A Basic Introduction to Algorithms

by Chuck Easttom

Introduction

Data structures and algorithms are a fundamental aspect of computer programming and computer science. It is difficult, if not impossible to be effective at programming without a knowledge of these two topics. In this paper we will examine the basic data structures and algorithms that are commonly studied in many college courses. This paper will be both descriptive and analytical. The purpose of this paper is to provide a broad overview of common data structures, algorithms, and the basics of algorithm analysis. Clearly one paper, regardless of length, cannot substitute for an entire textbook. However the key points needed to understand the topics, are covered.

As each algorithm is examined, particular attention will be paid to the relative efficiency of that algorithm. This can be measured in several ways (Drozdek 2001).

- The number of iterations the algorithm requires. In any algorithm, the process will need to be repeated (at least for sorting or searching algorithms). The number of iterations required, both on average, and worst case, is one way to measure the efficiency of the algorithm.
- The amount of resources required. Any algorithm will require a certain amount of resources, usually RAM (Random Access Memory). The resources required, is yet another way to evaluate the efficiency of an algorithm.

Specific methods for measuring and analyzing algorithms will be discussed in the section on algorithms.

A
l
g
o
r
i
t
h
m

H
a
n
d
o
u
t

1

Basic Data Structures

A data structure is a formal way of storing data that also defines how the data is to be processed. This means that the definition of any given data structure must detail how the data is going to be stored, and what methods are available for moving data into and out of that particular data structure. There are a number of well-defined data structures that are used in various situations. Different data structures are suited for different applications. In this section we will examine several of the more commonly known data structures.

List

A list is one of the simplest of data structures. It is a structure consisting of an ordered set of elements, each of which may be a number, another list, etc. A list is usually denoted (a_1, a_2, ..., a_n) or $\{a_1, a_2, \ldots, a_n\}$. Any basic array would constitute a list. Most other data structures are some sort of extension of this basic concept (NISTb 2005). A list can be either homogenous or heterogeneous. However the most common implementation of a list, in most programming languages, is the array. And in most programming languages an array is homogenous (i.e. this means all elements of the array are of the same data type). It should be noted that several object oriented programming languages over a type of list known as a collection, that is heterogeneous (i.e. elements of that list may be of diverse data types).

To add data to a list you simply add it on the end, or insert it at a given point based on the index number. To remove data you reference a specific index number and get that data. This just happens to be one major weakness with a list, it can quickly become disordered. The reason is that one can insert items anywhere in the list and remove them anywhere. Even if the list is perfectly sorted, adding items at random intervals can introduce informational entropy.

The ideal place to utilize a list is when you simply need to store data and the order of processing the data in or out of storage is not important. A list is an appropriate data structure to use, especially if your situation specifically requires that you be able to add and remove data at any point in the storage mechanism, rather than in a particular sequence, then the list is an ideal choice.

Queue

A queue is simply a special case of a list. It stores data in the same way that a list does, it is also often implemented as simply an array. The difference between a list and a queue is in the processing of the data contained in either data structure. A queue has a more formal process for adding and removing items, than the list does. In a queue data is processed on a first in first out basis (FIFO) (Drozdek 2001). Often there is some numerical pointer designating where the last data was input (often called a tail) and where the last data was extracted (often called a head). Putting data into the queue is referred to as enqueueing, and removing it is dequeueing. This figure should help clarify that point:

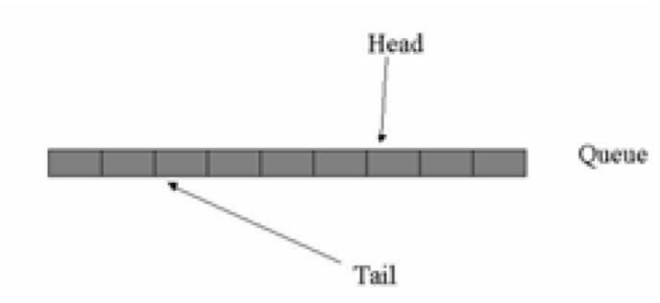

Figure 1 The Queue

Clearly the head and tail must also be moving in some direction. This is usually via a simple increment method (simply the ++ operator in C, C#, or Java). That would mean that in figure 1 they are moving to the right. The queue is a relatively simple data structure to implement. Below is a c-like pseudo code example of a queue implemented in a class:

```
class Q
{
  stringarray[20]; // this queue can hold up to 20
  strings int head, tail;
  void enqueue(string item)
  {
       stringarray[head] =
       item; head++;
   }
   string dequeue()
   {
      return stringarray[tail];
      tail++
   }
}
```

With a queue data is added and removed in a sequential fashion. This is very different from a list, where data may be added to any point in the list. With the queue data is always added at the next spot in the sequence, and processed similarly. However there are two problems that immediately present themselves with the queue. The first problem is how to handle the condition of reaching the end of the queue. The usual answer to this is to create what is known as a circular queue. When the head (or tail) reach the end of the queue, they simply start over at the beginning. Referring to figure 1, this would mean that if the head or tail reach the end, they are simply repositioned back to the beginning. In code you would add to the previous code sample something similar to this pseudo code:

```
if (head= =20)
   head = 0;
and
if (tail = = 20
   ) tail = 0
```

This still leaves us with the second problem, what happens if the head is adding new items faster than the tail is processing them? Without proper coding, the head will overtake the tail, and you will begin overwriting items in the queue that have not been processed, ergo they are lost and will never be processed. This means you will be adding items to a space in the queue that already contains unprocessed data. That unprocessed data will simply be lost. The answer to this is to stop allowing new items to be added, should the head catch up to the tail. This should be communicated to the end user via a ’queue is full’ message.

Of course there is another option, other than the circular queue, and that is the bounded queue. A bounded queue is simply one that can only contain a finite amount of data. When that limit is reached (i.e. when you reach the end of the queue) then the queue is done. Clearly this implementation of the queue is somewhat limited and you will

encounter it much less frequently than the unbounded queue (i.e. any queue which does not have an arbitrary finite limit, such as the circular queue).

The queue is a very efficient data structure and you will find it implemented in many diverse situations. The most common place to encounter a queue is with a printer. Printers usually have a queue in which print jobs are stored awaiting processing. And of course they utilize circular queues. If you overload that queue, then you will get a message telling you the print queue is full. If your programming problem requires an orderly and sequential processing of data on a first in first out basis, then the queue is an ideal data structure to use.

Stack

The stack is a data structure which is a special case of the list. With the stack, elements may be added to or removed from the top only. Adding an item to a stack is referred to as a push, and removing an item is referred to as a pop. In this scenario, the last item added must be the first one removed or Last in First Out (LIFO). A good analogy is to consider a stack of plates. Since the last item in is the first out, this data structure does not require two pointers (a head and tail), merely one. This figure may help clarify this:

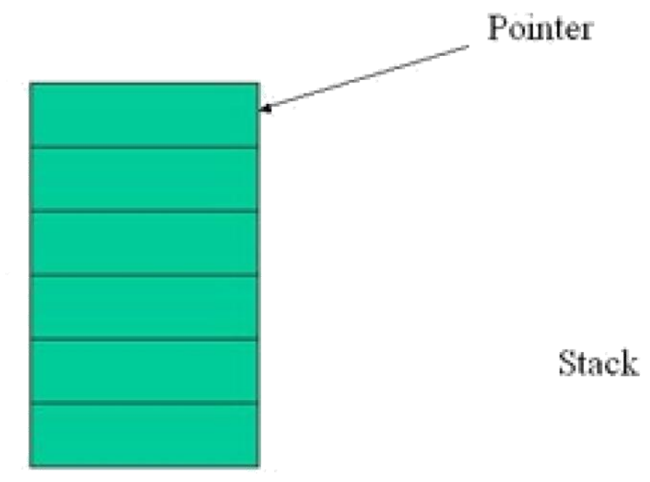

Figure 2 The Stack

The problem with this particular data structure is the LIFO processing. If you have accumulated a number of data items and must get to one of the first you put on the stack, you will first need to process all of the subsequent items. For this reason it is primarily used where a few items need to be stored for a short period of a time. You will find stacks in temporary memory storage. The following code sample shows a class that implements a stack:

```
public class clsstack
{
        private string[] mystack = new
        string[20]; private int count;
        public clsstack()
        {
                count = 0;
        }
```

7

```
        public string pop()
        {
            return mystack [count];
           count --;
        }
        public void push(string item)
        {
            count++;
            mystack [count] = item;
        }
}
```

The push and pop operations are the methods for adding or removing items from the stack. Clearly the push operation is only valid if the stack is not full. Also the pop operation can only be used if the stack is not empty. Our sample code above is simplified for educational purposes. A real implementation would validate the current conditions (is the stack full or empty) prior to executing a push or a pop.

As was mentioned, certain computer memory structures and the CPU registers will often use stacks. However their method of last in first out, makes them poor choices for many standard data processing situations. For example, if a network printer utilized a stack rather than a queue, then it would always attempt to process the last job first. Since new jobs are continually being added, it would be entirely possible for the first job sent to the printer to wait hours, if not days, to be printed. Clearly that would be unacceptable.

Linked List

A linked list is a data structure wherein each item as a pointer (or link) to next item in the list, or the item preceding it, but not both (a double linked list, which will be discussed later in this paper, does both). This can be quite useful since if an item is displaced, for any reason, it is aware of the next item in the list. Each item knows what comes after it, or what comes before it, but not both. The following c like pseudo code shows you how to implement a linked list with a pointer data type to track the previous item in the list

```
public class linkedlist
{
        private string[] mystack = new
        string[20]; private int count;
        private struct item
        {
              string content;
              item* linkto;
         }
       item[] myitems[20];
        public additem(string s)
        {
            item n;
             n.content = s;
              n->linkto = curitem;
```

```
                curitem = n;
```

9

```
        }
}
```

As you can see the distinctive element in the linked list is that it tracks the next member in the chain. A linked list can be implemented that either tracks the preceding or succeeding item. In some cases a linked list is used to implement a modified queue. Rather than each item in the queue being unaware of its surroundings, each one would know the next item in the list. Consider again the issue of a print queue. If implemented via a linked list, each document in the print queue would have a pointer to the next document in the print queue. This would be a more fail-safe method, preventing disruptions in the series of print jobs.

That scenario, however, brings up one of the serious complexities of the linked list. If you do not implement it in such a way that data is processed in an orderly, sequential fashion (such as first in first out) then items can be removed or added anywhere (as with a standard list). This means that when an item is inserted or deleted, you must adjust the pointer of the item is was inserted next too. For example consider this diagram:

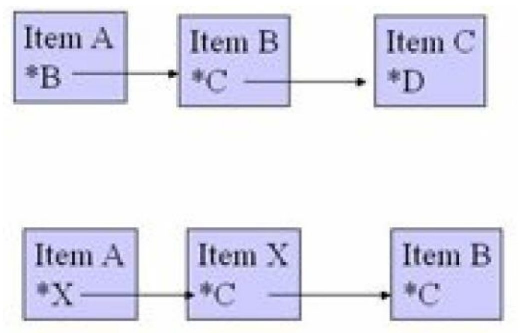

Figure 3 The Linked List

Each of the items (*B, *C, etc.) in the boxes are the pointers to the next item in the linked list. The * notation is the same as the C and C++ language use to denote pointers.

Note that when item X is inserted between item A and item B, that item A’s pointer must also be changed so that it now points to X rather than to B. The same situation occurs if an item is deleted. This problem is minimized, however, if items are always added or removed in a sequential fashion, for example FIFO.

Double Linked List

A double linked list is a data structure wherein each item as a pointer (or link) to the item in front of it, and the item behind it. This data structure is the logical next step after the linked list. It is highly efficient in that any data point has links to the preceding and succeeding data points. The source code is virtually identical to that of a linked list. It simply has two pointers, one for the preceding item and one for the succeeding item. This sort of data structure is highly efficient.

Note there is also a special form of the double linked list called Circularly-linked list circularly-linked list, the first and last nodes are linked together. This can be done for both singly and doubly linked lists. To traverse a circular linked list, you begin at any node and follow the list in either direction until you return to the original node. Viewed another way, circularly-linked lists can be seen as having no beginning or end. This type of list is very useful for managing buffers.

The circularly linked version of the double linked list makes it ideal for the printer buffer. It allows the printer buffer to have every document aware of the documents on either side. This is because it has both a double linked list (each item has a pointer to the previous item and the next item) and its final item is linked back to the first item. This type of structure provides and unbroken chain of items.

Whatever method you use to implement the double linked list (circular, etc.), it has the same complication as the linked list. When you insert or delete an item you must

11

update any other items which have pointers pointing to the item you deleted, or the space you are inserting an item into. The only difference between a double linked list and a single linked list is that when you insert or delete an item in a double linked list you must update the pointers on either side of that item, rather than only one (either preceding or next).

Binary Tree

Yet another interesting data structure is the binary tree. A binary tree is an ordered data structure in which each node has at most two children. Typically the child nodes are called left and right. So in the diagram below the node ‘Science’ has a left child node named ‘Physical’ and a right child node named ‘Biological.

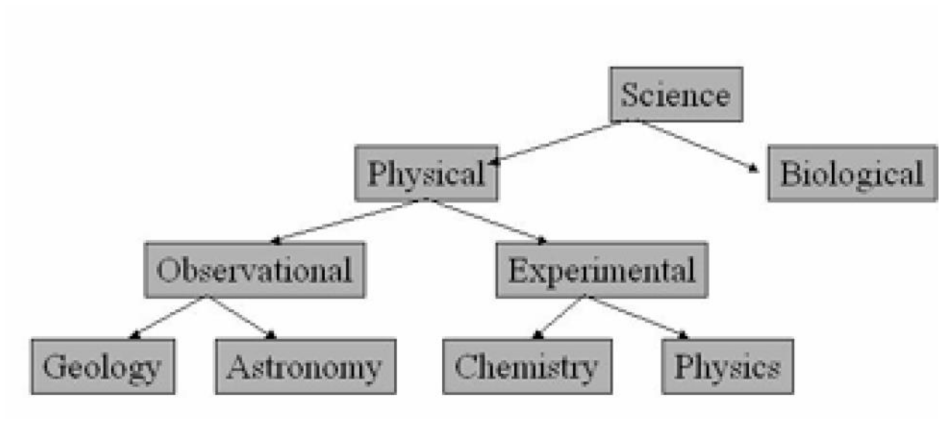

Figure 4 The binary tree

One common use of binary trees is binary search trees.. You can see in figure 4 that each item in the binary tree has child items that are related to the parent node. This is exactly the context in which binary trees are most useful, i.e. when there is a strong parent-child relationship between items. In other words a binary tree would not be useful in grouping items that did not have a clear relationship, for example an arbitrary group of

items such as unrelated items that just happen to be stored in the same warehouse

(perhaps lawn mowers, chairs, and dishes).

The following c-like pseudo code implements a binary

```
tree struct node
{
 int
key_value;
node *left;
node *right; };
```

This structure will then be used in the btree class shown

```
below: class btree
{
 node *root;
 btree();
 ~btree();
 void insert(int key, node *leaf);
 node *search(int key, node *leaf);
 public:
 void insert(int key);
node *search(int key);
void destroy_tree(); };
```

```
btree::btree()
```

13

```
{
 root=NULL;
}
void btree::insert(int key)
{
 if(root!=NULL)
 insert(key, root);
 else
  {
   root=new node; root-
   >key_value=key;
   root->left=NULL;
   root->right=NULL;
  }
}
void btree::insert(int key, node *leaf)
{
 if(key< leaf->key_value)
  {
   if(leaf->left!=NULL)
    insert(key, leaf-
   >left); else
```

{

14

```
      leaf->left=new node; leaf-
      >left->key_value=key;
      leaf->left->left=NULL; //Sets the left child of the child node to null
      leaf->left->right=NULL; //Sets the right child of the child node to null
    }
  }
  else if(key>=leaf->key_value)
  {
    if(leaf->right!=NULL)
    insert(key, leaf->right);
    else
    {
      leaf->right=new node; leaf-
      >right->key_value=key;
      leaf->right->left=NULL; //Sets the left child of the child node to null
      leaf->right->right=NULL; //Sets the right child of the child node to null
    }
  }
}
node *btree::search(int key, node *leaf)
{
  if(leaf!=NULL)
```

{

15

```
    if(key==leaf->key_value)

    return leaf; if(key<leaf-

    >key_value) return

    search(key, leaf->left); else


      return search(key, leaf->right);

  }

 else return NULL;

}
```

It should be noted that the binary tree is actually a special case of the tree. In a binary tree each parent has at most two child nodes. In a complete binary tree each node has precisely zero or two nodes (i.e.a complete binary tree can not have 1 child for any node). It is certainly possible to have a tree with any number of child nodes you wish, but those would simply not be binary trees. However binary configurations are much more efficient for search algorithms, and hence the binary tree is a commonly encountered data structure.

Any tree structure, binary or otherwise, is ideal for situations where the data being stored is easily catagorized in some sort of hierarchical fashion. The choice of a binary tree as opposed to some other tree structure (one with n child nodes per parent node) is purely for search optimization.

Heap

A heap is closely related to a binary tree. In fact it is a complete tree where every node has a key more extreme (greater or less) than or equal to the key of its parent. Usually

16

understood to be a binary heap. Because of its structure, the heap can be used as a priority queue. This is because the item with the largest key value (highest priority) will always be at the root of the heap. The Dequeue() function removes this root node, rebuilds the heap so as to maintain the heap quality, and returns the data stored in the root node. This figure shows the structure of a heap.

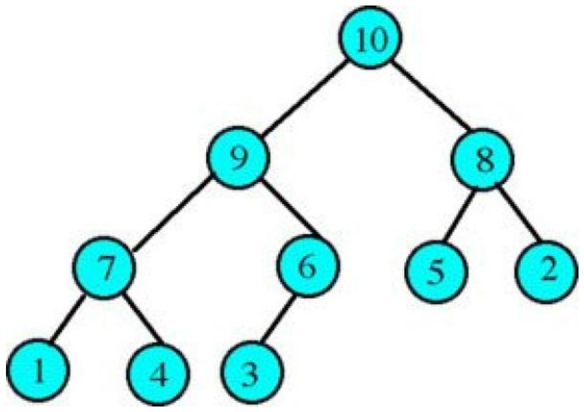

Figure 5 The Heap

Heaps, like binary trees, are a special case of the tree. Also like the binary tree, the heap is useful in sorting algorithms. In particular it is often used with the heap-sort algorithm.

Basic Algorithms

Before we can begin a study of algorithms we must first define what an algorithm is. An algorithm is simply a systematic way of solving a problem. A recipe for an apple pie is an algorithm. If you follow the procedure you get the desired results. Algorithms are a routine part of computer programming. Often the study of computer algorithms centers on sorting algorithms. The sorting of lists is a very common task and therefore a common topic in any study of algorithms.

It is also important that we have a clear method for analyzing the efficacy of a given algorithm. When considering any algorithm if the desired outcome is achieved, then clearly the algorithm worked. But the real question is how well did it work. If you are sorting a list of 10 items, the time it takes to sort the list is not of particular concern. However if your list has 1 million items, then the time it takes to sort the list, and hence the algorithm you choose, is of critical importance. Fortunately there are well-defined methods for analyzing any algorithm.

When analyzing algorithms we often consider the asymptotic upper and lower bounds (NIST 2005). Asymptotic Analysis is a process used to measure the asymptotic performance of computer algorithms. This type of performance is the based a factor called complexity. Usually this is a measure of either the time it takes for an algorithm to work, or the resources (memory) needed. It should be noted that one usually can only optimize time or resources, but not both. The asymptotic upper bound is simply the worst-case scenario for the given algorithm. Whereas the asymptotic lower bound is a best case.

Some analyst prefer to simply use an average case, however knowing the best case and worst can be useful in some situations. In simple terms, both the asymptotic upper bound and lower bounds must be within the parameters of the problem you are attempting to solve. You must assume that the worst case scenario will occur in some situations.

The reason for this disparity between the asymptotic upper and lower bounds has to do with the initial state of a set. If one is applying a sorting algorithm to a set that is at its maximum information entropy (state of disorder) then the time taken for the sorting algorithm to function will be the asymptotic upper bound. If, on the other hand, the set is very nearly sorted, then one may approach or achieve the asymptotic lower bound.

Perhaps the most common way to formally evaluate the efficacy of a given algorithm is Big O notation (Mathworld 2005). This method is measure of the execution of an algorithm, usually the number of iterations required, given the problem size n. In sorting algorithms n is the number of items to be sorted. . Stating some algorithm $f(n) = O(g(n))$ means it is less than some constant multiple of $g(n)$. The notation is read, "f of n is big oh of g of n". This means that saying an algorithm is 2N, means it will have to execute 2 times the number of items on the list. Big O notation essentially measures the asymptotic upper bound of a function. Big O is also the most often used analysis.

Big O Notation was first introduced by the mathematician Paul Bachmann in his 1892 book *Analytische Zahlentheorie*. The notation was popularized in the work of another mathematician named Edmund Landau. Because Landaue was responsible for popularizing this notation. it is sometimes referred to as a Landau symbol.

Ω

Omega notation is the opposite of Big O notation. It is the asymptotic lower bound of an algorithm and gives the best case scenario for that algorithm. It gives you the minimum running time for an algorithm (Cormen 2001).

Theta notation Θ combines Big O and Omega to give the average case (average being arithmetic mean in this situation) for the algorithm. In our analysis we will focus heavily on the Theta, also often referred to as the Big O running time. This average time gives a more realistic picture of how an algorithm executes (Cormen 2001).

Note: It can be confusing when a source refers to a Big O running time other than a theta. In writing this paper I found several online sources that used O notation, when clearly what they where providing was actually .Θ

Now that we have some idea of how to analyze the complexity and efficacy of a given algorithm, lets take a look at a few commonly studied sorting algorithms, and apply these analytical tools.

Merge Sort

The merge sort is a fairly simple algorithm that is reasonably efficient. It works in a recursive manner (i.e. by calling itself until the sort is complete). It first splits the list to be sorted into two equal halves, and places the two halves into separate arrays. Each array is then recursively sorted, and then finally merged back together to form the final sorted list. The merge sort has a Θ and an O of (n ln n). In other words its average and worst case situations are the same. This makes it attractive for some situations. If the average case, or theta, is adequate for a given situation then one need not be concerned about worst case scenarios. Its best case scenario is $\Theta(\frac{1}{2}nlogn)$

Elementary implementations of the merge sort sometimes utilize three arrays. This would be one array for each half of the data set and one to store the final sorted list in. There are non-recursive versions of the merge sort, but they don't yield any significant performance enhancement over the recursive algorithm on most machines. It should also be noted that almost every implementation you will find of merge sort will be recursive. Here is a C code implementation of this algorithm:

```
void mergeSort(int numbers[], int temp[], int array_size)
{
 m_sort(numbers, temp, 0, array_size - 1);
}
void m_sort(int numbers[], int temp[], int left, int right)
{
 int mid;
 if (right > left)
  {
   mid = (right + left) / 2;
   m_sort(numbers, temp, left, mid);
   m_sort(numbers, temp, mid+1, right);
   merge(numbers, temp, left, mid+1, right);
  }
}
void merge(int numbers[], int temp[], int left, int mid, int right)
```

{

21

```
int i, left_end, num_elements, tmp_pos;
left_end = mid - 1;
tmp_pos = left;
num_elements = right - left + 1;
while ((left <= left_end) && (mid <= right))
{
  if (numbers[left] <= numbers[mid])
  {
    temp[tmp_pos] = numbers[left];
    tmp_pos = tmp_pos + 1;
    left = left +1;
  }
  else
  {
    temp[tmp_pos] = numbers[mid];
    tmp_pos = tmp_pos + 1;
    mid = mid + 1;
  }
}
while (left <= left_end)
{
  temp[tmp_pos] = numbers[left];
```

```
   left = left + 1;
```

22

```
    tmp_pos = tmp_pos + 1;
  }
  while (mid <= right)
  {
    temp[tmp_pos] =
    numbers[mid]; mid = mid + 1;
    tmp_pos = tmp_pos + 1;
  }
  for (i=0; i <= num_elements; i++)
  {
    numbers[right] = temp[right];
    right = right - 1;
  }
}
```

You can see that this is relatively simple code. The name derives from the fact that the lists are divided, sorted, then merged. If one had an exceedingly large list, and could separate the two sub lists onto different processors then the efficacy of the merge sort would be significantly improved.

Quick Sort

The quick sort is a very commonly used algorithm. Like the merge sort, it is recursive, meaning that it simply calls itself repeatedly until the list is sorted. Some books will even refer to the quick sort as a more effective version of the merge sort. The quick sorts is the same as merge sort (n ln n) however the difference is that this is also

23

its best case scenario. However it has an O (n^2), which indicates that its worse case scenario is quite inefficient. So for very large lists, the worst case scenario may not be acceptable.

This recursive algorithm consists of three steps (which bear a strong resemblance to the merge sort):

1. Pick an element in the array to serve as a pivot point.
2. Split the array into two parts. The split will occur at the pivot point. So one array will have elements larger than the pivot and the other with elements smaller than the pivot. Clearly one or the other should also include the pivot point.
3. Recursively repeat the algorithm for both halves of the original array.

One very interesting aspect of this algorithm is that the efficiency of the algorithm is significantly impacted by which element is chosen as the pivot point. The worst-case scenario of the quick sort occurs when the list is sorted and the left-most element is chosen, this gives a complexity of , O(n2). Randomly choosing a pivot point rather than using the left-most element is recommended if the data to be sorted isn't random. As long as the pivot point is chosen randomly, the quick sort has an algorithmic complexity of O(n log n). The following source code shows an implementation of a quick sort.

```
void quickSort(int numbers[], int array_size)
{
  q_sort(numbers, 0, array_size - 1);
}
void q_sort(int numbers[], int left, int right)
```

{

24

```
int pivot, l_hold, r_hold;


l_hold = left;
r_hold = right;
pivot = numbers[left];
while (left < right)
{
  while ((numbers[right] >= pivot) && (left < right))
    right--;
  if (left != right)
  {
    numbers[left] =
    numbers[right]; left++;
  }
  while ((numbers[left] <= pivot) && (left <
    right)) left++;
  if (left != right)
  {
    numbers[right] =
    numbers[left]; right--;
  }
}
```

```
numbers[left] = pivot;
```

25

```
  pivot = left;
  left = l_hold;
  right = r_hold;
  if (left < pivot)
    q_sort(numbers, left, pivot-1);
  if (right > pivot) q_sort(numbers,
  pivot+1, right);
}
```

Insertion Sort

The insertion sort works just like its name suggests - it inserts each item into its proper place in the final list. The simplest implementation of this requires two list structures - the source list and the destination list. To save memory, many implementations use an in-place sort that works by moving the current item past the already sorted items and repeatedly swapping it with the preceding item until it is in place.

Like the bubble sort, the insertion sort has a and an O of (n^2). Like the merge sort its average and worst case scenarios are the same. The worst case of n^2 means that for particularly larges lists, the insertion sort would be a very poor choice. The best case scenario would be O(n). The following code sample illustrates an insertion sort.

```
void insertionSort(int numbers[], int array_size)
{
  int i, j, index;
```

```
for (i=1; i < array_size; i++)
```

26

```
  {
    index =
    numbers[i]; j = i;
    while ((j > 0) && (numbers[j-1] > index))
    {
      numbers[j] = numbers[j-
      1]; j = j - 1;
    }
    numbers[j] = index;
  }
}
```

Because of the slowness of the insertion sort it is not seen as often as the merge sort or quick sort. However it can be appropriately used on smaller sets of data.

Heap Sort

The heap sort is not the fastest algorithm, however it does not depend on complex recursive techniques in order to function. This makes it an attractive option for very large data sets of millions of items. It has a of (n log n) and is a special case of the selection sort algorithm (Sedgewick 1995). This algorithm has an $O(n^2)$ meaning that its upper asymptotic bound (worst case scenario) is as bad as the insertion sort.

The heap sort works as it name suggests, it begins by building a heap (recall the discussion of the heap data structure earlier in this paper) out of the data set, and then removing the largest item and placing it at the end of the sorted array. After removing the largest item, it reconstructs the heap and removes the largest remaining item and places it in the next open position from the end of the sorted array. This is repeated until there are no items left in the heap and the sorted array is full. Elementary implementations require two arrays. One array is used to hold the initial heap and the other to hold the sorted elements.

To do an in-place sort and save the space the second array would require, the algorithm implementation shown below uses a short cut. It simply uses the same array to store both the heap and the sorted array. Whenever an item is removed from the heap, it frees up a space at the end of the array that the removed item can be placed in.

```
void heapSort(int numbers[], int array_size)
{
 int i, temp;
 for (i = (array_size / 2)-1; i >= 0; i--)
 siftDown(numbers, i, array_size);
 for (i = array_size-1; i >= 1; i--)
  {
   temp = numbers[0];
   numbers[0] = numbers[i];
   numbers[i] = temp;
   siftDown(numbers, 0, i-1);
  }
}
void siftDown(int numbers[], int root, int bottom)
```

{

28

```
 int done, maxChild, temp;

 done = 0;

 while ((root*2 <= bottom) && (!done))

 {

   if (root*2 == bottom)

     maxChild = root * 2;

   else if (numbers[root * 2] > numbers[root * 2 + 1])

     maxChild = root * 2;

   else

     maxChild = root * 2 + 1;

   if (numbers[root] < numbers[maxChild])

   {

     temp = numbers[root]; numbers[root]

     = numbers[maxChild];

     numbers[maxChild] = temp;

     root = maxChild;

   }

   else

     done = 1;

 }

}
```

The heap sort, in the implementation shown here, does an excellent job of conserving space/memory. In many situations this can make up for what it loses in speed.

29

Bubble Sort

The bubble sort is the oldest and simplest sort in use. By simple I mean that from a programmatic point of view it is very easy to implement. Unfortunately, it's also one of the slowest. It has an O (n^2). This means that for very large lists, it is probably going to be too slow. As with most sort algorithms, its best case (lower asymptotic bound) is O(n)

The bubble sort works by comparing each item in the list with the item next to it, and swapping them if required. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items (in other words, all items are in the correct order). This causes larger values to "bubble" to the end of the list while smaller values "sink" towards the beginning of the list, thus the name of the algorithm.

The bubble sort is generally considered to be the most inefficient sorting algorithm in common usage. Under best-case conditions (the list is already sorted), the bubble sort can approach a constant O(n) level of complexity. General-case is an O(n2).

The following Visual Basic syle pseudo code clearly shows a bubble sort.

```
Bubble_Sort(myarray[])
   Do While Not Done
      Done = True
      For j = 0 To 39
         If myarray(j).AOPPercent < myarray(j + 1).AOPPercent Then
             tmp = myarray(j).AOPPercent
             tempname = myarray(j).AOPName
             myarray(j).AOPPercent = myarray(j + 1).AOPPercent
```

```
            myarray(j).AOPName = myarray(j + 1).AOPName
```

30

```
                myarray(j + 1).AOPPercent = tmp
                myarray(j + 1).AOPName = tempname
                Done = False
            End If
        Next
    Loop
End Sub
```

Even though this is one of the slower algorithms available, it is seen more often simply because it is so easy to implement. Many programmers who lack a thorough enough understanding of algorithm efficiency and analysis will depend on the bubble sort.

Conclusions

Data structures are an ordered way of storing data that also define how the data will be stored. They also occasionally form the basis for sorting algorithms, as with the heap. The selection of an appropriate data structure is largely contingent upon the type of data being stored.

Algorithms are simply systematic procedures for solving some problem. Sorting algorithms are commonly used in programming, and thus commonly studied in computer science. There are three primary ways to examine an algorithm. They Big O notation, which denotes worst case situations, the Omega notation which denotes best case situations, and Theta notation which shows the average case situation.

End Notes

Corman et. al. , *Introduction to Algorithms 2nd Ed*. (2001), Prentice Hall

31

Drozdeck Adam, *Data Structures and Algorithms in C++, 2nd Ed*, (2001), Thompson Learning

Harvard Basics of Big O notation, http://www.eecs.harvard.edu/~ellard/Q-97/HTML/root/node8.html, (accessed in February 2005).

MathWorld, http://mathworld.wolfram.com/Algorithm.html, (accessed January 2005).

National Institute of Science and Technology (NIST), http://www.nist.gov/dads/HTML/asymptoticUpperBound.html (Accessed February 2005) National Institute of Science and Technology (NISTb), http://www.nist.gov/dads/ (Accessed January 2005)

Sedgewick, Robert; Flajolet, Phillipe, *An Introduction to the Analysis of Algorithms 1st Ed* (1995), Addison Wesley