Chapter 3

Contax T

Musee de L'Armee

Fundamental Data Structures

Learning Objectives

- *Describe the performance characteristics of fundamental data structures used to implement collections*
- *Learn how to choose a collection based on its underlying data structure*
- *Describe the performance characteristics of an array*
- *Describe the performance characteristics of a linked list*
- *Describe the performance characteristics of a hash table*
- *Describe the performance characteristics of a red-black tree*
- *Describe the performance characteristics of a stack*
- *Describe the performance characteristics of a queue*

Introduction

In this chapter, I want to introduce you to the performance characteristics of several different types of foundational data structures. These include the *array*, *linked list*, *hash table*, and *red-black binary tree*. Knowing a little bit about how these data structures work and behave will make it easier for you to select the .NET collection type that's best suited for your particular application.

Array Performance Characteristics

As you know already from reading Chapter 8, an array is a contiguous collection of homogeneous elements. You can have arrays of value types or arrays of references to objects. The general performance issues to be aware of regarding arrays concern inserting new elements into the array at some position prior to the last element, accessing elements, and searching for particular values within the array.

When a new element is inserted into an array at a position other than the end, room must be made at that index location for the insertion to take place by shifting the remaining references one element to the right. This series of events is depicted in figures 3-1 through 3-3.

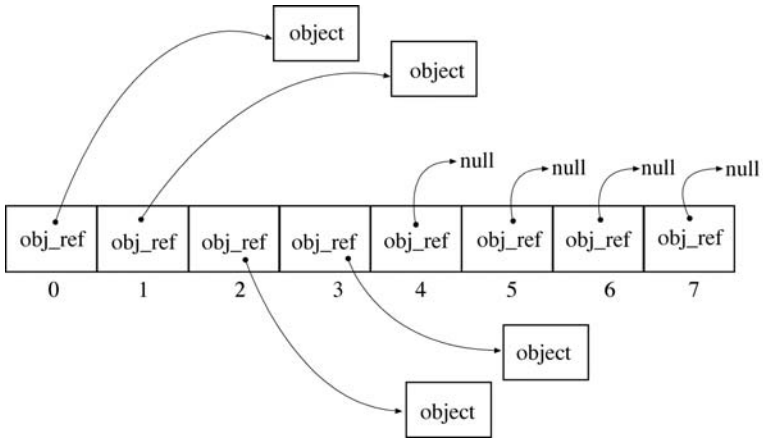

Figure 3-1: Array of Object References Before Insertion

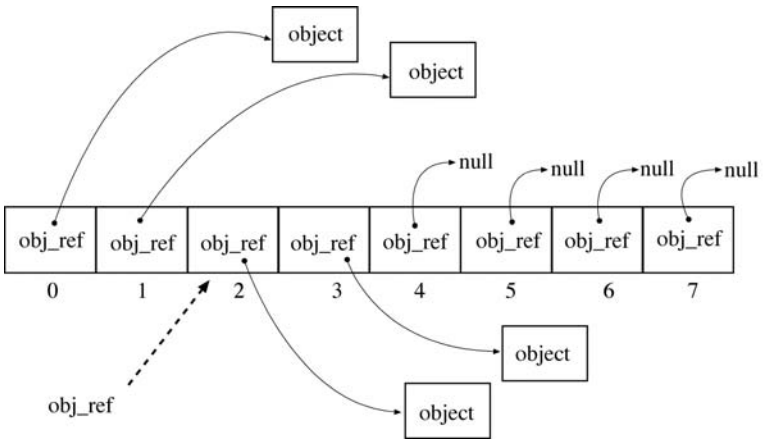

Figure 3-2: New Reference to be Inserted at Array Element 3 (index 2)

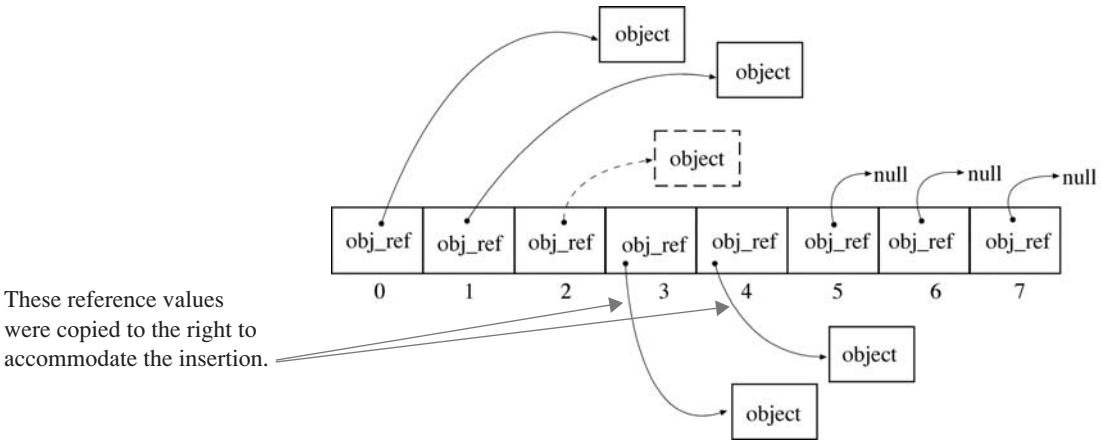

Figure 3-3: Array After New Reference Insertion

Referring to figures 3-1 through 3-3 — an array of object references contains references that may point to an object or to null. In this example, array elements 1 through 4 (index values 0 through 3) point to objects while the remaining array elements point to null.

A reference insertion is really just an assignment of the value of the reference being inserted to the reference residing at the target array element. To accommodate the insertion, the values contained in references located to the right of the target element must be reassigned one element to the right. (*i.e.,* They must be shifted to the right.) It is this shifting action that causes a performance hit when inserting elements into an array-based collection. If the insertion triggers the array growth mechanism, then you'll receive a double performance hit. The insertion performance penalty, measured in time, grows with the length of the array. Element retrieval, on the other hand, takes place fairly quickly because of the way array element addresses are computed. (*Refer to Chapter 4— Arrays*)

Linked List Performance Characteristics

A linked list is a data structure whose elements stand alone in memory. (And may indeed be located anywhere in the heap!) Each element is linked to another by a reference. Unlike the elements of an array, which are ordinary references, each linked list node is a complex data structure that contains a reference to the *previous* node in the list, the *next* node in the list, and a reference to an object payload, as Figure 3-4 illustrates.

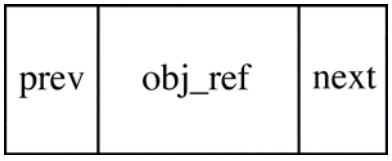

Figure 3-4: Linked List Node Organization

Whereas an array's elements are always located one right after the other in memory, and their memory addresses quickly calculated, a linked list's elements can be, and usually are, scattered in memory hither and yonder. The nice thing about linked lists is that element insertions take place fairly quickly because no element shifting is required. Figures 3-5 through 3-8 show the sequence of events for the insertion of a circular linked list node. Referring to figures 3-5 through 3-8 — a linked list contains one or more non-contiguous nodes. A node insertion requires reference rewiring. This entails setting the *previous* and *next* references on the new node in addition to resetting the affected references of its adjacent list nodes. If this looks complicated, guess what? It is! And if you take a data structures class you'll get the chance to create a linked list from scratch!

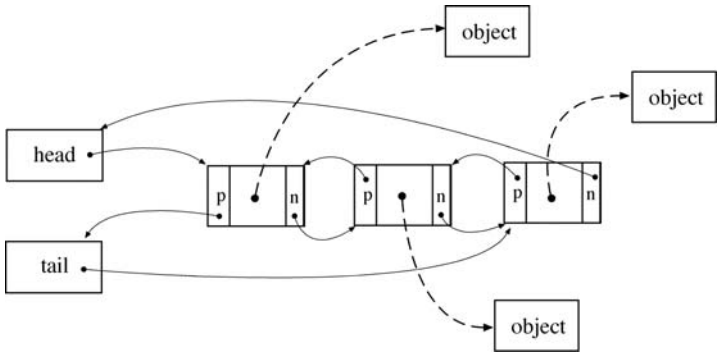

Figure 3-5: Linked List Before New Element Insertion

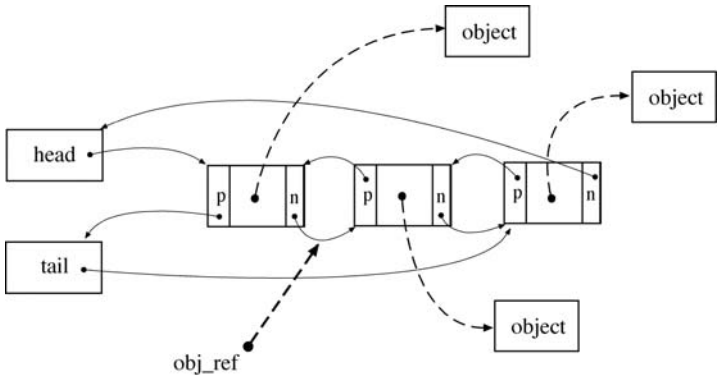

Figure 3-6: New Reference Being Inserted Into Second Element Position

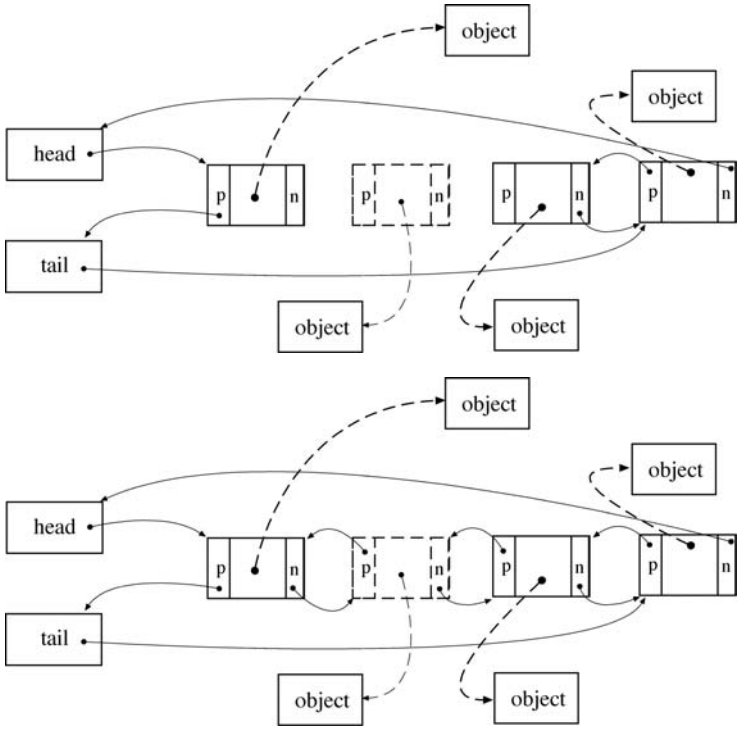

Figure 3-7: References of Previous, New, and Next List Elements must be Manipulated

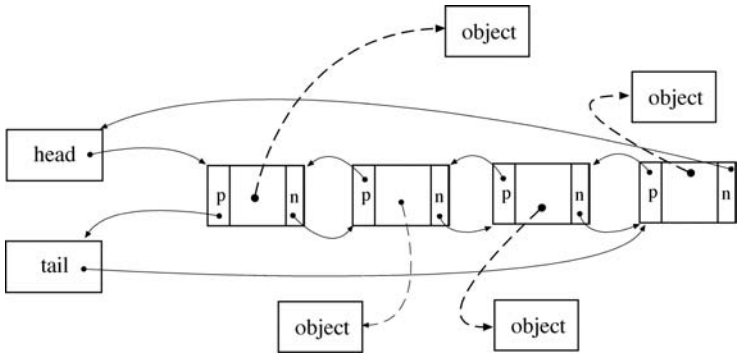

Figure 3-8: Linked List Insertion Complete

Hash Table Performance Characteristics

A hash table is an array whose elements can point to a series of nodes. Structurally, as you'll see, a hash table is a cross between an array and a one-way linked list. In an ordinary array, elements are inserted by index value. If there are potentially many elements to insert, the array space required to hold all the elements would be correspondingly large as well. This may result in wasted memory space. The hash table addresses this problem by reducing the size of the array used to point to its elements and assigning each element to an array location based on a *hash function* as Figure 3-9 illustrates.

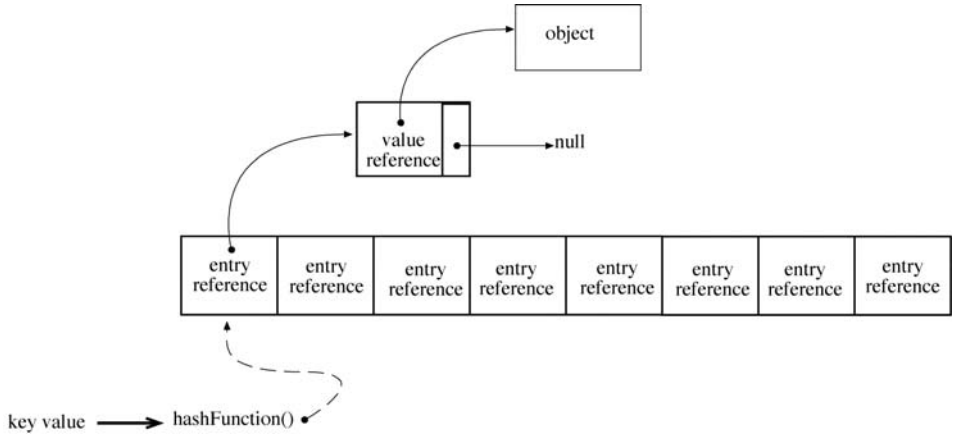

Figure 3-9: A Hash Function Transforms a Key Value into an Array Index

Referring to Figure 3-9 — the purpose of the hash function is to transform the key value into a unique array index value. However, sometimes two unique key values translate to the same index value. When this happens a *collision* is said to have occurred. The problem is resolved by chaining together nodes that share the same hash table index as is shown in Figure 3-10.

The benefits of a hash table include lower initial memory overhead and relatively fast element insertions. On the other hand, if too many insertion collisions occur, the linked elements must be traversed to insert new elements or to retrieve existing elements. List traversal extracts a performance penalty.

Chained Hash Table vs. Open-Address Hash Table

The hash table discussed above is referred to as a *chained hash table*. Another type of hash table, referred to as an *open-address hash table*, uses a somewhat larger array and replaces the linking mechanism with a *slot probe function* that searches for empty space when the table approaches capacity.

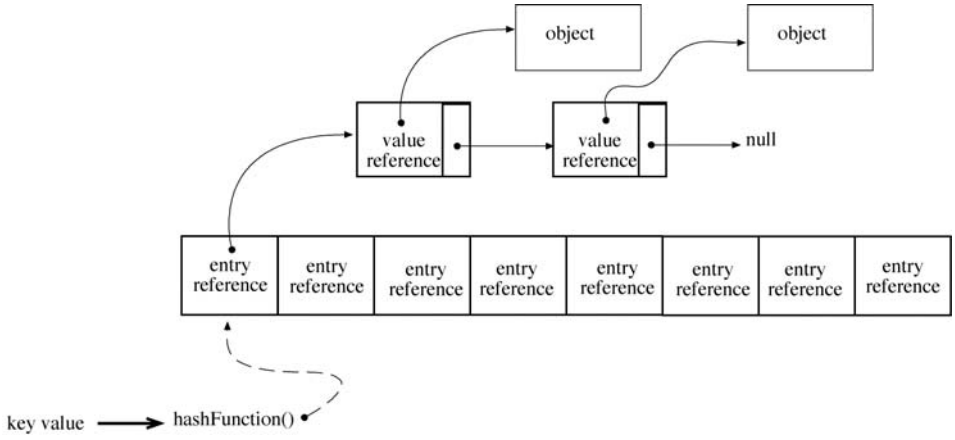

Figure 3-10: Hash Table Collisions are Resolved by Linking Nodes Together

Red-Black Tree Performance Characteristics

A *red-black tree* is a special type of *binary search tree* with a self-balancing characteristic. *Tree nodes* have an additional data element, *color*, that is set to either red or black. The data elements of a red-black tree node are shown in Figure 3-11.

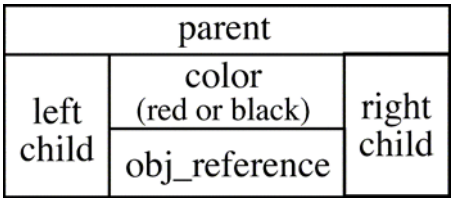

Figure 3-11: Red-Black Tree Node Data Elements

Insertions into a red-black tree are followed by a self-balancing operation. This ensures that all leaf nodes are the same number of black nodes away from the root node. Figure 3-12 shows the state of a red-black tree after inserting the integer values 1 through 9 in the following insertion order: 9, 3, 5, 6, 7, 2, 8, 4, 1. (Red nodes are shown lightly shaded.)

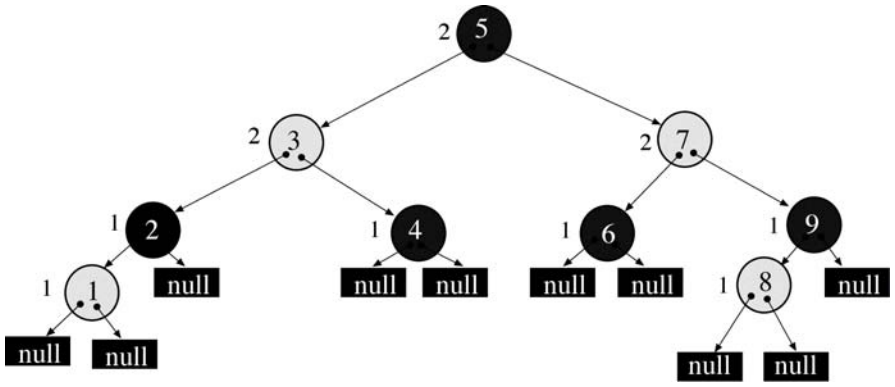

Figure 3-12: Red-Black Tree After Inserting Integer Values 9, 3, 5, 6, 7, 8, 4, 1

Referring to Figure 3-12 — the numbers appearing to the left of each node represent the height of the tree in black nodes. The primary benefit associated with a red-black tree is the generally overall good node search perfor-

mance regardless of the number of nodes the tree contains. However, because the tree reorders itself with each insertion, an insertion into a tree that contains lots of nodes incurs a performance penalty.

Think of it in terms of a clean room versus a messy room. You can store things really fast in a messy room because you just throw your stuff anywhere. Finding things in a messy room takes some time. You may have to look high and low before finding what you're looking for. Storing things in a clean room, conversely, takes a little while longer, but when you need something, you can find if fast!

Stacks And Queues

Two additional data structures you'll encounter in the collections API are *stacks* and *queues*. A stack is a data structure that stores objects in a *last-in-first-out* (LIFO) basis. Objects are placed on the stack with a *push* operation and removed from the stack with a *pop* operation. A stack operates like a plate dispenser, where you put in a stack of plates and take plates off the stack one at a time. The last plate inserted into the plate dispenser is the first plate dispensed when someone needs a plate. Figure 3-13 shows the state of a stack after several pushes and pops.

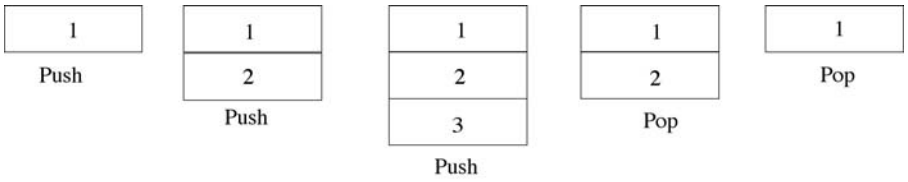

Figure 3-13: A Stack After Several Push and Pop Operations

A *queue* is a data structure that stores objects in a *first-in-first-out* (FIFO) basis. A queue operates like a line of people waiting for some type of service; the first person in line is the first person to be served. People arriving in line must wait for service until they reach the front of the line. Objects are added to a queue with an *enqueue* operation and removed with a *dequeue* operation. Figure 3-14 shows the state of a queue after several enqueues and dequeues.

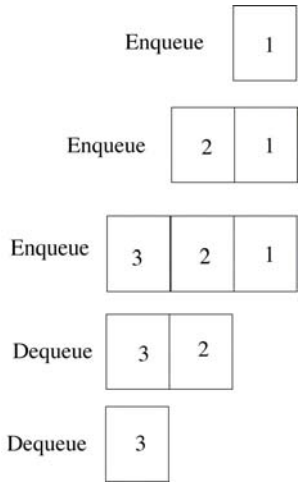

Figure 3-14: A Queue After Several Enqueue and Dequeue Operations

Summary

An *array* is a contiguous allocation of objects in memory. An array-based collection offers quick element access but slow element insertion, especially if the collection's underlying array must be resized and its contents shifted to accommodate the insertion.

A *linked list* consists of individual *nodes* linked to each other via references. To traverse a linked list, you must start at the beginning, or the end (head or tail) and follow each element to the next. Linked list-based collections can conserve memory space because memory need only be allocated on each object insertion. Insertions into linked list-based collections are relatively quick but element access is relatively slow due to the need to traverse the list.

A *chained hash table* is a cross between an array and a linked list and allows element insertion with key/value pairs. A *hash function* performed on the key determines the value's location in the hash table. A collision is said to occur when two keys produce the same hash code. When this happens, the values are chained together in a linked list-like structure. A hash function that produces a uniform distribution over all the keys is a critical feature of a hash table.

A red-black tree is a self-balancing binary tree. Insertions into a red-black tree take some time because of element ordering and balancing operations. Element access times for a red-black tree-based collection is fairly quick.

References

Microsoft Developer Network (MSDN) .NET Framework 3.5 Documentation [http://www.msdn.com]

John Franco. *Red-Black Tree Demonstration Applet*. [http://www.ececs.uc.edu/~franco/C321/html/RedBlack/redblack.html]

Thomas H. Cormen, et. al. *Introduction To Algorithms*. The MIT Press, Cambridge, MA. ISBN: 0-262-03141-8

Donald E. Knuth. *The Art of Computer Programming, Volume 1, Fundamental Algorithms*, *Third Edition*. Addison-Wesley, Reading, MA. ISBN: 0-201-89685-0

Donald E. Knuth. *The Art of Computer Programming, Volume 3, Sorting and Searching*, *Second Edition*. Addison-Wesley, Reading, MA. ISBN: 0-201-89685-0

Rick Miller. C# For Artists: The Art, Philosophy, and Science of Object-Oriented Programming. ISBN-13: 978-1-932504-07-1. Pulp Free Press.

Notes

