Array-Based Structures

Chapter 2

Array: A Data Structure Case Study

- Arrays are rooted in the mathematical concept of subscripted variables
  - Mathematicians used early computers to evaluate formulas with subscripted variables
  - They needed a way to store these variables
  - Arrays were conceived to fill that need
- Array’s design goals
  - Speed (because the variables would be used in many calculations)
  - Low overhead (because a large number of variables to be represented)

Array Design Considerations

- A dilemma was presented to the designers
  - Most often speed and low overhead are mutually exclusive
- The solution was found in the characteristics of subscripted variables
  - Each variable as a unique ordinal subscript
  - A minimum and maximum value of the subscript
  - All values stored in a set of variables are of one type
  - Insert and Delete operations not allowed (only Fetch and Update)
- These characteristics permit contiguous memory model accessed using a mapping function

Contiguous Memory Model

- Because
  - There is a minimum and maximum value of the subscript, and
  - All values stored in a set of variables are of one type
  -...the size of a contiguous memory area can be determined
- For example:
  - For: 0 <= subscript <= 9 and type int (4 bytes), the size of the contiguous area is 40 bytes = 10 * 4

Mapping Function

- Because
  - Each variable as a unique ordinal subscript that could be used as a node number, and
  - The designers stored the nodes in sequential order in contiguous memory
  -...the address of a node (or variable) could be calculated
- The formula to perform the calculation is called a mapping function

One-Dimensional Array Storage Scheme

Gives rise to the mapping function:
Address of xi = Address of x[i] = Ao + (i – io) * w
where: io is the lowest value of the subscript (array index),
Ao is the address of x0

7 Two-Dimensional Array Storage Scheme
• Mapping functions
  Address of Xij = Ao + (i * (jmax + 1) + j) * w  for row major order
  Address of Xij = Ao + (j * (imax + 1) + i) * w  for column major order

8 Design Goals Were Achieved
• Speed
  – Mapping function locates a node quickly
  – Only a calculation is necessary, not a memory search
• Low overhead
  – The only overhead is:
    • The address A0
    • The node width, w
    • The minimum index i0 (not necessary in Java, = 0)

9 Arrays Are a Data Structure
• Although most programmers use arrays, they fail to view them as a data structure
• The reason for this is that by default the data structure array is:
  – A homogeneous structure
  – A static structure (maximum number of nodes must be specified when the structure is created)
  – Accessed in the node number mode
  – Only the Fetch (a = x[2]) and Update (x[2] = 5) operations are allowed
  – Design goals: speed and low overhead

10 Programmer Defined Array-Based Structures
• These are data structures that
  – Use arrays in their implementation
  – Each array element is a “node” reference variable
  – Each array element stores the address of a node
• Unlike arrays, these structures
  – Are accessed in the key field mode
  – Support all four basic operations
• The three examples are:
  – Unsorted Array structure
  – Sorted Array structure
  – Unsorted-Optimized Array structure

11 Unsorted Array Structure
All elements of the array are initialized to null for this, any other array-based, structure.

- Insert—uses the next available null array element to store the new node location
- Delete—
  - Uses a sequential search
  - Moves up all nodes below the deleted node (garbage collection technique)
- Fetch—uses a sequential search
- Update—combines a Delete and Insert

### Unsorted Array Structure Insert Algorithm

```java
// No error checking performed
// Store a deep copy of the node to be
// inserted, e.g. newNode
data[next] = newNode.deepCopy()
```

### Unsorted Array Structure Delete Algorithm

```java
// Access the node (assumes the node is in
// the structure)
i = 0;
while(targetKey != data[i].key) // not found
{
  i++;
}
// Move all the references up to eliminate
// the node and collect the garbage
for( j = i; j < next - 1; j++)
{
  data[j] = data[j + 1];
}
next++; data[next] = null;
```

### Unsorted Array Structure Fetch Algorithm

```java
// Access the node (assumes the node is in
// the structure)
i = 0;
while ( targetKey != data[i].getKey() )
{
  i++;
}
```
// Return a clone of the node to the client
return data[i].deepCopy();

19 [[ Speed of the Unsorted Array Structure
    Containing n Nodes
    • Insert
      – No loops, 3 memory accesses, O(1)
    • Fetch
      – Loop in the sequential search executes an average of n/2 times, with 2 memory accesses each time, O(n)
    • Delete
      – Each loop executes an average of n/2 times, each performing 2 memory accesses (4 total) each time, O(n)
    • Update
      – Combines a Delete and Insert Operation, 2n+3 memory accesses, O(n)
    • Average operation:1.25n + 1.5 memory accesses, O(n)

21 [[ Sorted Array Structure
    • All elements of the array are initialized to null
    • Nodes are stored in sorted order based on key
    • Fetch uses a binary search
    • Delete
      – Uses a binary search and then ...
      – Moves up all node references below the deleted node (garbage collection technique)
    • Insert
      – First moves larger nodes references down in array
      – Then places inserted node in its sorted order position

23 [[ Sorted Array Structure Fetch Algorithm (Page 1)
    // Access the node using a binary search
    // (assumes it is in the structure)
    low = 0;
    high = next – 1;
    i = (low + high) / 2;

24 [[ Sorted Array Structure Fetch Algorithm (Page 2)
    // Binary search
    while (targetKey != data[i].key)
    {
      if( targetKey < data[i].key && high != low )
// Move high down to eliminate
// the upper half of array
{  high = i – 1; }
else
  // Move low up to eliminate
  // the lower half of the array
  {  low = i + 1; }
  i = (low + high) / 2;
}

25 Sorted Array Structure Fetch Algorithm (Page 3)
// Return the fetched node
return data[i].deepCopy();

27 Sorted Array Structure Delete Algorithm (Page 1)
  // Access the node using a binary search
  // (assumes it is in the structure)
  low = 0;
  high = next – 1;
  i = (low + high) / 2;

28 Sorted Array Structure Delete Algorithm (Page 2)
  // Binary search
  while (targetKey != data[i].key)
  {
    if( targetKey < data[i].key && high != low)
      // Move high down to eliminate
      // the upper half of array
      {  high = i – 1; }
    else
      // Move low up to eliminate
      // the lower half of the array
      {  low = i + 1; }
      i = (low + high) / 2;
  }

29 Sorted Array Structure Delete Algorithm (Page 3)
  // Move node references up to delete
  // the node and collect the garbage
  for(j = i; j < next - 1; j++)
  {
    data[j] = data[j + 1];
for(j = i; j < next - 1; j++) {
    data[j] = data[j + 1];
}
next--;
data[next] = null;

31 Sorted Array Structure Insert Algorithm (Page 1)
   // Find the new node's sorted order position
   // using a binary search,
   // targetKey is its key
   low = 0;
   high = next - 1;
   i = (low + high) / 2;

32 Sorted Array Structure Insert Algorithm (Page 2)
   // Binary search
   while (targetKey != data[i].key) {
       if (targetKey < data[i].key && high != low) {
           // Move high down to eliminate
           // the upper half of array
           high = i + 1;
       } else {
           // Move low up to eliminate
           // the lower half of the array
           low = i - 1;
       }
   }

33 Sorted Array Structure Insert Algorithm (Page 3)
   // Move all the node references down to
   // "open up" a spot for the new node
   for(j = next; j >= i; j--) {
       data[j] = data[j - 1];
   }
next++;
   // Add a deep copy of the new node to
   // the structure
   data[i] = newNode.deepCopy();

34 Speed of the Sorted Array Structure Containing n Nodes
   • Fetch
Reduces the average number of times subsequent sequential searches execute

Delete
- Loop to move up the node references executes an average of n/2 times, with 2 memory accesses each time, O(n)

Insert
- Loop to move the node references down executes an average of n/2 times, with 2 memory accesses each time, O(n)

Update
- Combines a Delete and Insert Operation, 2n memory accesses, O(n)

Average operation: n + \( \frac{1}{2} \log_2 n \) memory accesses, O(n)

Nodes in most data bases do not have equal probability of being fetched

•

35 Unsorted-Optimized Structure
- All elements of the array are initialized to null
- The Insert algorithm is the same as the Unsorted Array structure, O(1)
- The Fetch and Delete algorithms of the Unsorted Array structure are modified to improve their speed

36 Unsorted-Optimized Fetch Improvement
- A sequential search is still used to locate a node, however:
  - After a node is fetched, its reference is swapped with the reference above it
  - Most frequently fetched nodes “bubble-up” to the top of the array
- Reduces the average number of times subsequent sequential searches execute
  - Nodes in most data bases do not have equal probability of being fetched

38 Unsorted-Optimized Fetch Algorithm (Page 1)

```java
// Assumes the node is in the structure
i = 0;
// Sequential search
while (targetKey != data[i].key)
{
    i++;
}
// Copy the node before it is relocated
node = data[i].deepCopy();
```

39 Unsorted-Optimized Fetch Algorithm (Page 2)

```java
// Move the node reference up one position
if (i != 0)
{
    temp = data[i - 1];
    data[i - 1] = data[i];
    data[i] = temp;
}
```
data[i] = temp;
}
// Return a copy of the node
return node;

40 □ Unsorted-Optimized Delete Improvement

- A sequential search is still used to locate a node; however...
  - The loop to move up all the node references is eliminated
  - Only the last node reference is moved; it overwrites the deleted node's reference
- This eliminates the $n$ memory accesses doubling the speed of a Delete operation

42 □ Unsorted-Optimized Delete Algorithm

// Assumes the node is in the structure
// access the node
i = 0;

// Sequential search
while (targetKey != data[i].key)
{
    i++;
}

// Move the last node into the deleted
// node's position
data[i] = data[next - 1];
next++;
Best Array-Based Structure

- When all operations equally probable
  - Unsorted-Optimized Array is the fastest
  - Its average memory accesses of \(0.75n + 1.5\) is < Sorted Array \((n + \frac{1}{2}\log_2n)\) < Unsorted Array \((1.25n + 1.5)\)

- When most operations are:
  - Inserts: the Unsorted structures are the fastest
  - Fetches: the Sorted structure is the fastest

- All three structures have the same density

Density of the Array Based Structures

- Density = information bytes / total bytes
  - Information bytes = \(n \times w\)
    - \(n\) is the number of nodes, \(w\) is the bytes per node
  - Overhead = \(4n + 4\)
    - 4 bytes per array element + 4 bytes for next
  - Density = \(\frac{n \times w}{n \times w + 4n + 4}\)
    = \(\frac{1}{1 + 4 / w + 4 / (n \times w)}\)
  - As \(n\) gets large above approaches \(\frac{1}{1 + 1 / w}\)

Implementation of a Data Structure
As a Separate Class

- An data base application is coded as three separate classes
  - The data structure class (e.g., the Optimized(Array)
    - Implements the operation algorithms expanded to include error checking
  - The node definition class (e.g., class Listing)
  - The application class
    - Declares an object in the data structure class
    - Invokes the data structure class’ operation methods

- The data structure and node classes include utility methods

Error Checking
in the Unsorted-Optimized Structure

- The Insert algorithm should return false if
  - The structure is full
  - There is insufficient system memory to perform the deep copy

- If the node is not in the structure
  - The Fetch and Delete algorithms should return null and false respectively

Error Checking in the Optimized-Unsorted Array’s Insert Algorithm
• To detect structure full (array is completely used)
  – An integer variable, size, is added to the data structure to store the size of the array
  – The test for full is added to the front of the algorithm
    if (next == size) // structure full
      return false;

• To detect insufficient system memory
  – In this case, Java’s new operator returns null,
  – So, after the deep copy is performed add
    if(data[next] == null) // no system memory
      return false;

50 Error Checking in the Unsorted-Optimized Fetch and Delete Algorithms
• The Boolean condition in the algorithms’ while loop is expanded
  while (i < next && targetKey != data[i].key)
• When the loop ends
  – In the Fetch algorithm we add
    if(i == next); // node not found
      return null;
  – In the Delete algorithm we add
    if(i == next); // node not found
      return false;

51 Utility Methods
• A method to output all nodes, show
• A constructor to allow the application to specify max number of nodes
• A method to input the contents of a node, input

52 Dynamic Array-Based Structures
• Unlike arrays, array-based structures can expanded at run time (within the limits of system memory)
• The expansion algorithm is added to the Insert algorithm
  – It uses one of two copy techniques
• The expansion does decrease the speed of the insert operation that triggered the expansion

53 Array-Based Structure Expansion Algorithm
Added to the Insert Operation
• When a full structure is detected
  1- An array reference variable, temp, is set pointing to the structure’s filled array, data
  2- A larger array is declared, e.g., larger
  3- The reference variable data is set pointing to the array larger
4. The references from the filled array, temp, are copied into the array larger
5. temp is set to null to recycle the smaller array's storage

55. **Array Copy Techniques**
   - Two techniques to copy the array temp into the array data
     - Code a for loop
       ```java
       for(int i=0; i<temp.length; i++)
       {  data[i] = temp[i]; }
       ```
     - Use Java’s arraycopy method
       ```java
       System.arraycopy(temp, 0, data, 0, temp.length);
       ```
   - Use of arraycopy is 50% faster

56. **Generic Array-Based Structures**
   - Any type of node could be stored in the structure
   - Certain considerations must be followed in the design of the structure
   - The client application specifies the type of nodes that will be stored in the structure
   - The generic features of Java are used in the structure’s generic implementation

57. **Design Considerations for Generic Data Structures**
   - The node definition and the data structure are coded as two separate classes
   - The data structure cannot mention the names of the data fields that make up a node
   - If the structure is going to be encapsulated, a method to perform a deep copy of a node must be coded in the node definition class
   - If the structure is going to be accessed in the key field mode, a method to determine if a given key is equal to the key of a node must be coded in the node definition class
   - The data structure code cannot mention the name of the node class
   - The data structure class cannot mention the key field’s type

58. **Client Side Syntax of Generic Data Structures**
   - To declare instance, boston, of a generic data structure class, UOA, that will store Listing nodes, the client codes
     ```java
     UAO <Listing> boston = new <Listing> UAO();
     ```
   - Similarly, if the nodes were Car objects the declaration would be
     ```java
     UAO <Car> garage = new <Car> UAO();
     ```
   - An attempt to store nodes of any other type in these structures would result a
compile error

59 **Generic Implementation Techniques**
- A generic placeholder is coded at the end of the class heading e.g., public class UOA<T>

- The array reference (data) is declared to be of type T e.g., private T[] data;

- The array is declared to be an array of Object references whose location is coerced into the variable data
  
  ```java
  data = (T[]) new Object[100];
  ```

60 **Generic Implementation Techniques (continued)**
- The type of the nodes stored in the structure is type T and the type of the key is Object

  ```java
  public boolean update(Object targetKey, T newNode)
  ```

- An interface (implemented by the node definition class) must contain the signatures of the deepCopy, toString, and compareTo methods
  - In the data structure, these methods must operate on an instance of the interface

61 **Java’s API ArrayList Class**
- A generic data structure
  - Access is in the *node number* mode
  - Supports the *four* basic operations
  - Not encapsulated
  - Stores *objects*, but primitive types are wrapped automatically in Wrapper objects
  - expandable
- Client codes

  ```java
  ArrayList<Car> garage = new ArrayList<Car>(200)
  ```

  for an *initial* capacity of 200 Car objects (default is a capacity of 10)

62 **Four Basic Operations in the Class ArrayList**