Data Structures – topcoder

Even though computers can perform literally millions of mathematical computations per second, when a problem gets large and complicated, performance can nonetheless be an important consideration. One of the most crucial aspects to how quickly a problem can be solved is how the data is stored in memory.

To illustrate this point, consider going to the local library to find a book about a specific subject matter. Most likely, you will be able to use some kind of electronic reference or, in the worst case, a card catalog, to determine the title and author of the book you want. Since the books are typically shelved by category, and within each category sorted by author's name, it is a fairly straightforward and painless process to then physically select your book from the shelves.

Now, suppose instead you came to the library in search of a particular book, but instead of organized shelves, were greeted with large garbage bags lining both sides of the room, each arbitrarily filled with books that may or may not have anything to do with one another. It would take hours, or even days, to find the book you needed, a comparative eternity. This is how software runs when data is not stored in an efficient format appropriate to the application.

Simple Data Structures
The simplest data structures are primitive variables. They hold a single value, and beyond that, are of limited use. When many related values need to be stored, an array is used. It is assumed that the reader of this article has a solid understanding of variables and arrays.

A somewhat more difficult concept, though equally primitive, are pointers. Pointers, instead of holding an actual value, simply hold a memory address that, in theory, contains some useful piece of data. Most seasoned C++ coders have a solid understanding of how to use pointers, and many of the caveats, while fledgling programmers may find themselves a bit spoiled by more modern "managed" languages which, for better or worse, handle pointers implicitly. Either way, it should suffice to know that pointers "point" somewhere in memory, and do not actually store data themselves.

A less abstract way to think about pointers is in how the human mind remembers (or cannot remember) certain things. Many times, a good engineer may not necessarily know a particular formula/constant/equation, but when asked, they could tell you exactly which reference to check.

Arrays
Arrays are a very simple data structure, and may be thought of as a list of a fixed length. Arrays are nice because of their simplicity, and are well suited for situations where the number of data items is known (or can be programmatically determined). Suppose you need a piece of code to calculate the average of several numbers. An array is a perfect data structure to hold the individual values, since they have no specific order, and the required computations do not require any special handling other than to iterate through all of the values. The other big strength of arrays is that they can be accessed randomly, by index. For instance, if you have an array containing a list of names of students seated in a classroom, where each seat is numbered 1 through n, then studentName[i] is a trivial way to read or store the name of the student

in seat i.

An array might also be thought of as a pre-bound pad of paper. It has a fixed number of pages, each page holds information, and is in a predefined location that never changes.

Linked Lists
A linked list is a data structure that can hold an arbitrary number of data items, and can easily change size to add or remove items. A linked list, at its simplest, is a pointer to a data node. Each data node is then composed of data (possibly a record with several data values), and a pointer to the next node. At the end of the list, the pointer is set to null.

By nature of its design, a linked list is great for storing data when the number of items is either unknown, or subject to change. However, it provides no way to access an arbitrary item from the list, short of starting at the beginning and traversing through every node until you reach the one you want. The same is true if you want to insert a new node at a specific location. It is not difficult to see the problem of inefficiency.

A typical linked list implementation would have code that defines a node, and looks something like this:

```
class ListNode {
   String data;
   ListNode nextNode;
}
ListNode firstNode;
```

You could then write a method to add new nodes by inserting them at the beginning of the list:

```
ListNode newNode = new ListNode();
NewNode.nextNode = firstNode;
firstNode = newNode;

Iterating through all of the items in the list is a simple task:

ListNode curNode = firstNode;
while (curNode != null) {
   ProcessData(curNode);
   curNode = curNode.nextNode;
}
```

A related data structure, the doubly linked list, helps this problem somewhat. The difference from a typical linked list is that the root data structure stores a pointer to both the first and last nodes. Each individual node then has a link to both the previous and next node in the list. This creates a more flexible structure that allows travel in both directions. Even still, however, this is rather limited.

Queues
A queue is a data structure that is best described as "first in, first out". A real world example of a queue is

people waiting in line at the bank. As each person enters the bank, he or she is "enqueued" at the back of the line. When a teller becomes available, they are "dequeued" at the front of the line.

Perhaps the most common use of a queue within a topcoder problem is to implement a Breadth First Search (BFS). BFS means to first explore all states that can be reached in one step, then all states that can be reached in two steps, etc. A queue assists in implementing this solution because it stores a list of all state spaces that have been visited.

A common type of problem might be the shortest path through a maze. Starting with the point of origin, determine all possible locations that can be reached in a single step, and add them to the queue. Then, dequeue a position, and find all locations that can be reached in one more step, and enqueue those new positions. Continue this process until either a path is found, or the queue is empty (in which case there is no path). Whenever a "shortest path" or "least number of moves" is requested, there is a good chance that a BFS, using a queue, will lead to a successful solution.

Most standard libraries, such the Java API, and the .NET framework, provide a Queue class that provides these two basic interfaces for adding and removing items from a queue.

BFS type problems appear frequently on challenges; on some problems, successful identification of BFS is simple and immediately, other times it is not so obvious.

A queue implementation may be as simple as an array, and a pointer to the current position within the array. For instance, if you know that you are trying to get from point A to point B on a 50×50 grid, and have determined that the direction you are facing (or any other details) are not relevant, then you know that there are no more than 2,500 "states" to visit. Thus, your queue is programmed like so:

```
class StateNode {
   int xPos;
   int yPos;
   int moveCount;
}

class MyQueue {
   StateNode[] queueData = new StateNode[2500];
   int queueFront = 0;
   int queueBack = 0;

   void Enqueue(StateNode node) {
      queueData[queueBack] = node;
      queueBack++;
   }

   StateNode Dequeue() {
      StateNode returnValue = null;
      if (queueBack > queueFront) {
      returnValue = queueData[queueFront];
```

```
      QueueFront++;
   }
   return returnValue;
   }

   boolean isNotEmpty() {
      return (queueBack > queueFront);
   }
}
```

Then, the main code of your solution looks something like this. (Note that if our queue runs out of possible states, and we still haven't reached our destination, then it must be impossible to get there, hence we return the typical "-1" value.)

```
MyQueue queue = new MyQueue();
queue.Enqueue(initialState);
while (queue.isNotEmpty()) {
   StateNode curState = queue.Dequeue();
   if (curState == destState)
return curState.moveCount;
   for (int dir = 0; dir < 3; dir++) {
      if (CanMove(curState, dir))
         queue.Enqueue(MoveState(curState, dir));
   }
}
```

Stacks

Stacks are, in a sense, the opposite of queues, in that they are described as "last in, first out". The classic example is the pile of plates at the local buffet. The workers can continue to add clean plates to the stack indefinitely, but every time, a visitor will remove from the stack the top plate, which is the last one that was added.

While it may seem that stacks are rarely implemented explicitly, a solid understanding of how they work, and how they are used implicitly, is worthwhile education. Those who have been programming for a while are intimately familiar with the way the stack is used every time a subroutine is called from within a program. Any parameters, and usually any local variables, are allocated out of space on the stack. Then, after the subroutine has finished, the local variables are removed, and the return address is "popped" from the stack, so that program execution can continue where it left off before calling the subroutine.

An understanding of what this implies becomes more important as functions call other functions, which in turn call other functions. Each function call increases the "nesting level" (the depth of function calls, if you will) of the execution, and uses increasingly more space on the stack. Of paramount importance is the case of a recursive function. When a recursive function continually calls itself, stack space is quickly used as the depth of recursion increases. Nearly every seasoned programmer has made the mistake of writing a recursive function that never properly returns, and calls itself until the system throws up an "out of stack space" type of error.

Nevertheless, all of this talk about the depth of recursion is important, because stacks, even when not used explicitly, are at the heart of a depth first search. A depth first search is typical when traversing through a tree, for instance looking for a particular node in an XML document. The stack is responsible for maintaining, in a sense, a trail of what path was taken to get to the current node, so that the program can "backtrack" (e.g. return from a recursive function call without having found the desired node) and proceed to the next adjacent node.

Soma (SRM 198) is an excellent example of a problem solved with this type of approach.

Trees

Trees are a data structure consisting of one or more data nodes. The first node is called the "root", and each node has zero or more "child nodes". The maximum number of children of a single node, and the maximum depth of children are limited in some cases by the exact type of data represented by the tree.

One of the most common examples of a tree is an XML document. The top-level document element is the root node, and each tag found within that is a child. Each of those tags may have children, and so on. At each node, the type of tag, and any attributes, constitutes the data for that node. In such a tree, the hierarchy and order of the nodes is well defined, and an important part of the data itself. Another good example of a tree is a written outline. The entire outline itself is a root node containing each of the top-level bullet points, each of which may contain one or more sub-bullets, and so on. The file storage system on most disks is also a tree structure.

Corporate structures also lend themselves well to trees. In a classical management hierarchy, a President may have one or more vice presidents, each of whom is in charge of several managers, each of whom presides over several employees.

PermissionTree (SRM 218) provides an unusual problem on a common file system.

bloggoDocStructure (SRM 214) is another good example of a problem using trees.

Binary Trees

A special type of tree is a binary tree. A binary tree also happens to be one of the most efficient ways to store and read a set of records that can be indexed by a key value in some way. The idea behind a binary tree is that each node has, at most, two children.

In the most typical implementations, the key value of the left node is less than that of its parent, and the key value of the right node is greater than that of its parent. Thus, the data stored in a binary tree is always indexed by a key value. When traversing a binary tree, it is simple to determine which child node to traverse when looking for a given key value.

One might ask why a binary tree is preferable to an array of values that has been sorted. In either case, finding a given key value (by traversing a binary tree, or by performing a binary search on a sorted array) carries a time complexity of O(log n). However, adding a new item to a binary tree is an equally simple operation. In contrast, adding an arbitrary item to a sorted array requires some time-consuming

reorganization of the existing data in order to maintain the desired ordering.

If you have ever used a field guide to attempt to identify a leaf that you find in the wild, then this is a good way to understand how data is found in a binary tree. To use a field guide, you start at the beginning, and answer a series of questions like "is the leaf jagged, or smooth?" that have only two possible answers. Based upon your answer, you are directed to another page, which asks another question, and so on. After several questions have sufficiently narrowed down the details, you are presented with the name, and perhaps some further information about your leaf. If one were the editor of such a field guide, newly cataloged species could be added to field guide in much the same manner, by traversing through the questions, and finally at the end, inserting a new question that differentiates the new leaf from any other similar leaves. In the case of a computer, the question asked at each node is simply "are you less than or greater than X?"

Priority Queues

In a typical breadth first search (BFS) algorithm, a simple queue works great for keeping track of what states have been visited. Since each new state is one more operational step than the current state, adding new locations to the end of the queue is sufficient to insure that the quickest path is found first. However, the assumption here is that each operation from one state to the next is a single step.

Let us consider another example where you are driving a car, and wish to get to your destination as quickly as possible. A typical problem statement might say that you can move one block up/down/left/right in one minute. In such a case, a simple queue-based BFS works perfectly, and is guaranteed to provide a correct result.

But what happens if we say that the car can move forward one block in two minute, but requires three minutes to make a turn and then move one block (in a direction different from how the car was originally facing)? Depending on what type of move operation we attempt, a new state is not simply one "step" from the current state, and the "in order" nature of a simple queue is lost.

This is where priority queues come in. Simply put, a priority queue accepts states, and internally stores them in a method such that it can quickly pull out the state that has the least cost. (Since, by the nature of a "shortest time/path" type of problem, we always want to explore the states of least cost first.)

A real world example of a priority queue might be waiting to board an airplane. Individuals arriving at their gate earlier will tend to sit closest to the door, so that they can get in line as soon as they are called. However, those individuals with a "gold card", or who travel first class, will always be called first, regardless of when they actually arrived.

One very simple implementation of a priority queue is just an array that searches (one by one) for the lowest cost state contained within, and appends new elements to the end. Such an implementation has a trivial time-complexity for insertions, but is painfully slow to pull objects out again.

A special type of binary tree called a heap is typically used for priority queues. In a heap, the root node is always less than (or greater than, depending on how your value of "priority" is implemented) either of its children. Furthermore, this tree is a "complete tree" from the left. A very simple definition of a complete tree

is one where no branch is n + 1 levels deep until all other branches are n levels deep. Furthermore, it is always the leftmost node(s) that are filled first.

To extract a value from a heap, the root node (with the lowest cost or highest priority) is pulled. The deepest, rightmost leaf then becomes the new root node. If the new root node is larger than at at least one of its children, then the root is swapped with its smallest child, in order to maintain the property that the root is always less than its children. This continues downward as far as necessary. Adding a value to the heap is the reverse. The new value is added as the next leaf, and swapped upward as many times as necessary to maintain the heap property.

A convenient property of trees that are complete from the left is that they can be stored very efficiently in a flat array. In general, element 0 of the array is the root, and elements 2k + 1 and 2k + 2 are the children of element k. The effect here is that adding the next leaf simply means appending to the array.

Hash Tables

Hash tables are a unique data structure, and are typically used to implement a "dictionary" interface, whereby a set of keys each has an associated value. The key is used as an index to locate the associated values. This is not unlike a classical dictionary, where someone can find a definition (value) of a given word (key).

Unfortunately, not every type of data is quite as easy to sort as a simple dictionary word, and this is where the "hash" comes into play. Hashing is the process of generating a key value (in this case, typically a 32 or 64 bit integer) from a piece of data. This hash value then becomes a basis for organizing and sorting the data. The hash value might be the first n bits of data, the last n bits of data, a modulus of the value, or in some cases, a more complicated function. Using the hash value, different "hash buckets" can be set up to store data. If the hash values are distributed evenly (which is the case for an ideal hash algorithm), then the buckets will tend to fill up evenly, and in many cases, most buckets will have no more than one or only a few objects in them. This makes the search even faster.

A hash bucket containing more than one value is known as a "collision". The exact nature of collision handling is implementation specific, and is crucial to the performance of the hash table. One of the simplest methods is to implement a structure like a linked list at the hash bucket level, so that elements with the same hash value can be chained together at the proper location. Other, more complicated schemes may involve utilizing adjacent, unused locations in the table, or re-hashing the hash value to obtain a new value. As always, there are good and bad performance considerations (regarding time, size, and complexity) with any approach.

Another good example of a hash table is the Dewey decimal system, used in many libraries. Every book is assigned a number, based upon its subject matterï¿½ the 500′s are all science books, the 700′s are all the arts, etc. Much like a real hash table, the speed at which a person could find a given book is based upon how well the hash buckets are evenly dividedï¿½ It will take longer to find a book about frogs in a library with many science materials than in a library consisting mostly of classical literature.

In applications development, hash tables are a convenient place to store reference data, like state abbreviations that link to full state names. In problem solving, hash tables are useful for implementing a

divide-and-conquer approach to knapsack-type problems. In LongPipes, we are asked to find the minimum number of pipes needed to construct a single pipe of a given length, and we have up to 38 pieces of pipe. By dividing this into two sets of 19, and calculating all possible lengths from each set, we create hash tables linking the length of the pipe to the fewest number of segments used. Then, for each constructed pipe in one set, we can easily look up, whether or not we constructed a pipe of corresponding length in the other set, such that the two join to form a complete pipe of the desired length.

Conclusion

The larger picture to be seen from all of this is that data structures are just another set of tools that should be in the kit of a seasoned programmer. Comprehensive libraries and frameworks available with most languages nowadays preempt the need for a full understanding of how to implement each of these tools. The result is that developers are able to quickly produce quality solutions that take advantage of powerful ideas. The challenge lies in knowing which one to select.

Nonetheless, knowing a little about how these tools work should help to make the choices easier. And, when the need arises, perhaps leave the programmer better equipped to think up a new solution to a new problemï¿½ if not while on the job doing work for a client, then perhaps while contemplating the 1000 point problem 45 minutes into the coding phase of the next SRM.