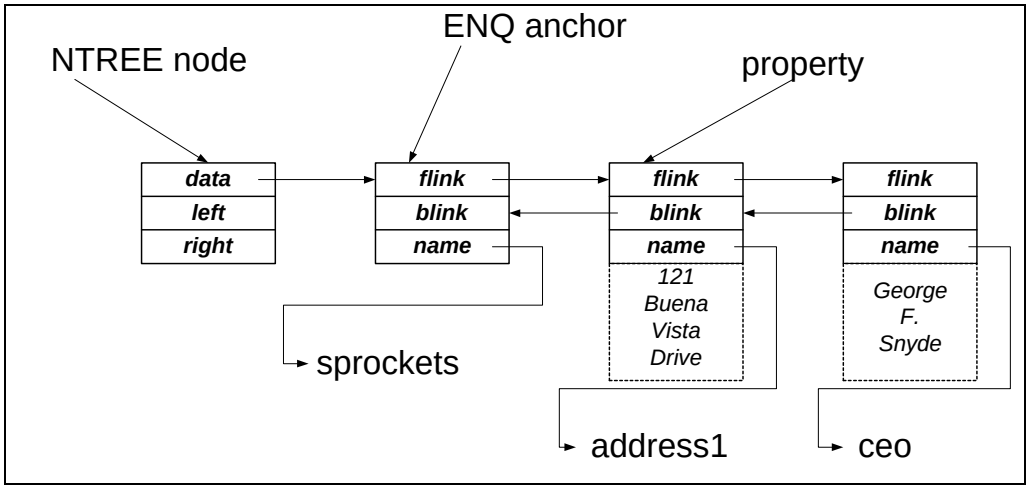


Figure 10-8 CDIR Implementation Strategy

Since our basic CDIR types will just be based on NTREE types the only thing we have to declare in our private header file will be the structure to represent a property. This will be an enqueuable item with a single field for the user data. This field is declared to be type char, but when we create an item we will allocate enough extra space for an entire value, and this field will contain the first byte. The complete private header file is shown in **Figure 10-9**.

10.4.10 CDIR_create_dir Implementation

The CDIR create method just has to create an n-ary tree, and add a root to it. Like all of our CDIR nodes, the root will need an anchor for a list of properties. Note that the root node never has a name provided by the user; it is always referred to in some generic way, such as *root, /,* \ or *dot (.)*. So we'll use the name of the root's anchor to store the name of the directory. Here is the implementation for this method.

```
CDIR_ID_t CDIR_create_dir( const char *name )
{
    CDIR_ID_t       cdir    = NTREE_create_tree();
    ENQ_ANCHOR_p_t  anchor  = ENQ_create_list( name );
    NTREE_add_root( cdir, anchor );

    return cdir;
}
```

```
#ifndef CDIRP_H
#define CDIRP_H

#include <cdir.h>
#include <enq.h>

typedef struct cdir__property_s
{
    ENQ_ITEM_t  item;
    char        value;
} CDIR__PROPERTY_t, *CDIR__PROPERTY_p_t;

#endif
```

Figure 10-9 CDIR Module Private Header File

10.4.11 CDIR_add_child Implementation

Adding a node to a directory is our most complex operation, because the DN must be formulated from the *names* array and interpreted as either a relative DN or FQDN (much of this complexity is embedded in *CDIR_get_node*). Since the *entire* names array defines the DN of the new node, the names array up through the next-to-last element must be the DN of the parent node. Here is the implementation.

```
CDIR_NODE_ID_t CDIR_add_child( CDIR_ID_t       cdir,
                               CDIR_NODE_ID_t  base,
                               const char      **names,
                               size_t          num_names
                             )
{
    CDIR_NODE_ID_t  node    = CDIR_NULL_NODE_ID;
    CDIR_NODE_ID_t  temp    = CDIR_NULL_NODE_ID;

    CDA_ASSERT( num_names > 0 );
    temp = CDIR_get_node( cdir, base, names, num_names - 1 );
    if ( temp != CDIR_NULL_NODE_ID )
    {
        ENQ_ANCHOR_p_t  anchor  =
            ENQ_create_list( names[num_names - 1] );
        node = NTREE_add_child( temp, anchor );
    }

    return node;
}
```

10.4.12 CDIR_add_property

This method will add a new property to a node's property list. Since the user passes in the node, all we have to do is obtain the anchor of the property list, create a new item containing the property value, and add it to the tail of the list. Here is the code.

```
const char *CDIR_add_property( CDIR_NODE_ID_t  node,
                               const char      *name,
                               const char      *value
                             )
{
    ENQ_ANCHOR_p_t      anchor  = NTREE_get_data( node );
    size_t              size    =
        sizeof(CDIR__PROPERTY_t) + strlen( value );
    CDIR__PROPERTY_p_t  prop    =
        (CDIR__PROPERTY_p_t)ENQ_create_item( name, size );

    strcpy( &prop->value, value );
    ENQ_add_tail( anchor, (ENQ_ITEM_p_t)prop );

    return value;
}
```

10.4.13 CDIR_get_node Implementation

As did CDIR_add_child, this method will have to formulate a DN out the *names* array passed by the user, and then interpret the DN as either relative or fully qualified. Note that an array of length zero must refer to either the node passed by the user (for a relative DN) or to the root node (for an FQDN). Here is the implementation.

```
CDIR_NODE_ID_t CDIR_get_node( CDIR_ID_t       cdir,
                              CDIR_NODE_ID_t  base,
                              const char      **names,
                              size_t          num_names
                            )
{
    CDIR_NODE_ID_t  node    = base;
    ENQ_ANCHOR_p_t  anchor  = NULL;
    int             inx     = 0;

    if ( node == CDIR_NULL_NODE_ID )
        node = NTREE_get_root( cdir );

    for ( inx = 0                                                    ;
          (inx < (int)num_names) && (node != CDIR_NULL_NODE_ID) ;
          ++inx
        )
    {
        CDA_BOOL_t  found   = CDA_FALSE;
        node = NTREE_get_child( node );
        while ( !found && (node != CDIR_NULL_NODE_ID) )
        {
            anchor = NTREE_get_data( node );
            if ( strcmp( names[inx], ENQ_GET_LIST_NAME( anchor )
                       ) == 0
               )
                found = CDA_TRUE;
```

```
                else
                    node = NTREE_get_sib( node );
            }
        }

        return node;
    }
```

10.4.14 CDIR_get_property Implementation

For this method all we have to do is obtain the property list from a node (which is passed by the caller) and traverse it until we find an item with the name of the target property. Once we find the item, we pass the address of the value back to the caller via the *value* parameter. If we reach the end of the property list without locating the target property then the property doesn't exist, in which case we return NULL for the value of the property, and CDA_FALSE for the function's return value.

```
CDA_BOOL_t CDIR_get_property( CDIR_NODE_ID_t  node,
                              const char      *name,
                              const char      **value
                            )
{
    CDA_BOOL_t          rcode   = CDA_FALSE;
    ENQ_ANCHOR_p_t      anchor  = NTREE_get_data( node );
    ENQ_ITEM_p_t        item    = ENQ_GET_HEAD( anchor );
    char                *propv  = NULL;

    while ( !rcode && (item != anchor) )
        if ( strcmp( ENQ_GET_ITEM_NAME( item ), name ) == 0 )
        {
            rcode = CDA_TRUE;
            propv = &((CDIR__PROPERTY_p_t)item)->value;
        }
        else
            item = ENQ_GET_NEXT( item );

    *value = propv;
    return rcode;
}
```

10.4.15 CDIR_destroy_dir Implementation

The implementation for this method is fairly simple: we just destroy the n-ary tree that encapsulates our directory. The one wrinkle is that each node in the directory contains a list of properties that have to be destroyed, but we easily handle that by using the NTREE destroy method's destroy-proc feature. Note that we don't have to supply our users (that is, the users of the CDIR module) with a destroy proc because none of the data in the directory actually belongs to the user; it all belongs to the CDIR implementation. Here is the code.

```
static NTREE_DESTROY_PROC_t destroyProc;
CDIR_ID_t CDIR_destroy_dir( CDIR_ID_t cdir )
{
    NTREE_destroy_tree( cdir, destroyProc );
    return CDIR_NULL_ID;
```

```
}

static void destroyProc( void *data )
{
    ENQ_destroy_list( data );
}
```

10.4.16 Directories Discussion Wrap-up

The discussion of directories, above, merely touched the surface of this complex topic. And our definition of a directory module is incomplete; for example, we should have methods to change or delete a property, and a method to delete a node (along with all of its children). Adding methods to change and delete a property is straightforward and is left to the student. Adding a method to delete a node is a little tougher. Deleting a node from our directory will entail adding an NTREE method to delete an NTREE node; this in turn means adding a BTREE method to delete a BTREE node, which requires studying up on the subject of binary tree pruning. But our purpose here was not to do a complete directory implementation; it was to introduce directories, and to show how a directory can be built using an n-ary tree. If you want to tackle the job of adding a delete-node method, your textbook has all the details you need on pruning a binary tree. And to learn more about directories and directory APIs, you should browse the Netscape DevEdge documentation for *Directory Server and LDAP*, at http://developer.netscape.com/docs/index.html.

Practice Final Examination

This lesson will prepare you for the final examination. The final examination will consist of 20 questions worth five percentage points each; partial credit may apply to some of the questions. To receive credit for the course, you must achieve a score of 70% on the final.

To prepare for the final, review your notes, assigned readings, and quizzes, and then answer the following sample questions. During the exam itself, you may use your books and notes. You will have two hours to complete the examination.

Sample Questions

1. Use **CDA_malloc** or **CDA_calloc** to allocate enough memory for an array consisting of **NELEMENTS** elements. Each element should be type **CDA_INT32**. Use a for loop to initialize each element of the array to –1.

   Suggestion: If you think you know the answer, bench test it; code your solution into a test driver and see if it really works like you think it does.

2. The X Window System makes frequent use of callback routines much like the destroy callbacks we wrote when designing some of our modules. Each such callback has return type of void, and takes three parameters, each of which is type void*. Use typedef statements to create the names XT_CBPROC_t, equivalent to the type of a callback function, and XT_CBPROC_p_t, equivalent to type pointer-to-callback function.

3. Complete the following subroutine, which traverses a linked list in search of an item with a given name. If found, the item is dequeued and returned, otherwise, NULL is returned.

```
ENQ_ITEM_p_t ENQ_deq_named_item( ENQ_ANCHOR_p_t anchor,
                                 const char     *name
                               )
{
    ENQ_ITEM_p_t item = NULL;









    return item;
}
```

   Suggestion: If you think you know the answer, bench test it; code your solution into a test driver and see if it really works like you think it does.

4. List two contexts in which the name of an array is not equivalent to the address of an array?
5. In our ENQ module, what is the definition of a list in the empty state?
6. According to our module naming standards, to which module does the function UI__dump_parms belong, and where is its prototype published?

7. In the context of our HASH module, what is the definition of a collision?
8. In the context of our HASH module, complete the following function, which will create an index out of the string of bytes pointed to by *string*. The length of *string* is given by *length*, and the size of the table is given by *table_size*.

```
static size_t keyHash( const CDA_UINT8_t *string,
                       size_t            length,
                       size_t            tableSize
                     )
{
    size_t index = 0;









    return index;
}
```

   Suggestion: If you think you know the answer, bench test it; code your solution into a test driver and see if it really works like you think it does.

9. Write the function *QUE_remove* as discussed in your notes. Assume that the queue is implemented via the *ENQ* module (also as discussed in your notes).

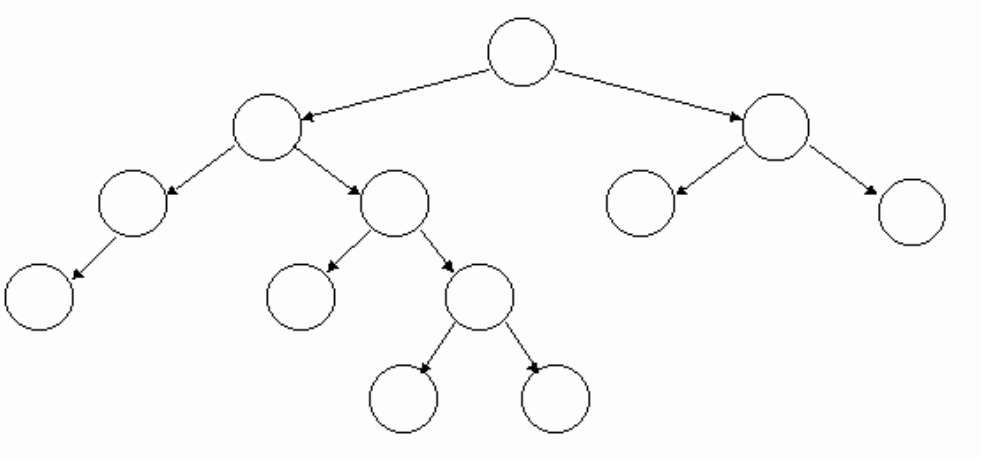
   Suggestion: If you think you know the answer, bench test it; code your solution into a test driver and see if it really works like you think it does.

10. Discuss the testing activities that occur during the design phase of the system life cycle.
11. What is a *regression test*?
12. According to your notes, what are the two major categories of *complexity*.
13. Describe the order in which a stack pointer is incremented and dereferenced in a push operation. What mechanism would you use to prevent stack overflow in a push operation?
14. Complete the following subroutine. It will count the number of items stored in a priority queue with the given priority; the queue is maintained as a single doubly linked list (just like our "simple" priority queue implementation). Refer to your notes for the declarations of PRQ_ID_t, etc.

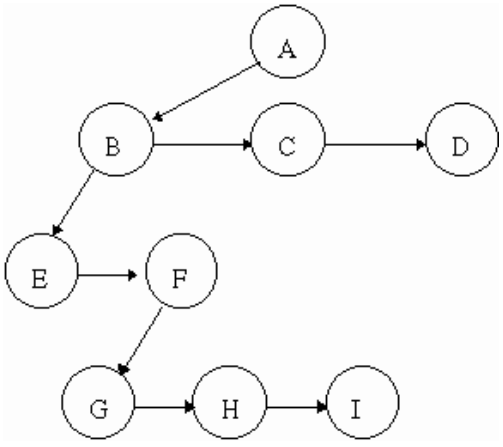
```
size_t PRQ_get_class_len( PRQ_ID_t queue, int class )
{
    int result=0;









    return result;
}
```

Suggestion: If you think you know the answer, bench test it; code your solution into a test driver and see if it really works like you think it does.

15. Assume that a binary tree is stored as an array, where the root node is stored in index 0 of the array.
    a) At what index will you find the parent of the node stored at index 197?
    b) At what index will you find the right child of the node stored at index 233?
16. Refer to the accompanying figure. Is the illustrated binary tree balanced? Why or why not?



17. List three ways to traverse a binary tree. Which method of traversal would you use to print the keys of an index in alphabetical order?
18. Refer to the accompanying figure, an n-ary tree implemented via a binary tree, and answer the following questions.
    a) Which node is the parent of node F?
    b) Is node G a sibling of node I?
    c) How many children does node E have?



19. ***(Note: this topic is not currently covered in your notes, and will not be represented on the final examination.)*** Examine the following source code, and answer the questions that follow.

```
#define FALSE (0)
typedef int BOOL;

static BOOL inited = FALSE;

static void print_err( char *err )
{
    int facility_id = 42;

    if ( inited )
        printf( "%d: err\n", facility_id, err );
}
```

a) Which components in the code represent *variable space requirements*?

b) Which components in the code represent *fixed space requirements?*

Answers

1.
```
CDA_INT32 *array = CDA_calloc( NELEMENTS, sizeof(CDA_INT32) );
int       inx    = 0;

for ( inx = 0 ; inx < NELEMENTS ; ++inx )
    array[inx] = -1;
```
2.
```
typedef void XT_CBPROC_t( void *, void *, void * );
typedef XT_CBPROC_t *XT_CBPROC_p_t;
```
3.
```
ENQ_ITEM_p_t ENQ_deq_named_item( ENQ_ANCHOR_p_t anchor,
                                 const char     *name
                               )
{
    ENQ_ITEM_p_t rval = NULL;
    ENQ_ITEM_p_t item = ENQ_GET_HEAD( anchor );

    while ( (item != anchor) && (rval == NULL) )
        if ( strcmp( item->name, name ) == 0 )
            rval = item;
        else
            item = item->flink;

    return rval;
}
```
4. When it is used as the operand of the **sizeof** operator; a pointer may be used as an *lvalue*, but the name of an array may not.
5. The *flink* and *blink* of the anchor point to the anchor.
6. The function belongs to the UI module. Since UI is followed by a double underscore, it must be a private function, so its prototype is published in uip.h.
7. A collision occurs when two different keys hash to the same element of the hash table's array.
8.
```
static size_t keyHash( const CDA_UINT8_t *string,
                       size_t            length,
                       size_t            tableSize
                     )
{
    size_t     index = 0;
    size_t     inx   = 0;
    const char *temp = string;

    for ( inx = 0 ; inx < length ; ++inx )
        index += *temp++;
    index %= tableSize;

    return index;
}
```

9.
```
QUE_ITEM_p_t QUE_remove( QUE_ID_t queue )
{
    ENQ_ANCHOR_p_t anchor = queue->anchor;
    ENQ_ITEM_p_t   item   = ENQ_deq_head( anchor );

    if ( item == anchor )
        item = NULL;

    return (QUE_ITEM_p_t)item;
}
```

10. Acceptance criteria for programs and modules are documented. Requirements are certified as testable. A proof of concept prototype may be built. A high-level test plan for verifying that the implementation conforms to the design is created.
11. A *regression* is a new flaw that is created when another flaw is fixed. A *regression test* attempts to find regressions in a system that has been changed.
12. The two major categories of complexity are *space complexity* and *time complexity*.
13. In a bottom-up stack implementation, first the stack pointer is dereferenced, adding an item to the stack, then the stack pointer is incremented, making it point to the next free item on the stack.

    To prevent a stack overflow, use a conditional statement that throws SIGABRT if a push operation would cause an overflow:
    ```
    if ( stack is full )
        abort();
    push
    ```
    Note that an assertion would be inappropriate, because assertions are disabled in production code.
14.
```
size_t PRQ_get_class_len( PRQ_ID_t queue, CDA_UINT32_t class )
{
    int            result = 0;
    ENQ_ANCHOR_p_t anchor = queue->anchor;
    PRQ_ITEM_p_t   item   = NULL;

    item = (PRQ_ITEM_p_t)ENQ_GET_HEAD( anchor );
    while ( (ENQ_ITEM_p_t)item != anchor &&
            item->priority != class
          )
        item = (PRQ_ITEM_p_t)ENQ_GET_NEXT( &item->enq_item );

    while ( (ENQ_ITEM_p_t)item != anchor &&
            item->priority == class
          )
    {
        ++result;
        item = (PRQ_ITEM_p_t)ENQ_GET_NEXT(&item->enq_item );
    }

    return result;
}
```

15. a) (197 - 1) / 2 = 98
    b) 2 * 233 + 2 = 468
16. No. The distances between the root and each leaf sometimes vary by more than 1.
17. Three methods to traverse a binary tree are *preorder*, *postorder* and *inorder*. To print the keys of an index in alphabetical order use an *inorder* traversal.
18. a) B
    b) No; siblinghood is a one-way relationship, so I is a sibling of G, but G is not a sibling of I.
    c) None
19. a) Variable space requirements: The function's parameter, **err**; the function's automatic variable, **facility_id**; and the parameters passed to **printf**.
    b) Fixed space requirements: The static variable **inited** and the instructions comprising the function **print_err**.

Quizzes

Quiz 1

Due Week 2

1. Write the typedefs for the following data types:

   `CDA_BOOL_t`: type int.

   `OBJECT_t`: a structure whose first two members, named *flink* and *blink*, are type "pointer to `OBJECT_t`"; and whose third member, named *object_id*, is type (char *).

   `OBJECT_p_t`: type pointer to `OBJECT_t`.

   `XQT_PROC_t`: a function which returns `CDA_BOOL_t`, and takes a single argument which is type pointer to `OBJECT_t`.

   XQT_PROC_p_t: type pointer to XQT_PROC_t.

   Use XQT_PROC_t to declare a static function named *thread_proc*.
2. Use CDA_malloc to allocate an array of 100 elements of type `char*`. Use a for loop to initialize each element of the array to NULL.
3. Write cover routines for *calloc* and *realloc* like the one that we wrote for *malloc* in class. Call the covers *CDA_calloc* and *CDA_realloc*, respectively.

Quiz 2

Due Week 3

1. Review the definition for an item in the *unenqueued state*, and an anchor in the *empty state*. Now review the implementation of *ENQ_deq_item*. One step at a time, show what will happen if you pass an unenqueued item to *ENQ_deq_item*. One step at a time, show what will happen if you pass an anchor in the empty state to *ENQ_deq_item.*
2. Complete the following subroutine which will traverse one of our linked lists, printing out the name of every item in the list:

```
void printNames( ENQ_ANCHOR_p_t list )
{
    ENQ_ITEM_p_t item = NULL;
    for ( item = ENQ_GET_HEAD( list )     ;
          . . .                           ;
          . . .
        )
        . . .
}
```

3. Write a function that will test whether or not your linked list function ENQ_deq_tail will work correctly if passed the anchor of an empty list.

Quiz 3

Due Week 4

1. Briefly discuss the difference between a *selection sort* and a *bubble sort*.
2. Examine the following code:

```
int arr[5] = { 30, 20, 50, 70, 10 };
int *parr  = &arr[4];
int inx    = 0;

inx = *parr++;
```

   a) Is the above code legal?

   b) After executing the above code, what will be the value of *inx*?

   c) After executing the above code, where will *parr* be pointing?

3. Examine the following subroutine:

```
static int storage[100];
static int *pstore = storage;

store( int number )
{
    *pstore++ = number;
}
```

   The routine *store* is to be called repeatedly. If it is called more than 100 times it will malfunction; i.e., it will try to store an integer in a location *outside* the array *store*. Add an *if* statement to *store* that will call *abort* if the operation would be illegal.

Quiz 4

Due Week 5

1. Do a module's private functions have external scope?
2. A static global variable has scope that extends to what?
3. Where will the private declarations for the module PRT be published?
4. Where should you place the declarations used only by the source file prt.c?
5. Write the minimum amount of code required to implement the private header file for the INT module
6. In the context of an abstract data type, give an example of an exception.
7. What are three ways to handle an exception?

Quiz 5

Due Week 6

1. Write the code for *STK_pop_item* as discussed in your notes, in **Section 6.4.4**.
2. Complete the following function. This function will push strings passed by a caller onto a stack until the caller passes NULL, then it will pop the strings off the stack and print them.

```
#define MAX_STACK_SIZE (100)
static STK_ID__t stack = STK_NULL_ID;
void stringStuff( char *string )
{
    if ( stack == STK_NULL_ID )
    {
        . . .
    }

    if ( string != NULL )
        . . .
    else
    {
        /* Note: as you pop strings off the stack, how will you
         * know when you're done?
        */
        . . .
    }
}
```

Quiz 6

Due Week 7

1. Complete routine PRQ_create_queue for the simple priority queue implementation as discussed in your notes.
2. Revise the control structure for our PRQ module so that it can handle the optimized implementation described in your notes.
3. Rewrite the PRQ create method to work with the optimized implementation described in your notes.
4. A PRQ queue contains one priority 10 item, one priority 5 item and one priority 0 item. If PRQ_remove_item is called three times, in what order will the items be returned?
5. Write a function that can be used to test whether your PRQ_remove_item method works if passed the ID of an empty queue.

Quiz 7

Due Week 8

1. As defined in your notes, what are the phases of the system life cycle?
2. In what phases of the system life cycle is the module test plan used?
3. Describe the difference between *unit testing* and *acceptance testing.*

Quiz 8

Due Week 9

1. When is a binary tree balanced?
2. Complete routine BTREE_is_empty as described in Section 8.3.9 of your notes.
3. Complete routine BTREE_is_leaf as described in Section 8.3.10 of your notes.
4. Design and implement a method to delete a property from a node in the CDIR module as discussed in the *N-ary tree* section of your notes.