Chapter 1. What is the course about. Vaguely speaking, a data structure is a conceptual way to store data. Examples of data structures are “lists”, “queues”, “trees” and “graphs”. We use the word “conceptual” because a discussion of data structures can be completely divorced from the implementation of that structure. Each data structure has associated to it a set of operations. Every “list” for example can have items inserted into it and deleted from it. Insertion and deletion are two of the basic operations of every list.

A data structure, together with its full set of operations is often called an “abstract data type” or ADT. In this course we will learn of a variety of ADT’s AND how to implement them in Java. Java is a very good language to study ADT’s because there is a very natural way to implement them, as classes.

Many of the basic operations that we perform on some data structures are of considerably more than academic importance. Sorting a list, for example, is basic to information processing of all kinds. It will turn out that the implementation of a particular ADT can greatly effect what algorithms can be used on that ADT as well as the efficiency of the algorithms. Learning how to choose appropriate data structures and algorithms in order to solve a particular problem is the goal of this course.

Before we begin that process, however, we need to decide what we mean by “efficiency” and find out how to measure it....

Definitions:

A (primitive) type is merely a collection of values: \{ 0, 1 \} or \{ ...-1, 0, 1, 2, 3...\}

A (primitive) data type is a (primitive) type together with a set of operations on that type...e.g. integers together with all the usual integer oprations.

A (complex) type is a composite of primitive types or other complex types, and a (complex) data type is a complex type together with a set of operations on that type. An example of this is the ComplexNumber type that my students completed in Cmpt 184.

A data item is an item of some type, either primitive or complex.

An abstract data type, ADT, defines a data type strictly in terms of its type and operations. It does not specify any implementations....they are completely encapsulated.

A data structure is the physical implementation of an ADT. Each operation is implemented by one or more methods or procedures. Data structures naturally correspond to classes.
A **file structure** is a representation of data on permanent media usually on disk (either magnetic or optical) or tape.

Examples: The mathematical concept of an integer together with the ordinary integer operations is an ADT. The wrapper class Integer (almost) implements that ADT. (In a computer, integers are not unbounded)

**Definition:** An **algorithm** is a process for solving a particular problem. It takes some set of inputs and returns some set of outputs. In Java, every algorithm is a method or set of methods. Every algorithm must:
- be correct...think how you might prove that an algorithm was correct
- be concrete...each step must be completely understandable
- be unambiguous...there should only be one possible next step
- be finite...couldn’t be implemented otherwise
- terminate...

Read chapter 2. All of these ideas should be familiar to you from Calculus and discrete math. We will return to it as necessary.

**Chapter 3. Algorithm Analysis...how to tell which of two algorithms is faster?**

The obvious way is to write programs which implement both algorithms, and then test them both on the same input. There are several problems with this approach:
- One program might be more efficiently written than the other.
- Some particular feature of the input may make one algorithm appear better. The behavior of the algorithms may be depend on the size of the inputs.

The preferred way to test algorithm behavior is to perform what is called **asymptotic algorithm analysis**. This is a mathematical technique which will tell us how the running time of an algorithm will increase as the input increases. Book: “Of primary consideration when estimating an algorithm’s performance is the number of “basic operations” required by the algorithm to process an input of a certain “size”. Those terms are pretty vague, but a simple example will make it clear.

Example 3.1. Find the largest value in an array of positive integers...see p43 for the actual Java code. The size of the input is length of the array (denoted by n) and the basic operations are comparison of two integers and the assignment an integer to a variable. We make the very reasonable assumption that the time required to perform the basic operations is the same for any integer in the array. To be specific, assume that it takes “a” nanoseconds to make a comparison and “b” nanoseconds to make an assignment. We analyze the algorithm as follows:

It “costs” b to assign 0 to currLargest.

The loop executes n times and each time through there is a comparison giving a “cost” of na and during the execution of the loop there are occasionally assignments made. The minimum number of assignments is 1 and the maximum number is n, so the assignments cost us between b and nb.
Summing up: the minimum cost of the algorithm is \( b + na + b = 2b + na \)

- the maximum cost of the algorithm is \( b + na + nb = b + n(a+b) \)
- the (estimated) average cost is \( b + n + \frac{n}{2}b \)

Note that \( b \) is some number of nanoseconds, and that term doesn’t change no matter how large the input is. When we do this formally, we’ll ignore such insignificant costs making the maximum cost, \( T(n) = n(a+b) \). Finally, \( a+b \) is constant, so if we replace it by the letter \( c \), we’ll get the same answer as the book.

Example 3.3. This example shows a nested loop with one basic operation, the increment operator. If we again ignore the initialization statement, we analyze the run time as follows. Let \( c \) represent the time required to perform the basic operation. The time required for the inner loop is exactly \( nc \). The outer loop consists exactly of the inner loop executed \( n \) times, so the time for the outer loop is exactly \( n \) times the time for the inner loop or \( n^2c \).

To see the significance of the formulas let’s assume an input size of 100. The first algorithm costs \( 100c \), while the second costs \( 10000c \). Now increase the size of the input 10 fold, i.e., the input size is now 1000. The first algorithm now costs \( 1000c \) but the second costs a whopping \( 1000000c \). Notice that the timing calculation, \( c \), is constant so it really isn’t necessary to make the comparison between algorithms.

The calculations above illustrate an important concept...that of “growth rate”. An algorithm whose running time is \( cn \) (\( c \) constant) is said grow linearly (since the graph of the equation \( t = cn \)) is a straight line. We can similarly talk about quadratic grow (\( t = cn^2 \)) and even exponential growth (\( t = 2^n \)). No matter what the constant is, an algorithm with linear growth rate will eventually be better than one with quadratic growth.

Example: The time for algorithm 1 is \( 1000n \), the time for algorithm 2 is \( 2n^2 \)...for what size input will algorithm 1 perform more efficiently than algorithm 2?

Best, Worst and Average.

For many algorithms the size of the input is not the only consideration. Consider the famous Bubblesort algorithm. As we’ll see, the average, asymptotic run time for the algorithm is \( n^2 \), but if the input list is already sorted for some reason, the run time is \( n \). This best case scenario, is highly unlikely but possible. When analyzing algorithms, we will normally analyze the average or the worst case. Both are useful and a worst case analysis has the further advantage of guaranteeing that the algorithm will usually run faster than our analysis would indicate.

The asymptotic run time hierarchy: \( \lg n, n, n \lg n, n^2, n^3 \lg n, \ldots 2^n \)

Asymptotic Analysis.

The most common way that computer scientists talk about algorithm efficiency is via “Big-O” notation, as in \( O(n^2) \) which is read “Big-O of \( n \)-squared”. Big-O is a mathematical way of calculating the upper-bound of the running time of an algorithm. This should not be confused with the worst-case scenario analysis...we can and will calculate the upper-bound of an algorithm
given the best, worst and average cases.

Example: Quick sort is \(O(n \log n)\) in the average or worst cases while it is \(O(n)\) in the best case.

Definition: Let \(T(n)\) and \(f(n)\) be positively valued functions (whose domain is the integers). \(T\) is said to be in the set \(O(f(n))\) if there is an integer \(n_0 > 0\) and a constant \(c\) such that \(T(n) \leq f(n)\) for all integers bigger than \(n_0\).

Example 3.4 The average time required to find a specific value in an array of \(n\) values is \(kn/2\) where \(k\) is a constant expressing the time required to compare to values (Why?) (What is the worst case?). \(kn/2\) is \(O(n)\) where \(n_0\) is 1 and \(c\) is \(k/2\).

Example 3.5 Suppose that \(T(n) = kn^2 + k'n\). \(T(n)\) is \(O(n^2)\)...\(T(n) \leq k n^2 + k' n^2 = (k + k') n^2\). For every \(n\), so we can choose \(n_0\) to be 1 and \(c = k+k'\).

Example 3.6. If \(T(n)\) is constant, i.e. \(T(n) = k\), then we can choose \(f(n)\) to be the constant function \(f(n) = 1\) and \(T(n)\) is \(O(1)\).

Note: We will always seek the smallest function for which Big-O is true. We can make a \(O(n)\) function \(O(n^2)\), but not visa-versa.

Big Omega

We can also calculate the lower bound on the running time of an algorithm.

Definition: \(T(n)\) is \(S(g(n))\) if there is an integer \(n_0 > 0\) and a constant \(c\) such that \(T(n) \geq g(n)\) whenever \(n > n_0\)

Definition: \(T(n)\) is \(1\( g(n)\) if it both \(S(g(n))\) and \(O(n)\)

Some simplifying rules...the following rules follow directly from the definition of \(O\).

1. If \(f(n)\) is \(O(g(n))\) and \(g\) is \(O(h(n))\), the \(f(n)\) is \(O(h(n))\)
2. For any constant \(k\), if \(f(n)\) is \(O(kg(n))\), then \(f(n)\) is \(O(g(n))\)
3. If \(f(n)\) is \(O(g(n))\) and \(f'(n)\) is \(O(h(n))\), then \(f(n) + f'(n)\) is \(O(\max (g(n), h(n)))\)
4. If \(f(n)\) is \(O(g(n))\) and \(f'(n)\) is \(O(h(n))\), then \(f(n) \cdot f'(n)\) is \(O(g(n)*h(n))\)

Consequences:

rule 2 : \(1000 \cdot n^2\) is \(O(n^2)\)
rule 3 : \(n^2 + n\) is \(O(n^2)\)
rule 4 : \(n(n-1)\) is \(O(n^2)\)
rule 1. \(n(n-1)/2\) is \(O(n^2)\)
Example Find the upper bound of Bubble Sort.

For \( i = 1, i <= n, i++ \)
    For \( j = i, j <= n, j++ \)
        if \( a[j] > a[j+1] \) interchange \( a[j] \) and \( a[j+1] \);

The maximum cost is incurred when an exchange has to be made for every single pair of elements. If we assume that comparison cost 1 and interchange costs \( k \), we can calculate an upper bound for Bubble sort by multiplying \( 1+k \) times the maximum number of times the operation happens. That turns out to be: \((n-1) + (n-2) + ... + 2 + 1 = n(n-1)/2\) which is of course \( O(n^2) \). It is fact \( 1^2 \).

Calculating the running time of an algorithm

1. Assignment costs 1.
2. Any algebraic operation costs 1.

// so any arithmetic expression has constant time and is therefore \( O(1) \)

3. Given a statement of the form
   if (condition)
       complex statement 1
   else
       complex statement 2
   the worst case cost is max ( complex statement 1, complex statement 2 ) which is therefore an upper bound.

4. Given a loop of the form:
   loop control{
       complex statement
   }
   then \( O( \text{the loop} ) = O(\text{number of times the loop executes}) \times (\text{cost of the complex statement}) \)

5. A statement which is a call to a method has the same cost as that of the method.

6. The cost of a recursive method may be calculated using a recurrence relation.

// Factorial (n) = n * Factorial(n-1), so \( T(n) = T(n-1) + \text{constant...where } T(0) \text{ is 1, so you recall} // // \text{from discrete math} T(n) \text{ is } O(n) \)

Example 3.10

assignment costs 1
Each execution of the for loop costs 2,
the for loop costs \( 2n \).
\( T(n) = 2n + 1 \) is \( O(n) \)
Example 3.12

part 1.
assignment costs 1
inside loop cost n, so
outside loop costs n*T(inside loop ) = n^2
which is O(n^2 )

part 2.
assignment costs 1
inside loop costs 1 the first time, 2 the second time etc. so the cost of the
outside loop is 1+2+3+4...+n = n(n-1)/2
which is O(n^2 )

Example 3.13

part 1
inner loop executes n times, while the outer loop execute lgn times so T(n) is O(nlgn)

part 2
the inner loop executes 1 time, then 2 times, the 4 times...up to n...if we assume that n is 2^m for
some m then the outer loop costs 2^m+1 ...however m is in fact lgn so T(n) = 2^lg^{2}^{n} = n.

Analyzing the cost of solving a problem is slightly different from that of finding the run time of
an algorithm...the upper bound of the cost of a problem is the upper bound of the running time of
the most efficient algorithm which can solve it, similarly the lower bound is the largest lower
bound of any algorithm which solves the problem.

consider the sorting problem....the lower bound, S(n) is clearly n since we have to read each
element in a list at least once. The upper bound turns out to be O(nlgn) for a number of
algorithms that we’ll study later. ( Theoreticians have since proven that S is also nlgn ). For
many problems the difference between lower and upper bound is quite large and trying to bridge
the gap is a central problem in Comp Sci. There are lots of problems in which we can figure out
a lower bound for a problem, but for which every known algorithm that solves the problem has a
higher run time...Will someone figure out a higher lower bound or will some one fine a more
efficient algorithm....note, if the first, then we can give up trying to find the second.

Multiple parameters...it will happen that the cost of an algorithm will depend on more than
input...read sec. 3.7. In this example, the cost of the algorithm depends on two parameters: the
number of colors, c and the number of pixels p. Either one or the other of these might be large
compared to the other, so the cost must include both, to wit: O( c+plgp)

The great space-time trade off. Every algorithm requires some amount of main memory or disk
storage. Often, we will have to choose which of these we want to optimize.
Example: Most data can be compressed so that it consumes fewer bytes. The percentage savings in size usually depends on the kind of data and the type of compression used. If data is compressed the time required to perform any operation will increase since the algorithm must first decompress the data before operating on it.

Case 1. An operating database. Since transactions must take place as rapidly as possible, it is usually not a good idea to compress it.

Case 2. Pictures taken by space craft must transmitted long distances, and since the images contain a lot of absolutely black pixels, they can be highly compressed. It makes a good deal of sense to transmit compressed data.

Exercises p 71. # 3.6 and 3.9

First Programming Exercise. Make a class called Student with the following instance variables:

   int id;
   String firstName;
   String lastName;
   String major;
   float gpa;

Provide at least a constructor for the class, methods to retrieve all the instance variables. i.e,

   String getFirstName();

and a method which copies the values of one student into another, as in:

   void copyStudent( Student copyee);

Create a driver program which makes a ArrayList of Students; allows the user to enter new Students into the ArrayList and further prints out the ArrayList, one Student at a time, one instance variable per line. Finally, test the copy procedure by interchanging the first and second elements of the ArrayList and print out the ArrayList again.

Programming standards....my web sight.

Lists.

Definition. A list is an ordered sequence of data items, called elements. Ordered means that each element has a position in the list, first, second etc. Common terminology: The number of elements in the list is called its length, The first element in the list is called it’s head and the last it’s tail. If the length of he list is zero, it is said to be empty.

The list ADT.

   Assuming that we have some sort of list, the list ADT has the following methods (
written in pigeon Java and slightly different from the book):

```java
void clear() // remove all elements from the list
void insert( element, position) // insert a new element at the specified position
void append(element) // same as insert(element, tail+1)
void remove( position) // delete the element at the specified position
int length() // return the length of the list
Boolean isEmpty() // returns true if the list is empty
Boolean search( element) // returns true is element is somewhere in the list
int search( element ) // returns the position in the list of the first occurrence
// of element if it’s in the list, -1 if not
element retrieve( position) // returns the element in the specified position
void traverse() // returns all of the elements in the list
```

Look at the book’s version starting on p 78. It is the signature of the Java list interface, using a member variable “current position”. Print is similar to traverse.

Important note. All of the books examples assume that list elements support the interface Elem (p. 80). Elem requires only that each list element support the method:

```java
abstract int key();
```

A key value is one that uniquely identifies a element. The ADT that I’ve shown you would have some small changes, notably

```java
int search( keyvalue )
```

read method “find” also on p 80.

next 4.1 the array implementation of a list.

The simplest way to implement a list is as an array. This implementation makes several of the list methods are very simple and efficient, others are quite nasty and slow.

Assume the following declarations:

```java
private int maximumListSize; // Books msize
private int listSize // Books numInList
private Object[] listArray;
```

Easy methods:

```java
void append(element) // same as insert(element, tail+1)
```

is essentially:

```java
if listSize < maximumListSize
    listArray[listSize++] = element; // note increment after using listSize
```
else System.out.println("Error: List overflow")

void clear() // remove all elements from the list is:
    listSize = 0;

int length() // return the length of the list is:
    return( listSize)

element retrieve( position) // returns the element in the specified position is
    if (position < listSize && position >= 0 )
        return ( listArray[position])
    else ERROR

Boolean isEmpty() // returns true if the list is empty is
    if (listSize == 0)
        return ( True )
    else
        return ( False )

All of these are in \( O(1) \)

void traverse() // returns all of the elements in the list is
    // could be some other operation besides print
    for ( int i = 0; i<listSize; i++ )
        System.out.println( listArray[i]);

Boolean search( element) // returns true is element is somewhere in the list is
    Boolean found = False;
    for ( int i=0; i < listSize; i++)
        if ( listArray[i].compareTo( element) == 0 )
            found = True;
    return ( found );

all of these are \( O(n) \) // Expected run time is \( n/2 \)

Nasty

void insert( element, position) // insert a new element at the specified position is:
    if ( listSize < maximumListSize ){

for (int i = listSize; i > position; i--) // You have to make room
    listArray[i] = listArray[i-1]; // for the new element
listArray[position] = element;

} else ERROR

void remove(position) // delete the element at the specified position

// is similar. Once the element is removed, // everybody in the list has to move up.

all of these are \( 1(n) \) as well. The Books version is on p 83.

Important note...look at the book’s version on p 83...this is not the classical list implementation, but it DOES implement the List interface.

Homework: Write int search(element) in either version. Due next class.

It is clear that the array implementation of a list suffers from two serious problems:
1. It is of fixed size
   i. It therefore has an absolute maximum size, and
   ii. It may take up a lot of space even when it is empty
2. A couple of its basic methods are nasty

A new implementation of a list, the linked list.

The basic idea of a linked list is as follows:
The user knows the address of only the first element in the list (called the head of the list). Each element in the list knows the location of at most one other member of the list, (called its successor.) One element has no successor, it’s called the tail of the list. In order to move through the list (Traverse) you must visit the first element, find its successor, visit the successor etc. The last node in the list has NULL for its successor.

In order to implement the linked list, the elements of the list must be combined with these references to successors...we generally call these constructs “nodes” or “links”. We will make use of the fact that Java uses object references...in C++ and other languages you must explicitly create these references, called pointers.

The trick to working with linked lists is to maintain the links...

It is often useful to picture a linked list, book p 88. Look at how insertion and removal have to work.

Book’s Link class

class Link { // A singly linked list node
private Object element; // Object for this node
private Link next; // Pointer to next node in list

Link(Object it, Link nextval){ // Constructor 1, Given Object
    element = it;
    next = nextval;
}

Link(Link nextval){ // Constructor 2
    next = nextval;
}

Link next(){ // represents the links successor
    return next;
}

Link setNext(Link nextval){ // set the link’s successor
    return next = nextval;
}

Object element(){ // returns the link’s element
    return element;
}

Object setElement(Object it){ // sets the link’s element
    return element = it;
}

} // class Link

The book’s class LList also implements the List interface...note the name change

class LinkedList implements List { // A Linked list class
    private Link head; // Pointer to list header
    private Link tail; // Pointer to last Object in list
    protected Link curr; // Pointer to current Object’s predecessor

    // We really NEED a current value unlike the array implementation

    LinkedList(int sz) { // Constructor -- Ignore sz
        setup();
    }

    LinkedList() { // Constructor
        setup();
    }

    // ...
```java
private void setup() { // Initialize the list
    curr = new Link(null); // At the beginning all objects
    head = curr; // references are the same, null
    tail = curr; // because the list has no elts yet
}

public void clear() { // Remove all Objects from list
    head.setNext(null); // Drop access to rest of links
    curr = tail = head; // Reinitialize
}

public void insert(Object it) { // Insert Object at current position
    Assert.notNull(curr, "No current element"); // This actually works
    curr.setNext(new Link(it, curr.next())); // Check the Link constructor
    if (tail == curr) // Appended new Object
        tail = curr.next(); // Draw a picture.
}

public void append(Object it) { // Insert Object at end of list
    tail.setNext(new Link(it, null));
    tail = tail.next();
}

public boolean isInList() { // True if curr is within list
    return (curr != null) && (curr.next() != null);
}

public Object remove() { // Remove and return current Object
    Object it;
    if (!isInList()) return null;
    it = curr.next().element(); // Remember the element in current
    if (tail == curr.next()) // if you removed the tail, reset the value of tail
        tail = curr;
    curr.setNext(curr.next().next()); // Remove the node from the list
    return it; // Return value removed
}

public void setFirst() { // Set curr to first position
    curr = head;
}
```
public void next() { // Move curr to next position
    if (curr != null)
        curr = curr.next();
}

public void prev() { // Move curr to previous position
    Link temp = head; // Start at front of list
    if ((curr == null) || (curr == head)) { // No previous Object
        curr = null;
        return; // so just return
    }
    while ((temp != null) && (temp.next() != curr))
    
    temp = temp.next();
    curr = temp; // Found previous link
}

public int length() { // Return current length of list
    int cnt = 0;
    for (Link temp = head.next(); temp != null; temp = temp.next())
    
        cnt++; // Count the number of Objects
    return cnt;
}

public void setPos(int pos) { // Set curr to specified position
    curr = head;
    for(int i=0; (curr!=null) && (i<pos); i++)
        curr = curr.next();
}

public void setValue(Object it) { // Set current Object's value
    Assert.notFalse(isInList());
    curr.next().setElement(it);
}

public Object currValue() { // Return value of current Object
if (!isInList()) return null;
return curr.next().element();
}

public boolean isEmpty() { // Return true if list is empty
    return head.next() == null;
}

public void print() { // Print out the list's elements - Traverse
    if (isEmpty())
        System.out.println("()");
    else {
        System.out.print("( ");
        for (setFirst(); isInList(); next())
            System.out.print(currValue() + " ");
        System.out.println(")");
    }
}

} // class LinkedList

Programming exercise. Modify the code for a linked list so that the contents of each node are in fact integers ( small i ). Create the list and test it by writing a driver program which asks the user to enter a sequence of integers ( one at a time ), and then when the user is finished finds the sum of all the integers.

Discussion?

Due dates. Documentation:
    Working program:

Programming standards....my web sight.

    Note that a linked list has no theoretical limit to its size and that the list is exactly as big as it has to be. The asymptotic run time of all the methods is $O(1)$ except for Traverse and Search ( not shown ) which have to be $O(n)$. One further note, the linked list node is slightly larger that its array implementation counterpart...it has to have a few extra bits for the link.

    The cost of creating new nodes and garbage collecting of discarded nodes can in fact be very expensive under some circumstances. If a linked list is constantly under going expansion and then shrinking a programmer can achieve a significant improvement in performance by managing the nodes from the program. The idea is to maintain what’s called a “freelist” of available node. As a node is deleted, it is placed on this freelist, and if another node is needed,
the first place the program looks is the freelist...see pp92,93.

4.1.4 A note on Element Implementations. In all of our examples, the elements of the list have been of class Object. That is, elements are really Object references, with the actual Objects stored somewhere else. It is also possible to construct lists whose elements are primitive types like integers. There are pros and cons to both approaches:

Object references make lists slightly larger since the object references themselves take up a few bytes, they can however point to anything making their use more flexible. The elements of most lists are all of the same type (homogeneous structures). You actually have to be careful in Java to implement homogeneous lists. Probably the best way to do this is to write the Link constructor so that as new nodes are created, they are checked to see that they are of the correct type.

Doubly Linked Lists

As the name implies, each node in such a list has two links, one to its successor node and one to its predecessor. There are two clear consequences to this fact:
1. You can navigate the list in two directions
2. Every operation requires us to maintain both links

Also as before, we have a choice of how to handle the “current” node...the book again chooses to have it point to the predecessor of the actual current.

// Source code example for "A Practical Introduction
// to Data Structures and Algorithm Analysis"
// Copyright 1998 by Clifford A. Shaffer

class DoubleLink {                // A doubly-linked list node
    private Object element;    // Object for this node
    private DoubleLink next;        // Pointer to next node in list
    private DoubleLink prev;        // Pointer to previous node in list

    DoubleLink(Object it, DoubleLink n, DoubleLink p){              // Constructor 1
        element = it;  //  Given Object
        next = n;
        prev = p;
    }

    DoubleLink(DoubleLink n, DoubleLink p) {  // Constructor 2
        next = n;
        prev = p;
    }

    DoubleLink next() {
        return next;
    }
Now investigate the concept of insertion, deletion and append in figures 4.15 and 4.16...Note that insert puts the new node BEFORE the existing current element...code on p 98 does make sense.

A doubly linked list does cost slightly more than a singly linked one both in space and time. Each node is slightly bigger and slightly more processing is required for insertion and deletion. There is one decided advantage to using a doubly linked list that is sorted to search for a particular node. A singly linked list must always be searched from the front, so the expected number of nodes that have to be searched is $\frac{1}{2} n$. In a doubly list you can start from the previous position of “current”...you can prove that the expected number of nodes that have to searched is in fact $\frac{1}{3} n$.

The last variation of the linked list is a natural extension of the concept, a circular linked list. The idea of course is that the last node’s next field points to the head of the list..a circular linked list can be either sinly or doubly linked.

Stacks.

Definition. A stack is a special kind of list in which insertion and deletion can only take place at the head of the list.

They are called stacks because they’re supposed to remind you of the stacks of plates in a diner!
The terminology of stacks is slightly different from other lists. The first element of the list is called the “top” of the stack, insertion is called “push” and deletion is called “pop” (almost). An attempt to push an element on a full stack is called “overflow” and an attempt to pop an empty list is called “underflow”. No other list operations are appropriate for stacks...all we ever care about is the top.

Pop is actually more than simply deletion, in any stack implementation pop returns the value of the element at the top of the stack and then deletes it.

StackElement pop( Stack aStack) {....

Example: The run time system keeps track of which methods are currently operating by means of a stack. When a method is called, an “activation frame” is pushed on the stack...when a method is finished its activation frame is popped and the method underneath it rises to the top of the stack.

It may a little vague to you now but stacks provide a sort of primitive memory for the program.

Implementations: Any list can implement a stack...

The array implementation. There are two candidates for the “top” of the stack....the first element and the last. If you use the first element, then pushing (insertion) or popping requires us to move all the previous elements of the stack and are therefore \( \mathcal{O}(n) \). If we choose the last element the pushing and popping are \( \mathcal{O}(1) \). The Java code for the array based stack implementation is on p 104. Note that isEmpty() is easy and that the Stack interface does not have an isFull() method...if you do implement a stack as an array you should probably provide one. In fact micro exercise...write the isFull() method for the array implementation of a stack.

The linked implementation is similarly very easy. The only data element is “top” (we don’t have to maintain a pointer to the current stack element...that’s always top). Push merely sets the next pointer of any new element to be the previous top and sets the new top to be the new element. Pop (after returning the value stored in top) merely sets top to be top.next. The Java code is on p 106.

There is little to choose between the two implementations. The array implementation does use more space and can become full. Each element of a linked stack requires a few more bytes than its array based counterpart.

Third programming exercise. A palindrome is a string that reads the same forwards and backwards.

Examples: axbbxa A man a plan a canal Panama

A special kind of palindrome has a special kind of character to mark the inside of the palindrome, as in axb$bx$a. It is fairly easy to check and see if a string is a palindrome of this...
special type using a stack. Your project is to

1. Implement a stack as a linked structure, and
2. Write a program which accepts input from the user, one character at a time and when the user finishes inputting characters, decides if a palindrome has been entered.

Discussion??

Due dates:
The algorithm: next class.
Documentation:
Working program.

There is one canonical variation of the array based stack, that is using an array to store multiple stacks. Two stacks can be stored by allowing one to grow from the front and one from the back. What advantage could there be to using a 200 element array to hold 2 stacks versus putting each stack in its own 100 element array?

Queues.

Definition. A Queue is a special kind of list in which insertion takes place on only one end of the list and deletion takes place only at the other end. The insertion point is called the “tail” of the queue and the deletion point is called the “head”. You will occasionally see the term “enqueue” for insertion and “dequeue” for deletion.

The array based implementation of the queue is simple and easily figured out. You have two operations and two logical places to do them...them beginning and end of the array. It makes very good sense to enter new objects at the end of the array (we’ve been calling this appending objects) and to remove them from the front. The enqueue operation is clearly then $O(1)$ but the dequeue is $O(n)$ because after an object has been removed, you have to move all the remaining objects one space up.
Again there is a very simple way to make dequeue $O(1)$...

There are a couple of questions still to be addressed...

First, under our current plan, we are going to use up the space in the queue very rapidly...under our original scheme, dequeueing releases space that can then be reused, now however, once the tail of the queue reaches the end of the queue we’re stuck, so...

This unfortunately leads to a whole other set of problems...how to tell if the queue is empty or full?

suppose that we have instance variables called head and tail (the book calls these front and rear)...cases we need to worry about

1. the queue is empty
2. the queue has exactly one object in it.
3. the queue is full

1. set tail equal to head -1 mod n.
2. head and tail to be equal
3. which is it?...either tail is head-1 mod n or tail is head.

second try.. set tail to be equal to the current last index +1.

1. head and tail are equal
2. tail is head + 1 mod n
3. which is it? head and tail are equal or tail is head + 1 mod n

Solution...well pick one, but use another instance variable which contains not the size of the array, but the actual size of the queue!

Solution..pick one, but put an extra space in the array...i.e. restrict the number of elements in the queue to n but allow the array to have n+1 spaces...This is fairly subtle so to see how it works draw it out

Suppose we’re in case 2, i.e. tail is the current last index + 1. Then
1. head and tail are equal
2. tail is head + 1 mod n+1, and
3. tail is head - 1 mod n + 1

Queue size is 3, the array size is 4

<table>
<thead>
<tr>
<th>head and tail</th>
<th></th>
<th></th>
</tr>
</thead>
</table>

This queue is empty

<table>
<thead>
<tr>
<th>head ( the only elt)</th>
<th>tail</th>
</tr>
</thead>
</table>

This queue has exactly one element

|( The second elt ) |( The last elt ) |tail |head ( The first elt ) |

This queue is full.

The actual code for the array based queue is on p 114. I don’t think that there is any thing new and unusual ...

The linked queue implementation presents no new challenges... the head of the queue is the end of the linked list and tail of the queue is the beginning of the list.
With our current implementation all queue operations are $O(1)$.

Non-exercise: For fun, create a Queue class which is a subset of our previous LinkedList class...

Why doesn’t Java do this?

ANACHRONISM...jumping ahead little...one standard variation on the plain vanilla queue is something called a “priority” queue The idea is simple, instead of items being removed from the queue in the exact order in which they are placed on the queue, elements are given a score or weight and then are removed from the queue according to their weight.

Example: In a hospital emergency room, patients are seen in the order of the severity of their medical problem.

Example: In a multi-user computer system processes are assigned a priority. System processes get highest priority, student programs the lowest...No low level process will be executed before all the high level processes are complete.

One of the ways to implement a priority queue is as a list in which new elements are inserted in the list according to their weight. As with a normal queue, removal always takes place at the head. This changes the time requirement of insertion to $O(n)$.

And now for something completely different...

Binary Trees.

Definition. Given a set ( of nodes ) an edge is a ( ordered ) pair of nodes. If the edge is \{a,b\} then we say that a is the parent of the edge and b is the child.

Definition: A Binary Tree consists of a finite set of nodes and a set of edges subject to the following conditions:

- One node does not appear as a child in any edge. This node is called the ”root”.
- Every other node appears as the child in exactly one edge.
- A node can appear as the parent in at most two edges. ( This is the binary part ) Nodes that do not appear as the parent in any edge ( have no children ) are called leaves. Non-leaf nodes are called interior nodes.

Examples.

Terminology. A path is a set of edges $e_1,e_2...e_k$ such that the child of $e_1$ is parent of $e_2$ etc.

One of the consequences of the definition of a binary tree is that there is a path from the root to every other node in the tree.

Terminology. The length of a path is the number of edges in the path.
Terminology. The depth of any node is the length of the path from the root to that node.

Terminology. The height of the binary tree is the largest depth of any of the nodes.

Terminology. If there is a path from node a to node b, then we say that b is a descendant of a and a is an ancestor of b.

Terminology: The set of all descendants of node a is called the subtree whose root is a.

ASIDE: It’s easy to figure out what the maximum number of nodes in a binary tree of any depth...

Terminology. A binary tree is said to be full if every leaf occurs at the same level and every interior node has two children.

Terminology. A binary tree of depth d is said to be complete if
1. Every leaf is either at depth d or d-1
2. The bottom level leaves fill in from left to right.

Binary Trees in computers are actually very similar to double linked lists. Each node contains some element of class Object and two links one of which is called left-child and right-child or just “left” and “right”. The Java interface BinNode is shown on p 127.

Binary trees can be used in a variety of ways to implement lists. One of the things that varies the most from our previous lists structures, is he fact that there is no definitive, obvious technique for traversing the Binary Tree. There are in fact several ways to visit every node: InOrder traversal, PreOrder traversal and PostOrder traversal (and others). All three of these are recursive and kind of like marching:

- PreOrder: Root, Left, Right
- InOrder: Left, Root, Right
- PostOrder: Left, Right, Root

That is, in pidgen Java

```java
void inOrderTraversal( BinNode route) {
    if ( route == null ) return; // recursion ends at the children
    else {
        inOrderTraversal( route.left()); // traverse the left subtree
        doSomethingTo( route); // process the element
        inOrderTraversal( route.right() ); // traverse the right subtree
    }
}
```

practice with the tree of section 5.1...p 122.
Binary Tree implementations

1. Using pointers, the code for an implementation of a BinNode, the poorly named BinNodePtr is on page 129. It is common in most usages to declare a BinaryTree just to be an object of type BinNodePtr which is just the root...in figure 5.7 if we had the following declaration

    BinNodePtr BinaryTree();

    BinaryTree would be the node labeled A.

   Note that naming the tree for the root of the tree makes recursive traversals easy since the a traversal called on the entire tree is no different than one called on any other node.

   In some applications, every node of a tree will carry some useful piece of information, in others only the leaves have actual data in them and in others the root and/or the interior nodes and/or the leaves may carry different types of information....

Example 1. A parsing tree... (actually only a parser for expressions). A you recall, a parsing tree has labels that correspond to binary arithmetic operators on its interior nodes and roots and constants or identifiers on the leaves. See fig 5.8 on p 130. Parsing trees, if read in order produce a usual arithmetic expression...

     (((4*x)*((2*x)+a))-c)

What would the tree look like for (((4*x)*((2*x)+a))-c)

An interesting thing happens if you read the same tree post order..

Example 2. Assume that we are given an input stream consisting of some number of letters. H, K, C, E, J, T and A

    Place H at the root of a binary tree H. As each letter is read, place it on the tree by following the existing nodes in the tree according to the alphabetical ordering of the letters. Now read the tree in order.

In the cases where we must make a distinction between nodes of a binary tree, Java makes it easy: we just need to derive either interior nodes or leaf nodes from our existing BinNodePtr class...

Look at LeafNode and InternalNode carefully...note that the parameter given to the traverse procedure is of class BinNode so that it might be either an interior node or a leaf...

Why didn’t the author make LeafNode and InternalNode a subclass of BinNodePtr ??

Space considerations...The way we’ve set up the tree, each node has 2 pointers and a data element. so the size of the tree is n*( 2*p + d ) where p stands for the size of the pointer and d stands for the size of the data...this of course 1(n) but could really be quite large in a very large
tree...There are couple of things that we can try to shrink the size of the tree, especially since we’ve gone to the trouble of creating two node types...
1. Remove the pointers from the leaves...since about half the nodes are leaves, the size calculation becomes: \((n/2)*(2*p+d) + (n/2)*d = n*(p+d)\)
2. Remove data from the interior nodes...this sounds odd, but in some circumstances this means that the tree can searched more rapidly...at any rate, this would reduce the size further to:
\[(n/2)*(2*p) + (n/2)*d = n*p + (n*d)/2\]

Contra this the overhead required to use two separate types adds at lot more bytes...but there is yet another trick....one node type with a single byte to separate the “type” of the node...gets you back of the advantages.

The array implementation. Doesn’t seem like an array would be suitable data structure for a tree....but at least for binary trees it really quite easy....the root is index 0 and for every other node residing in index i has children locate in indices \(2*i+1\) and \(2*i+2\).

Example: Consider the following almost complete Binary Tree

```
       B
     /   \
    K     C
   / \   /  \
  M   P  D   J
 /  \  \  
R    O
```

Place these into an array.....

Read chapter 5.4 on Huffman Codes...this is the basis for a common compression technique.

INTERRUPT...up until now, we haven’t much cared what elements are in the nodes of our structures, but in fact real applications involve elements that hold some real data. This data, whatever it is will usually have a key value...

Definition: A key value is a field or set of fields that uniquely identify an element.

I suppose that its possible to use a non-comparable data type as a key, but we won’t...we assume that every key is of a comparable class.

Binary Search Trees.

Definition. A Binary Search Tree ( BST ) is a binary tree such that, given any interior node, its left hand subtree contains only key values that are less than the key value of the node and its right hand subtree contains only key values that are greater...p147 has 2 examples...

recall that in all of list so far, the time required to search for a particular element has been \(1(n)\),
but BST’s allow a much faster search technique...look at the BST of fig 5.21, and imagine that you are searching for the element whose key value is 35.

The root of the tree has key value 37 so because of the ordering properties of a BST the subtree containing 35 must be the left hand sub-tree, so we continue by checking the node labeled 24...etc. Note that the maximum number of comparisons that we have to make ( whether or not the element is eventually found ) is exactly the depth of the tree. Consider a BST tree with n nodes. What is its depth?

The worst that can happen is that the smallest key element arrives first, then the next smallest etc. In this case the BST will look exactly like a linked list and its depth is n. The best that can happen is the somehow the key values are distributed exactly evenly so the depth of the tree is just \( \lg(n) \). It turns out that the best and average search times are \( \frac{1}{2}(\lg(n)) \) and the worst case is \( \frac{1}{2}(n) \)

Note also that an InOrder traversal of the BST will visit the nodes from smallest key value to largest. Further note that traversing the BST always costs \( \frac{1}{2}(n) \).

Some notes on the code...books p 148.

The interface Elem does nothing but guarantee that the element has a key field.

The authors call their search method “find” which is non-standard. The actual work of the search procedure is handled by a private member method called “findhelp”. You don’t have to do this but organizing it this way contributes to the encapsulation of the search method.

The search method itself is recursive...my version. I assume that each element implements Elem and therefore has a member variable called key. The key can be of any comparable type...to make the code cleaner, I invented a class called KeyClass.

```java
class KeyClass {
    public KeyClass(int k) {
        key = k;
    }
    int key;
}
```

```java
public Element search ( BSTnode route, KeyClass K) {
    if ( route == null ) return null
    else if K < route.key return (search ( route.left , K ))
        else if K > route.key return (search ( route.right, K))
    else return route.element;
}
```

Some other BST methods:

```java
public void insert ( BSTree tree, Element newNode){...}
```

insertion. The traditional technique for inserting a new element in a BST is as follows:

1. Use a version of the search method to find the correct position in which to append the new node.
2. Place the new node in the BST by setting the appropriate pointer.
The books technique is different and clever enough to be worth studying. At every step of the recursive method “inserthelp” the procedure returns a pointer to a subtree containing the new node. This allows us to use the recursion to keep track of where we are in the tree...and what path we have followed from the root to the eventual insertion position.

The books example: In order to insert 35 you must make calls to insert help on:
- the root elem 37
- it’s left child elem 24
- it’s right child elem 32
- it’s right child which is null

return the new node to the line
  rt.setRight(...) on the node with elem 32
return node 32 with its new child to the line
  rt.setRight(...) on the node with elem 24..etc.

Deletion of a node in a BST requires a little effort but it is all very logical.

- If the deleted node is a leaf, all we have to do is to set the appropriate pointer in its parent to null.
- If the deleted node has only one child, you can safely delete it by reassigning the appropriate node in its parent to its only descendant.
- If the node has two children, we have to do a little more work. The simplest solution is arbitrarily choose one of the children, say the left, reassign the parent pointer as in step 2, and the re-insert all the nodes in the right subtree. A more practical solution is to replace the deleted node with some other node in such a way so as to preserve the BST property. There are in fact two candidates for the job, the node containing the SMALLEST KEY value in the RIGHT SUBTREE or the node containing the LARGEST KEY value in the LEFT-SUBTREE...to see that these choices are correct look at figure 5.19a. and look at what the tree would look like under either strategy if node 42 were deleted...

  Find the smallest key in any subtree is easy..it is housed in the node that is reached by going left at every node.

  In the case of a node with two children, the removal process consists of finding the node with the smallest key in the right subtree of that node, deleting it ( this is easy it has to either be a leaf or only have one child ) and then plugging this node into the spot previously occupied by the deleted node.

One final note. BST’s are a simple efficient data structure that allows efficient retrieval of data and traversal as long as the tree doesn’t accidently grow in an unbalanced way...people have invented all sorts variations of BSTs to prevent that from happening...maybe later.

Heaps and priority queues again.

We briefly discussed priority queues before and decided that insertion in a linear priority queue was \( O(n) \). A special kind of binary tree, called a heap, can reduce that time to \( O(\log(n)) \).
Definition A max-heap is an almost complete binary tree that has the further property that the key value of the element in any interior node is larger than the key value in its children’s elements. A min-heap is defined similarly.

Example:

```
       6
      /   \
     4    5
    / \   /  \
   1  2  3
```

Insertion of elements in a max-heap is neat......the element is attached in the correct position so as to maintain the tree structure, and the bubbled it up the tree by comparing its value to that of its parent and exchanging if necessary, until it reaches its correct position so as to maintain the ordering properties....insert 7 and you get the max-heap of fig. 5.25 a.2

Removal of elements in a max -heap is also interesting

1. Remove the root. Note that the root always has the largest weight.
2. Place the element in the last position of the heap and place it in the root.
3. Bubble the root element down the tree by exchanging it with the larger of its children’s weights, until it reaches the correct position in the tree in order to maintain the ordering properties.

Example:

Note that in the array implementation of a binary tree it is very easy to find both the children and the parents of any node.

What is the time complexity of the insertion and deletion? The operations involved are simply comparisons between the weights of a node and either its children or its parents...the maximum number of comparisons that have to be made in order to place a node in the proper location is...the depth of the tree....so the time complexity is $1(\lg(n))$ for both insertion and deletion.

NOTE: A heap is the best implementation of a priority queue. It costs $1(\lg(n))$ to put somebody on the queue and $1(\lg(n))$ to remove them.

General Trees. Trees don’t always have to be restricted to two children per node. This may appear to present programming difficulties since a node in that tree has an unpredictable number of children, however it turns out that any tree can be represented by a binary node. The trick is rather than thinking of the two pointers in the node as both pointing to children, think of the right hand pointer as pointing to a node’s sibling.

```
        Alexander
       /     \
      Edward  Sophie
     /        \
```

The book devotes an entire chapter to general trees, we’ll come back later if we have time

Now, for something completely different Graphs.

Definition: A graph consists of two sets, \( V \) and \( E \), where \( V \) is called the set of vertices and every element of \( E \) is a pair of elements of \( V \) and is called an edge.

Example: Figure 7.2 would look like:

\[ V = \{ 0,1,2,3,4,5,6,7 \} \text{ and } E = \{ \{0,1\}, \{0,4\}, \{1,4\}, \{2,4\}, \{1,3\}, \{5,6\} \} \]

Definition: A graph is said to be directed if in fact \( E \) consists of ordered pairs of elements of \( V \). People often refer to directed graphs as “digraphs”. A directed edge is usually called an arc.

In either case if there is an edge between any two vertices, we say that they are adjacent.

Example: Figure 7.1.b would like

\[ V = \{ 0,1,2,3,4 \} \text{ and } E = \{ (0,1), (0,4), (4,1),(2,4),(1,3),(3,2) \} \]

Definition: A graph or digraph is said to weighted or labeled if a weight (resp label) is assigned to every edge.

Graphs are very useful to model problems in many problem domains.
1. Anytime you are attempting to model the flow of something, you can use a directed digraph. That flow could be oil, cars, information or the flow of control among the components of an application. Putting weights on the edges adds information to the model.
2. Anytime you wish to keep track of the relationship between things, the natural data structure is a graph. An interesting and useful example of this in computer science is used by transaction management systems and is called the “wait-for” graph.
3. In many applications, including artificial intelligence problems, the technique of the program is to methodically search through a collection of “states” trying to find the “optimal” state...the natural way to represent these states (and the transitions between them) is as a graph.
A path in a graph is a set of vertices $v_1, v_2, \ldots, v_n$ such that each pair $v_i, v_{i+1}$ is an edge (or an arc). The path is simple if it doesn’t contain repeated vertices. The length of a path is the number of vertices in it.

A path is called a cycle if there is an edge from $v_n$ to $v_1$. The cycle is simple if it doesn’t contain repeated vertices other than the first and last vertices.

Definition: an undirected graph is called connected if there is a path from every edge to every other edge. (A digraph that has this property is called strongly connected).

Definition: A disconnected graph in which each connected component is a tree is called a forest.

Definition: A free tree is a connected, undirected graph with no cycles.

Definition: A directed graph with no directed cycles is called acyclic or often a Directed Acyclic Graph or a DAG.

Definition: A connected, directed graph that has no cycles, directed or otherwise is a plain ol’ tree.

Implementation: There are two basic ways to implement graphs, each of which has its own pros and cons.

Definition: An adjacency matrix and has one row and one column for each vertex in a graph. Place a 1 in position $(i, j)$ if the graph has an edge from $v_i$ to $v_j$. This implementation can be made to work for either edges or arcs.

Example: The adjacency matrix for the connected part of figure 7.2 is

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The adjacency matrix can also easily store weights...do 7.1.c.

The second implementation is even simpler. For every vertex, just keep a list of the vertices that are adjacent to it:
Example: again the connected part of figure 7.2 is

0: 1,4
1: 0,4
2: 4
3: 1
4: 0,1,2

You can implement the lists any way you want, but it makes more sense to use a linked list.

The space requirements are easy to calculate. An adjacency matrix requires $|V|^2$ entries while each of the edge lists has at most $|E|$ entries so the space required is at most $|V| + |E|$. This difference can be substantial if the graph has relatively few edges, a so-called sparse graph.

The Edge interface and the Graph interface are very instructive...they can be found on pp 195 and 195....and they can implemented as either an adjacency matrix or an edge list

interface Edge { // Interface for graph edges
    public int v1(); // Return the vertex it comes from ( more properly its source )
    public int v2(); // Return the vertex it goes to ( more properly its destination )
} // interface Edge

How would you do this for a adjacency matrix? What is an edge anyhow?
Note that an edge has no member variables.Why?

interface Graph { // Graph class ADT

    // Assumes that vertices are simply denoted by integers.

    public int n(); // Number of vertices
    public int e(); // Number of edges
    public Edge first(int v); // Get first edge for vertex - doesn’t really make sense
    public Edge next(Edge w); // Get next edge for a vertex
    public boolean isEdge(Edge w); // True if this is an edge
    public boolean isEdge(int i, int j); // True if this is an edge
    public int v1(Edge w); // Where edge came from - source or origin
    public int v2(Edge w); // Where edge goes to - destination
    public void setEdge(int i, int j, int weight); // Set edge weight
    public void setEdge(Edge w, int weight); // Set edge weight
    public void delEdge(Edge w); // Delete edge w
    public void delEdge(int i, int j); // Delete edge (i, j)
    public int weight(int i, int j); // Return weight of edge
    public int weight(Edge w); // Return weight of edge
    public void setMark(int v, int val); // Set Mark for v - put a label on the vertex
    public int getMark(int v); // Get Mark for v - read the label.
Labels on a vertex can be used to indicate that a particular edge has been visited during a traversal or in certain other algorithms that required us to keep track of which vertices have been seen.

The only interesting thing here is how to keep track of which edges are marked...ideas? Other than that, I don’t think there is anything interesting in the adjacency matrix implementation of this interface:

How do you find the first edge adjacent to a given vertex? How to find the next? Hint Mark ‘em

Traversing graphs is a little more complicated than traversing trees. First there is no special place to start, second, the graph may not be connected, so that you may not be able to reach every vertex from one starting spot and finally, the graph may have cycles. Our ability to mark vertices will be the tool that we need.

The first traversal, Depth First Search (DFS). The idea is to choose a starting vertex, mark it and then recursively choose an unmarked adjacent vertex and perform DFS from there. If the graph is connected and there are no unmarked adjacent vertices remaining, return. If the graph is not connected, choose a new unmarked vertex and start again.

static void DFS(Graph G, Vertex v) {

While (there are UNVISITED vertices of G) {
    choose an UNVISITED vertix of G, v
    preVisit(G,v); // Do something to the vertex if appropriate
    G.setMark(v, VISITED); // VISITED is a final var
    while (there are UNVISITED vertices adjacent to v, w)
        DFS(G, w);
    postVisit(G,v); // Do something to the vertex if appropriate
}

More precise code is in the book on p 202...note that the books method presumes a connected graph.

Example: Perform DFS on the graph of figure 7.2 starting at node 4. The nodes will be marked:

4 0 1 3 2 6 5 7 or
4 1 3 0 2 6 5 7

The second traversal is called Breadth First Search (BFS). Again choose a vertex and mark it. This time however, mark all the vertices adjacent to this one. Then, one by one, visit these
vertices and mark all the vertices adjacent to them until all vertices are marked (if the graph is connected). The usual technique for making sure that you mark all the edges in a BFS in an organized way is to place the marked vertices on a queue, then as you finish marking all the vertices adjacent to a particular vertex, remove it from the queue, and proceed to process the next vertex on the queue.

Example: Perform BFS on the example of figure 7.2 starting at node 3:

Mark 3 and place it on the queue.
Mark 1 and place it on the queue (queue is 3,1)
Remove 3 from the queue
Mark 0 and place it on the queue (queue is 1,0)
Mark 4 and place it on the queue (queue is 1,0,4)
Remove 1 from the queue
Remove 0 from the queue
Mark 2 and place it on the queue (queue is 4,2)
Remove 4 from the queue
Remove 2 from the queue....

Code ??

Definition: Given a DAG whose vertices are labeled with a comparable type. The DAG is said to be partially ordered if, for any arc, the label on the source of the arc is less than the label on the destination.

Definition. Given a partially ordered DAG, a topological sort of the graph’s vertices is a listing of the labels in such a way that if label i comes before label j in the DAG than label i comes before label j in the list.

Example: Figure 7.12.

Topological sorts are very useful in many contexts.

Constructing a topological sort is merely a matter of performing a variation of DFS. Starting at any vertex, perform DFS but perform no action during the preVisit on any vertex. After the recursive call the postVisit action is to list the vertex’s label. The list produced in this way is the topological sort in reverse order.

Example. Figure 7.12 Start at J1...then start at J3 finally assume that J1 is missing.

There is also a clever algorithm for constructing a topological sort using a queue....

1. To each vertex, v, associate the number of arcs of the form (x,v). (This is called the in-degree).
2. Place any vertices with in-degree 0 on the queue.
3. Remove the head of the queue, print out its label and decrement the in-degree of all its...
adjacent vertices.

4. If the in-degree of any vertex becomes 0, append it to the queue

If every the label on every vertex appears in the list, the list is a topological sort. If some label
does not appear on the list no topological sort is possible.

Shortest Path problems are interesting and we’ll come back if time allows.

Internal Sorting. That is sorting lists which are small enough to fit in main memory. We will
assume that the lists contain elements that have a key value of a comparable type, the problem
can be more formally stated as:

Internal Sorting:

Input: a list containing n elements, e_1,e_2,...e_n
Output: a list with those elements rearranged so that if i < j the e_i.key < e_j.key.

We already know a couple of sorting algorithms: Insertion sort, Selection sort and Bubble sort.
The time analysis of these algorithms is straight forward...

Insertion Sort

static void innsort( Element[] array) {
    for ( i = 1; i < array.length; i++ )
        for ( j = i; ( j> 0 ) && ( array[j].key() < array[j-1].key(); j– )
            swap( array, j,j-1);
}

The cost of the first loop execution is at most 1, the second at most 2 etc. so that the cost of
innsort is:

1+2+3....+ array.length ( n) = (n+1)*n/2 = O(n^2 ).

Bubble Sort and Selection Sort are similarly 1 ( n^2 ).

Advanced Sorts:

Merge Sort, is an example of what’s called a “divide and conquer algorithm”. It is a recursive
algorithm but nevertheless is pretty simple to understand. The list is divided in half and each half
is recursively “sent away” to be sorted. When the recursion finishes, the two half lists are merged
to form one sorted list...hence the name.

My approach is slightly different from the book...

First we need to be able to merge two previously sorted lists:
List merge (List aList, bList) { // Assumes that both lists are sorted according to
// value of element.key
    List newList;
    int i = 0; int j = 0; int k = 0;

    while (k < aList.length + bList.length) {
        if (aList[i].key <= bList[j].key) { // this is not exactly right
            newList[k] = aList[i];
            k++;
            i++;
        } else {
            newList[k] = bList[j];
            k++;
            j++;
        }

        return (newList);
    }

    The commented line should be

    if ((i < aList.length) && (aList[i].key <= bList[j].key || j == bList.length))

Another useful method splits a list at its middle:

void splitList (List inputList, firstHalfList, secondHalfList) {
    int midPoint = inputList.length /2;

    for (int i = 0; i <= midPoint; i++)
        firstHalfList[i] = inputList[i];
    for (int i = midPoint+1; i <= inputList.length; i++)
        secondHalfList[i] = inputList[i];
}

// A reasonable implementation of a List might include a subList (i, k) method. The splitList
// method
//would then just be firstHalfList = inputList.subList(0, midPoint) etc.

Finally mergeSort

List mergeSort( List inputList) {
    List firstHalfList, secondHalfList;

    if (inputList.length = 0 || inputList.length = 1)
return ( inputList ); // the recursion ends here
else {
    splitList( inputList, firstHalfList, secondHalfList);
    return ( merge( mergeSort( firstHalfList), mergeSort( secondHalfList )
    }
}

// Where is the rest?

The time complexity of mergeSort is best understood using recurrence relations.

mergeSort ( n ) = SplitList ( n ) + mergeSort ( n/2 ) + mergeSort ( n/2 ) + merge( n/2,n/2)

that is

mergeSort( n ) = n + mergeSort ( n/2 ) + mergeSort ( n/2 ) + n = 2*n + 2* mergeSort ( n/2 );

Where mergeSort( 0 ) = mergeSort( 1 ) =1

This is the familiar recurrence relation:

T( n ) = b if n = 1, and
T(n) = aT(n/c) + bn,  if n > 1

where a = 2, b = 2, c = 2

whose solution is O( n lgn )

and we conclude that its time complexity is 1( nlgn)

A completely un- obvious sorting algorithm Quicksort.

The idea:

Quicksort is another example of a divide and conquer algorithm, we will somehow divide
the list into two pieces such that the key value of every element in the first sub-list is smaller
than the key value of every element in the second sub-list. ( The sub-lists themselves will not in
general be sorted ). We will them sort both sub-lists and concatenate them to form the final,
sorted list.

Forming the sub-lists is the trick to the efficiency of Quicksort. We begin by choosing
some key value which is close to median key value of all the keys values in the list, this value is
called the “pivot”. We will place all elements in the list whose key value is less than the pivot in
the first sub-list and all those whose key value is larger in the second half list. To do this
efficiently requires the use of two pointers, Left and Right. Left begins the beginning ( left side )
of the list and moves towards the middle, Right begins at the end ( right side ) of he list and
again moves towards the middle. The procedure is at follows:
Look at the key value of the element that Left is pointing to
if it is less than the pivot, move to the next element in the list.
else pause
Look at the Key value of the element that Right is pointing to
if it is greater than or equal to the pivot, move to the preceding element in the list
else interchange the left and right elements.

The process ends when the left and right elements meet. The common element being pointed to
is the first element in the second half list.

Note that the maximum number of exchanges that must be made is n/2 and that each element is
pointed to by Left or Right exactly once so that the number of operations required for this
“partitioning” phase in \(1(n)\).

Example. Pivot is 5

\[
\begin{array}{cccccc}
2 & 6 & 4 & 9 & 8 & 1 \\
L & & & & & R \\
2 & 1 & 4 & 9 & 8 & 6 \\
LR \\
\end{array}
\]

Example. Pivot is 5

\[
\begin{array}{cccccc}
2 & 6 & 4 & 9 & 3 & 1 \\
L & & & & & R \\
2 & 1 & 4 & 9 & 3 & 6 \\
LR \\
\end{array}
\]

Example. Pivot is 5

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 6 & 9 \\
L & & & & & R \\
\end{array}
\]

Example: Pivot is 5

\[
\begin{array}{cccccc}
9 & 6 & 4 & 3 & 2 & 1 \\
LR \\
\end{array}
\]

The last thing we have to consider are the extreme conditions, i.e. the list empty or it has only
one element....but they’re easy because these lists are by definition sorted, so the algorithm
doesn’t need to do anything..
The trickiest theoretical part is how to choose the pivot. Clearly the most desirable thing to do is to actually calculate the median value, but this actually requires you to sort the list! So some suggestions:

1. Just use the first or last key value. This is very simple but gives disastrous results if the list is already sorted or sorted in reverse

2. Use the average of the first two elements

3. Find the median value of the first three elements of the list.

// Methods 2 and 3 guarantee that each sub-list has at least one element

4. Find the middle value in the list, i.e. theList[(theList.length+1)/2]

The code in the book is a simplified version. It assumes that the list is implemented as an array and uses method #4 to choose the pivot. see p 237

The asymptotic run time analysis: is a little like the analysis for Mergesort. We need to know the number of recursive calls times the cost of each call. Oddly, the total cost for each call is \( \frac{1}{n} \) because no matter how the list is divided up, the total lengths of all the sub-lists is always \( n \). A problem arises however because the number of recursive calls depends on how well the pivot is chosen.

The best case occurs when the pivot is exactly the median value in every call. This means that the analysis is the same as that for Mergesort, \( \frac{1}{n \ln n} \).

The worst case occurs if the partitioning always results in lists of size 1 and \( n-1 \) respectively. This occurs if the list is already sorted and some simple pivot technique ( like #2 above is used ). The number of recursive calls is then \( n-1 \) and the total cost is \( \frac{1}{n^2} \).

The usual case, of course, is somewhere in the middle. The correct calculation is very hairy, but can be made almost understandable by assuming that every possible partition combination is equally likely. In that case the recurrence relation of p 240 gives the solution. The closed form solution to this equation is \( n \ln n \) making the EXPECTED cost \( \frac{1}{n \ln n} \).

In point of fact, Quicksort has been shown ( by repeated trials ) to be faster on average than other advanced sorts including Mergesort. In order to avoid the worst case, the algorithm may do a quick scan of the list to see if its already sorted . It is the sorting technique employed by UNIX’s qsort function.

BinaryTreeSort, and HeapSort are two other advanced sorts that make use of stuff that we already know. In BinaryTreeSort, as we read the items in a list, we place them in a Binary Search Tree. Once every item has been placed in the tree, we perform an inOrder traversal. The list item come out sorted. Since the average time required to place an item in a tree is \( \ln n \) and
there are n item to be placed, the average time required to build the BST is \( nlgn \). Similarly the
time required to retrieve every item from the list is \( nlg \) making the time required for
BinaryTreeSort, \( 1( nlgn ) \). BinaryTreeSort unfortunately has
\( 1( n^2 ) \) worst case performance.

HeapSort in nearly identical, except that instead of putting the list elements in a BST we put
them in a minimum heap. Once the heap is created we repeated “deheap” it. Since the minimum
heap always has the smallest key value in the root, repeated deheap returns the key in sorted
order. Again, the time required to build the heap is \( nlgn \) and the time to completely deheap it is
also \( nlg \) making the running time \( 1( nlgn ) \) in all cases.

Both of these sorts have proven experimentally to be slower than quick sort but have the
advantage of returning elements of the list in sorted order as they are found as opposed to merge
sort and quick sort which do not produce any output until the entire list is sorted.

BucketSort.is an advanced sort of a different sort. It sorts an input list by using a set of labeled
lists each of which is called a bucket. An example is the simplest way to understand what’s
going on, so consider the books example in which we know that the key values are all two digit
integers ( i.e. range from 00 to 99. The buckets are in fact 10 linked lists each of which is labeled
by a digit from 0 to 9.

On the first pass through the list, elements are assigned to buckets according to the last
digit in their key value...i.e. the key value 59 is assigned to bucket 9. ( See p 251 ). Otherwise
said an element with key value K is assigned to list ( bucket ) K mod 10. ( K % 10 in Java )
The buckets are then read in label order and assigned to a second set of 10 similarly
labeled buckets, but this time each element is assigned to the bucket that corresponds to its first
digit. ( See p251 again ). Note this number is K div 10.

This idea can be generalized in two ways. The first is to use a different number of
buckets. The number of buckets used is called the “radix”.

Example: Do the example on p251 using 12 buckets

The second way to generalize any of these is to allow a wider range of key values. To handle
these we must allow for more passes, so if the keys can range from 000 to 999, using radix ten,
we must make three passes, the first on the rightmost digit, the second on the second digit and
the third on the leftmost digit. The right most is \( K \ % \ 10 \), the rightmost is \( K \ / \ 100 \) what is the
middle digit?

Time complexity is interesting. Assume an input list of length n and r buckets in each pass. The
number of steps in the sort is the number of passes times the amount of work done in each pass.
In each pass the algorithm must look at r buckets and process n elements, so if denote the
number of passes by p.
the time is \( p*(r+n) \). Now r should be considered to constant so the run time is \( 1(n*p) \) and the
sixty-four dollar question is “What is \( p \)?”. I hope that its clear that \( p \) depends, not directly on n,
but rather on the range of possible key values.....but THAT may depend on n !
Question: What is the maximum size list that you can have if you have unique key values with two decimal digits? three?

Question: If a list has 10,000 records, what is the fewest number of digits in the key?

In general, using a key value in base r, the smallest possible key size for a list with n elements is \( \log_r(n) \)
so the time complexity of this version of radix/bucket sort is \( \Theta(n \log_r(n)) \). Which is \( \Theta(n \log n) \).

Programming Exercise: Write a program that will sort a list of 10 Students. You may use any advanced sort algorithm. In addition to sorting the list, your program should count how many operations are required to complete the sort. This means that you must maintain a counter and increment it every time your program performs an arithmetic operation, an assignment or a comparison.

File Processing and External Sorting

A file is:

- **physically**, a collection of bytes. We are concerned mostly with files that are stored on secondary storage—magnetic or optical; disk or tape.
- **logically**, it a “destination” to which data is sent and from which data is retrieved.

Physical considerations.

Permanent storage devices come in two basic flavors:

- Direct Access Storage Devices (DASD) which are disks...
- Sequential access devices...tapes.

We could also segregate devices into magnetic or optical media

The organization of disks:

A disk drive usually contains several “platters”. Each platter contains one or two usable surfaces. The surfaces are “logically” divided in concentric circles called tracks (tracks that occupy identical positions on all the surfaces are grouped into a “cylinder”). Each track is further subdivided into what are called sectors. Each sector typically can contain a fixed number of bytes, usually 512. The capacity of a disk is consequently calculated as:

\[
(\text{bytes per sector}) \times (\text{sectors per track}) \times (\text{tracks per surface}) \times \text{number of surfaces}
\]

A sector is the smallest amount of disk space that can be retrieved at one time. To read a sector, the disk controller must move the “boom” until it is above the correct track (cylinder). It must then wait for the disk’s rotation to bring the desired sector beneath the head. A sector is sometimes called a “physical” record.
Usually a file will occupy more than one sector. In this case, we would like to store the sectors that comprise a file “adjacent” to one another...what adjacent means depends on the disk. Older disk controllers require so much time to process the data from one sector that they cannot immediately begin to read the sector which is physically adjacent to the previous one. In these cases the sectors may have to be interleaved...ie logically adjacent sectors may have to physically offset by some number sectors ( that offset has to be relatively prime to the number of sectors per track ). And a track will actually have to rotate multiple times to read all of its contents once. More modern disk have controllers that are fast enough so the minuscule amount of physical space that separates sectors is sufficient for them to reset so they can read an entire track in one rotation.

A cluster is some integral number of “adjacent” sectors. A cluster is the minimum number of sectors that can be allocated to a file when it needs more space. Since the sectors of a cluster are adjacent it is not necessary to reposition the read/write head in order to read the entire cluster. The number of clusters per file can be set by the systems administrator and is typically 3 or 4.

An extent is some number of cluster all of which can be read or written without repositioning the read write head...the maximum number of extents in a file is the same as the number of clusters, the minimum number is 1.

The fixed sector and cluster sizes can lead to difficulties. Suppose a file contains records of fixed length, 300 bytes. We could construct a file system in which record were allowed to cross over the boundaries of sectors ( records “span” sectors ). If we do this then its possible, in fact likely, that it will require 2 disk accesses in order to retrieve only one record...very inefficient. The alternative is to forbid records from spanning sectors, in which case we can only fit one record in each sector!. This intrinsic waste of disk space because of the sector structure is called internal fragmentation.

Some Operating systems, notably IBM systems allow users to choose the size of the “physical records”, called blocks. This avoids the problem of internal fragmentation by letting programmers tailor the size of the blocks to the size of the records in their file. Blocks can either be of fixed or variable length. Blocks usually must contain enough information about themselves to allow the file system to make sense of them. This extra information is contained in “sub-blocks” which contain for example the size of the block itself.

Whatever scheme is used for physical file organization, there is a certain amount of overhead in the form of wasted space. On a sector-addressable disk, a certain amount of space in each sector must be allocated to information like the sector address, capacity and whether or not the sector is usable. There are also small gaps between sectors and synchronization marks. In
block addressable the overhead is considerably more since interblock and inter subblock gaps have to used aside from the previously mentioned information necessary to read the blocks.

Cost of disk access:

**Seek time**: The time required to move the read/write head from its current position to the correct track. The amount of time required to perform this task can estimated by knowing the speed with which the head moves. Interestingly, the average amount of distance the head travels can be shown to be 1/3 of total width of the disk. For our purposes, we can use the figure of 10 msecs for this activity, although some disks are actually faster.

**Rotational Delay**: The time required for the correct position on the track to rotate underneath the read/write head. This depends entirely on the speed of the disk but 6msec’s is a good estimate. Rotational delay can be greatly influenced by the physical placement of a file. If a file requires more than one cylinder, it can stored in such a way that as soon as the first cylinder is filled, the read/write head moves to the second cylinder and begins to write right away so there is absolute no rotational delay when the file is read...this is called “disk striping”

**Burst transmission time**. This is the amount of time actually spent reading the file. It can be calculated by: (number of sectors read) * (time to read each sector ). The time to read each sector depends exactly on the number of sectors per track and the speed of the rotating disk and is estimated at 0.036 msec per sector.

The most important thing to learn is that reading from a disk is orders of magnitude slower than reading from main memory.

Field and record organization.

A stream file has no intrinsic organization. It is up to us (read the person who creates the file and writes the bytes into it) to impose structure on the stream of bytes...This “artificial organization” must also communicate this organization to others who want to use the file...THERE IS THE RUB.

Field Structure: How to tell where one field ends and another one begins...

**Method 1**: Force the field to be of fixed length (fixed number of bytes)...this is kind of the method most people think of first because it mimics the usual record/struct construct. This method is very easy to program but has serious drawbacks from a practical point of view...the fact that a field contains a fixed number of bytes means that data may have to be padded in order to fill up the field (wasted space) or worse it may have to be truncated to fit...

Example... suppose you decide to allot 10 bytes to a person’s first name field...then “Christopher” would become “Christophe” AND if someone else had allotted 12 bytes to the first name field, the two values would not match when compared.

**Method 2**: Use variable length fields and identify the length of the field by preceding the
field with an indicator of its length...In order to get the information contained in a field, one must read its length and then read that many more bytes...This approach increases the processing necessary to read a field... and automatically adds some number of bytes to each field.

Method 3. Separate fields by a delimiting character...We are used to this because we are used to delimiting statements in a program. An important consideration in this method is the character used as delimiter...it must be a character that does not legitimately occur in our data...this means the a space is not a good delimiter.

Method 4. Use a keyword to identify the value of fields. A record with this kind of structure looks like

LNAME = “Ames” FNAME =”Mary”...

In order to retrieve a fields value you just search for the field’s keyword and the copy the value...This technique seems very wasteful but in fact very useful for a record which might contain very many fields most of which are empty or assume default values...then only the fields which contain ( non-default ) have to be specified.

A record is a collection of fields which provide a model for a higher level of abstraction ( entity in the database jargon ). Similar to being able to distinguish between two fields, we must be able to separate records out of a stream file.

Method 1: Fixed length records. These work well with fixed length fields but work with other field types as well. Note that there is a fixed number of bytes in a cluster and if records are not allowed to span sectors this provides a de facto record length.

Method 2: Using a fixed number of fields. This is easy to implement and since most implementations have a fixed number of fields this is a useful method.

Method 3: Use variable length records and precede each record with an indicator of it’s length.

Method 4: Use an index.. In its simplest form, an index is a set of records of the form

[ record_id, record_locator ]

where the record_locator is some sort of way to find the beginning of the record indicated by the record_id...a good record record_locator is simply its byte offset from the beginning of the file.

Method 5: Use a delimiter to make the end of each record. EOLN is a frequently used delimiter.

Obviously some combinations of field and record structures make more sense than others..variable length records and fixed length fields don’t make a lot of sense.

Notes on representing the length of a record or a field....the length of course has to be an integer, but the file is really just a stream of bytes, so the preferred method is to represent the length as a sequence of digital characters.. 0,1,2,3,4,5,6,7,8,9 . This means even more work for the
application however since it must now be able to change an integer into a sequence of characters and visa-versa. In C these functions are called “itoa” and “atoi”. Some implementations of C++ provide these functions but our C++ does not...if you haven’t already written them as exercises you must do so now.

Sorting Preliminaries

...cosequential processing...matching the values in two files, finding their intersection or merging two sorted files.

In what follows, we’ll assume that all files are opened and correctly initialized. We’ll also assume that we have correctly implemented functions

void input(filename); which inputs the next cluster of filename, and
void input(filename, key_variable); which retrieves the next record from the named file.
void reset(filename) which re-initializes the file to read it from the beginning.

The matching algorithm

File Nested Loop Match(File file_1,file_2)
{
  open and initialize file_1, file_2 and the output_file;
  while not EOF(file_1)
  {
    input(file_1);
    while not EOF(file_2)
    {
      If the current blocks of file_1 and file_2 contain any matching records place them in the output buffer;
      if the output buffer fills up, output it to a temporary file (or the terminal)
        Input(file_2);
    } // end while not EOF(file_2)
    reset(file_2);
    input(file_1);
  } //end while not EOF(file_1)
}

the cost in disk I/O’s of executing N-L match is

\[ T_{N-L \text{ match}} = \Theta(m+m*n) \text{ + size of temp output file} \], where \( m \) is the size of file_1 in clusters and \( n \) is the size of file_2 in clusters.

The second algorithm assumes that the files are sorted...

The merging algorithm
File merge(File file_1; file_2) {

keytype key_1, key_2;

{ 
initialize both input files and the output file;
// get the initial key values for both files
input( file_1, key_1);
input( file_2, key_2);
while more records exist
{
  if ( key_1 < key_2 )
  {
    write(output_file, record_1);
    Input(file_1, key_1); // get the next record
  }
  else if ( key_1 > key_2 )
  {
    write( output_file, record_2 );
    input(file_2,key_2 );
  }
  else
  {
    Write(output_file, record_1);
    Input(file_1,key_1);
    Input(file_2, key_2)
  }
}
}

NOTE: This method closely mirrors the merge method for two lists

It is simple to rewrite the merge algorithm so that an arbitrary number of sorted files are merged ... the naive run time analysis is
\[ T_{merge \ k \ files} = \Theta(k*(n_1+...n_k)) \]
where \( n_i \) is the number of records in file_\( i \).

The k term comes from the fact that the current list of competing key values has to be searched linearly. You can improve this by using a heap...we’ll talk about this later... but in fact, we don’t care that much, because the cost of searching among the key values in main memory is insignificant compared the cost of reading all the files. Actually, we calculate the cost as
\[ T_{merge \ k \ files} = \Theta(B_1+B_2+...+B_n) \] where \( B_i \) is the size of file_\( i \) in Clusters.

The algorithm is simple to modify to form procedure match or intersection.

Sorting.
Given that the file is too large to fit entirely in main memory, our strategy must be:

- repeat until finished
  - read the file in a piece at a time,
  - sort that piece,
  - output that piece
- end repeat
- put the pieces together to form a sorted whole.

Merging will be our strategy for that last step, so let's begin by talking about what should happen in main memory.

Classic sorting algorithms like Quick Sort and Merge Sort aren't really suitable for this situation...the records are not sorted completely until the very last operation. We can do better with heap sort. Recall that arrays are a very good way to implement a heap. This technique is easily adapted to input buffers and records. The sorting method allows the CPU to input and sort records simultaneously because as the buffer blocks that correspond to the heap are emptied the blocks at the rear can be refilled with new records.

Using Heap sort while sorting a file:

```plaintext
// Assuming 5 buffer blocks, use four for input and one for output.

  Input 4 blocks of the file;
  Build a heap containing the records in these blocks,
  Read the heap, placing each "output" record in the output buffer
  As the output buffer fills up, output it and input a new block into the vacated buffer block.
```

**Algorithm Merge Sort:**

- Repeat until no more blocks exist.
  - Read as many blocks as possible into the input buffers
    - Sort the blocks (using Heap sort as above)
  - Output the sorted blocks into a temporary file. (called a "run")
- end repeat
- Apply the k-way merge algorithm to "runs"

**Notes on Merge Sort:**
- There is no limit on the size of the file.
- The initial reading of the file is sequential so some seeks can be avoided
Tape is a suitable medium
I/O and sorting operations can be overlapped
If a large number of runs is required, each cluster of the individual runs might have to read more than once...

Time Analysis of the merge:
// Assumes that the File contains n clusters and that there are k buffer blocks ( each block
// equaling one cluster) available.

-- Each run is then k clusters in size.
-- There n/k runs.
-- For the sort, the buffer space must divided into n/k pieces, one for every run, so the size
  of each piece is k/(n/k) = k^2 / n clusters.
-- If k^2 / n >= 1, then each run gets at least one buffer block to use during merge so the run
  has to be read only once but in general each cluster of each run has to be read sigt(n/k^2)
  times.
-- The total number of clusters that must be read during merge is n, so the number of seeks
  required is

  sigt(n/k^2 )*n =sigt (n^2 /k^2 ) which since k is constant is \( O( n^2 ) \)

Note: the cost of creating the runs is order n as is the cost of outputting the result, so the total is
\( O( n^2 ) \)

Sample calculation:
File 1 is 512 M clusters ( One Tbyte )
Ram is 16 Mbytes = 8K clusters
There would be 64K runs of 8K clusters each
Each cluster would be read 8 times during merge, so the total cost would be
512M clusters to create the runs +
4096M clusters for the merge +
512M cluster to output the result = 5KM clusters = 5G clusters.

EXERCISE: What would the cost be if the file were 4 times larger i.e. 2048M ( 2G ) clusters?

What can be done to speed up sorting?

Hardware..increase RAM = increased run size, better use of buffers.
Overlapping of physical requirement:
-- Worst 1 disk
-- Best A separate disk for each run with separate I/O channels for each device. Allows us to
  simultaneously read and write various runs
-- A better algorithm...

The multistep merge algorithm.
Input: A file, F, of size n clusters
output: that file, sorted
// Assumes : K input buffers of 1 cluster each and 1 out put buffer.
Repeat until no more clusters exist {
    Read K clusters of F
    Sort this part of the file in main memory;
    Output the run
} end repeat
/* at this point there are n/k runs. */
R = the number of runs;
While ( R> K){
    While there are runs remaining {
        Merge the first(next) K runs into a new super-run
    }End while
    R = number of super-runs;
}end while;
Merge the final R super runs;
}

Cost of the multi-step merge:

At each step, the number of clusters read is exactly n...the question then is how many steps are needed to produce the sorted file. If R_i represents the number of runs after step i, then
R_1 = n/K;
R_2 =R_1 /K = n/K^2 ;
.
.
R_i = R_{i-1}/K.

by brute force, if the number of steps is S, then
n/K^S < K < n/K^{S-1}
taking the log_k of each term
(since log is monotonically increasing )
\log_k (n/K^S) < 1 < \log_k (n/K^{S-1})
\log_k(n) - S < 1 < \log_k(n) -(S-1)
\log_k(n) < S+1 < \log_k(n)+1,...

Rounding up
S = \log_k(n) which is \epsilon(lgn)

Repeating our previous example

Recall n = 2G clusters, Buff size = 16K clusters:
first run produces n/Buff size = 128K clusters runs each of 16K clusters.
First merge produces 256K/Buff size = 4 runs each of size 256M clusters which can then be merged.
Cost = 3*2G clusters = 6G clusters.
Decreasing run time by increasing run size...

Recall from our discussion of Heap Sort, that after we output a buffer full of records, we can read another cluster into the vacated input buffer... Some of these new records - those with key values larger than those already output - could safely be added into our heap...this has the effect of adding to the length of the run...

Assume that we have enough buffer space large enough to hold exactly 3 integers and consider the input string:

21,67,12,47,5,16

Old way:

<table>
<thead>
<tr>
<th>On disk</th>
<th>In memory</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>21,67,12</td>
<td>16,47,5</td>
<td></td>
</tr>
<tr>
<td>21,67,12</td>
<td>5,16,47</td>
<td>//memory is sorted</td>
</tr>
<tr>
<td>21,67</td>
<td>12,16,47</td>
<td>5 //5 is output</td>
</tr>
<tr>
<td>21</td>
<td>67,12,47</td>
<td>5,16 //16 is output</td>
</tr>
<tr>
<td>21,67,12</td>
<td>5,16,47</td>
<td>//47 is output</td>
</tr>
<tr>
<td>12,21,67</td>
<td></td>
<td>// memory is sorted...etc</td>
</tr>
</tbody>
</table>

Using the new technique

| 21,67,12 | 16, 47, 5 |
| 21,67,12 | 5,16,47 |
| 21,67 | 12,16,47 |
| 21 | 67,16,47 |
| 21,67,47 | 5,12 |
| 5,12,16 //21 can still be added to the run! In fact they all can be... |

Called the replacement selection technique.

Surprising result due to Donald Knuth:

Given a random input, the average size of each run is doubled! Using this technique

Other External Sorting Techniques:

It is quite easy to adapt radix sort to an external sorting technique. Instead of creating an array of lists based on some radix value, create sub-files. For each subfile, subdivide it again using the second radix-digit etc. Eventually each of these sub-sub-files will become small enough that they can be sorted in main memory. The small files can then be concatenated to retrieve the fully sorted file.

The run-time of this radix sort is also \( \Theta(n \log(n)) \).
More on shortest path problems.