Java Data Structures (2nd edition)

August 4th, 2017

www.theparticle.com
Main
Services
Products
Resources
Forum
About Us
Contact

Particle Revelation
Hardware Destruction
Quotes
Humor [alpha]
Murphy's Laws

Programming
Java
Java Data Structures
C# Data Structures
Database Design
Graphics Tutorial
Artificial Intelligence

Downloads
SQLRunner
Graphics Tutorials
Hacking Tutorials
Java Applets
MIDI Music
Gov & Misc Docs

Games
Chess Game
Asteroids
Tic-Tac-Toe
Tetris

Applets
DRAW!
FlightBox
pWobble 3D
pRunner
NYU HWs
Swarms
Geometry
Chaos
Machine Learning

Academic
CISC 751[02]X
Networks (old)
OS (old)
AI (old)
App Dev (old)
C++ (old)
OOP (old)
Web (old)
Perl (old)
DBMS (old)
Perl (old)
ProgLangs (old)
PHP (old)
MltMedia (old)
Oracle (old)

Misc
Privacy Policy
Publications

Java Data Structures (2nd edition)

[**Printer Friendly Version**]

Introduction...

Welcome to ***Java Data Structures*** (2nd edition). This document was created with an intent to show people how easy Java really is, and to clear up a few *things* I've missed in the previous release of the document.

This is a growing document; as new features are added to the language, new techniques are discovered or realized, this document shall be updated to try to accommodate them all. If you have suggestions, or requests, (or spelling/grammar errors) just e-mail them, and I'll try to add the suggested topics into the subsequent release. Because this document is changing so much, I've decided to implement a version number. This release is: *v2.2.11*, updated: *May 7th, 2002*.

Current release of the document, including all the sources, can be downloaded here:

[**download zip with sources and everything**]

Of course, this document is free, and I intend to keep it that way. Selling of this document is NOT permitted. You *WILL* go to hell if you do (*trust me*). (not that anybody would want to buy it...) You may distribute this document (in *ANY* form), provided you don't change it. (yes, you CAN include it in a book provided you notify me and give me credit <and give me one free copy of the book>) To my knowledge, this document has already been reproduced and distributed within some corporations, schools and colleges, but has yet to be formally published in a book.

I take no responsibility for *ANYTHING*. I am only responsible for all the good things you like about the article. So, remember, if it's bad, don't blame me, if it's good, thank me (give me credit).

All the source has been compiled and tested using *JDK v1.2*. Although most things should work flawlessly with previous versions, there are things where *JDK 1.2* is more appropriate. If you find problems and/or errors, please let me know.

Although this document should be read in sequence, it is divided into several major sections, here they are:

profphreak.com

BLUE RIBBON CAMPAIGN

In contrast to what most people think about Java, it being a language with no pointers, data structures are quite easy to implement. In this section, I'll demonstrate few basic data structures. By learning how easy they are to implement in Java, you'll be able to write any implementation yourself.

I also think that this document is a pretty good introduction to ***Data Structures*** in general. All these concepts can be applied in any programming language. Incidentally, most of these programs are ported from their C++ counterparts. So, if you want to learn Data Structures in C/C++, you'll still find this document useful! Java is an Object Oriented language, but more so than C++, so, most data structure concepts are expressed and illustrated *"more naturally"* in Java! (*try not to raise your blood pressure from all the caffeine*)

I suggest that you be familiar with Java format, and know some other programming language in advance. Coincidentally, I and a couple of my friends are in the process of writing a C language book, which deals with all that "start up" stuff.

The way most examples are executed is through the JDK's command line Java interpreter. (at the prompt, you just type "`java`" and the name of the class to run.)

Variables...

Variables are the key to any program. There are variables called registers inside every CPU (Central Processing Unit). Every program ever written uses some form of variables. Believe it or not, the way you use variables can significantly impact your program. This section is a very simple introduction to what variables are, and how they're used in programs.

Usually, a variable implies a memory location to hold one instance of one specific type. What this means is that if there is an integer variable, it can only hold one integer, and if there is a character variable, it can only hold one character.

There can be many different types of variables, including of your own type. A sample declaration for different variable types is given below.

```
boolean t;
byte b;
char c;
int i;
long l;
```

I believe the above is straight forward, and doesn't need much explanation. Variable `'t'` is declared as `boolean` type, and `'b'` as of `byte` type, etc.

The above variables are what's know as primitive types. Primitive types in Java means that you don't have to create them, they're already available as soon as you declare them. (you'll see what I mean when we deal with Objects) It also means that there is usually some hardware equivalent to these variables. For example, an `int` type, can be stored in a 32 bit hardware register.

The other types of variables are instances of classes or Objects. Java is a very *Object Oriented* language, and everything in it, is an object. An object is an instance of a class. Your Java programs consist of classes, in which you manipulate objects, and make the whole program do what you want. This concept will be familiar to you if you've ever programmed C++, if not, think of objects as structures. An example of a simple class would be:

```
public class pSimpleObject{
    int i;
    public pSimpleObject(){
        i = 0;
    }
    public int get(){
        return i;
    }
    public void set(int n){
        i = n;
    }
}
```

As you can see, first we specify that the class is `public`, this means that it can be visible to other objects outside it's file. We later say that it's a `class`, and give it's name, which in this case is: `pSimpleObject`. Inside of it, the class contains an integer named `'i'`, and three functions. The first function named `pSimpleObject()`, is the constructor. It is called every time an object is created using this class. The `set()` and `get()` functions set and get the value of `'i'` respectively. One useful terminology is that functions in objects are not called functions, they're called methods. So, to refer to function `set()`, you'd say "method `set()`." That's all there is to objects!

The way you declare a variable, or in this case, an object of that class, is:

```
pSimpleObject myObject;
myObject = new pSimpleObject();
```

or

```
pSimpleObject myObject = new pSimpleObject();
```

The first example illustrates how you declare an object named `myObject`, of class `pSimpleObject`, and later instantiate it (a process of actual creation, where it calls the object's constructor method). The second approach illustrates that it all can be done in one line. The object does not get created when you just declare it, it's only created when you do a `new` on it.

If you're familiar with C/C++, think of objects as pointers. First, you declare it, and then you allocate a new object to that pointer. The only limitation seems to be that you can't do math on these pointers, other than that, they behave as plain and simple C/C++ pointers. (You might want to think of objects as references however.)

Using variables is really cool, and useful, but sometimes we'd like to have more. Like the ability to work with hundreds or maybe thousands of variables at the same time. And here's where our next section starts, the Arrays!

Arrays...

One of the most basic data structures, is an array. An array is just a number of items, of same type, stored in linear order, one after another. Arrays have a set limit on their size, they can't grow beyond that limit. Arrays usually tend to be easier to work with and generally more efficient than other structural approaches to organizing data; way better than a *no formal structure* approach.

For example, lets say you wanted to have 100 numbers. You can always resort to having 100 different variables, but that would be a pain. Instead, you can use the clean notation of an array to create, and later manipulate those 100 numbers. For example, to create an array to hold 100 numbers you would do something like this:

```
int[] myArray;
myArray = new int[100];
```

or

```
int[] myArray = new int[100];
```

or

```
int myArray[] = new int[100];
```

The three notations above do exactly the same thing. The first declares an array, and then it creates an array by doing a `new`. The second example shows that it can all be one in one line. And the third example shows that Java holds the backwards compatibility with C++, where the array declaration is: `int myArray[];` instead of `int[] myArray;`. To us, these notations are exactly the same. I do however prefer to use the Java one.

Working with arrays is also simple, think of them as just a line of variables, we can address the 5th element (counting from 0, so, it's actually the 6th element) by simply doing:

```
int i = myArray[5];
```

The code above will set integer `'i'` to the value of the 5th (counting from 0) element of the array. Similarly, we can set an array value. For example, to set the 50th element (counting from 0), to the value of `'i'` we'd do something like:

```
myArray[50] = i;
```

As you can see, arrays are fairly simple. The best and most convenient way to manipulate arrays is using loops. For example, lets say we wanted to make an array from 1 to 100, to hold numbers from 1 to 100 respectively, and later add seven to every element inside that array. This can be done very easily using two loops. (actually, it can be done in one loop, but I am trying to separate the problem into two)

```
int i;
for(i=0;i<100;i++)
    myArray[i] = i;
for(i=0;i<100;i++)
    myArray[i] = myArray[i] + 7;
```

In Java, we don't need to remember the size of the array as in C/C++. Here, we have the length variable in every array, and we can check it's length whenever we need it. So to print out any array named: `myArray`, we'd do something like:

```
for(int i = 0;i<myArray.length;i++)
    System.out.println(myArray[i]);
```

This will work, given the objects inside the myArray are printable, (have a corresponding `toString()` method), or are of primitive type.

One of the major limitations on arrays is that they're fixed in size. They can't grow or shrink according to need. If you have an array of 100 max elements, it will not be able to store 101 elements. Similarly, if you have less elements, then the unused space is being wasted (doing nothing).

Java API provides data storage classes, which implement an array for their storage. As an example, take the `java.util.Vector` class (JDK 1.2), it can grow, shrink, and do some quite useful things. The way it does it is by reallocating a new array every time you want to do some of these operations, and later copying the old array into the new array. It can be quite fast for small sizes, but when you're talking about several megabyte arrays, and every time you'd like to add one more number (or object) you might need to reallocate the entire array; that can get quite slow. Later, we will look at other data structures where we won't be overly concerned with the amount of the data and how often we need to resize.

Even in simplest situations, arrays are powerful storage constructs. Sometimes, however, we'd like to have more than just a plain vanilla array.

Array Stack...

The next and more serious data structure we'll examine is the Stack. A stack is a FILO (First In, Last Out), structure. For now, we'll just deal with the array representation of the stack. Knowing that we'll be using an array, we automatically think of the fact that our stack has to have a maximum size.

A stack has only one point where data enters or leaves. We can't insert or remove elements into or from the middle of the stack. As I've mentioned before, everything in Java is an object, (since it's an Object Oriented language), so, lets write a stack object!

```
public class pArrayStackInt{
    protected int head[];
    protected int pointer;

    public pArrayStackInt(int capacity){
        head = new int[capacity];
        pointer = -1;
    }
    public boolean isEmpty(){
        return pointer == -1;
    }
    public void push(int i){
        if(pointer+1 < head.length)
            head[++pointer] = i;
    }
    public int pop(){
        if(isEmpty())
            return 0;
        return head[pointer--];
    }
}
```

As you can see, that's the stack class. The constructor named `pArrayStackInt()` accepts an integer. That integer is to initialize the stack to that specific size. If you later try to `push()` more integers onto the stack than this capacity, it won't work. Nothing is complete without testing, so, lets write a test driver class to test this stack.

```
import java.io.*;
import pArrayStackInt;

class pArrayStackIntTest{
    public static void main(String[] args){
        pArrayStackInt s = new pArrayStackInt(10);
        int i,j;
        System.out.println("starting...");
        for(i=0;i<10;i++){
            j = (int)(Math.random() * 100);
            s.push(j);
            System.out.println("push: " + j);
        }
        while(!s.isEmpty()){
            System.out.println("pop: " + s.pop());
        }
        System.out.println("Done ;-)");
    }
}
```

The test driver does nothing special, it inserts ten random numbers onto the stack, and then pops them off. Writing to standard output exactly what it's doing. The output gotten from this program is:

```
starting...
push: 33
push: 66
push: 10
push: 94
push: 67
push: 79
push: 48
push: 7
push: 79
push: 32
pop: 32
pop: 79
pop: 7
pop: 48
pop: 79
pop: 67
pop: 94
pop: 10
pop: 66
pop: 33
Done ;-)
```

As you can see, the first numbers to be pushed on, are the last ones to be popped off. A perfect example of a FILO structure. The output also assures us that the stack is working properly.

Now that you've had a chance to look at the source, lets look at it more closely.

The `pArrayStackInt` class is using an array to store it's data. The data is `int` type (for simplicity). There is a head data member, that's the actual array. Because we're using an array, with limited size, we need to keep track of it's size, so that we don't overflow it; we always look at `head.length` to check for maximum size.

The second data member is `pointer`. Pointer, in here, points to the top of the stack. It always has the position which had the last insertion, or -1 if the stack is empty.

The constructor: `pArrayStackInt()`, accepts the maximum size parameter to set the size of the stack. The rest of the functions is just routine initialization. Notice that pointer is initialized to -1, this makes the next position to be filled in an array, 0.

The `isEmpty()` function is self explanatory, it returns `true` if the stack is empty (`pointer` is -1), and `false` otherwise. The return type is `boolean`.

The `push(int)` function is fairly easy to understand too. First, it checks to see if the next insertion will not overflow the array. If no danger from overflow, then it inserts. It first increments the pointer and then inserts into the new location pointed to by the updated `pointer`. It could easily be modified to actually make the array grow, but then the whole point of "simplicity" of using an array will be lost.

The `int pop()` function is also very simple. First, it checks to see if stack is not empty, if it is empty, it will return 0. In general, this is a really bad error to pop of something from an empty stack. You may want to do something more sensible than simply returning a 0 (an exception throw would not be a bad choice). I did it this way for the sake of simplicity. Then, it returns the value of the array element currently pointed to by pointer, and it decrements the pointer. This way, it is ready for the next `push` or `pop`.

I guess that just about covers it. Stack is very simple and is very basic. There are tons of useful algorithms which take advantage of this FILO structure. Now, lets look at alternative implementations...

Given the above, a lot of the C++ people would look at me strangely, and say: "All this trouble for a stack that can only store integers?" Well, they're probably right for the example above. It is too much trouble. The trick I'll illustrate next is what makes Java my favorite Object Oriented language.

In C, we have the `void*` type, to make it possible to store "generic" data. In C++, we also have the `void*` type, but there, we have very useful templates. Templates is a C++ way to make generic objects, (objects that can be used with any type). This makes quite a lot of sense for a data storage class; why should we care what we're storing?

The way Java implements these kinds of generic classes is by the use of parent classes. In Java, every object is a descendent of the `Object` class. So, we can just use the `Object` class in all of our structures, and later cast it to an appropriate type. Next, we'll write an example that uses this technique inside a generic stack.

```
public class pArrayStackObject{
    protected Object head[];
    protected int pointer;

    public pArrayStackObject(int capacity){
        head = new Object[capacity];
        pointer = -1;
    }
    public boolean isEmpty(){
        return pointer == -1;
    }
    public void push(Object i){
        if(pointer+1 < head.length)
            head[++pointer] = i;
    }
    public Object pop(){
        if(isEmpty())
            return null;
        return head[pointer--];
    }
}
```

The above is very similar to the `int` only version, the only things that changed are the `int` to `Object`. This stack, allows the `push()` and `pop()` of any `Object`. Lets convert our old test driver to accommodate this new stack. The new test module will be inserting `java.lang.Integer` objects (not `int`; not primitive type).

```
import java.io.*;
import pArrayStackObject;

class pArrayStackObjectTest{
    public static void main(String[] args){
        pArrayStackObject s = new pArrayStackObject(10);
        Integer j = null;
        int i;
        System.out.println("starting...");
        for(i=0;i<10;i++){
            j = new Integer((int)(Math.random() * 100));
            s.push(j);
            System.out.println("push: " + j);
        }
        while(!s.isEmpty()){
            System.out.println("pop: " + ((Integer)s.pop()));
        }
        System.out.println("Done ;-)");
    }
}
```

And for the sake of being complete, I'll include the output. Notice that here, we're not inserting elements of `int` type, we're inserting elements of `java.lang.Integer` type. This means, that we can insert any `Object`.

```
starting...
push: 45
push: 7
push: 33
push: 95
push: 28
push: 98
push: 87
push: 99
push: 66
push: 40
pop: 40
pop: 66
pop: 99
pop: 87
pop: 98
pop: 28
pop: 95
pop: 33
pop: 7
pop: 45
Done ;-)
```

I guess that covers stacks. The main idea you should learn from this section is that a stack is a FILO data structure. After this section, non of the data structures will be working with primitive types, and everything will be done solely with objects. (now that you know how it's done...)

And now, onto the array relative of Stack, the Queue.

Array Queues...

A queue is a FIFO (First In, First Out) structure. Anything that's inserted first, will be the first to leave (kind of like the real world queues.) This is totally the opposite of what a stack is. Although that is true, the queue implementation is quite similar to the stack one. It also involves pointers to specific places inside the array.

With a queue, we need to maintain two pointers, the `start` and the `end`. We'll determine when the queue is empty if `start` and `end` point to the same element. To determine if the queue is full (since it's an array), we'll have a `boolean` variable named `full`. To insert, we'll add one to the `start`, and mod (the `%` operator) with the size of the array. To remove, we'll add one to the `end`, and mod (the `%` operator) with the size of the array. Simple? Well, lets write it.

```
public class pArrayQueue{
    protected Object[] array;
    protected int start,end;
    protected boolean full;

    public pArrayQueue(int maxsize){
        array = new Object[maxsize];
        start = end = 0;
        full = false;
    }

    public boolean isEmpty(){
        return ((start == end) && !full);
    }

    public void insert(Object o){
        if(!full)
            array[start = (++start % array.length)] = o;
        if(start == end)
            full = true;
    }

    public Object remove(){
        if(full)
            full = false;
        else if(isEmpty())
            return null;
        return array[end = (++end % array.length)];
    }
}
```

Well, that's the queue class. In it, we have four variables, the `array`, the `start` and `end`, and a `boolean full`. The constructor `pArrayQueue(int maxsize)` initializes the queue, and allocates an `array` for data storage. The `isEmpty()` method is self explanatory, it checks to see if `start` and `end` are equal; this can only be in two situations: when the queue is empty, and when the queue is full. It later checks the `full` variable and returns whether this queue is empty or not.

The `insert(Object)` method, accepts an `Object` as a parameter, checks whether the queue is not `full`, and inserts it. The insert works by adding one to `start`, and doing a mod with `array.length` (the size of the `array`), the resulting location is set to the incoming object. We later check to see if this insertion caused the queue to become full, if yes, we note this by setting the `full` variable to `true`.

The `Object remove()` method, doesn't accept any parameters, and returns an `Object`. It first checks to see if the queue is `full`, if it is, it sets `full` to `false` (since it will not be `full` after this removal). If it's not `full`, it checks if the queue is empty, by calling `isEmpty()`. If it is, the method returns a `null`, indicating that there's been an error. This is usually a pretty bad bug inside a program, for it to try to remove something from an empty queue, so, you might want to do something more drastic in such a situation (like an exception throw). The method continues by removing the end object from the queue. The removal is done in the same way insertion was done. By adding one to the `end`, and later mod it with `array.length` (`array` size), and that position is returned.

There are other implementations of the same thing, a little re-arrangement can make several `if()` statements disappear. The reason it's like this is because it's pretty easy to think of it. Upon insertion, you add one to `start` and mod, and upon removal, you add one to `end` and mod... easy?

Well, now that we know how it works, lets actually test it! I've modified that pretty cool test driver from the stack example, and got it ready for this queue, so, here it comes:

```
import java.io.*;
import pArrayQueue;

class pArrayQueueTest{
    public static void main(String[] args){
        pArrayQueue q = new pArrayQueue(10);
        Integer j = null;
        int i;
        System.out.println("starting...");
        for(i=0;i<10;i++){
            j = new Integer((int)(Math.random() * 100));
            q.insert(j);
            System.out.println("insert: " + j);
        }
        while(!q.isEmpty()){
            System.out.println("remove: " + ((Integer)q.remove()));
        }
        System.out.println("Done ;-)");
    }
}
```

As you can see, it inserts ten random `java.lang.Integer Objects` onto the queue, and later prints them out. The output from the program follows:

```
starting...
insert: 3
insert: 70
insert: 5
insert: 17
insert: 26
insert: 79
insert: 12
insert: 44
insert: 25
insert: 27
remove: 3
remove: 70
remove: 5
remove: 17
remove: 26
remove: 79
remove: 12
remove: 44
remove: 25
remove: 27
Done ;-)
```

I suggest you compare this output to the one from stack. It's almost completely different. I guess that's it for this array implementation of this FIFO data structure. And now, onto something more complex...

Array Lists...

The next step up in complexity is a list. Most people prefer to implement a list as a linked list (and I'll show how to do that later), but what most people miss, is that lists can also be implemented using arrays. A list has no particular structure; it just has to allow for the insertion and removal of objects from both ends, and some way of looking at the middle elements.

A list is kind of a stack combined with a queue; with additional feature of looking at the middle elements. Preferably, a list should also contain the current number of elements. Well, lets not just talk about a list, but write one!

```
public class pArrayList{
    protected Object[] array;
    protected int start,end,number;

    public pArrayList(int maxsize){
        array = new Object[maxsize];
        start = end = number = 0;
    }
    public boolean isEmpty(){
        return number == 0;
    }
    public boolean isFull(){
        return number >= array.length;
    }
    public int size(){
        return number;
    }
    public void insert(Object o){
        if(number < array.length){
            array[start = (++start % array.length)] = o;
            number++;
        }
    }
    public void insertEnd(Object o){
        if(number < array.length){
            array[end] = o;
            end = (--end + array.length) % array.length;
            number++;
        }
    }
    public Object remove(){
        if(isEmpty())
            return null;
        number--;
        int i = start;
        start = (--start + array.length) % array.length;
        return array[i];
    }
    public Object removeEnd(){
        if(isEmpty())
            return null;
        number--;
        return array[end = (++end % array.length)];
    }
    public Object peek(int n){
        if(n >= number)
            return null;
        return array[(end + 1 + n) % array.length];
    }
}
```

The class contains four data elements: `array`, `start`, `end`, and `number`. The `number` is the number of elements inside the array. The `start` is the starting pointer, and the `end` is the ending pointer inside the `array` (kind of like the queue design).

The constructor, `pArrayList()`, and methods `isEmpty()`, `isFull()`, and `size()`, are pretty much self explanatory. The `insert()` method works exactly the same way as an equivalent queue method. It just increments the `start` pointer, does a mod (the `%` symbol), and inserts into the resulting position.

The `insertEnd(Object)` method, first checks that there is enough space inside the `array`. It then inserts the element into the `end` location. The next trick is to decrement the `end` pointer, add the `array.length`, and do a mod with `array.length`. This had the effect of moving the `end` pointer backwards (as if we had inserted something at the end).

The `Object remove()` method works on a very similar principle. First, it checks to see if there are elements to remove, if not, it simply returns a `null` (no `Object`). It then decrements `number`. We're keeping track of this `number` inside all insertion and removal methods, so that it always contains the current `number` of elements. We then create a temporary variable to hold the current position of the `start` pointer. After that, we update the `start` pointer by first decrementing it, adding `array.length` to it, and doing a mod with `array.length`. This gives the appearance of removing an element from the front of the list. We later return the position inside the array, which we've saved earlier inside that temporary variable `'i'`.

The `Object removeEnd()` works similar to the `insert()` method. It checks to see if there are elements to remove by calling `isEmpty()` method, if there aren't, it returns `null`. It then handles the `number` (number of elements) business, and proceeds with updating the `end` pointer. It first increments the `end` pointer, and then does a mod with `array.length`, and returns the resulting position. Simple?

This next `Object peek(int n)` method is the most tricky one. It accepts an integer, and we need to return the number which this integer is pointing to. This would be no problem if we were using an `array` that started at `0`, but we're using our own implementation, and the list doesn't necessarily start at `array` position `0`. We start this by checking if the parameter `'n'` is not greater than the `number` of elements, if it is, we return `null` (since we don't want to go past the bounds of the `array`). What we do next is add `'n'` (the requesting number) to an incremented `end` pointer, and do a mod `array.length`. This way, it appears as if this function is referencing the array from `0` (while the actual start is the incremented `end` pointer).

As I've said previously, the code you write is useless, unless it's working, so, lets write a test driver to test our list class. To write the test driver, I've converted that really cool Queue test driver, and added some features to test out the specifics of lists. Well, here it is:

```java
import java.io.*;
import pArrayList;

class pArrayListTest{
    public static void main(String[] args){
        pArrayList l = new pArrayList(10);
        Integer j = null;
        int i;
        System.out.println("starting...");
        for(i=0;i<5;i++){
            j = new Integer((int)(Math.random() * 100));
            l.insert(j);
            System.out.println("insert: " + j);
        }
        while(!l.isFull()){
            j = new Integer((int)(Math.random() * 100));
            l.insertEnd(j);
            System.out.println("insertEnd: " + j);
        }
        for(i=0;i<l.size();i++)
            System.out.println("peek "+i+": "+l.peek(i));
        for(i=0;i<5;i++)
            System.out.println("remove: " + ((Integer)l.remove()));
        while(!l.isEmpty())
            System.out.println("removeEnd: " + ((Integer)l.removeEnd()));
        System.out.println("Done ;-)");
    }
}
```

The test driver is nothing special, it inserts (in front) five random numbers, and the rest into the back (also five). It then prints out the entire list by calling `peek()` inside a `for` loop. It then continues with the removal (from front) of five numbers, and later removing the rest (also five). At the end, the program prints "Done" with a cute smiley face ;-)

The output from this test driver is given below. I suggest you examine it thoroughly, and make sure you understand what's going on inside this data structure.

```
starting...
insert: 14
insert: 72
insert: 71
insert: 11
insert: 27
insertEnd: 28
insertEnd: 67
insertEnd: 36
insertEnd: 19
insertEnd: 45
peek 0: 45
peek 1: 19
peek 2: 36
peek 3: 67
peek 4: 28
peek 5: 14
peek 6: 72
peek 7: 71
peek 8: 11
peek 9: 27
remove: 27
remove: 11
remove: 71
remove: 72
remove: 14
removeEnd: 45
removeEnd: 19
removeEnd: 36
removeEnd: 67
removeEnd: 28
Done ;-)
```

Well, if you really understand everything up to this point, there is nothing new anybody can teach you about arrays (since you know all the basics). There are however public tools available to simplify your life. Some are good, some are bad, but one that definitely deserves to have a look at is the `java.util.Vector` class; and that's what the next section is about!

The Vector...

The `java.util.Vector` class is provided by the Java API, and is one of the most useful array based data storage classes I've ever seen. The information provided here is as far as JDK 1.2 goes, future versions may have other implementations; still, the functionality should remain the same. A vector, is a growing array; as more and more elements are added onto it, the array grows. There is also a possibility of making the array smaller.

But wait a minute, all this time I've been saying that arrays can't grow or shrink, and it seems Java API has done it. Not quite. The `java.util.Vector` class doesn't exactly grow, or shrink. When it needs to do these operations, it simply allocates a new array (of appropriate size), and copies the contents of the old array into the new array. Thus, giving the impression that the array has changed size.

All these memory operations can get quite expensive if a `Vector` is used in a wrong way. Since a `Vector` has a similar architecture to the array stack we've designed earlier, the best and fastest way to implement a `Vector` is to do stack operations. Usually, in programs, we need a general data storage class, and don't really care about the order in which things are stored or retrieved; that's where `java.util.Vector` comes in very useful.

Using a `Vector` to simulate a queue is very expensive, since every time you insert or remove, the entire array has to be copied (not necessarily reallocated but still involves lots of useless work).

`Vector` allows us to view it's insides using an `Enumerator`; a class to go through objects. It is very useful to first be able to look what you're looking for, and only later decide whether you'd like to remove it or not. A sample program that uses `java.util.Vector` for it's storage follows.

```
import java.io.*;
import java.util.*;

class pVectorTest{
    public static void main(String[] args){
        Vector v = new Vector(15);
        Integer j = null;
        int i;
        System.out.println("starting...");
        for(i=0;i<10;i++){
            j = new Integer((int)(Math.random() * 100));
            v.addElement(j);
            System.out.println("addElement: " + j);
        }
        System.out.println("size: "+v.size());
        System.out.println("capacity: "+v.capacity());

        Enumeration enum = v.elements();
        while(enum.hasMoreElements())
            System.out.println("enum: "+(Integer)enum.nextElement());

        System.out.println("Done ;-)");
    }
}
```

The example above should be self evident (if you paid attention when I showed test programs for the previous data structures). The main key difference is that this one doesn't actually remove objects at the end; we just leave them inside. Removal can be accomplished very easily, and if you'll be doing anything cool with the class, you'll sure to look up the API specs.

Printing is accomplished using an `Enumerator`; which we use to march through every element printing as we move along. We could also have done the same by doing a `for` loop, going from `0` to `v.size()`, doing a `v.elementAt(int)` every time through the loop. The output from the above program follows:

```
starting...
addElement: 9
addElement: 5
addElement: 54
addElement: 49
addElement: 60
addElement: 81
addElement: 8
addElement: 91
addElement: 76
addElement: 81
size: 10
capacity: 15
enum: 9
enum: 5
enum: 54
enum: 49
enum: 60
enum: 81
enum: 8
enum: 91
enum: 76
```

```
enum: 81
Done ;-)
```

You should notice that when we print the `size` and `capacity`, they're different. The `size` is the current number of elements inside the `Vector`, and the `capacity`, is the maximum possible without reallocation.

A trick you can try yourself when playing with the `Vector` is to have `Vectors` of `Vectors` (since `Vector` is also an `Object`, there shouldn't be any problems of doing it). Constructs like that can lead to some interesting data structures, and even more confusion. Just try inserting a `Vector` into a `Vector` ;-)

I guess that covers the `Vector` class. If you need to know more about it, you're welcome to read the API specs for it. I also greatly encourage you to look at `java.util.Vector` source, and see for yourself what's going on inside that incredibly simple structure.

Nodes...

The other type of data structures are what's called Node based data structures. Instead of storing data in it's raw format, Node based data structures store nodes, which in turn store the data. Think of nodes as being elements, which may have one or more pointers to other nodes.

Yes, I did say the "pointer" word. Many people think that there are no pointers in Java, but just because you don't see them directly, doesn't mean they're not there. In fact, you can treat any object as a pointer.

Thus, the Node structure should have a data element, and a reference to another node (or nodes). Those other nodes which are referenced to, are called child nodes. The node itself is called the parent node (or sometimes a "father" node) in reference to it's children. (nice big happy family)

Well, the best way to visualize a node is to create one, so, lets do it. The node we'll create will be a one child node (it will have only one pointer), and we'll later use it in later sections to build really cool data structures. The source for our one child node follows:

```
public class pOneChildNode{
    protected Object data;
    protected pOneChildNode next;

    public pOneChildNode(){
        next = null;
        data = null;
    }
    public pOneChildNode(Object d,pOneChildNode n){
        data = d;
        next = n;
    }
    public void setNext(pOneChildNode n){
        next = n;
    }
    public void setData(Object d){
        data = d;
    }
    public pOneChildNode getNext(){
        return next;
    }
    public Object getData(){
        return data;
    }
    public String toString(){
        return ""+data;
    }
}
```

Go over the source, notice that it's nothing more than just set and get functions (pretty simple). The two data members are the `data` and `next`. The `data` member holds the data of the node, and `next` holds the pointer to the next node. Notice that `next` is of the same type as the class itself; it effectively points to the object of same class!

The `String toString()` method is the Java's standard way to print things. If an object wants to be printed in a special way, it will define this method, with instructions on how to print this object. In our case, we just want to print the `data`. Adding `data` to a bunch of quotation marks automatically converts it to type `String` (hopefully, our data will also have a `toString()` method defined on it). Without this method, we get the actual binary representation of the data members of this class (not a pretty nor meaningful printout).

Node based data structures provide for dynamic growing and shrinking, and are the key to some complex algorithms (as you'll see later). Now that we know how to implement a Node, lets get to something cool...

Linked Lists...

A linked list is just a chain of nodes, with each subsequent node being a child of the previous one. Many programs rely on linked lists for their storage because these don't have any evident restrictions. For example, the array list we did earlier could not grow or shrink, but node based ones can! This means there is no limit (other than the amount of memory) on the number of elements they can store.

A linked list has just one node, that node, has links to subsequent nodes. So, the entire list can be referenced from that one node. That first node is usually referred to as the head of the list. The last node in the chain of nodes usually has some special feature to let us know that it's last. That feature, most of the time is a `null` pointer to the next node.

```
[node0]->[node1]->[node2]->[node3]->[node4]->null
```

The example above illustrates the node organization inside the list. In it, `node0` is the head node, and `node4` is the last node, because it's pointer points to `null`. Well, now that you know how it's done, and what is meant by a linked list, lets write one. (I mean, that's why we're here, to actually write stuff!)

```
import pOneChildNode;

public class pLinkedList{
    protected pOneChildNode head;
    protected int number;

    public pLinkedList(){
        head = null;
        number = 0;
    }
    public boolean isEmpty(){
        return head == null;
    }
    public int size(){
        return number;
    }
    public void insert(Object obj){
        head = new pOneChildNode(obj,head);
        number++;
    }
    public Object remove(){
        if(isEmpty())
            return null;
        pOneChildNode tmp = head;
        head = tmp.getNext();
        number--;
        return tmp.getData();
    }
    public void insertEnd(Object obj){
        if(isEmpty())
            insert(obj);
        else{
            pOneChildNode t = head;
            while(t.getNext() != null)
                t=t.getNext();
            pOneChildNode tmp =
                new pOneChildNode(obj,t.getNext());
            t.setNext(tmp);
            number++;
        }
    }
    public Object removeEnd(){
        if(isEmpty())
            return null;
        if(head.getNext() == null)
            return remove();
        pOneChildNode t = head;
        while(t.getNext().getNext() != null)
            t = t.getNext();
        Object obj = t.getNext().getData();
        t.setNext(t.getNext().getNext());
        number--;
        return obj;
    }
    public Object peek(int n){
        pOneChildNode t = head;
        for(int i = 0;i<n && t != null;i++)
            t = t.getNext();
        return t.getData();
    }
}
```

Before we move on, lets go over the source. There are two data members, one named `head`, and the other named `number`. Head is the first node of the list, and `number` is the total number of nodes in the list. Number is primarily used for the `size()` method. The constructor, `pLinkedList()` is self explanatory. The `size()` and `isEmpty()` methods are also pretty easy.

Here comes the hard part, the insertion and removal methods. Method `insert(Object)` creates a new `pOneChildNode` object with `next` pointer pointing to the current `head`, and `data` the data which is inserted. It then sets the `head` to that new node. If you think about it, you'll notice that the `head` is still saved, and the new node points to it.

Method `Object remove()` works in a very similar fashion, but instead of inserting, it is removing. It first checks to see if the list is `isEmpty()` or not, if it is, it returns a `null`. It then saves the current `head` node, and then changes it to accommodate the removal (think about the logic), decrements the `number`, and returns the `data` from the previously saved node.

In the method `insertEnd(Object)`, we're actually inserting at the end of the list. We first check to see if the list is `isEmpty()`, if it is, we do a regular insertion (since it really doesn't matter which direction we're coming from if the list is `empty`). We then setup a loop to search for the end. The end is symbolized by the `next` pointer of the node being `null`. When we get to the end, we create a new node, and place it at the end location. Incrementing `number` before we return.

Method `Object removeEnd()` works in a similar fashion as `insertend(Object)` method. It also goes through the whole list to look for the end. At the beginning, we check if the list is not `isEmpty()`, if it is, we return a `null`. We then check to see if there is only one element in the list, if there is only one, we remove it using regular `remove()`. We then setup a loop to look for the node who's child is the last node. It is important to realize that if we get to the last node, we won't be able to erase it; we need the last node's parent node. When we find it, we get the `data`, setup necessary links, decrement `number`, and return the `data`.

The `Object peek(int)` method simply goes through the list until it either reaches the element requested, or the end of the list. If it reaches the end, it should return a `null`, if not, it should return the correct location inside the list.

As I have said before, it is very important to actually test. The ideas could be fine, and logical, but if it doesn't work, it doesn't work. So, lets convert our `pArrayListTest` driver to accommodate this class.

```
import java.io.*;
import pLinkedList;

class pLinkedListTest{
    public static void main(String[] args){
        pLinkedList l = new pLinkedList();
        Integer j = null;
        int i;
        System.out.println("starting...");
        for(i=0;i<5;i++){
            j = new Integer((int)(Math.random() * 100));
            l.insert(j);
            System.out.println("insert: " + j);
        }
        for(;i<10;i++){
            j = new Integer((int)(Math.random() * 100));
            l.insertEnd(j);
            System.out.println("insertEnd: " + j);
        }
        for(i=0;i<l.size();i++)
            System.out.println("peek "+i+": "+l.peek(i));
        for(i=0;i<5;i++)
            System.out.println("remove: " + ((Integer)l.remove()));
        while(!l.isEmpty())
            System.out.println("removeEnd: " + ((Integer)l.removeEnd()));
        System.out.println("Done ;-)");
    }
}
```

The test driver is nothing special, it's just a pretty simple conversion of the old test driver, so, I wont spend any time discussing it. The output follows.

```
starting...
insert: 65
insert: 78
insert: 21
insert: 73
insert: 62
insertEnd: 82
insertEnd: 63
insertEnd: 6
insertEnd: 95
insertEnd: 57
peek 0: 62
peek 1: 73
peek 2: 21
peek 3: 78
peek 4: 65
peek 5: 82
peek 6: 63
peek 7: 6
peek 8: 95
peek 9: 57
remove: 62
remove: 73
remove: 21
remove: 78
remove: 65
removeEnd: 57
removeEnd: 95
removeEnd: 6
removeEnd: 63
```

```
removeEnd: 82
Done ;-)
```

Look over the output, make sure you understand why you get what you get. Linked lists are one of the most important data structures you'll ever learn, and it really pays to know them well. Don't forget that you can always experiment. One exercise I'd like to leave up to the reader is to create a circular list. In a circular list, the last node is not pointing to `null`, but to the first node (creating a circle). Sometimes, lists are also implemented using two pointers; and there are many other variations you should consider and try yourself. You can even make it faster by having a "dummy" first node and/or "tail" node. This will eliminate most special cases, making it faster on insertions and deletions.

I guess that's it for the lists, next, I'll show you how to do simple and easy tricks to re-use code.

Reusing Tricks...

We have already written quite a lot of useful stuff, and there might come a time, when you're just too lazy to write something new, and would like to reuse the old source. This section will show you how you can convert some data structures previously devised, to implement a stack and a queue, with almost no creativity (by simply reusing the old source).

Before we start, lets review the function of a stack. It has to be able to push and pop items of from one end. What structure do we know that can do something similar? A list! Lets write a list implementation of a stack.

```
import pLinkedList;

public class pEasyStack{
    protected pLinkedList l;

    public pEasyStack(){
        l = new pLinkedList();
    }
    public boolean isEmpty(){
        return l.isEmpty();
    }
    public void push(Object o){
        l.insert(o);
    }
    public Object pop(){
        return l.remove();
    }
}
```

See how easily the above code simulates a stack by using a list? It may not be the best implementation, and it's certainly not the fastest, but when you need to get the project compiled and tested, you don't want to spend several unnecessary minutes writing a full blown optimized stack. Test for the stack follows:

```
import java.io.*;
import pEasyStack;

class pEasyStackTest{
    public static void main(String[] args){
        pEasyStack s = new pEasyStack();
        Integer j = null;
        int i;
        System.out.println("starting...");
        for(i=0;i<10;i++){
            j = new Integer((int)(Math.random() * 100));
            s.push(j);
            System.out.println("push: " + j);
        }
        while(!s.isEmpty()){
            System.out.println("pop: " + ((Integer)s.pop()));
        }
        System.out.println("Done ;-)");
    }
}
```

The stack test program is exactly the same as for the previous stack version (doesn't really need explanation). For the completion, I'll also include the output.

```
starting...
push: 23
push: 99
push: 40
push: 78
push: 54
push: 27
push: 52
push: 34
push: 98
push: 89
pop: 89
pop: 98
pop: 34
pop: 52
pop: 27
pop: 54
pop: 78
pop: 40
pop: 99
pop: 23
Done ;-)
```

You've seen how easily we can make a stack. What about other data structures? Well, we can just as easily implement a queue. One thing though, now instead of just inserting and removing, we'll be removing from the end (the other from the one we're inserting).

```
import pLinkedList;

public class pEasyQueue{
    protected pLinkedList l;

    public pEasyQueue(){
        l = new pLinkedList();
    }
    public boolean isEmpty(){
        return l.isEmpty();
    }
    public void insert(Object o){
        l.insert(o);
    }
    public Object remove(){
        return l.removeEnd();
    }
}
```

Pretty easy huh? Well, the test driver is also easy.

```
import java.io.*;
import pEasyQueue;

class pEasyQueueTest{
    public static void main(String[] args){
        pEasyQueue s = new pEasyQueue();
        Integer j = null;
        int i;
        System.out.println("starting...");
        for(i=0;i<10;i++){
            j = new Integer((int)(Math.random() * 100));
            s.insert(j);
            System.out.println("insert: " + j);
        }
        while(!s.isEmpty()){
            System.out.println("remove: " + ((Integer)s.remove()));
        }
        System.out.println("Done ;-)");
    }
}
```

I guess you get the picture, reusing code may not always be the best choice, but it sure is the easiest! Definitely, if you have time, always write a better implementation; these approaches are only good for the deadline, just to compile and test the code before the actual hard work of optimizing it. For the sake of completeness, the output of the above program follows:

```
starting...
insert: 77
insert: 79
insert: 63
insert: 59
insert: 22
insert: 62
insert: 54
insert: 58
insert: 79
insert: 25
remove: 77
remove: 79
remove: 63
remove: 59
remove: 22
remove: 62
remove: 54
remove: 58
remove: 79
remove: 25
Done ;-)
```

I guess that covers code reuse! And remember, you can also reuse the code in the Java API. The reuse can be of many forms. You can reuse it the way I've shown in this section, or you can extend that particular class to provide the necessary functions. (Extending a `java.util.Vector` class is very useful ;-)

Trees...

The next major set of data structures belongs to what's called Trees. They are called that, because if you try to visualize the structure, it kind of looks like a tree (root, branches, and leafs). Trees are node based data structures, meaning that they're made out of small parts called nodes. You already know what a node is, and used one to build a linked list. Tree Nodes have two or more child nodes; unlike our list node, which only had one child.

Trees are named by the number of children their nodes have. For example, if a tree node has two children, it is called a binary tree. If it has three children, it is called tertiary tree. If it has four children, it is called a quad tree, and so on. Fortunately, to simplify things, we only need binary trees. With binary trees, we can simulate any tree; so the need for other types of trees only becomes a matter of simplicity for visualization.

Since we'll be working with binary trees, lets write a binary tree node. It's not going to be much different from our `pOneChildNode` class; actually, it's going to be quite the same, only added support for one more pointer. The source for the follows:

```
public class pTwoChildNode{
    protected Object data;
    protected pTwoChildNode left,right;

    public pTwoChildNode(){
        data = null;
        left = right = null;
    }
    public pTwoChildNode(Object d){
        data = d;
        left = right = null;
    }
    public void setLeft(pTwoChildNode l){
        left = l;
    }
    public void setRight(pTwoChildNode r){
        right = r;
    }
    public void setData(Object d){
        data = d;
    }
    public pTwoChildNode getLeft(){
        return left;
    }
    public pTwoChildNode getRight(){
        return right;
    }
    public Object getData(){
        return data;
    }
    public String toString(){
        return ""+data;
    }
}
```

This node is so much similar to the previous node we did, that I'm not even going to cover this one. (I assume you'll figure it out by simply looking at the source) The children of the node are named `left` and `right`; these will be the left branch of the node and a right branch. If a node has no children, it is called a leaf node. If a node has no parent (it's the father of every node), it is called the root of the tree. This weird terminology comes from the tree analogy, and from the family tree analogy.

Some implementations of the tree node, might also have a back pointer to the parent node, but for what we'll be doing with the nodes, it's not necessary. The next section will talk about a generic binary tree which will be later used to create something cool.

Generic Tree...

Binary Trees are quite complex, and most of the time, we'd be writing a unique implementation for every specific program. One thing that almost never changes though is the general layout of a binary tree. We will first implement that general layout as an abstract class (a class that can't be used directly), and later write another class which extends our layout class.

Trees have many different algorithms associated with them. The most basic ones are the traversal algorithms. Traversals algorithms are different ways of going through the tree (the order in which you look at it's values). For example, an in-order traversal first looks at the left child, then the data, and then the right child. A pre-order traversal, first looks at the data, then the left child, and then the right; and lastly, the post-order traversal looks at the left child, then right child, and only then data. Different traversal types mean different things for different algorithms and different trees. For example, if it's binary search tree (I'll show how to do one later), then the in-order traversal will print elements in a sorted order.

Well, lets not just talk about the beauties of trees, and start writing one! The code that follows creates an abstract Generic Binary Tree.

```
import pTwoChildNode;

public abstract class pGenericBinaryTree{
    private pTwoChildNode root;

    protected pTwoChildNode getRoot(){
        return root;
    }
    protected void setRoot(pTwoChildNode r){
        root = r;
    }
    public pGenericBinaryTree(){
        setRoot(null);
    }
    public pGenericBinaryTree(Object o){
        setRoot(new pTwoChildNode(o));
    }
    public boolean isEmpty(){
        return getRoot() == null;
    }
    public Object getData(){
        if(!isEmpty())
            return getRoot().getData();
        return null;
    }
    public pTwoChildNode getLeft(){
        if(!isEmpty())
            return getRoot().getLeft();
        return null;
    }
    public pTwoChildNode getRight(){
        if(!isEmpty())
            return getRoot().getRight();
        return null;
    }
    public void setData(Object o){
        if(!isEmpty())
            getRoot().setData(o);
    }
    public void insertLeft(pTwoChildNode p,Object o){
        if((p != null) && (p.getLeft() == null))
            p.setLeft(new pTwoChildNode(o));
    }
    public void insertRight(pTwoChildNode p,Object o){
        if((p != null) && (p.getRight() == null))
            p.setRight(new pTwoChildNode(o));
    }
    public void print(int mode){
        if(mode == 1) pretrav();
        else if(mode == 2) intrav();
        else if(mode == 3) postrav();
    }
    public void pretrav(){
        pretrav(getRoot());
    }
    protected void pretrav(pTwoChildNode p){
        if(p == null)
            return;
        System.out.print(p.getData()+" ");
        pretrav(p.getLeft());
        pretrav(p.getRight());
    }
    public void intrav(){
        intrav(getRoot());
    }
    protected void intrav(pTwoChildNode p){
        if(p == null)
            return;
        intrav(p.getLeft());
        System.out.print(p.getData()+" ");
        intrav(p.getRight());
    }
    public void postrav(){
        postrav(getRoot());
    }
    protected void postrav(pTwoChildNode p){
        if(p == null)
            return;
        postrav(p.getLeft());
        postrav(p.getRight());
        System.out.print(p.getData()+" ");
    }
}
```

Now, lets go over it. The `pGenericBinaryTree` is a fairly large class, with a fair amount of methods. Lets start with the one and only data member, the `root`! In this `abstract class`, `root` is a `private` head of the entire tree. Actually, all we need is `root` to access anything (and that's how you'd implement it in other languages). Since we'd like to have access to `root` from other places though (from derived classes, but not from the "outside," we've also added two methods, named `getRoot()`, and `setRoot()` which get and set the value of `root` respectively.

We have two constructors, one with no arguments (which only sets `root` to null), and another with one argument (the first element to be inserted on to the tree). Then we have a standard `isEmpty()` method to find out if the tree is empty. You'll also notice that implementing a counter for the number of elements inside the tree is not a hard thing to do (very similar to the way we did it in a linked list).

The `getData()` method returns the data of the `root` node. This may not be particularly useful to us right now, but may be needed in some derived class (so, we stick it in there just for convenience). Throughout data structures, and mostly entire programming world, you'll find that certain things are done solely for convenience. Other "convenient" methods are `getLeft()`, `getRight()` and `setData()`.

The two methods we'll be using later (for something useful), are: `insertLeft(pTwoChildNode,Object)`, and `insertRight(pTwoChildNode,Object)`. These provide a nice way to quickly insert something into the left child (sub-tree) of the given node.

The rest are just print methods. The trick about trees are that they can be traversed in many different ways, and these print methods print out the whole tree, in different traversals. All of these are useful, depending on what you're doing, and what type of tree you have. Sometimes, some of them make absolutely no sense, depending on what you're doing.

Printing methods are recursive; a lot of the tree manipulation functions are recursive, since they're described so naturally in recursive structures. A recursive function is a function that calls itself (kind of like `pretrav()`, `intrav()`, and `postrav()` does).

Go over the source, make sure you understand what each function is doing (not a hard task). It's not important for you to understand why we need all these functions at this point (for now, we "just" need them); you'll understand why we need some of them in the next few sections, when we `extend` this class to create a really cool sorting engine.

Comparing Objects...

Comparing Objects in Java can be a daunting task, especially if you have no idea how it's done. In Java, we can only compare variables of `native` type. These include all but the objects (ex: `int`, `float`, `double`, etc.). To compare Objects, we have to make objects with certain properties; properties that will allow us to compare.

We usually create an `interface`, and `implement` it inside the objects we'd like to compare. In our case, we'll call the `interface pComparable`. Interfaces are easy to write, since they're kind of like `abstract` classes.

```
public interface pComparable{
    public int compareTo(Object o);
}
```

As you can see, there is nothing special to simple interfaces. Now, the trick is to `implement` it. You might be saying, why am I covering comparing of objects right in the middle of a binary tree discussion... well, we can't have a binary search tree without being able to compare objects. For example, if we'd like to use integers in our binary search tree, we'll have to design our own integer, and let it have a `pComparable` interface.

Next follows our implementation of `pInteger`, a number with a `pComparable interface`. I couldn't just `extend` the `java.lang.Integer`, since it's `final` (cannot be `extended`) (those geniuses!).

```
public class pInteger implements pComparable{
    int i;

    public pInteger(){
    }
    public pInteger(int j){
        set(j);
    }
    public int get(){
        return i;
    }
    public void set(int j){
        i = j;
    }
    public String toString(){
        return ""+get();
    }
    public int compareTo(Object o){
        if(o instanceof pInteger)
            if(get() > ((pInteger)o).get())
                return 1;
            else if(get() < ((pInteger)o).get())
                return -1;
        return 0;
    }
}
```

I believe most of the `interface` is self explanatory, except maybe for the `compareTo(Object)` method. In the method, we first make sure that the parameter is of type `pInteger`, and later using casting, and calling methods, we compare the underlying `native` members of `pInteger` and return an appropriate result.

A note on JDK 1.2: In the new versions of the JDK, you won't need to `implement` your own `pComparable`, or your own `pInteger`; since it's built in! There is a `Comparable interface`, and it's already `implemented` by the built in `java.lang.Integer`, `java.lang.String`, and other classes where you might need comparisons. I'm doing it this way only for compatibility with the older versions of JDK. I'll talk about JDK 1.2 features later in this document (hopefully).

Binary Search Trees...

And now, back to the show, or shall I say Binary Trees! A binary tree we'll be designing in this section will be what's known as binary search tree. The reason it's called this is that it can be used to sort numbers (or objects) in a way, that makes it very easy to search them; traverse them. Remember how I've said that traversals only make sense in some specific context, well, in binary search tree, only the in-traversal makes sense; in which numbers (objects) are printed in a sorted fashion. Although I'll show all traversals just for the fun of it.

A binary search tree will extend our pGenericBinaryTree, and will add on a few methods. One that we definitely need is the insert() method; to insert objects into a tree with binary search in mind. Well, instead of just talking about it, lets write the source!

```
import pComparable;

public class pBinarySearchTree extends pGenericBinaryTree{

    public pBinarySearchTree(){
        super();
    }

    public pBinarySearchTree(Object o){
        super(o);
    }

    public void print(){
        print(2);
    }

    public void insert(pComparable o){
        pTwoChildNode t,q;
        for(q = null, t = getRoot();
            t != null && o.compareTo(t.getData()) != 0;
            q = t,t = o.compareTo(t.getData()) < 0
                ? t.getLeft():t.getRight());
        if(t != null)
            return;
        else if(q == null)
            setRoot(new pTwoChildNode(o));
        else if(o.compareTo(q.getData()) < 0)
            insertLeft(q,o);
        else
            insertRight(q,o);
    }
}
```

As you can obviously see, the insert(pComparable) method is definitely the key to the whole thing. The method starts out by declaring two variables, 't', and 'q'. It then falls into a for loop. The condition inside the for loop is that 't' does not equal to null (since it was initially set to getRoot(), which effectively returns the value of root), and while the object we're trying to insert does not equal to the object already inside the tree.

Usually, a binary search tree does not allow duplicate insertions, since they're kind of useless; that's why we're attempting to catch the case where we're trying to insert a duplicate. Inside the for loop, we set 'q' to the value of the next node to be examined. We do this by first comparing the data we're inserting with the data in the current node, if it's greater, we set 't' to the right node, if less, we set it to the left node (all this is cleverly disguised inside that for statement).

We later check the value of 't' to make sure we've gotten to the end (or leaf) of the tree. If 't' is not null, that means we've encountered a duplicate, and we simply return. We then check to see if the tree is empty (didn't have a root), if it didn't, we create a new root by calling setRoot() with a newly created node holding the inserted data.

If all else fails, simply insert the object into the left or the right child of the leaf node depending on the value of the data. And that's that!

Understanding binary search trees is not easy, but it is the key to some very interesting algorithms. So, if you miss out on the main point here, I suggest you read it again, or get a more formal reference (where I doubt you'll learn more).

Anyway, as it was with our stacks and queues, we always had to test everything, so, lets test it! Below, I give you the test module for the tree.

```
import java.io.*;
import pInteger;
import pBinarySearchTree;

class pBinarySearchTreeTest{
    public static void main(String[] args){
        pBinarySearchTree tree = new pBinarySearchTree();
        pInteger n;
        int i;
        System.out.println("Numbers inserted:");
        for(i=0;i<10;i++){
            tree.insert(n=new pInteger((int)(Math.random()*1000)));
            System.out.print(n+" ");
        }
        System.out.println("\nPre-order:");
        tree.print(1);
        System.out.println("\nIn-order:");
        tree.print();
        System.out.println("\nPost-order:");
        tree.print(3);
    }
}
```

As you can see, it's pretty simple (and similar to our previous tests). It first `inserts` ten `pInteger` (`pComparable`) objects in to the tree, and then traverses the tree in different orders. These different orders print out the whole tree. Since we know it's a binary search tree, the in-order traversal should produce an ordered output. So, lets take a look at the output!

```
Numbers inserted:
500 315 219 359 259 816 304 902 681 334
Pre-order:
500 315 219 259 304 359 334 816 681 902
In-order:
219 259 304 315 334 359 500 681 816 902
Post-order:
304 259 219 334 359 315 681 902 816 500
```

Well, our prediction is confirmed! The in-order traversal did produce sorted results. There is really nothing more I can say about this particular binary search tree, except that it's worth knowing. This is definitely not the fastest (nor was speed an issue), and not necessarily the most useful class, but it sure may proof useful in teaching you how to use trees.

And now, onto something completely different! NOT! We're going to be doing trees for a while... I want to make sure you really understand what they are, and how to use them. (and to show you several tricks other books try to avoid <especially Java books>)

Tree Traversals...

I've talked about tree traversals before in this document, but lets review what I've said. Tree's are created for the sole purpose of traversing them. There are two major traversal algorithms, the depth-first, and breadth-first.

So far, we've only looked at depth-first. Pre-traversal, in-traversal, and post-traversal are subsets of depth-first traversals. The reason it's named depth-first, is because we eventually end up going to the deepest node inside the tree, while still having unseen nodes closer to the root (it's hard to explain, and even harder to understand). Tracing a traversal surely helps; and you can trace that traversal from the previous section (it's only ten numbers!).

The other type of traversal is more intuitive; more "human like." Breadth-first traversal goes through the tree top to bottom, left to right. Lets say you were given a tree to read (sorry, don't have a non-copyrighted picture I can include), you'd surely read it top to bottom, left to right (just like a page of text, or something).

Think of a way you visualize a tree... With the root node on top, and all the rest extending downward. What *Breadth-First* allows us to do is to trace the tree from top to bottom as you see it. It will visit each node at a given tree depth, before moving onto the the next depth.

A lot of the algorithms are centered around *Breadth-First* method. Like the search tree for a *Chess* game. In chess, the tree can be very deep, so, doing a *Depth-First* traversal (search) would be costly, if not impossible. With *Breadth-First* as applied in *Chess*, the program only looks at several moves ahead, without looking too many moves ahead.

The Breadth-First traversal is usually from left to right, but that's usually personal preference. Because the *standard* consul does not allow graphics, the output may be hard to correlate to the actual tree, but I will show how it's done.

As with previous examples, I will provide some modified source that will show you how it's done. An extended `pBinarySearchTree` is shown below:

```
import pTwoChildNode;
import pBinarySearchTree;
import pEasyQueue;

public class pBreadthFirstTraversal extends pBinarySearchTree{

    public void breadth_first(){
        pEasyQueue q = new pEasyQueue();
        pTwoChildNode tmp;
        q.insert(getRoot());
        while(!q.isEmpty()){
            tmp = (pTwoChildNode)q.remove();
            if(tmp.getLeft() != null)
                q.insert(tmp.getLeft());
            if(tmp.getRight() != null)
                q.insert(tmp.getRight());
            System.out.print(tmp.getData()+" ");
        }
    }
}
```

As you can see, the class is pretty simple (only one function). In this demo, we're also using `pEasyQueue`, developed earlier in this document. Since breadth first traversal is not like depth first, we can't use recursion, or stack based methods, we need a queue. Any recursive method can be easily simulated using a stack, not so with breadth first, here, we definitely need a queue.

As you can see, we start by first inserting the `root` node on to the queue, and loop while the queue is not `isEmpty()`. If we have a left node in the node being examined, we insert it in to the queue, etc. (same goes for the right node). Eventually, the nodes inserted in to the queue, get removed, and subsequently, have their left children examined. The process continues until we've traversed the entire tree, from top to bottom, left to right order.

Now, lets test it. The code below is pretty much the same code used to test the tree, with one minor addition; the one to test the breadth-first traversal!

```
import java.io.*;
import pInteger;
import pBinarySearchTree;

class pBreadthFirstTraversalTest{
    public static void main(String[] args){
        pBreadthFirstTraversal tree = new pBreadthFirstTraversal();
        pInteger n;
        int i;
        System.out.println("Numbers inserted:");
        for(i=0;i<10;i++){
            tree.insert(n=new pInteger((int)(Math.random()*1000)));
            System.out.print(n+" ");
        }
        System.out.println("\nPre-order:");
        tree.print(1);
        System.out.println("\nIn-order:");
        tree.print();
        System.out.println("\nPost-order:");
        tree.print(3);
        System.out.println("\nBreadth-First:");
        tree.breadth_first();
    }
}
```

As you can see, nothing too hard. Next, goes the output of the above program, and you and I will have to spend some time on the output.

```
Numbers inserted:
890 474 296 425 433 555 42 286 724 88
Pre-order:
890 474 296 42 286 88 425 433 555 724
In-order:
42 88 286 296 425 433 474 555 724 890
Post-order:
88 286 42 433 425 296 724 555 474 890
Breadth-First:
890 474 296 555 42 425 724 286 433 88
```

Looking at the output in this format is very abstract and is not very intuitive. Lets just say we have some sort of a tree, containing these numbers above. We were looking at the root node. Now, looking at the output of this program, can you guess what the root node is? Well, it's the first number in breadth-first: `890`. The left child of the root is: `474`. Basically, the tree looks like:

```
            |--[890]
            |
       |--[474]--|
       |         |
  |--[296]--|  [555]--|
  |         |         |
[ 42]--|  [425]--|  [724]
       |         |
  |--[286]     [433]
  |
[ 88]
```

As is evident from this picture, I'm a terrible ASCII artist! (If it's screwed up, sorry).

What you can also see is that if you read the tree form left to right, top to bottom, you end up with the breadth first traversal. Actually, this is a good opportunity for you to go over all traversals, and see how they do it, and if they make sense.

There are many variations to these traversals (and I'll show a few in subsequent sections), for now, however, try to understand these four basic ones.

Well, see you again in the next section, where we examine some ways you can speed up your code and take a look at JDK 1.2 features, which will simplify our life a LOT!

Node Pools...

Since you've looked at node based data structures, and their flexibility, you might wonder why would we want to use anything else. Well, sometimes, the situation comes where the standards are not enough. Where we need that extra boost of performance, which is currently occupied by the Java's garbage collection.

In any environment, allocating and releasing memory is slow! Most of the time, you should try to avoid the procedure, unless you really need it (or it's not in a time critical loop). You certainly wouldn't want to allocate and/or release memory every time you do anything to a list, or a binary tree. What can we do to avoid the natural way? Plenty!

The most common way around these kinds of problems is to use our own allocation technique, which is perfectly suited for our purposes, and will generally be much faster than the current system's approach (but always make sure; system techniques are quite fast as well).

Using our own allocation scheme does imply that we'll have to restrict our selves, but most of the times (when you really need it), the restriction is worth while. What we will be doing is simulating the allocation and release of nodes, using an array of nodes, stored in a linked list fashion. It may seem complex, but in reality, it's not.

This array of nodes, is usually referred to as the *"Node Pool"* (we *"catch"* one when we need one, and *"throw it back"* when we don't). I know, I know, I haven't been the same since that girl on IRC has stolen my humor.

There are two considerations we should think about before we start programming, however. These two are whether we want to make the node-pool itself static or dynamic? Shall we allocate the node-pool with every new instance of the object, or shall we just share one common node-pool for all the objects of that class?

These two approaches are both quite useful in different situations. For example, when was the last time you've ever used more than 52 cards in a deck of cards? That implies that your deck of cards class may have no more than 52 cards at a time; a perfect candidate for a Node Pool! Inside your game, you have this one class, which can only hold 52 cards, no more. Then again, what if your card game has to have several decks? You might want to extend that a little more; like have a separate deck for each deck! (am I making any sense?)

The reason I describe this deck of cards example is a historical one; it was the first program I've ever used with Node Pools (long time ago; in C++). The one static node-pool allows other instances of the class (inside the same program) to know which cards are missing from this universal deck, and which are present. On the other hand, if you have a local (non-static) node-pool, each deck handles it's own cards, and non of the other decks know anything about it.

It is important to understand this concept, since it makes a lot of sense <*sometimes*> (and will allow you to write robust code that doesn't waste memory). Then again, sometimes it makes absolutely no sense to use a node-pool at all!

The example we will design in the next few sections will deal with a binary tree, which does not use dynamic memory when inserting (or removing) nodes. It will have a non-static node-pool; meaning that you'll have to specify the maximum number of elements when it's instantiated (create an instance of it). I believe the example will be general enough for you to learn to apply node-pools in various different situations. Since I've already showed how to use my own `pComparable` for JDK 1.1 (or lower), this next example will deal only with JDK 1.2 `java.lang.Comparable interface` for most purposes. (A `java.lang.Comparable` interface is worth learning, since it's very useful; I only wonder why they haven't come up with something similar earlier?)

Node Pool Nodes...

As with everything else, there are TONS of ways to write exactly the same thing! With nodes, there is no exception. I prefer to implement node-pool nodes with integer children; where the integer points to the next child inside the node-pool array. This may not be the best *Object Oriented* way to do this, since it's binding the node-pool node to the node-pool tree. However, there are many other solutions to this, and many are more *"elegant"* than the one I'll be using here. So, if you don't like it, implement your own version!

If you've paid close attention to the discussion about regular binary trees (somewhere in this document), you'll find this discussion of node-pool stuff pretty easy and similar to that "regular" stuff. Enough talk, lets go and write this node-pool node!

```
public class pNodePoolTreeNode{

    private Object data;
    private int left;
    private int right;

    public pNodePoolTreeNode(){
        left = right = -1;
        data = null;
    }
    public pNodePoolTreeNode(int l,int r){
        left = l;
        right = r;
        data = null;
    }
    public Object getData(){
        return data;
    }
    public void setData(Object o){
        data = o;
    }
    public int getLeft(){
        return left;
    }
    public void setLeft(int l){
        left = l;
    }
    public int getRight(){
        return right;
    }
    public void setRight(int r){
        right = r;
    }
    public String toString(){
        return new String(data.toString());
    }
}
```

The code above is pretty much self explanatory (if it's not, take a look back at binary tree nodes section, and you'll figure it out). The only significant thing you should notice is that children of the node are now integers, instead of references to other node objects. Although, it later turns out that what these integer children are referencing are in fact nodes.

Well, that was easy enough, are you ready to figure out how a tree actually manipulates these?

Node Pool Generic Trees...

A node-pool tree is not that much different from a regular tree. The only key differences is that it has to maintain a linked list (more or less a "stack") of free nodes, and allocate them when needed.

Another tricky thing is that now, the node's children are pointed to by integers, which are references into a node-pool array; so, watch out. You'll notice that a lot of the "tree stuff" has been re-used from the previous version, and you're right! It makes it a lot easier to understand on your part, and a lot easier to write on mine! (oh, I just ran out of coffee...)

```
import pNodePoolTreeNode;
import java.io.*;
```

```
public abstract class pNodePoolArrayTree{

    private int numNodes,nextAvail;
    private pNodePoolTreeNode[] arrayNodes;
    protected int root;

    private void init(){
        int i;
        root = -1;
        arrayNodes = new pNodePoolTreeNode[numNodes];
        nextAvail = 0;
        for(i=0;i<numNodes;i++)
            arrayNodes[i] = new pNodePoolTreeNode(-1,i+1);
        getNode(i-1).setRight(-1);
    }
    public pNodePoolArrayTree(){
        numNodes = 500;
        init();
    }
    public pNodePoolArrayTree(int n){
        numNodes = n;
        init();
    }
    protected int getNode(){
        int i = nextAvail;
        nextAvail = getNode(nextAvail).getRight();
        if(nextAvail == -1){
            nextAvail = i;
            return -1;
        }
        return i;
    }
    protected void freeNode(int n){
        getNode(n).setRight(nextAvail);
        nextAvail = n;
    }
    public boolean isEmpty(){
        return getRoot() == -1;
    }
    public boolean isFull(){
        int i = getNode();
        if(i != -1){
            freeNode(i);
            return false;
        }
        return true;
    }
    protected Object getData(){
        if(isEmpty())
            return null;
        return getNode(getRoot()).getData();
    }
    protected void setData(Object o){
        if(isEmpty())
            root = getNode();
        getNode(root).setData(o);
        getNode(root).setLeft(-1);
        getNode(root).setRight(-1);
    }
    protected int getLeft(){
        if(isEmpty())
            return -1;
        return getNode(getRoot()).getLeft();
    }
    protected void setLeft(int n){
        if(isFull())
            return;
        if(isEmpty()){
            root = getNode();
            getNode(getRoot()).setRight(-1);
        }
        getNode(getRoot()).setLeft(n);
    }
    protected void setRight(int n){
        if(isFull())
            return;
        if(isEmpty()){
            root = getNode();
            getNode(getRoot()).setLeft(-1);
        }
        getNode(getRoot()).setRight(n);
    }
    protected int getRight(){
        if(isEmpty())
            return -1;
        return getNode(root).getRight();
    }
    protected int getRoot(){
        return root;
    }
    protected pNodePoolTreeNode getNode(int n){
        if(n != -1)
            return arrayNodes[n];
        else return null;
    }
    protected void insertLeft(int node,Object o){
        if((node != -1) && (getNode(node).getLeft() == -1)
            && !isFull()){
            int i = getNode();
            getNode(i).setData(o);
            getNode(i).setRight(-1);
            getNode(node).setLeft(i);
        }
    }
    protected void insertRight(int node,Object o){
        if((node != -1) && (getNode(node).getRight() == -1)
            && !isFull()){
            int i = getNode();
            getNode(i).setData(o);
            getNode(i).setRight(-1);
            getNode(node).setRight(i);
        }
    }
    public void print(int mode){
        switch(mode){
        case 1:
            pretrav();
            break;
        case 2:
            intrav();
            break;
        case 3:
            postrav();
            break;
        }
    }
    public void pretrav(){
        pretrav(getRoot());
    }
    private void pretrav(int p){
        if(p == -1)
            return;
        System.out.print(getNode(p).getData()+" ");
        pretrav(getNode(p).getLeft());
        pretrav(getNode(p).getRight());
    }
    public void intrav(){
        intrav(getRoot());
    }
    private void intrav(int p){
        if(p == -1)
            return;
        intrav(getNode(p).getLeft());
        System.out.print(getNode(p).getData()+" ");
        intrav(getNode(p).getRight());
    }
    public void postrav(){
        postrav(getRoot());
    }
    private void postrav(int p){
        if(p == -1)
            return;
        postrav(getNode(p).getLeft());
        postrav(getNode(p).getRight());
        System.out.print(getNode(p).getData()+" ");
    }
}
```

There are not that many hard methods in this one; so, lets focus in on the hard ones, and I'll leave you to figure out the *"easy"* ones yourself.

One thing I'd like to mention of the top so that it causes as little confusion as possible: now that we're not using objects, but integers, we still have to note the "`null` node" somehow, so, in our example, I'm using `-1` as the "`null` node" indicator.

Data members `numNodes`, `nextAvail`, and `arrayNodes` are there to keep track of the linked list of free and occupied nodes. It's not exactly a linked list (although many people like to think of it that way), it's more or less a simple array stack we've implemented at the beginning of this document. The `numNodes` is the maximum number of nodes inside this tree; we could just as well substitute it for `arrayNodes.length`, but I think this "separation" is more convenient.

The `nextAvail` member is the one that points to the next available node. Our allocation methods, `getNode()` and `freeNode(int n)`, uses this particular *pointer* to allocate and release nodes. And `arrayNodes` is just an array holding the node-pool. All of the integer child references are into this `arrayNodes` array; which we allocate dynamically during instantiation (creation) of the object.

Inside the `init()` function, we allocate the `arrayNodes` array to hold the node-pool. We later loop through all the nodes inside of it, and turn them into a linked list (with right child pointing to the next available node and left child being `-1`). The last node in the list is apparently marked `-1`; since it's last!

I suggest you go over the source for `getNode()` and `freeNode(int n)`; try to understand what it's doing. Which is nothing more complex than what we've already done! (look over the linked list description if you find it confusing)

The rest of the functions are pretty much self explanatory (you should be able to figure them out). Most of them are just a port of the old tree functions to use integer children with node-pool nodes. Well, are you ready for a binary sort tree implementing a node-pool (ready or not, we're going there)!

Node Pool Sort Trees...

As I've mentioned before, node-pool implementations are fairly similar to the regular ones, so, our old sort tree will not change much. The only things that will change however, are the node handling methods. Since we already know what needs to be done (review the binary sort example described previously in this document), we can just go ahead and write the source.

```
import pNodePoolTreeNode;
import pNodePoolArrayTree;
import java.lang.*;

public class pNodePoolSortArrayTree extends pNodePoolArrayTree{

    public pNodePoolSortArrayTree(){
        super();
    }
    public pNodePoolSortArrayTree(int n){
        super(n);
    }
    public void print(){
        print(2);
    }
    public void insert(Comparable obj){
        int t,q = -1;
        t = getRoot();
        while(t != -1 && !(obj.equals(getNode(t).getData()))){
            q = t;
            if(obj.compareTo(getNode(t).getData()) < 0)
                t = getNode(t).getLeft();
            else
                t = getNode(t).getRight();
        }
        if(t != -1)
            return;
        if(q == -1){
            setData(obj);
            return;
        }
        if(obj.compareTo(getNode(q).getData()) < 0)
            insertLeft(q,obj);
        else
            insertRight(q,obj);
    }
}
```

Now, the following only makes sense inside a JDK 1.2 (or above) context. Earlier version of the JDK do not support the `Comparable interface`. I just though it would be nice to show how to use all these cool interfaces introduced with JDK 1.2!

The code does nothing special, and chances are, you can figure it out on your own. The only seeming difference between this source and the one presented earlier, is that in this one, whenever we test for a null node, we're not testing the node, we're testing whether if it has value of `-1`. Other than that, the code looks almost identical.

This code is pretty bad, however, in illustrating the beauty of Object Oriented programming. With our approach, we needed to change the whole thing just to implement node-pools. Ideally, we should strive to only have to change one part of the system. For example, we could have only changed the `abstract` parent class, and leave everything else as it was. Or we could have changed the node a little, and make the node itself support stuff, but I just thought this was a clearer approach, without dragging all that Object Oriented stuff into the issue.

I think I've made this Node Pool section longer than it should be; so, let me quickly wrap it up by getting right to the point. Lets write a test driver (which will be an almost identical copy of the old test driver).

```
import java.io.*;
import pNodePoolSortArrayTree;
```

```
public class pNodePoolSortArrayTreeTest{

    public static void main(String[] args){
        pNodePoolSortArrayTree tree =
            new pNodePoolSortArrayTree(100);
        int i,n;
        System.out.println("Numbers inserted:");
        for(i=0;i<10;i++){
            tree.insert(new Integer(n=(int)(Math.random()*1000)));
            System.out.print(n+" ");
        }
        System.out.println("\nPre-order:");
        tree.print(1);
        System.out.println("\nIn-order:");
        tree.print(2);
        System.out.println("\nPost-order:");
        tree.print(3);
    }
}
```

You'll notice that the code above uses `java.lang.Integer` object, instead of our own `pInteger` one. That's because in JDK 1.2, an `Integer` is `implementing Comparable interface`, simplifying a whole lot of stuff. Other than that, the code works identically to the one shown earlier in the document. Just to be complete, however, lets include the output from the above program.

```
Numbers inserted:
103 82 165 295 397 828 847 545 766 384
Pre-order:
103 82 165 295 397 384 828 545 766 847
In-order:
82 103 165 295 384 397 545 766 828 847
Post-order:
82 165 295 397 384 828 545 766 847 103
```

I guess that wraps it up. (finally!) Now, I'm gonna take a little break, and try to come up with some cool programs that use these data structures. The only way to actually learn what they are, and how to use them, is to see them in action. And some of these can be VERY useful in a wide variety of applications, ranging form databases, to graphics programming.

So, now that you know all the basics of everything about data structures, the remainder of this document will mostly concentrate on interesting approaches, and algorithms using these. (You'll be surprised how far a binary sort tree can go when it comes to creating virtual 3D worlds!)

Priority Vectors, etc.

We've learned about the structure of vectors, lists, queues, etc., but what if you need some order in the way data is arranged? You have several options to this; you can simply implement a sort method inside the data storage class, or you can make the class itself a priority one.

Priority classes are widely used in programs because they simplify a lot of things. For example, you don't have to worry about sorting data, the class itself does it for you. Imagine a simulation of a real world queue at a doctor's office, where people with life threatening injuries get priority over everyone else. The queue has to accept all people, and sort them according to the seriousness of their illness; with most serious ending up seeing the doctor first.

The priority concept can be applied anywhere, not just to queues. It can easily be extended to lists, and vectors. In this section, we will talk about creating a priority vector (since it's fairly simple). We will also cover some *"well known"* sorting algorithms.

How simple is it? The answer is: very! All we need is to extend the `java.util.Vector` class, add one insert function, and we are done. And here's the source:

```
import java.lang.Comparable;

public class pPriorityVector extends java.util.Vector{
    public void pAddElement(Comparable o){
        int i,j = size();
        for(i=0;i<j&&(((Comparable)(elementAt(i))).compareTo(o)<0);i++);
        insertElementAt(o,i);
    }
}
```

As you can see, it's VERY simple. We simply use this class the way we would use any other `java.util.Vector`, accept, when we use `pAddElement(Comparable)` method, we are inserting in accenting order. We could have just as well written a descending order one; I think you get the picture...

Lets test this class, and see if it really works.

```
import java.io.*;
import java.util.*;
import pPriorityVector;

class pPriorityVectorTest{
    public static void main(String[] args){
        pPriorityVector v = new pPriorityVector();
        System.out.print("starting...\nadding:");
        for(int i=0;i<10;i++){
            Integer j = new Integer((int)(Math.random()*100));
            v.pAddElement(j);
            System.out.print(" " + j);
        }
        System.out.print("\nprinting:");
        Enumeration enum = v.elements();
        while(enum.hasMoreElements())
            System.out.print(" "+(Integer)enum.nextElement());
        System.out.println("\nDone ;-)");
    }
}
```

The above test program simply inserts ten random `java.lang.Integer` objects, and then prints them out. If our class works, the output should produce a sorted list of number. Our prediction is correct, the output follows.

```
starting...
adding: 95 85 62 16 39 73 84 43 77 61
printing: 16 39 43 61 62 73 77 84 85 95
Done ;-)
```

As you can see, we are inserting numbers in an unsorted order, and get a sorted list when we finally print them out. The technique illustrated above sorts the numbers upon insertion, some algorithms do sorting when retrieving data.

Sorting...

Sorting, in general, means arranging something in some meaningful manner. There are TONS of ways of sorting things, and we will only look at the most popular ones. The one we won't cover here (but one you should know) is Quick Sort. You can find a really good Quick Sort example in the JDK's sample applets. We have also learned that we can sort using binary trees, thus, in this section, we won't cover that approach.

We will be using JDK 1.2's `java.lang.Comparable` interface in this section, thus, the examples will not work with earlier versions of the JDK. The sorting algorithms we'll look at in this section are: *Bubble Sort*, *Selection Sort*, *Insertion Sort*, *Heap Sort*, and lastly, *Merge Sort*. If you think that's not enough, you're welcome to find other algorithms from other sources. One source I'd recommend is the JDK's sources. The JDK has lots of code, and lots of algorithms that may well be very useful.

We won't exactly cover the algorithms; I will simply illustrate them, and you'll have to figure out how they work from the source. (By tracing them.) I don't have the time to go over each and every algorithm; I'm just taking my old C source, converting it to Java, and inserting it into this document ;-)

```
import java.io.*;
import java.util.*;
import java.lang.*;

public class pGeneralSorting{

    public static void BubbleSort(Comparable[] a){
        boolean switched = true;
        for(int i=0;i<a.length-1 && switched;i++){
            switched = false;
            for(int j=0;j<a.length-i-1;j++)
                if(a[j].compareTo(a[j+1]) > 0){
                    switched = true;
                    Comparable hold = a[j];
                    a[j] = a[j+1];
                    a[j+1] = hold;
                }
        }
    }

    public static void SelectionSort(Comparable[] a){
        for(int i = a.length-1;i>0;i--){
            Comparable large = a[0];
            int indx = 0;
            for(int j = 1;j <= i;j++)
                if(a[j].compareTo(large) > 0){
                    large = a[j];
                    indx = j;
                }
            a[indx] = a[i];
            a[i] = large;
        }
    }

    public static void InsertionSort(Comparable[] a){
        int i,j;
        Comparable e;
        for(i=1;i<a.length;i++){
            e = a[i];
            for(j=i-1;j>=0 && a[j].compareTo(e) > 0;j--)
                a[j+1] = a[j];
            a[j+1] = e;
        }
    }

    public static void HeapSort(Comparable[] a){
        int i,f,s;
        for(i=1;i<a.length;i++){
            Comparable e = a[i];
            s = i;
            f = (s-1)/2;
            while(s > 0 && a[f].compareTo(e) < 0){
                a[s] = a[f];
                s = f;
                f = (s-1)/2;
            }
            a[s] = e;
        }
        for(i=a.length-1;i>0;i--){
            Comparable value = a[i];
            a[i] = a[0];
            f = 0;
            if(i == 1)
                s = -1;
            else
                s = 1;
            if(i > 2 && a[2].compareTo(a[1]) > 0)
                s = 2;
            while(s >= 0 && value.compareTo(a[s]) < 0){
                a[f] = a[s];
                f = s;
                s = 2*f+1;
                if(s+1 <= i-1 && a[s].compareTo(a[s+1]) < 0)
                    s = s+1;
                if(s > i-1)
                    s = -1;
            }
            a[f] = value;
        }
    }

    public static void MergeSort(Comparable[] a){
        Comparable[] aux = new Comparable[a.length];
        int i,j,k,l1,l2,size,u1,u2;
        size = 1;
        while(size < a.length){
            l1 = k = 0;
            while((l1 + size) < a.length){
                l2 = l1 + size;
                u1 = l2 - 1;
                u2 = (l2+size-1 < a.length) ?
                    l2 + size-1:a.length-1;
                for(i=l1,j=l2;i <= u1 && j <= u2;k++)
                    if(a[i].compareTo(a[j]) <= 0)
                        aux[k] = a[i++];
                    else
                        aux[k] = a[j++];
                for(;i <= u1;k++)
                    aux[k] = a[i++];
                for(;j <= u2;k++)
                    aux[k] = a[j++];
                l1 = u2 + 1;
            }
            for(i=l1;k < a.length;i++)
                aux[k++] = a[i];
            for(i=0;i < a.length;i++)
                a[i] = aux[i];
            size *= 2;
        }
    }

    public static void main(String[] args){
        Integer[] a = new Integer[10];
        System.out.print("starting...\nadding:");
        for(int i=0;i<a.length;i++){
            a[i] = new Integer((int)(Math.random()*100));
            System.out.print(" " + a[i]);
        }
        BubbleSort(a);
        System.out.print("\nprinting:");
        for(int i=0;i<a.length;i++){
            System.out.print(" " + a[i]);
        }
        System.out.println("\nDone ;-)");
    }
}
```

The above program both illustrates the algorithms, and tests them. Some of these may look fairly complicated; usually, the more complicated a sorting algorithm is, the faster and/or more efficient it is. That's the reason I'm not covering Quick Sort; I'm too lazy to convert that huge thing! Some sample output from the above program follows:

```
starting...
adding: 22 63 33 19 82 59 70 58 98 74
printing: 19 22 33 58 59 63 70 74 82 98
Done ;-)
```

I think you get the idea bout sorting. You can always optimize the above sorting methods, use native types, etc. You can also use these in derived classes from `java.util.Vector`, using the `java.util.Vector` data directly. And just when you though we were done with sorting, here we go again...

Sorting JDK 1.2 Style...

Wasn't the last section scary and a bit confusing? If you said *"yes,"* then you're not alone. Java implementers also think so; that's why a lot of the sorting thingies are already built in! JDK 1.2 introduced the `Collection` interface; using it, we can do all kinds of data structures effects like sorting and searching. For example, to sort a `Vector`, all we need to do is call a `sort()` method of the `java.util.Collections` class.

```
import java.io.*;
import java.util.*;

public class pJDKVectorSortingUp{
    public static void main(String[] args){
        Vector v = new Vector();
        System.out.print("starting...\nadding:");
        for(int i=0;i<10;i++){
            Integer j = new Integer((int)(Math.random()*100));
            v.addElement(j);
            System.out.print(" " + j);
        }
        Collections.sort(v);
        System.out.print("\nprinting:");
        Enumeration enum = v.elements();
        while(enum.hasMoreElements())
            System.out.print(" "+(Integer)enum.nextElement());
        System.out.println("\nDone ;-)");
    }
}
```

See how simple it is? The output follows...

```
starting...
adding: 91 37 16 53 11 15 89 44 90 58
printing: 11 15 16 37 44 53 58 89 90 91
Done ;-)
```

All this is nice and useful, but what if you need descending order, instead of the *"default"* accenting one? Guess what?, the implementers of the language have though about that as well! We have a `java.util.Comparator` interface. With this `Comparator`, we can specify our own compare function, making it possible to sort in descending order (or any other way we want... i.e.: sorting absolute values, etc.).

```
import java.io.*;
import java.util.*;

public class pJDKVectorSortingDown implements Comparator{

    public int compare(Object o1,Object o2){
        return -((Comparable)o1).compareTo(o2);
    }

    public static void main(String[] args){
        Vector v = new Vector();
        System.out.print("starting...\nadding:");
        for(int i=0;i<10;i++){
            Integer j = new Integer((int)(Math.random()*100));
            v.addElement(j);
            System.out.print(" " + j);
        }
        Collections.sort(v,new pJDKVectorSortingDown());
        System.out.print("\nprinting:");
        Enumeration enum = v.elements();
        while(enum.hasMoreElements())
            System.out.print(" "+(Integer)enum.nextElement());
        System.out.println("\nDone ;-)");
    }
}
```

As you can see, this time, we're sending a `Comparator` object to the `Collections.sort()` method. This technique is really cool and useful. The output from the above program follows:

```
starting...
adding: 9 96 58 64 13 99 91 55 51 95
printing: 99 96 95 91 64 58 55 51 13 9
Done ;-)
```

I suggest you go over all those JDK classes and their methods. There is a LOT of useful stuff there. Now, say good bye to sorting for a while; we're moving into something totally different; NOT!

Sorting using Quicksort...

Sorting using Quicksort is something that I wasn't planning to cover in this document due to the fact that the algorithm solely depends on the data it receives. If the data has certain properties, quicksort is one of the fastest, if not, quicksort can be excruciatingly slow. By now, you probably already figured out that *bubble sort* (*covered earlier*) is pretty slow; put it another way: *forget about using bubble sort for anything important*.

Bubble sort tends to be `O(n^2)`; in other words, pretty slow. Now, quicksort can perform quite fast, on average about `O(n log n)`, but it's worst case, is a humiliating `O(n^2)`. For quicksort, the worst case is usually when the data is already sorted. There are many algorithms to make this *"less likely to occur,"* but it does happen. (the `O notation` is paraphrased *"on the order of"*)

If you need a *no-worst-case* fast sorting algorithm, then by all means, use *heap sort* (*covered earlier*). In my opinion, heap sort is more elegant than any other sort <it is *in place* `O(n log n)` *sort*>. However, on average, it does tend to perform twice as slow as quicksort.

UPDATE: I've recently attended a conference at the [insert fancy name here] Science Society, and heard a nice talk by a professor from Princeton. The talk was totally dedicated to Quicksort. Apparently, if you modify the logic a bit, you can make Quicksort optimal! This is a very grand discovery! (unfortunately, I didn't write down the name of anybody; not even the place!) The idea (among other things) involves moving duplicate elements to one end of the list, and at the end of the run, move all of them to the middle, then sort the left and right sides of the list. Since now Quicksort performs well with lots of duplicate values (really well actually ;-), you can create a radix-like sort that uses embedded Quicksort to quickly (very quickly) sort things. There's also been a Dr.Dobbs article (written by that same professor ;-) on this same idea, and I'm sorry for not having any more references than I should. I still think this is something that deserved to be mentioned.

We will start out by writing a general (*simplest*) implementation of quicksort. After we thoroughly understand the algorithm we will re-write it implementing all kinds of tricks to make it faster.

Quicksort is naturally recursive. We partition the array into two sub-arrays, and then re-start the algorithm on each of these sub-arrays. The partition procedure involves choosing some object (usually, already in the array); If some other object is greater than the chosen object, it is added to one of the sub-arrays, if it's less than the chosen object, it's added to another sub-array. Thus, the entire array is partitioned into two sub-arrays, with one sub-array having everything that's larger than the chosen object, and the other sub-array having everything that's smaller.

The algorithm is fairly simple, and it's not hard to implement. The tough part is making it fast. The implementation that follows is recursive and is not optimized, which makes this function inherently slow (most sorting functions try to avoid recursion and try to be as fast as possible). If you need raw speed, you should consider writing a native version of this algorithm.

```
public class pSimpleQuicksort{

    public static void qsort(Comparable[] c,int start,int end){
        if(end <= start) return;
        Comparable comp = c[start];
        int i = start,j = end + 1;
        for(;;){
            do i++; while(i<end && c[i].compareTo(comp)<0);
            do j--; while(j>start && c[j].compareTo(comp)>0);
            if(j <= i)   break;
            Comparable tmp = c[i];
            c[i] = c[j];
            c[j] = tmp;
        }
        c[start] = c[j];
        c[j] = comp;
        qsort(c,start,j-1);
        qsort(c,j+1,end);
    }

    public static void qsort(Comparable[] c){
        qsort(c,0,c.length-1);
    }

    public static void main(String[] args){
        int i;
        Integer[] arr = new Integer[20];
        System.out.println("inserting: ");
        for(i=0;i<arr.length;i++){
            arr[i] = new Integer((int)(Math.random()*99));
            System.out.print(arr[i]+" ");
        }
        pSimpleQuicksort.qsort(arr);
        System.out.println("\nsorted: ");
        for(i=0;i<arr.length;i++)
            System.out.print(arr[i]+" ");
        System.out.println("\nDone ;-)");
    }
}
```

As you can see, the `qsort()` functions itself is fairly short. One of them is just a wrapper for the other one. The idea is that `qsort()` receives the array, and then the `start`, and `end` pointer between which you want everything sorted. So, the starting call starts at array position `zero`, and ends at the last valid position in the array. Some sample output follows:

```
inserting:
58 52 82 27 23 67 37 63 68 18 95 41 87 6 53 85 65 30 10 3
sorted:
3 6 10 18 23 27 30 37 41 52 53 58 63 65 67 68 82 85 87 95
Done ;-)
```

The sorts starts out by first checking to see if the `end` pointer is less then or equal to the `start` pointer. It if is less, then there is nothing to sort, and we return. If they are equal, we have only one element to sort, and an element by itself is always already sorted, so we return.

If we did not return, we make a pick. In our example, we simply choose the first element in the array to use as our partition element (some call it *pivot* element). To some people, this is the most critical point of the sort, since depending on what element you choose, you'll get either good performance, or bad. A lot of the times, instead of choosing the first, people choose the last, or take a median of the first, last and middle elements. Some even have a random number generator to pick a random element. However, all these techniques are useless against random data, in which case, all those tricky approaches can actually prove to worsen results. Then again, most of the times, they do work quite nicely... (it really depends on the type of data you are working with)

After we have picked the element, we setup `i` and `j`, and fall into the a `for` loop. Inside that loop, we have two inner loops. Inside the first inner loop, we scan the array looking for the first element which is larger than our picked element, moving from left to right of the array. Inside the second inner loop, we scan to find an element smaller than the picked, moving from right to left in the array. Once we've found them (fallen out of both loops), we check to see if the pointers haven't crossed (since if they did, we are done). We then swap the elements, and continue on to the next iteration of the loop.

You should notice that in all these loops, we are doing one simple thing. We are making sure that only one element gets to it's correct position. We deal with one element at a time. After we find that correct position of our chosen element, all we need to do is sort the elements on it's right, and left sides, and we're done. You should also notice that the algorithm above is lacking in optimization. The inner loops, where most of the time takes place are not as optimized as they should be. However, for the time being, try to understand the algorithm; and we will get back to optimizing a bit later.

When we fall out of the outer loop, we put our chosen element into it's correct position, calculate the upper and lower bounds of the left and right arrays (from that chosen elements), and recursively call the method on these new computed arrays.

That's basically it for the *general* algorithm. In the next section, you'll be amazed at how much faster we can make this work. Basically, in the next section, we will implement a sort function to use everywhere (it will be faster than most other approaches).

Optimizing Quicksort...

Optimizing Quicksort written in Java is not such a great task. Whatever tricks we use, we will still be restrained by the speed of the Java Virtual Machine (JVM). In this section, we will talk about algorithmic improvements, rather than quirky tricks.

The first thing what we should do is look at the above code of the un-optimized sort, and see what can be improved. One thing that should be obvious is that it's way too much work if the array is very small. There are a LOT simpler sorts available for small arrays. For example, simple *insertion sort* has almost no overhead, and on small arrays, is actually faster than *quicksort*! This can be fixed rather easily, by including an `if` statement to see if the size of the array is smaller than some particular value, and doing a simple *insertion sort* if it is. This threshold value can only be determined from doing actual experiments with sample data. However, usually any value less than `15` is pretty good; and that is why we will choose `7` for our upcoming example.

What makes quicksort so fast is the simplicity of it's inner loops. In our previous example, we were doing extra work in those inner loops! We were checking for the array bounds; we should have made some swaps beforehand to make sure that the inner loops are as simple as possible. Basically, the idea is to make sure that the first element in the array is less than or equal to the second, and that the second is less than or equal to the last. Knowing that, we can remove those array bounds checking from the inner loops (in some cases speeding up the algorithm by about two times).

Another thing that should be obvious is recursion. Because sort methods are usually very performance hungry, we would like to remove as much function calls as possible. This includes getting rid of recursion. The way we can get rid of recursion is using a stack to store the upper and lower bounds of arrays to be sorted, and using a loop to control the sort.

A question automatically arises: *How big should that stack be?* A simple answer is: *It should be big enough.* (*you'll be surprised how many books avoid this topic!*) We cannot just use `java.util.Vector` object for it's dynamic growing ability, since that would be way too much overhead for simple stack operations that should be as quick as possible. Now, what would be a *big enough* stack? Well, a formula below calculates the size:

```
size = 2 * ln N / ln 2
```

Where `ln` is natural logarithm, and `N` is the number of elements in the array. Which all translates to:

```
size = (int)(Math.ceil(2.0*Math.log(array.length)/Math.log(2.0));
```

Believe it or not, but it's actually not worth using this equation. If you think about it, you will notice that the size of the stack grows VERY slowly. For example, if the `array.length` is `0xFFFFFFFF` in length (highest `32 bit` integer value), then size will be `64`! We will make it `128` just in case we will ever need it for `64 bit` integers. (If you don't like magic values inside your program, then by all means, use the equation.)

Having gotten to this point, we are almost ready to implement our optimized version of quicksort. I say *almost* because it is still not optimized to it's fullest. If we were using native types instead of `Comparable` objects, then the whole thing would be faster. If we implementing it as native code, it would be even faster. Basically, most other speed-ups are left up to these system or program specific quirky optimizations. And now, here's our optimized version:

```
import java.lang.*;
import java.io.*;

public class pQuicksort{

    public static void sort(Comparable[] c){
        int i,j,left = 0,right = c.length - 1,stack_pointer = -1;
        int[] stack = new int[128];
        Comparable swap,temp;
        while(true){
            if(right - left <= 7){
                for(j=left+1;j<=right;j++){
                    swap = c[j];
                    i = j-1;
                    while(i>=left && c[i].compareTo(swap) > 0)
                        c[i+1] = c[i--];
                    c[i+1] = swap;
                }
                if(stack_pointer == -1)
                    break;
                right = stack[stack_pointer--];
                left = stack[stack_pointer--];
            }else{
                int median = (left + right) >> 1;
                i = left + 1;
                j = right;
                swap = c[median]; c[median] = c[i]; c[i] = swap;
                /* make sure: c[left] <= c[left+1] <= c[right] */
                if(c[left].compareTo(c[right]) > 0){
                    swap = c[left]; c[left] = c[right]; c[right] = swap;
                }if(c[i].compareTo(c[right]) > 0){
                    swap = c[i]; c[i] = c[right]; c[right] = swap;
                }if(c[left].compareTo(c[i]) > 0){
                    swap = c[left]; c[left] = c[i]; c[i] = swap;
                }
                temp = c[i];
                while(true){
                    do i++; while(c[i].compareTo(temp) < 0);
                    do j--; while(c[j].compareTo(temp) > 0);
                    if(j < i)
                        break;
                    swap = c[i]; c[i] = c[j]; c[j] = swap;
                }
                c[left + 1] = c[j];
                c[j] = temp;
                if(right-i+1 >= j-left){
                    stack[++stack_pointer] = i;
                    stack[++stack_pointer] = right;
                    right = j-1;
                }else{
                    stack[++stack_pointer] = left;
                    stack[++stack_pointer] = j-1;
                    left = i;
                }
            }
        }
    }

    public static void main(String[] args){
        int i;
        Integer[] arr = new Integer[20];
        System.out.println("inserting: ");
        for(i=0;i<arr.length;i++){
            arr[i] = new Integer((int)(Math.random()*99));
            System.out.print(arr[i]+" ");
        }
        pQuicksort.sort(arr);
        System.out.println("\nsorted: ");
        for(i=0;i<arr.length;i++)
            System.out.print(arr[i]+" ");
        System.out.println("\nDone ;-)");
    }
}
```

The output closely follows:

```
inserting:
17 52 88 79 91 41 31 57 0 29 87 66 94 22 19 30 76 85 61 16
sorted:
0 16 17 19 22 29 30 31 41 52 57 61 66 76 79 85 87 88 91 94
Done ;-)
```

This new sort function is a bit larger than most other sorts. It starts out by setting `left` and `right` pointers, and the `stack`. Stack allocation could be moved outside the function, but making it static is not a good choice since that introduces all kinds of multithreading problems.

We then fall into an infinite loop in which we have an `if()` and an `else` statement. The `if()` statement finds out if we should do a simple *insertion sort*, `else`, it will do quicksort. I will not explain *insertion sort* here (since you should already be familiar with it). So, after *insertion sort*, we see if the `stack` is not empty, if it is, we `break` out of the infinite loop, and `return` from the function. If we don't `return`, we pop the `stack`, set new `left` and `right` pointers (from the stack), and continue on with the next iteration of the loop.

Now, if threshold value passed, and we ended up doing quicksort we first find the `median`. This `median` is used here to make the case with bad performance *less likely to occur*. It is useless against random data, but does help if the data is in somewhat sorted order.

We swap this `median` with the second value in the array, and make sure that the first value is less than or equal than the second, and the second is less than or equal than the last. After that, we pick our partition element (or pivot), and fall into an infinite loop of finding that pivot's correct place.

Notice that the most inner loops are fairly simple. Only one increment or decrement operation, and a compare. The compare could be improved quite a bit by using native types; instead of calling a function. The rest of this part of the sort is almost exactly like in the un-optimized version.

After we find the correct position of the pivot variable, we are ready to continue the sort on the right sub-array, and on the left sub-array. What we do next is check to see which one of these sub-arrays is bigger. We then insert the bigger sub-array bounds on to the stack, and setup new `left` and `right` pointers so that the smaller sub-array gets processed next.

I guess that's it for quicksort. If you still don't understand it, I doubt any other reference would be of any help. Basically go through the algorithm; tracing it a few times, and eventually, it will seem like second nature to you. The above function is good enough for most purposes. I've sorted HUGE arrays (*~100k*) of random data, and it performed quite well.

Radix Sort...

Radix sort is one of the nastiest sorts that I know. This sort can be quite fast when used in appropriate context, however, to me, it seems that the context is never appropriate for radix sort.

The idea behind the sort is that we sort numbers according to their base (sort of). For example, lets say we had a number 1024, and we break it down into it's basic components. The 1 is in the thousands, the 0 is in the hundreds, the 2 is in the tens, and 4 is in some units. Anyway, given two numbers, we can sort them according to these bases (i.e.: 100 is greater than 10 because first one has more 'hundreds').

In our example, we will start by sorting numbers according to their least significant bits, and then move onto more significant ones, until we reach the end (at which point, the array will be sorted). This can work the other way as well, and in some cases, it's even more preferable to do it 'backwards'.

Sort consists of several passes though the data, with each pass, making it more and more sorted. In our example, we won't be overly concerned with the actual decimal digits; we will be using base 256! The workings of the sort are shown below:

Numbers to sort: `23 45 21 56 94 75 52` we create `ten` queues (one queue for each digit): `queue[0 .. 9]`

We start going thought the passes of sorting it, starting with least significant digits.

```
queue[0] = { }
queue[1] = {21}
queue[2] = {52}
queue[3] = {23}
queue[4] = {94}
queue[5] = {45,75}
queue[6] = {56}
queue[7] = { }
queue[8] = { }
queue[9] = { }
```

Notice that the queue number corresponds to the least significant digit (i.e.: queue `1` holding `21`, and queue `6` holding `56`). We copy this queue into our array (top to bottom, left to right) Now, numbers to be sorted: `21 52 23 94 45 75 56` We now continue with another pass:

```
queue[0] = { }
queue[1] = { }
queue[2] = {21,23}
queue[3] = { }
queue[4] = {45}
queue[5] = {52,56}
queue[6] = { }
queue[7] = {75}
queue[8] = { }
queue[9] = {94}
```

Notice that the queue number corresponds to the most significant digit (i.e.: queue `4` holding `45`, and queue `7` holding `75`). We copy this queue into our array (top to bottom, left to right) and the numbers are sorted: `21 23 45 52 56 75 94`

Hmm... Isn't that interesting? Anyway, you're probably wondering how it will all work within a program (or more precisely, how much book keeping we will have to do to make it work). Well, we won't be working with

10 queues in our little program, we'll be working with 256 queues! We won't just have least significant and most significant bits, we'll have a whole range (from 0xFF to 0xFF000000).

Now, using arrays to represent Queues is definitely out of the question (most of the times) since that would be wasting tons of space (think about it if it's not obvious). Using dynamic allocation is also out of the question, since that would be extremely slow (since we will be releasing and allocating nodes throughout the entire sort). This leaves us with little choice but to use node pools. The node pools we will be working with will be really slim, without much code to them. All we need are nodes with two integers (one for the data, the other for the link to next node). We will represent the entire node pool as a two dimensional array, where the height of the array is the number of elements to sort, and the width is two.

Anyway, enough talk, here's the program:

```
import java.lang.*;
import java.io.*;

public class RadixSort{

    public static void radixSort(int[] arr){
        if(arr.length == 0)
            return;
        int[][] np = new int[arr.length][2];
        int[] q = new int[0x100];
        int i,j,k,l,f = 0;
        for(k=0;k<4;k++){
            for(i=0;i<(np.length-1);i++)
                np[i][1] = i+1;
            np[i][1] = -1;
            for(i=0;i<q.length;i++)
                q[i] = -1;
            for(f=i=0;i<arr.length;i++){
                j = ((0xFF<<(k<<3))&arr[i])>>(k<<3);
                if(q[j] == -1)
                    l = q[j] = f;
                else{
                    l = q[j];
                    while(np[l][1] != -1)
                        l = np[l][1];
                    np[l][1] = f;
                    l = np[l][1];
                }
                f = np[f][1];
                np[l][0] = arr[i];
                np[l][1] = -1;
            }
            for(l=q[i=j=0];i<0x100;i++)
                for(l=q[i];l!=-1;l=np[l][1])
                        arr[j++] = np[l][0];
        }
    }

    public static void main(String[] args){
        int i;
        int[] arr = new int[15];
        System.out.print("original: ");
        for(i=0;i<arr.length;i++){
            arr[i] = (int)(Math.random() * 1024);
            System.out.print(arr[i] + " ");
        }
        radixSort(arr);
        System.out.print("\nsorted: ");
        for(i=0;i<arr.length;i++)
            System.out.print(arr[i] + " ");
        System.out.println("\nDone ;-)");
    }
}
```

Yeah, not exactly the most friendliest code I've written. A few things to mention about the code. One: it's NOT fast (far from it!). Two: It only works with positive integers. Sample output:

```
original: 1023 1007 583 154 518 671 83 98 213 564 572 989 241 150 64
sorted: 64 83 98 150 154 213 241 518 564 572 583 671 989 1007 1023
Done ;-)
```

I don't like this sort much, so, I won't talk much about it. However, a few words before we go on. This sort can be rather efficient in conjunction with other sorts. For example, you can use this sort to pre-sort the most significant bits, and then use insertion sort for the rest. Another very crucial thing to do to speed it up is to make the node pool and queues statically allocated (don't allocate them every time you call the function). And if you're sorting small numbers (i.e.: `1-256`), you can make it `4` times as fast by simply removing the outer loop.

This sort is not very expandable. You'd have problems making it work with anything other than numbers. Even adding negative number support isn't very straight forward. Anyway, I'm going to go and think up more reasons not to use this sort... and use something like quick-sort instead.

Improving Radix Sort...

A day after I wrote the above section, I realized I made a blunder. Not only does that suck, but it's slow and has tons of useless code in it as well! For example, I realized that I didn't really need to care about the node pool that much. Even if I didn't setup the node pool, things would still work, since node pool gets re-initialized every time though the loop (look at the code in the above article). I later realized that I don't even need a pointer to the next free node on the node pool! Since we are allocating a node per number, and we are doing that in a loop, we could just use the loop counter to point to our free node!

After I got the code down to a mere 14 lines, I still wasn't satisfied with the inner loop that searched for the last node on the queue. So, created another array, just to hold the backs of queues. That eliminated lots of useless operations, and increased speed quite a bit, especially for large arrays.

After all that, I've moved the static arrays out of the function (just for the fun of it), since I didn't want to allocate exactly the same arrays every single time.

With each and every improvement, I was getting code which sucked less and less. Later, I just reset the 32 bit size to 16 bit integer size (to make it twice as fast, since the test program only throws stuff as large as 1024). I didn't go as far as unrolling the loops, but for a performance needy job, that could provide a few extra cycles.

Anyway, I won't bore you any longer, and just give you the new and improved code (which should have been the first one you've saw, but... I was lazy, and didn't really feel like putting in lots of effort into radix sort).

```
import java.lang.*;
import java.io.*;

public class pRadixSort{

    private static int q[],ql[];
    static{
        q = new int[256];
        ql = new int[256];
        for(int i=0;i<q.length;q[i++] = -1);
    }

    public static void radixSort(int[] arr){
        int i,j,k,l,np[][] = new int[arr.length][2];
        for(k=0;k<2;k++){
            for(i=0;i<arr.length;np[i][0]=arr[i],np[i++][1]=-1)
```

```
                if(q[j=((255<<(k<<3))&arr[i])>>(k<<3)]==-1)
                    ql[j] = q[j] = i;
                else
                    ql[j] = np[ql[j]][1] = i;
            for(l=q[i=j=0];i<q.length;q[i++]=-1)
                for(l=q[i];l!=-1;l=np[l][1])
                    arr[j++] = np[l][0];
        }
    }

    public static void main(String[] args){
        int i;
        int[] arr = new int[15];
        System.out.print("original: ");
        for(i=0;i<arr.length;i++){
            arr[i] = (int)(Math.random() * 1024);
            System.out.print(arr[i] + " ");
        }
        radixSort(arr);
        System.out.print("\nsorted: ");
        for(i=0;i<arr.length;i++)
            System.out.print(arr[i] + " ");
        System.out.println("\nDone ;-)");
    }
}
```

This code is so much better than the previous one, that I'd consider it as fast as quicksort (in some cases). Counting the looping, we can get an approximate idea of how fast is it... First, we have a loop which repeats 2 times (for 16 bit numbers), inside of it, we have 2 loops, both of which repeat on the order of N. So, on average, I'd say this code should be about 2*2*N, which is 4N... (not bad... <if I did it correctly>), imagine, if you have 1000 elements in the array, the sort will only go through about 4000 loops (for bubble sort, it would have to loop for 1,000,000 times!). True, these loops are a bit complicated compared to the simple bubble sort loop, but the magnitude is still huge.

Anyway, I should really get some sleep right about now (didn't get much sleep in the past few days due to this little 'thought' which kept me awake). Never settle for code which you truly believe sucks (go do something about it!).

Reading and Writing Trees (Serialization)...

Reading and writing binary trees to/from files is not as simple as reading and writing arrays. They're not linear, and require some algorithm to accomplish the task.

One simple approach would be to convert the tree to a node-pool one, and simply save the node-pool, and the first node (or something along those lines). That's the approach taken by lots of programs in saving trees. Actually, most programs implement the tree as a node-pool from the start, so, the whole thing is much simpler.

But what do you do when you're not using node-pools, but would still like to save the tree (and it's structure)? You can use a technique known as serialization.

There are many ways of serializing a binary tree, and incidentally, I've lost the *"formal algorithm,"* so, I'll re-derive it again, maybe in a little bit different form. The basic idea is to first write the size of data inside the node, and then the data itself. If there is no data you write zero, and return (without writing the data). If you've written any data, you then recursively write the left and right children of that node. There can be many variations of this; the most obvious is switching the order of left and right child.

The reading process has to know the way in which the tree was written. It has to first read the size of the data, if the size is zero, it sets the pointer to null, and returns. If the size is not zero, it allocates a new node in the tree, and reads the data into that node. Then, it recursively, reads the left and then the right children for that node.

This simple algorithm is very effective, and can be used to read and write data of different sizes (i.e.: character strings). The explanation of it may not be too clear at first; that's why I've written some source to clear up the confusion. The source that follows creates a binary tree holding strings, saves the tree to a file, and later reads that file.

```
import java.lang.*;
import java.io.*;

public class pBinTreeStringWR{
    public pBinTreeStringWR left,right;
    public String data;

    public static String[] strings = {"one","two",
        "three","four","five","six","seven","eight",
        "nine","ten","zero","computer","mouse","screen",
        "laptop","book","decimal","binary","quake"};

    public static pBinTreeStringWR tree_AddString(
        pBinTreeStringWR r,String s){
        if(r == null){
            r = new pBinTreeStringWR();
            r.left = r.right = null;
            r.data = s;
        }else if(r.data.compareTo(s) < 0)
            r.right = tree_AddString(r.right,s);
        else
            r.left = tree_AddString(r.left,s);
        return r;
    }

    public static void tree_InOrderPrint(
        pBinTreeStringWR r){
        if(r != null){
            tree_InOrderPrint(r.left);
            System.out.print(" "+r.data);
            tree_InOrderPrint(r.right);
        }
    }

    public static void tree_FileWrite(
        pBinTreeStringWR r,
        DataOutputStream output) throws IOException{
        if(r != null){
            byte[] tmp = r.data.getBytes();
            output.writeInt(tmp.length);
            output.write(tmp);
            tree_FileWrite(r.left,output);
            tree_FileWrite(r.right,output);
        }else
            output.writeInt(0);
    }

    public static pBinTreeStringWR tree_FileRead(
        pBinTreeStringWR r,
        DataInputStream input) throws IOException{
        int n = input.readInt();
        if(n != 0){
            byte[] tmp = new byte[n];
            input.read(tmp);
            r = new pBinTreeStringWR();
            r.data = new String(tmp);
            r.left = tree_FileRead(r.left,input);
            r.right = tree_FileRead(r.right,input);
        }else
            r = null;
        return r;
    }

    public static boolean tree_Compare(
        pBinTreeStringWR a,pBinTreeStringWR b){
        if(a != null && b != null){
            return a.data.compareTo(b.data) == 0 &&
                tree_Compare(a.left,b.left) &&
                tree_Compare(a.right,b.right);
        }else if(a == null && b == null)
            return true;
        else
            return false;
    }

    public static void main(String[] args){
        File file = new File("pBinTreeStringWR.dat");
        pBinTreeStringWR read_tree = null,tree = null;
        System.out.print("inserting: ");

        for(int i=0;i<strings.length;i++){
            String s = new String(strings[i]);
            System.out.print(" "+s);
            tree = tree_AddString(tree,s);
        }

        System.out.print("\ntree: ");
        tree_InOrderPrint(tree);

        System.out.println("\nwriting to "+file);
        try{
            tree_FileWrite(tree,
                new DataOutputStream(
                new FileOutputStream(file)));
        }catch(IOException e){
            System.out.println(e);
        }

        System.out.println("reading from "+file);
        try{
            read_tree = tree_FileRead(read_tree,
                new DataInputStream(
                new FileInputStream(file)));
        }catch(IOException e){
            System.out.println(e);
        }

        System.out.print("read tree: ");
        tree_InOrderPrint(read_tree);

        if(tree_Compare(tree,read_tree))
            System.out.println(
                "\nThe two trees are identical.");
        else
            System.out.println(
                "\nThe two trees are different.");
        System.out.println("done ;-)");
    }
}
```

The program both illustrates the algorithm and tests it. `pBinTreeStringWR` class is itself a node. These nodes are manipulated using `static` methods. I did it this way to reduce the number of needed classes (combining the program class with the node class).

In the program, we start out by creating a `File` object with the file we're about to write and read. We then declare two trees, one named `tree`, the other named `read_tree`. We then loop to add `static` strings to the `tree`. The add function adds them in a binary search tree fashion; it is using the `Comparable` interface to do the comparisons.

Then we open a file, create a `DataOutputStream` object to that file, and write the tree to that file using `tree_FileWrite()` function. This `tree_FileWrite()` closely matches the algorithm described earlier. If the node is not `null`, it writes the `length` of the `String`, and then the `String` itself (converted to `bytes`). It then recursively writes the `left` and `right` children of that node. If the node is initially `null`, the function simply writes a zero for the `length`, and returns.

The program continues by reading that file it has just written. It stores the read tree in `read_tree` variable. The writing process, `tree_FileRead()` also closely matches the algorithm described earlier. It is basically the opposite of writing algorithm. We first read the `length` of the `String`, if it's zero, we set the node to `null`, and continue on. If the `length` is not zero, we read that number of `bytes`, and store them in that node's `data String`. We then continue recursively by reading the `left` and `right` children of that `tree`.

The program then calls a compare function, which compares the trees (including the tree structure). Tests indicate that the program is working correctly, since the test function always returns that the tree written, and the tree read are identical. In the middle of all this, there is an in-order print function, but I don't think I need to describe it since I assume you know the basics of trees. (Basically, go over the source and you'll understand the whole thing.) The output from the program follows.

```
inserting:  one two three four five six seven eight
nine ten zero computer mouse screen laptop book decimal binary quake

tree:  binary book computer decimal eight five four laptop mouse
nine one quake screen seven six ten three two zero

writing to pBinTreeStringWR.dat
reading from pBinTreeStringWR.dat

read tree:  binary book computer decimal eight five four laptop
mouse nine one quake screen seven six ten three two zero

The two trees are identical.
done ;-)
```

Another useful trick that you can use when you are not reading or writing variable length data like strings is not to write the `length`, but simply some marker. For example, lets say your tree holds integers, and you know all integers are `32` bits. Thus, your `length` parameter can simply be a `boolean` value of whether there exists a next node or not. The next example illustrates this by storing a `boolean` value instead of the `length` parameter.

```
import java.lang.*;
import java.io.*;

public class pBinTreeIntegerWR{
    public pBinTreeIntegerWR left,right;
    public Integer data;

    public static pBinTreeIntegerWR tree_AddNumber(
        pBinTreeIntegerWR r,Integer n){
        if(r == null){
            r = new pBinTreeIntegerWR();
            r.left = r.right = null;
            r.data = n;
        }else if(r.data.compareTo(n) < 0)
            r.right = tree_AddNumber(r.right,n);
        else
            r.left = tree_AddNumber(r.left,n);
        return r;
    }
    public static void tree_InOrderPrint(
        pBinTreeIntegerWR r){
        if(r != null){
            tree_InOrderPrint(r.left);
            System.out.print(" "+r.data);
            tree_InOrderPrint(r.right);
        }
    }

    public static void tree_FileWrite(
        pBinTreeIntegerWR r,
        DataOutputStream output) throws IOException{
        if(r != null){
            output.writeBoolean(true);
            output.writeInt(r.data.intValue());
            tree_FileWrite(r.left,output);
            tree_FileWrite(r.right,output);
        }else
            output.writeBoolean(false);
    }

    public static pBinTreeIntegerWR tree_FileRead(
        pBinTreeIntegerWR r,
        DataInputStream input) throws IOException{
        if(input.readBoolean()){
            r = new pBinTreeIntegerWR();
            r.data = new Integer(input.readInt());
            r.left = tree_FileRead(r.left,input);
            r.right = tree_FileRead(r.right,input);
        }else
            r = null;
        return r;
    }

    public static boolean tree_Compare(
        pBinTreeIntegerWR a,pBinTreeIntegerWR b){
        if(a != null && b != null){
            return a.data.compareTo(b.data) == 0 &&
                tree_Compare(a.left,b.left) &&
                tree_Compare(a.right,b.right);
        }else if(a == null && b == null)
            return true;
        else
            return false;
    }

    public static void main(String[] args){
        File file = new File("pBinTreeIntegerWR.dat");
        pBinTreeIntegerWR read_tree = null,tree = null;
        System.out.print("inserting: ");
        for(int i=0;i<10;i++){
            Integer n = new Integer((int)(Math.random()*100));
            System.out.print(" "+n);
            tree = tree_AddNumber(tree,n);
        }
        System.out.print("\ntree: ");
        tree_InOrderPrint(tree);
        System.out.println("\nwriting to "+file);
        try{
            tree_FileWrite(tree,
                new DataOutputStream(
                new FileOutputStream(file)));
        }catch(IOException e){
            System.out.println(e);
        }

        System.out.println("reading from "+file);
        try{
            read_tree = tree_FileRead(read_tree,
                new DataInputStream(
                new FileInputStream(file)));
        }catch(IOException e){
            System.out.println(e);
        }

        System.out.print("read tree: ");
        tree_InOrderPrint(read_tree);

        if(tree_Compare(tree,read_tree))
            System.out.println(
                "\nThe two trees are identical.");
        else
            System.out.println(
                "\nThe two trees are different.");
        System.out.println("done ;-)");
    }
}
```

The program above is almost identical to the one before, except it stores `java.lang.Integer` objects instead of `java.lang.String` objects. Since both of these implement `java.lang.Comparable interface`, the functions look pretty much the same. The only functions which look different are the file reading and writing. You should quickly notice that no `length` parameter is involved, but a simple `boolean` value telling us whether there exists a next node. Output from the program above follows:

```
inserting:  29 59 25 16 32 43 68 32 8 43
tree:  8 16 25 29 32 32 43 43 59 68
writing to pBinTreeIntegerWR.dat
reading from pBinTreeIntegerWR.dat
read tree:  8 16 25 29 32 32 43 43 59 68
The two trees are identical.
done ;-)
```

The funny thing that I always save till the end is that in practice, you'll probably never use any of these approaches yourself. Java provides a fairly useful `java.io.Serializable` interface, which given a tree, will nicely store it into a file. Anyway, it does pay to know how these kinds of things are done...

Deleting items from a Binary Search Tree...

Every database system must provide for deletion of items. A Binary Search Tree is a very good option for implementing a database (fast searches, sorts, inserts, etc.). One problem that stands in the way though is that it's not very straight forward to delete an item from a database represented by a binary tree. More precisely, the problem is in maintaining the tree structure while the deletion process.

Deleting items from a Binary Search Tree can be rather tricky. On one hand, you'd like to remove the object from the tree, on the other, you don't want the process to destroy the tree structure. There are many algorithms for this, and they all vary in degree of simplicity and efficiency.

The simplest one (which we will not cover here), is to simply have a `boolean` variable inside a node. That `boolean` variable tells us if that node is valid or not. Whenever we want to delete that node, we simply set that valid variable to `false`; making all the traversal functions simply skip that node. The actual removal can take place when the tree is rebuilt. This may seem like a waste of space, but most of the time, it's not (in today's world, several extra bytes inside the tree structure don't make much of a difference).

The above can actually be used in conjunction with a more complicated approach. For example, lets take a regular company database. Throughout the day, there are thousands of transactions, deletions, insertions, etc., all this is handled by the algorithm described above (a `boolean` valid variable). At the end of the day (night), when the system becomes free, the program does a routine clean-up of the tree from these *"deleted"* items. Since setting or unsetting a `boolean` variable can be fast, the database is really fast during the day, i.e.: when it matters.

Cleaning up a tree from these types of *"deleted"* items can be accomplished in many different ways. If there are too many of them, then it might be worthwhile to simply rebuild the tree from scratch (making sure it doesn't loose it's advantageous structure, i.e.: doesn't become a linked list). The program could also fire up a *more complicated* approach to delete each node individually. (While at it, the system could also be optimizing the tree structure ;-)

Now, what is that *"more complicated"* approach that I keep talking of? It is the approach of switching the node being deleted with the one currently in the tree, only at lower level. Deleting a node that has no children is pretty simple, we just remove that node (and set the parent's pointer to it to `null`). Deleting a node that has one child is also easy. We delete the node, and make it's parent point to that only child of the deleted node.

The problem comes when you try to delete a node which has two valid children. Which one do you pick to be it's successor (take it's parent's place)? Actually, in most cases, neither!

You have to realize that we're dealing with a binary search tree. Search trees have very specific properties. For example, if we need to remove a node, we look for nearest right child that doesn't have a left son (or nearest left child, that doesn't have a right son). We then replace that newly found node with the one we are trying to delete (and making sure that all the links go where they should). The process of deleting that right child with no left son actually involves a simple removal described above: i.e.: the node is simply replaced by it's only child (in this case the right child; since there is no left). Confusing? Lets jump into the code to clear it up...

```
import java.lang.*;
import java.io.*;

public class pBSTRemoveNode{

    public pBSTRemoveNode left,right;
    public Comparable data;

    public static pBSTRemoveNode tree_AddNumber(
        pBSTRemoveNode r,Comparable n){
        if(r == null){
            r = new pBSTRemoveNode();
            r.left = r.right = null;
            r.data = n;
        }else if(r.data.compareTo(n) < 0)
            r.right = tree_AddNumber(r.right,n);
        else if(r.data.compareTo(n) > 0)
            r.left = tree_AddNumber(r.left,n);
        return r;
    }

    public static pBSTRemoveNode tree_removeNumber(
        pBSTRemoveNode r,Comparable n){
        if(r != null){
            if(r.data.compareTo(n) < 0){
                r.right = tree_removeNumber(r.right,n);
            }else if(r.data.compareTo(n) > 0){
                r.left = tree_removeNumber(r.left,n);
            }else{
                if(r.left == null && r.right == null){
                    r = null;
                }else if(r.left != null && r.right == null){
                    r = r.left;
                }else if(r.right != null && r.left == null){
                    r = r.right;
                }else{
                    if(r.right.left == null){
                        r.right.left = r.left;
                        r = r.right;
                    }else{
                        pBSTRemoveNode q,p = r.right;
                        while(p.left.left != null)
                            p = p.left;
                        q = p.left;
                        p.left = q.right;
                        q.left = r.left;
                        q.right = r.right;
                        r = q;
                    }
                }
            }
        }
        return r;
    }

    public static void tree_InOrderPrint(
        pBSTRemoveNode r){
        if(r != null){
            tree_InOrderPrint(r.left);
            System.out.print(" "+r.data);
            tree_InOrderPrint(r.right);
        }
    }

    public static void main(String[] args){
        pBSTRemoveNode tree = null;
        int[] numbers = {56,86,71,97,82,99,65,36,16,10,28,52,46};
        System.out.print("inserting: ");
        for(int i = 0;i<numbers.length;i++){
            Integer n = new Integer(numbers[i]);
            System.out.print(" "+n);
            tree = tree_AddNumber(tree,n);
        }
        System.out.print("\ntree: ");
        tree_InOrderPrint(tree);
        for(int j = 0;j < numbers.length;j++){
            Integer n = new Integer(numbers[j]);
            System.out.print("\nremove: "+n+" tree: ");
            tree = tree_removeNumber(tree,n);
            tree_InOrderPrint(tree);
        }
        System.out.println("\ndone ;-)");
    }
}
```

If you look through the above code, you'll quickly realize that the most relevant function (to this section), is the `tree_removeNumber()`. It accepts a reference to the `root` of the tree, and a number to remove. Actually, the way it's implemented, it doesn't necessarily has to be a number. It could just as well be a `java.lang.String` object; anything that's implementing a `Comparable interface` will work. The title of the function is a bit misleading; telling you that you can only work with numbers.

The `main()` is rather straight forward; it first adds numbers to the tree, and then removes them, displaying what it does at each step. The `tree_AddNumber()` function will not be explained here (since it's already been explained somewhere within this document).

The `tree_removeNumber()` method is where most of the interesting action takes place. We first check to see if the root of the tree is not `null`, since if it is, we have nothing to remove, and we simply `return`. The next two `if()` and `else if()` statements do what is know as binary search. If the item that we're looking for is greater than the value of the current node, we recursively search the right child of the current node. If it's less than the value of the current node, then we recursively search the left child of the current node.

The remainder of the function is kind of a big `else` statement. Once we're there, it means we have found the item we were looking for. We start by first checking for simple cases, where the node we're removing has no children, or has only one child. If it has two valid children, we fall into another `else` statement.

Once we know it is not one of the simple cases, we have to do a bit of thinking. First, we check a bit easier case, where the right child of the node doesn't have a left child. If that's the case, we need go no further, we simply replace the removing node with it's right child, and make sure the links are not lost. If the right child has a left child, we have to loop to find the closest right child with no left child, and that's what that `while()` loop is doing. The moment we find that node we would like to put in place of the one being deleted, we remove the found node. This removal is rather simple, since the node only has a right child. We then simply replace the removed node with the new one, making sure we don't loose any links, and we are done.

Inside `main()`, I have picked numbers to work with, and to construct the tree (sample data). The numbers generate a pretty good binary tree. The removal starts with the root node, so, the example does quite extensive testing in all the cases described above. The output from the above program follows:

```
inserting:  56 86 71 97 82 99 65 36 16 10 28 52 46
tree:  10 16 28 36 46 52 56 65 71 82 86 97 99
remove: 56 tree:  10 16 28 36 46 52 65 71 82 86 97 99
remove: 86 tree:  10 16 28 36 46 52 65 71 82 97 99
remove: 71 tree:  10 16 28 36 46 52 65 82 97 99
remove: 97 tree:  10 16 28 36 46 52 65 82 99
remove: 82 tree:  10 16 28 36 46 52 65 99
remove: 99 tree:  10 16 28 36 46 52 65
remove: 65 tree:  10 16 28 36 46 52
remove: 36 tree:  10 16 28 46 52
remove: 16 tree:  10 28 46 52
remove: 10 tree:  28 46 52
remove: 28 tree:  46 52
remove: 52 tree:  46
remove: 46 tree:
done ;-)
```

Trace the output if you like, it's quite interesting. This type of removing actually improves the tree structure, it makes it shorter (decreases tree's depth), thus, making searches faster. One thing that I'd like to mention before we go on, is that this example is for JDK 1.2 or above. It will not compile (nor run) on anything less due to the fact that it uses the `java.util.Comparable interface`, which is not supported in earlier versions of the JDK. Anyway, I guess that's it for this topic.

Determining Tree Depth...

Tree related algorithms depend on tree depth being small (otherwise, they become linked list algorithms ;-). Determining what is the depth of a tree can sometimes lead to optimizations, and other interesting things like that. How then, do you determine tree depth?

The task seems rather simple, however, there are a few tricks involved. Since most trees are defined recursively, our algorithm will also be recursive. We will need a wrapper method to initialize the maximum depth to zero, and then, recursively go through the tree determining the depth at each node. We will use a counter to keep the count of the current depth. Whenever we enter a recursive method, we will increment the counter, whenever we leave, we will decrement it. If that counter is greater than the maximum tree depth encountered so far, we will set the maximum tree depth to the value of that counter.

Once the algorithm completes, the variable holding the maximum tree depth will hold the max tree depth (sounds logical doesn't it?) Anyway, talk is cheap, lets go write it!

```
import java.lang.*;
import java.io.*;

public class pBinTreeDepth{
    public pBinTreeDepth left,right;
    public Integer data;
    private static int tree_depth,curr_depth = 0;
    public static int[] numbers = {7,3,11,2,5,9,12,4,6,8,10};

    public static pBinTreeDepth add(pBinTreeDepth r,Integer n){
        if(r == null){
            r = new pBinTreeDepth();
            r.left = r.right = null;
            r.data = n;
        }else if(r.data.compareTo(n) < 0)
            r.right = add(r.right,n);
        else
            r.left = add(r.left,n);
        return r;
    }

    public static void print(pBinTreeDepth r){
        if(r != null){
            print(r.left);
            System.out.print(" "+r.data);
            print(r.right);
        }
    }

    public static void _getdepth(pBinTreeDepth r){
        if(r != null){
            curr_depth++;
            if(curr_depth > tree_depth)
                tree_depth = curr_depth;
            _getdepth(r.left);
            _getdepth(r.right);
            curr_depth--;
        }
    }

    public static int getdepth(pBinTreeDepth r){
        tree_depth = 0;
        _getdepth(r);
        return tree_depth;
    }

    public static void main(String[] args){
        pBinTreeDepth tree = null;
        System.out.print("inserting: ");
        for(int i=0;i<numbers.length;i++){
            Integer n = new Integer(numbers[i]);
            System.out.print(" "+n);
            tree = add(tree,n);
        }
        System.out.print("\ntree: ");
        print(tree);
        System.out.println("\ndepth: "+getdepth(tree));
        System.out.println("done ;-)");
    }
}
```

The code above creates a binary search tree, and then calls the `getdepth(pBinTreeDepth)` method to get the tree's depth. This method in turn calls `_getdepth(pBinTreeDepth)`, which is the actual recursive procedure. The whole thing is implemented as `static` methods; this is just to make quick & simple implementation easier.

Advanced Linked Lists...

We have already talked about linked lists previously in this document, and the assumption was that you'll learn how to use them. We didn't however explain a lot of the more used types of lists (not many people use a plain old linked list). As it stands, the linked list explained earlier is pretty bad and inefficient.

The most critical problem is that insertions and deletions from the tail of the list are slow. To delete something from the end, we would have to loop until we hit the end, and only then remove the item. This limitation is fairly obvious, since it would be extremely inefficient to use that kind of a list (single linked) to implement a queue.

The solution is to use a doubly linked list; where each node has two pointers, one to it's left neighbor, and one to it's right, and to have a `head` and a `tail` pointer. We will later implement this type of a list.

Another critical problem with the previous lists is that they have special cases in insert and remove methods. Ideally, we would just like to insert and/or delete, without any special cases (like `null` head pointer). How can we speed-up, and simplify the insertion and deletion operations? Easy! We simply have a dummy `head` pointer which is always there (but is not storing any data). Since we're interested in doubly linked lists, we would have two dummy pointers; the dummy `head`, and dummy `tail`.

Some may argue that having nodes that store no data is useless, and wastes memory. That may be true for some cases, where memory is critical, but for most purposes, the speed and simplicity gained greatly outweigh the wasted memory disadvantage.

You should still keep in mind the above. Sometimes, you end up with arrays of arrays of arrays of linked lists, and in those cases, those few wasted bytes, can add up to hundreds of megabytes. For example, it's pointless to have dummy pointers if the lists never get beyond two or three elements. Before implementing *anything*, think about the approach you're using.

Another strange thing our previous lists had was a `peek(int)` method. We used it to go through the list, and view it's contents. This may have worked quite well when the list was implemented using an array (where we directly jump to that location), but when we were using linked lists, this `peek(int)` procedure got quite slow. Given that we're looping through every element, and every time, we have to loop until we hit that number inside the list, it starts to become obvious that it's a waste of time.

What can we use to go through the items in the list, do it safely, and more efficiently? In C++ world, programmers are quite familiar in writing iterators. Iterators are used to iterate through objects contained in some data structure (usually some data container class). In Java, we have something similar available to us. It is called the `Enumeration`. Java provides the standard `java.util.Enumeration` object for us to use to go through the items in the list. In fact, because the `java.util.Enumeration` is standard, even `java.util.Vector` class uses it!

So, whenever we need to iterate through every element in the list, we simply get the `Enumeration` for the class, and use that to go through the elements. Details of the implementation are described later.

For now, we have improved our view of the list quite a bit. You should still never forget to be inventive. There are other ways to represent a linked list (for example, make it circular, with `head` and `tail` being the same node). Hopefully, we'll later examine tree representation of a linked list. A tree representation gives you the best of both worlds, linked structure, and fast insertions and deletions (more on that later, hopefully).

Doubly Linked Lists (with Enumeration)...

Doubly linked lists are not much different from singly linked lists; we just have an extra pointer to worry about. As usual, if you don't understand something, it help to draw it out on paper. Yeah, go ahead and draw a linked list, then go through the operations by drawing or erasing links.

Anyway, lets get right to the point, and write it.

```
import java.lang.String;
import java.io.*;
import java.util.*;
import pTwoChildNode;
```

```
public class pDoublyLinkedList{

    private pTwoChildNode head,tail;
    protected long num;

    protected pTwoChildNode getHead(){
        return head;
    }

    protected pTwoChildNode getTail(){
        return tail;
    }

    protected void setHead(pTwoChildNode p){
        head = p;
    }

    protected void setTail(pTwoChildNode p){
        tail = p;
    }

    public pDoublyLinkedList(){
        setHead(new pTwoChildNode());
        setTail(new pTwoChildNode());
        getTail().setLeft(head);
        getHead().setRight(tail);
        num = 0;
    }

    public long size(){
        return num;
    }

    public boolean isEmpty(){
        return num == 0;
    }

    public void addHead(Object o){
        pTwoChildNode p = new pTwoChildNode(o);
        p.setLeft(getHead());
        p.setRight(getHead().getRight());
        getHead().setRight(p);
        p.getRight().setLeft(p);
        num++;
    }

    public Object removeHead(){
        Object o = null;
        if(!isEmpty()){
            pTwoChildNode p = getHead().getRight();
            getHead().setRight(p.getRight());
            p.getRight().setLeft(getHead());
            o = p.getData();
            num--;
        }
        return o;
    }

    public void addTail(Object o){
        pTwoChildNode p = new pTwoChildNode(o);
        p.setRight(getTail());
        p.setLeft(getTail().getLeft());
        getTail().setLeft(p);
        p.getLeft().setRight(p);
        num++;
    }

    public Object removeTail(){
        Object o = null;
        if(!isEmpty()){
            pTwoChildNode p = getTail().getLeft();
            getTail().setLeft(p.getLeft());
            p.getLeft().setRight(getTail());
            o = p.getData();
            num--;
        }
        return o;
    }

    public void add(Object o){
        addHead(o);
    }

    public Object remove(){
        return removeHead();
    }

    public Enumeration elementsHeadToTail(){
        return new Enumeration(){

            pTwoChildNode p = getHead();

            public boolean hasMoreElements(){
                return p.getRight() != getTail();
            }

            public Object nextElement(){
                synchronized(pDoublyLinkedList.this){
                    if(hasMoreElements()){
                        p = p.getRight();
                        return p.getData();
                    }
                }
                throw new NoSuchElementException(
                    "pDoublyLinkedList Enumeration");
            }
        };
    }

    public Enumeration elementsTailToHead(){
        return new Enumeration(){

            pTwoChildNode p = getTail();

            public boolean hasMoreElements(){
                return p.getLeft() != getHead();
            }

            public Object nextElement(){
                synchronized(pDoublyLinkedList.this){
                    if(hasMoreElements()){
                        p = p.getLeft();
                        return p.getData();
                    }
                }
                throw new NoSuchElementException(
                    "pDoublyLinkedList Enumeration");
            }
        };
    }

    public static void main(String[] args){
        pDoublyLinkedList list = new pDoublyLinkedList();
        int i;
        System.out.println("inserting head:");
        for(i=0;i<5;i++){
            Integer n = new Integer((int)(Math.random()*99));
            list.addHead(n);
            System.out.print(n+" ");
        }
        System.out.println("\ninserting tail:");
        for(i=0;i<5;i++){
            Integer n = new Integer((int)(Math.random()*99));
            list.addTail(n);
            System.out.print(n+" ");
        }
        System.out.println("\nhead to tail print...");
        Enumeration enum = list.elementsHeadToTail();
        while(enum.hasMoreElements())
            System.out.print(((Integer)enum.nextElement())+" ");
        System.out.println("\ntail to head print...");
        enum = list.elementsTailToHead();
        while(enum.hasMoreElements())
            System.out.print(((Integer)enum.nextElement())+" ");
        System.out.println("\nremoving head:");
        for(i=0;i<5;i++){
            Integer n = (Integer)list.removeHead();
            System.out.print(n+" ");
        }
        System.out.println("\nremoving tail:");
        while(!list.isEmpty()){
            Integer n = (Integer)list.removeTail();
            System.out.print(n+" ");
        }
        System.out.println("\ndone ;-)");
    }
}
```

The above code both implements the doubly linked list, and tests it. The code also uses `pTwoChildNode` object we've developed earlier. (It is simply a node with two children ;-) Output from the above program follows:

```
inserting head:
0 39 33 14 51
inserting tail:
42 25 76 43 56
head to tail print...
51 14 33 39 0 42 25 76 43 56
tail to head print...
56 43 76 25 42 0 39 33 14 51
removing head:
51 14 33 39 0
removing tail:
56 43 76 25 42
done ;-)
```

You can try tracing the output (it helps sometimes), or you can just look at the source and see what's happening. The testing procedure should seem like second nature by this time...

The list is built around two dummy nodes, the `head` and `tail`. These are created at the time of the constructor call, and remain valid until the class gets swept away by garbage collection (when it goes out of scope). The `addHead()` method simply inserts the new node right after the `head` dummy node, and `addTail()` right before the `tail` node. The remove functions do their appropriate functions.

There really isn't much to explain; just look at the source, and you'll figure it out (there is nothing here more complex than what you've already seen). What you should be curious about is the `elementsHeadToTail()` and `elementsTailToHead()` methods. These methods return an `Enumeration` object of the `java.util` package.

The `Enumeration` object is created (and declared), inside the functions! (don't you just love Java?) All this object contains is a pointer to a node inside the list. The `java.util.Enumeration` interface has two functions, and we simply use these two functions to make it possible to step through the elements of the list. If you've ever used iterators in C++, or used `java.util.Enumeration` object with `java.util.Vector`, then this should be pretty easy to comprehend.

This step-through-the-list method is fairly safe, since we're not giving away the safety of our list structure, and we're controlling everything (the user can't just access protected members of the list class, yet the user gets a fairly fast and efficient way to loop through every element of the list as if they were able to access the inside elements of the list.) This code is much more superior to the `Object peek(int)` method we used in previous lists.

The main point of this section was not to show you one particular implementation, but to help you realize that linked lists are much more flexible than it is evident from their first appearance.

Binary Space Partition (BSP) Trees...

- Dog3D DEMO - (Click here to see the demo for this section)

As mentioned previously, tree data structures are wonderful in some cases for certain purposes. One of these purposes is Hidden Surface Removal (HSR). HSR in graphics turns out to be quite a complicated problem. To paraphrase one book, "HSR is a thorn in a graphics programmer's back".

Most graphics programmers today are more concerned with an efficient way of eliminating hidden (or unseen) surfaces, than with the inner workings of pixel plotting and/or texture mapping. Games like *Wolfenstain 3D*, *DOOM*, and *Quake* by *id Software*, are mostly reflections of innovations in the field of HSR. (and a bit of added processing power ;-)

First, lets define HSR in a more understandable manner. Imagine you're standing in a room, you can only see the walls of the room you're in, and not the walls of other rooms (which are beside your room). The walls of your room cover up your view, so, you can't see anything else other than the walls of your room. This relatively simple concept turns out to be quite a bundle when it comes to computers. There are literally hundreds of different approaches to this, and they all have their advantages and disadvantages.

One solution used in *Wolfenstain 3D* is *ray casting*. Ray casting simply passes a horizontal *"ray"* (or two rays) across a map (a map in such a case is simply a 2D array of values representing blocks); if a ray hits a *"filled"* block on the map, then a vertical scaled texture is drawn, if not, the ray continues on to the next block. This produces a pretty blocky world, evidenced by *Wolfenstain 3D*.

Another solution is *ray tracing*, it's a bit more involved than *ray casting* (actually, *ray casting* came from simplifying *ray tracing*). In this, a ray is passed over all pixels on the screen, and when a ray hits an object in 3D space, that pixel is drawn having a texture color of that object. This sounds like a lot of work, and it is. Ray tracing, currently, is only good for high quality images, and not for real time games where images are generated on the fly. (it can easily take minutes or even hours to ray trace a scene)

There are many others, like Z-Buffering, Painter's Algorithm, Portals, etc., and the one we're here to talk about is Binary Space Partition (BSP), BSP was used successfully in games like DOOM and Quake, and has the potential for a lot more. It is a process of recursively subdividing space, building a binary tree, and later traversing the tree, knowing what to draw and when.

Imagine for example that you had to draw two rooms, one beside the other, with a small door in between. What you can do is draw the walls of the farther room, then draw the door, and draw the walls of the room you're in, overwriting parts of already drawn walls of the farther room. Now, how can you figure out where you're located (in which room), so that you can first draw the farther room, and then draw the room you're in? Easy, you create a binary tree of the world; then, given your point in space, you can easily determine your location relative to the world. Thus, you can determine what to draw first, and what to draw later. (to produce a nice, real looking 3D world)

To understand this concept you need to be able to visualize the tree, and how you traverse it (have a solid understanding of the tree structure). First, you create a tree of the world, by selecting a line (or plane in 3D), adding that line (or plane) to this node; you later use that line (or plane), to sort the whole world into two. One side with lines (or planes) that are in "front" of that selected line (or plane), and the rest which are in the back. The front and back are determined from the line (or plane) equation, and the x, y, z parameters from the lines (or planes) being checked. If there comes a time where some line (or plane) is neither in front or in back (has points on both sides), it is split (partitioned) into two, one side goes in front, and the other side in back. The process recursively continues with each of these new subsets until there are no more lines to select.

The result is a tree representing the world. To traverse it, you evaluate the player's location in relation to the root node, if it is in front, you recursively draw the back, and then the front, if it is in back, you recursively draw the front, and then the back. This simple procedure produces a nicely sorted list of *"walls"* to draw. The above describes a back to front traversal, you do totally the opposite for a front to back traversal, which seems to be getting more popular now a days.

In a back to front traversal, you don't have to worry about clipping; everything looks perfect after drawing, since everything unwanted is overwritten (i.e.: Painter's Algorithm). In a front to back traversal, you've got quite a lot to worry about because of clipping. You need an efficient way of remembering which pixels have been drawn, etc. For now, we'll be mostly concerned with back to front traversal because of it's simple nature.

In this type of a discussion, it helps to keep things simple; I will only describe how to do this type of a thing for a very primitive 2D case. We will take a bunch of line coordinates, create a binary space partition tree, and later traverse that tree, displaying a 3D looking world (which is actually 2D). Don't feel too bad. 2D is simple and lets you understand the structure. DOOM is totally 2D, and still looks really cool. 3D is full of math problems which will only complicate the matter at this point. Besides, once you understand the structure, writing your own 3D implementation shouldn't be a problem (if you can get through the 3D math ;-)

Well, lets get to it. The first thing that we need for any kind of tree structure is a node. In our case, the node should contain the partition plane (in our case the line equation of a line dividing this node), and a list of lines currently spanning this node. Of course, it wouldn't be a tree node without at least two pointers to it's children; so, we'll include those too! Follows the source for this simple, yet useful node.

```
class javadata_dog3dBSPNode{
    public float[] partition = null;
    public Object[] lines = null;
    public javadata_dog3dBSPNode front = null;
    public javadata_dog3dBSPNode back = null;
}
```

As you can see, there is nothing tricky or hard to the piece of code above. Right now is a good time to figure out the conventions used in this program. A point is represented by a two element integer array. A line is represented by a five element integer array; the first four are for the starting point and ending point respectively, and the last is for the color. A partition plane (line equation) is represented by a three element floating point array. Because our partition planes are not normalized, we could as well used an integer array, but I doubt the several floating point calculations would have made much difference (even for a below-Pentium system!).

You'll also notice that the class above is not public, that's because all the Dog 3D classes are contained within one file (*javadata_dog3d.java* in case you're interested). (this program is not very modular, and separate parts make little sense outside of the program)

What we need next is our Binary Space Partition Tree to manipulate the nodes we've just created. The tree should be able to accept a list of lines, and build itself. It should also be able to traverse itself (in our case render itself). And lastly, it should contain (or have access to) all the necessary methods for working with points and lines (i.e.: comparison functions, splitting functions, etc.). The source for the Binary Space Partition Tree follows:

```
class javadata_dog3dBSPTree{
    private javadata_dog3dBSPNode root;
    public int eye_x,eye_y;
    public double eye_angle;
    private javadata_dog3d theParent = null;

    private final static int SPANNING = 0;
    private final static int IN_FRONT = 1;
    private final static int IN_BACK = 2;
    private final static int COINCIDENT = 3;

    public javadata_dog3dBSPTree(javadata_dog3d p){
        root = null;
        eye_x = eye_y = 0;
        eye_angle = 0.0;
        theParent = p;
    }

    private float[] getLineEquation(int[] line){
        float[] equation = new float[3];
        int dx = line[2] - line[0];
        int dy = line[3] - line[1];
        equation[0] = -dy;
        equation[1] = dx;
        equation[2] = dy*line[0] - dx*line[1];
        return equation;
    }

    private int evalPoint(int x,int y,float[] p){
        double c = p[0]*x + p[1]*y + p[2];
        if(c > 0)
            return IN_FRONT;
        else if(c < 0)
            return IN_BACK;
        else return SPANNING;
    }

    private int evalLine(int[] line,float[] partition){
        int a = evalPoint(line[0],line[1],partition);
        int b = evalPoint(line[2],line[3],partition);
        if(a == SPANNING){
            if(b == SPANNING)
                return COINCIDENT;
            else return b;
        }if(b == SPANNING){
            if(a == SPANNING)
                return COINCIDENT;
            else return a;
        }if((a == IN_FRONT) && (b == IN_BACK))
            return SPANNING;
        if((a == IN_BACK) && (b == IN_FRONT))
            return SPANNING;
        return a;
    }

    public int[][] splitLine(int[] l,float[] p){
        int[][] q = new int[2][5];
        q[0][4] = q[1][4] = l[4];
        int cross_x = 0,cross_y = 0;
        float[] lEq = getLineEquation(l);
        double divider = p[0] * lEq[1] - p[1] * lEq[0];
        if(divider == 0){
            if(lEq[0] == 0)
                cross_x = l[0];
            if(lEq[1] == 0)
```

```
                cross_y = l[1];
            if(p[0] == 0)
                cross_y = (int)-p[1];
            if(p[1] == 0)
                cross_x = (int)p[2];
        }else{
            cross_x = (int)((-p[2]*lEq[1] + p[1]*lEq[2])/divider);
            cross_y = (int)((-p[0]*lEq[2] + p[2]*lEq[0])/divider);
        }
        int p1 = evalPoint(l[0],l[1],p);
        int p2 = evalPoint(l[2],l[3],p);
        if((p1 == IN_BACK) && (p2 == IN_FRONT)){
            q[0][0] = cross_x;  q[0][1] = cross_y;
            q[0][2] = l[2];     q[0][3] = l[3];
            q[1][0] = l[0];     q[1][1] = l[1];
            q[1][2] = cross_x;  q[1][3] = cross_y;
        }else if((p1 == IN_FRONT) && (p2 == IN_BACK)){
            q[0][0] = l[0];     q[0][1] = l[1];
            q[0][2] = cross_x;  q[0][3] = cross_y;
            q[1][0] = cross_x;  q[1][1] = cross_y;
            q[1][2] = l[2];     q[1][3] = l[3];
        }else
            return null;
        return q;
    }

    private void build(javadata_dog3dBSPNode tree,Vector lines){
        int[] current_line = null;
        Enumeration enum = lines.elements();
        if(enum.hasMoreElements())
            current_line = (int[])enum.nextElement();
        tree.partition = getLineEquation(current_line);
        Vector _lines = new Vector();

        _lines.addElement(current_line);
        Vector front_list = new Vector();
        Vector back_list = new Vector();
        int[] line = null;
        while(enum.hasMoreElements()){
            line = (int[])enum.nextElement();
            int result = evalLine(line,tree.partition);
            if(result == IN_FRONT)          /* in front */
                front_list.addElement(line);
            else if(result == IN_BACK)      /* in back */
                back_list.addElement(line);
            else if(result == SPANNING){    /* spanning */
                int[][] split_line = splitLine(line,tree.partition);
                if(split_line != null){
                    front_list.addElement(split_line[0]);
                    back_list.addElement(split_line[1]);
                }else{
                    /* error here! */
                }
            }else if(result == COINCIDENT)
                _lines.addElement(line);
        }
        if(!front_list.isEmpty()){
            tree.front = new javadata_dog3dBSPNode();
            build(tree.front,front_list);
        }if(!back_list.isEmpty()){
            tree.back = new javadata_dog3dBSPNode();
            build(tree.back,back_list);
        }
        tree.lines = new Object[_lines.size()];
        _lines.copyInto(tree.lines);
    }

    public void build(Vector lines){
        if(root == null)
            root = new javadata_dog3dBSPNode();
        build(root,lines);
    }

    private void renderTree(javadata_dog3dBSPNode tree){
        int[] tmp = null;
        if(tree == null)
            return; /* check for end */
        int i,j = tree.lines.length;
        int result = evalPoint(eye_x,eye_y,tree.partition);
        if(result == IN_FRONT){
            renderTree(tree.back);
            for(i=0;i<j;i++){
                tmp = (int[])tree.lines[i];
                if(evalPoint(eye_x,eye_y,getLineEquation(tmp)) == IN_FRONT)
                    theParent.renderLine(tmp);
            }
            renderTree(tree.front);
        }else if(result == IN_BACK){
            renderTree(tree.front);
            for(i=0;i<j;i++){
                tmp = (int[])tree.lines[i];
                if(evalPoint(eye_x,eye_y,getLineEquation(tmp)) == IN_FRONT)
                    theParent.renderLine(tmp);
            }
            renderTree(tree.back);
        }else{   /* the eye is on the partition plane */
            renderTree(tree.front);
            renderTree(tree.back);
        }
    }

    public void renderTree(){
        renderTree(root);
    }
}
```

The above might look intimidating, but it's actually really simple. The are a lot of data members; some familiar, some are not. The `root` data member is obvious, it's the `root` of the tree! The next several are the eye's current position. We need this since we don't want to pass them as a parameter every time we render. The next data member is `theParent`, which is a reference back to the original applet. This member is used during the rendering process (not very modular). The last data members are constants for the point and line comparison functions.

The constructor takes the parent applet as it's parameter, and initializes `theParent` and other data members. The actual insertion of data into the tree is accomplished with the `build(java.util.Vector)` method. This method calls a more complicated method of the same name, but with more parameters. The `build()` method goes through the given `java.util.Vector`. It first selects the first line in the list to be the splitting plane (for the current node). It then goes through the rest of the list, sorting the lines in relation to that splitting plane. If a line is in front (determined by the evaluation functions) then it's added to the front list. If a line is in back, it's added to the back list. If a line is spanning the splitting node (has end points on both sides of the splitting node), then that line is split by a `splitLine()` function; one part goes in front, and the other into the back. If some line is actually coincident with the splitting plane, then it's added to the list of the current node. After all that, we end up with two lists of lines; one list for the back, and one list for the front. We then recursively go through the two lists.

The process described above is fairly hard to describe (an oxymoron?). If you want a more formal (*maybe better?*) description, you can search the Internet for the *"BSPFAQ."* It is a document thoroughly describing BSP trees in a more formal way.

Once the tree is built, you are ready to traverse it! The traversing is accomplished by calling the `renderTree()` method. What it does is first evaluate the eye's position in relation to the current splitting plane of the node. If it's in `front`, we recursively render the `back` child, if it's in `back`, we recursively render the `front` child. The rendering itself is accomplished by looping through the lines of the current node and drawing them. The evaluation function call inside that loop is doing back-face-culling (back-face-removal). It is a process of making sure that we're not drawing lines (walls) which are not facing us. The drawing is done by calling `renderLine()` method of `theParent` (which is a reference back to the original applet).

If you've survived this far, you're in good shape. The remaining part of this applet is simply the initialization and rendering, which is not really related to data structures. Anyway, here we go again, diving into some source before explaining it...

```
public class javadata_dog3d extends Applet implements Runnable{
    private Thread m_dog3d = null;
    private String m_map = "javadata_dog3dmap.txt";
    private int m_width,m_height;
    private int m_mousex = 0,m_mousey = 0;
    private Vector initial_map = null;
    private javadata_dog3dBSPTree theTree = null;
    private Image double_image = null;
    private Graphics double_graphics = null;
    public int eye_x = 220,eye_y = 220;
    public double eye_angle = 0;
    private boolean KEYUP=false,KEYDOWN=false,
        KEYLEFT=false,KEYRIGHT=false;
    private boolean MOUSEUP=true;

    public void renderLine(int[] l){
        double x1=l[2];
        double y1=l[3];
        double x2=l[0];
        double y2=l[1];
        double pCos = Math.cos(eye_angle);
        double pSin = Math.sin(eye_angle);
        int[] x = new int[4];
        int[] y = new int[4];
        double pD=-pSin*eye_x+pCos*eye_y;
        double pDp=pCos*eye_x+pSin*eye_y;
        double rz1,rz2,rx1,rx2;
        int Screen_x1=0,Screen_x2=0;
        double Screen_y1,Screen_y2,Screen_y3,Screen_y4;
        rz1=pCos*x1+pSin*y1-pDp;     //perpendicular line to the players
        rz2=pCos*x2+pSin*y2-pDp;     //view point
        if((rz1<1) && (rz2<1))
            return;
        rx1=pCos*y1-pSin*x1-pD;
        rx2=pCos*y2-pSin*x2-pD;
        double pTan = 0;
        if((x2-x1) == 0)
            pTan = Double.MAX_VALUE;
        else
            pTan = (y2-y1)/(x2-x1);
        pTan = (pTan-Math.tan(eye_angle))/(1+
            (pTan*Math.tan(eye_angle)));
        if(rz1 < 1){
            rx1+=(1-rz1)*pTan;
            rz1=1;
        }if(rz2 < 1){
            rx2+=(1-rz2)*pTan;
            rz2=1;
        }
        double z1 = m_width/2/rz1;
        double z2 = m_width/2/rz2;
        Screen_x1=(int)(m_width/2-rx1*z1);
        Screen_x2=(int)(m_width/2-rx2*z2);
        if(Screen_x1 > m_width)
            return;
        if(Screen_x2<0)
            return;
        int wt=88;
        int wb=-40;
        Screen_y1=(double)m_height/2-(double)wt*z1;
        Screen_y4=(double)m_height/2-(double)wb*z1;
        Screen_y2=(double)m_height/2-(double)wt*z2;
        Screen_y3=(double)m_height/2-(double)wb*z2;
        if(Screen_x1 < 0){
            Screen_y1+=(0-Screen_x1)*(Screen_y2-Screen_y1)
                /(Screen_x2-Screen_x1);
            Screen_y4+=(0-Screen_x1)*(Screen_y3-Screen_y4)
                /(Screen_x2-Screen_x1);
            Screen_x1=0;
        }if(Screen_x2 > (m_width)){
            Screen_y2-=(Screen_x2-m_width)*(Screen_y2-Screen_y1)
                /(Screen_x2-Screen_x1);
            Screen_y3-=(Screen_x2-m_width)*(Screen_y3-Screen_y4)
                /(Screen_x2-Screen_x1);
            Screen_x2=m_width;
        }if((Screen_x2-Screen_x1) == 0)
            return;
        x[0] = (int)Screen_x1;
        y[0] = (int)(Screen_y1);
        x[1] = (int)Screen_x2;
        y[1] = (int)(Screen_y2);
        x[2] = (int)Screen_x2;
        y[2] = (int)(Screen_y3);
        x[3] = (int)Screen_x1;
        y[3] = (int)(Screen_y4);
        double_graphics.setColor(new Color(l[4]));
        double_graphics.fillPolygon(x,y,4);
    }

    private void loadInputMap(){
        initial_map = new Vector();
        int[] tmp = null;
        int current;
        StreamTokenizer st = null;
        try{
            st = new StreamTokenizer(
                (new URL(getDocumentBase(),m_map)).openStream());
        }catch(java.net.MalformedURLException e){
            System.out.println(e);
        }catch(IOException f){
            System.out.println(f);
        }
        st.eolIsSignificant(true);
        st.slashStarComments(true);
        st.ordinaryChar('\'');
        try{
            for(st.nextToken(),tmp = new int[5],current=1;
            st.ttype != StreamTokenizer.TT_EOF;
            st.nextToken(),current++){
                if(st.ttype == StreamTokenizer.TT_EOL){
                    if(tmp != null)
                        initial_map.addElement(tmp);
                    tmp = null; tmp = new int[5];
                    current=0;
                }else{
                    if(current == 1)
                        System.out.println("getting: "+st.nval);
                    else if(current == 2)
                        tmp[0] = (int)st.nval;
                    else if(current == 3)
                        tmp[1] = (int)st.nval;
                    else if(current == 4)
                        tmp[2] = (int)st.nval;
                    else if(current == 5)
                        tmp[3] = (int)st.nval;
                    else if(current == 6)
                        tmp[4] = (int)Integer.parseInt(st.sval,0x10);
                }
            }
        }catch(IOException e){
            System.out.println(e);
        }
    }

    public void init(){
        String param;
        param = getParameter("map");
        if (param != null)
            m_map = param;
        m_width = size().width;
        m_height = size().height;

        loadInputMap();

        double_image = createImage(m_width,m_height);
        double_graphics = double_image.getGraphics();

        theTree = new javadata_dog3dBSPTree(this);
        theTree.build(initial_map);

        theTree.eye_x = eye_x;
        theTree.eye_y = eye_y;
        theTree.eye_angle = eye_angle;

        repaint();
    }

    public void paint(Graphics g){
        g.drawImage(double_image,0,0,null);
    }

    public void update(Graphics g){
        double_graphics.setColor(Color.black);
        double_graphics.fillRect(0,0,m_width,m_height);
        theTree.renderTree();
        paint(g);
    }

    public void start(){
        if(m_dog3d == null){
            m_dog3d = new Thread(this);
            m_dog3d.start();
        }
    }

    public void run(){
        boolean call_update;
        while(true){
            call_update = false;
            if(MOUSEUP){
                if(KEYUP){
                    eye_x += (int)(Math.cos(eye_angle)*10);
```

```
                    eye_y += (int)(Math.sin(eye_angle)*10);
                    call_update = true;
                }if(KEYDOWN){
                    eye_x -= (int)(Math.cos(eye_angle)*10);
                    eye_y -= (int)(Math.sin(eye_angle)*10);
                    call_update = true;
                }if(KEYLEFT){
                    eye_angle += Math.PI/45;
                    call_update = true;
                }if(KEYRIGHT){
                    eye_angle -= Math.PI/45;
                    call_update = true;
                }if(call_update){
                    theTree.eye_x = eye_x;
                    theTree.eye_y = eye_y;
                    theTree.eye_angle = eye_angle;
                    repaint();
                }
            }
            try{
                Thread.sleep(5);
            }catch(java.lang.InterruptedException e){
                System.out.println(e);
            }
        }
    }

    public boolean keyUp(Event evt,int key){
        if(key == Event.UP){
            KEYUP = false;
        }else if(key == Event.DOWN){
            KEYDOWN = false;
        }else if(key == Event.LEFT){
            KEYLEFT = false;
        }else if(key == Event.RIGHT){
            KEYRIGHT = false;
        }
        return true;
    }

    public boolean keyDown(Event evt,int key){
        if(key == Event.UP){
            KEYUP = true;
        }else if(key == Event.DOWN){
            KEYDOWN = true;
        }else if(key == Event.LEFT){
            KEYLEFT = true;
        }else if(key == Event.RIGHT){
            KEYRIGHT = true;
        }
        return true;
    }

    public boolean mouseDown(Event evt, int x, int y){
        MOUSEUP = false;
        m_mousex = x;
        m_mousey = y;
        return true;
    }

    public boolean mouseUp(Event evt, int x, int y){
        MOUSEUP = true;
        m_mousex = x;
        m_mousey = y;
        return true;
    }

    public boolean mouseDrag(Event evt, int x, int y){
        if(m_mousey > y){
            eye_x += (int)(Math.cos(eye_angle)*(7));
            eye_y += (int)(Math.sin(eye_angle)*(7));
        }
        if(m_mousey < y){
            eye_x -= (int)(Math.cos(eye_angle)*(7));
            eye_y -= (int)(Math.sin(eye_angle)*(7));
        }
        if(m_mousex > x){
            eye_angle += Math.PI/32;
        }
        if(m_mousex < x){
            eye_angle -= Math.PI/32;
        }
        theTree.eye_x = eye_x;
        theTree.eye_y = eye_y;
        theTree.eye_angle = eye_angle;
        m_mousex = x;
        m_mousey = y;
        repaint();
        return true;
    }
}
```

The above should be quite easy (if you've ever written an applet). I will not explain the IO nor the threading in this code. The code starts up by getting the map file and loading it. The format of the map file is shown below:

```
1  100 800 100 500 "FFFFFF"
2  200 500 200 400 "FFFF00"
3  400 800 100 800 "FF00FF"
4  400 700 400 800 "00FFFF"
5  500 700 400 700 "FF0000"
6  500 800 500 700 "0000FF"
7  800 800 500 800 "FFFF00"
8  800 500 800 800 "FF00FF"
9  700 500 800 500 "00FFFF"
10  700 400 700 500 "FF00FF"
11  800 400 700 400 "00FF00"
12  800 100 800 400 "FF00FF"
13  500 100 800 100 "00FF00"
14  500 200 500 100 "FFFF00"
15  400 200 500 200 "FF00FF"
16  400 100 400 200 "00FFFF"
17  100 100 400 100 "FF0000"
18  100 400 100 100 "0000FF"
19  200 400 100 400 "00FF00"
20  100 500 200 500 "0000FF"
21  300 400 300 500 "FFFF00"
22  400 400 300 400 "00FFFF"
23  400 300 400 400 "FF00FF"
24  500 300 400 300 "FFFFFF"
25  500 400 500 300 "FFFF00"
26  600 400 500 400 "FF00FF"
27  600 500 600 400 "00FFFF"
28  500 500 600 500 "FFFFFF"
29  500 600 500 500 "FF00FF"
30  400 600 500 600 "FFFFFF"
31  400 500 400 600 "00FF00"
32  300 500 400 500 "FF00FF"
```

With first column being the number of the line (wall), the second column being the x coordinate of the starting point of the line. The third column being the y coordinate of the starting point of the line. The forth column being the x coordinate of the ending point of the line. The fifth column being the y coordinate of the ending point of the line. And lastly, the sixth column is the color of the line (wall) in RGB format (similar to the way color is represented in HTML documents).

Once the map is loaded, we create the tree. Once that's done, we create a double buffer surface to draw into. All that action inside the `init()` method of the applet! After that, we simply fall into the main loop (the `run()` method), and render the tree! The main loop checks to see if there are arrow keys pressed, if they are, then the eye's position is updated and `redraw()` method is called. The `redraw()` method effectively calls the `update()` method, which clears the double buffer surface, and call's tree's method to render the tree. It then calls the `paint()` method to paint the double buffer surface to the screen area of the applet.

The `renderLine()` method, referred to earlier, is not the best that it could be (it even has lots of bugs!). It is badly written, and is not very efficient. But still, it does a rather satisfactory job at rendering a perspective representation of the line onto the double buffer surface. (besides, this is not even the subject of this section)

You can see the whole thing in action by **clicking here**!

Well, that's mostly it for Binary Space Partition Trees. What I'd suggest you do is expand on the above applet. Start by improving the `renderLine()` method. Then, try to do a front to back tree traversal instead of the easy back to front traversal approach taken in this tutorial. Anyway, I guess you get the picture: Trees are wonderful data structures, and there are TONS of useful algorithms that use them.

Doing a bit more: This part is added some time after I've initially written this section. Just to show what exactly can be done with VERY minimal effort. You can easily implement lighting! You already have the Z distance of each line, so, all you'll have to do is brighten or darken up the color, and you're done! For example, placing the following lines at the end of `renderLine()` would do the trick:

```
double light = (z1 + z2) / 3;
int R = (R=(int)(light*((l[4] & 0xff0000)>>16))) > 0xFF ? 0xFF:R;
int G = (G=(int)(light*((l[4] & 0x00ff00)>>8))) > 0xFF ? 0xFF:G;
int B = (B=(int)(light*(l[4] & 0x0000ff))) > 0xFF ? 0xFF:B;
double_graphics.setColor(new Color(R,G,B));
```

This might be an over-simplified approach (and will probably suck too much for anything professional), but for our little applet, it's just perfect! **Click here** to see this new version. I will not include the sources for this modified version in this document, however, they are available inside the ZIP file linked on top.

Kitchen Sink Methods...

In Java, as with everything else in this world, there are additional bells and whistles. They do not restrict us, and offer new ways of doing things. For the purpose of this tutorial, I've named these bells and whistles the *Kitchen Sink Methods* (since they're not directly part of Data Structures, and their inclusion metaphorically adds a kitchen sink to something which has everything *but* the kitchen sink) OK! I admit it, I'm not a creative person when it comes to picking names.

Programs in this section might not be portable nor implemented in the most efficient way. They serve as guides in introducing some of these bells and whistles, which might or might not be useful.

Java Native Interface (JNI)...

We have all heard about the *Java Native Interface* (JNI), and about a ton of reasons to not use it. The simple truth, however, is that at times, Java is not as fast as we would like it to be. It does not offer system specific features which make other programming languages so powerful.

One might argue that we do not need any system specific features, and could happily exist in the sandbox that the JavaVM provides. In some real world applications, however, performance is a requirement. Imagine writing a full blown game in Java. Without Java3D, you would have quite a headache trying to use your 3D acceleration hardware. Or imagine writing a disk defragmenter in Java, how would you go about doing that?

The answer lies in *Java Native Interface* (JNI). It allows Java application to access methods written in other programming languages (most commonly in C/C++). The process goes like this: we write Java code which defines these `native` methods, use `javac` to compile the java code, then run `javah` on the resulting `class` file. That generates the `.h` include file (for C/C++). We use the method definition from the `.h` file to implement our `native` version of the method. When done, we compile the C/C++ module into a shared library (DLL under Win32, shared lib under UNIX). We then `load` the library from within our Java application, and call the `native` method just as if it were a regular Java method. Sounds interesting, doesn't it?

It gets better. Since all we are using is a shared library, we could theoretically use ANY programming language to write our native code. We could even use Assembler, COBOL, or any other arcane language. We will not do it in this tutorial though, and will go with the plain old C.

Here is where the system specific part of this section creeps in. Under Microsoft Windows, we will use Microsoft Visual C++ v6 Enterprise Edition to compile our C code (any other compiler capable of generating DLL files should do). Under UNIX (or more specifically: SunOS v5.7 running on Spark 5), we will use standard gcc. The generated DLL file, and a Solaris shared lib will be included with the sources archive for this document)

To get started, lets prepare a few things. Under Windows, go to your JDK (or Java SDK as they like to call it now a days) directory and copy all the files from the include directory into your VC++ include directory. Do not forget the file(s) in the <JDK dir>/include/win32 directory; copy all of them to your VC++ include directory. Do the same for all the lib file(s) in <JDK dir>/lib directory; copy those into your VC++ lib directory.

Under Solaris, unless you are the administrator, you will not be able to write to the gcc standard include directory, so, your best bet is to copy all the above mentioned to your project directory (that will make it simpler to point to them). Note that in the JDK 1.2 Solaris version there is no lib file that needs to be linked into your native code.

Once that is done, you are ready to write code! For the purpose of illustrating the uses of JNI, we will convert our quicksort method into native C code. The example is nice enough to illustrate the use of Java arrays in native code, and is practically useful (the C version is faster than the identical Java version).

Note, that this section will not explain the workings for Quicksort. You are recommended to go and read the Quicksort section of this document before continuing. Also realize the the things described in this particular section are not hard; they're quite easy. As soon as you successfully call your first `native` method, everything will be clear.

Continuing from that encouraging sentence, lets get to writing the code! We will start by implementing our main application code (which will change later). So far, all we need is this:

```
import java.lang.*;
import java.io.*;

public class pQuicksortNative{

    public static native void qsort(int[] c);

    public static void main(String[] args){
        int i;
        int[] arr = new int[20];
        System.out.println("inserting: ");
        for(i=0;i<arr.length;i++){
            arr[i] = (int)(Math.random()*99);
            System.out.print(arr[i]+" ");
        }
        qsort(arr);
        System.out.println("\nsorted: ");
        for(i=0;i<arr.length;i++)
            System.out.print(arr[i]+" ");
        System.out.println("\nDone ;-)");
    }
}
```

This code looks nearly identical to the testing code in Quicksort section. A thing you should definitely notice is the absence of actual implementation of `qsort()` method. The declaration of `qsort()` contains a keyword `native`, which tells the Java compiler that the implementation will be loaded as a library at runtime. Implementation can be written in any language, as long as it is a library load-able at runtime. (I'm sure there are many ways of bending this definition, but that's what is *generally* assumed by the `native` declaration.)

After compiling the above code, run `javah` on the resulting `class` file.

```
> javac pQuicksortNative.java
> javah pQuicksortNative
```

This should generate a file named `pQuicksortNative.h`, which is your C/C++ include file. It contains the declarations for all the `native` methods of a given class. In our simple example, the include file should look something like:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class pQuicksortNative */

#ifndef _Included_pQuicksortNative
#define _Included_pQuicksortNative
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     pQuicksortNative
 * Method:    qsort
 * Signature: ([I)V
 */
JNIEXPORT void JNICALL Java_pQuicksortNative_qsort
  (JNIEnv *, jclass, jintArray);

#ifdef __cplusplus
}
#endif
#endif
```

The scary message on top doesn't mean much (of course you can edit the file), but it makes very little point to do so. This generated file can be used with either C or C++ code. Our example will use plain C, but the few key differences between C++ will be pointed out.

All we need to do now, is write our C module containing the implementation of `Java_pQuicksortNative_qsort`. To make the process simple, we will simply cut and paste the declaration into a new file, and continue from there. We then take the Java source which we did in our Quicksort section, convert it to C, and that's more or less the whole job. The C module (named `qsort.c`) follows.

```
#include "pQuicksortNative.h"

JNIEXPORT void JNICALL
Java_pQuicksortNative_qsort(JNIEnv * jniEnv,
    jclass javaClass,jintArray arr){

    int i,j,left = 0,right,stack_pointer = -1;
    int stack[128];
    int swap,temp;

    /* get actual array & it's size */
    jint* c = (*jniEnv)->GetIntArrayElements(jniEnv,arr,0);
    right = (*jniEnv)->GetArrayLength(jniEnv,arr) - 1;
```

```
        for(;;){
            /* see if to do insertion sort or quicksort */
            if(right - left <= 7){
                /* simple insertion sort */
                for(j=left+1;j<=right;j++){
                    swap = c[j];
                    i = j-1;
                    while(i>=left && c[i] > swap)
                        c[i+1] = c[i--];
                    c[i+1] = swap;
                }
                if(stack_pointer == -1)
                    break;
                right = stack[stack_pointer--];
                left = stack[stack_pointer--];
            }else{
                /* quicksort */

                /* find the median */
                int median = (left + right) >> 1;
                i = left + 1;
                j = right;

                /* swap the median */
                c[median] ^= c[i];
                c[i] ^= c[median];
                c[median] ^= c[i];

                /* make sure: c[left] <= c[left+1] <= c[right] */
                if(c[left] > c[right]){
                    c[left] ^= c[right];
                    c[right] ^= c[left];
                    c[left] ^= c[right];
                }if(c[i] > c[right]){
                    c[i] ^= c[right];
                    c[right] ^= c[i];
                    c[i] ^= c[right];
                }if(c[left] > c[i]){
                    c[i] ^= c[left];
                    c[left] ^= c[i];
                    c[i] ^= c[left];
                }
                temp = c[i];
                for(;;){
                    do i++; while(c[i] < temp);
                    do j--; while(c[j] > temp);
                    if(j < i)
                        break;
                    c[i] ^= c[j];
                    c[j] ^= c[i];
                    c[i] ^= c[j];
                }
                c[left + 1] = c[j];
                c[j] = temp;
                if(right-i+1 >= j-left){
                    stack[++stack_pointer] = i;
                    stack[++stack_pointer] = right;
                    right = j-1;
                }else{
                    stack[++stack_pointer] = left;
                    stack[++stack_pointer] = j-1;
                    left = i;
                }
            }
        }
```

```
    /* release array */
    (*jniEnv)->ReleaseIntArrayElements(jniEnv,arr,c,0);
}
```

The code changed only slightly from it's Java implementation. One thing that should strike you as strange is the use of function pointers in structures (the `GetArrayLength()`, etc.). It is correct to assume that in C++, this code becomes a bit simpler. And something like:

```
(*jniEnv)->ReleaseIntArrayElements(jniEnv,arr,c,0);
```

Will reduce to something like the following in C++:

```
jniEnv->ReleaseIntArrayElements(arr,c,0);
```

We're not in C++ however. Don't think that the C++ way is faster or more efficient however; it does exactly the same thing as the C method, and thus, takes exactly the same time to execute.

Now, what are those weird functions which we call? Why do we need that `GetIntArrayElements()` or `GetArrayLength()`, and why do we need to call `ReleaseIntArrayElements()` when we are finished sorting? Good question; the answer is in the way JavaVM works.

In C, we have the good old `malloc()` & `free()` to allocate/free memory. In C++, we have the all versatile `new` & `delete` operators. In Java, we have the `new` operator and the good old garbage collector. This fact that memory in Java is not fully controlled by the programmer, but by the Virtual Machine makes for some interesting issues when letting C code play around with memory managed by the JavaVM.

The `native` module expects memory to stay in one place. It does not want things to be wiped out or garbage collected when it is using them. The primary reason for calling `GetIntArrayElements()` method is to notify the JavaVM that we want to use that block of memory. The JavaVM has several options at this point. It can pin-down this block of memory (prevent it from being moved), or it can create a copy of the original data, and let us play around with the copy. No matter what it does, when we are done using the memory, we have to notify the JavaVM that we are done using it (so that it can unpin the memory, or copy the new memory block over the old one).

It is interesting to note that under Microsoft Windows it primarily seems to give you the pinned down memory, while under Solaris, it tends to give you a copy to work with. You can tell whether it's a copy or not by passing the address of a `jboolean` variable as the fourth parameter to `GetIntArrayElements()`. I suggest you look though the JDK and JavaVM documentation for a more detailed explanation.

If you are really into it, you might have noticed other changes in the code. Some of the swap code now uses `XOR` instead of a temporary variable to swap numbers. I think it's kind of cute. (but only works for numbers, not objects)

Gotten this far, it would be a shame not to compile it! Using command line options to generate a `DLL` under Microsoft Windows:

```
> cl /GD /LD qsort.c
```

and under UNIX (Solaris):

```
> gcc -shared -I $HOME/javadata -o qsort.a qsort.c
```

Assuming you are in the correct directory, your `PATH` is setup correctly, and everything else works, you should have a `qsort.dll` under Windows, and/or a `qsort.a` shared lib under UNIX.

Now that you got that done, you can go ahead and modify your Java application to load the library. The code for a contemporary `load()` procedure would look something like this:

```
static{
    System.load("c:/projects/javadata/qsort.dll");
}
```

Of course, the absolute path to the library will be different. This `System.load()` method requires that we pass the complete full path of the shared library to it. There is another method: `System.loadLibrary()` that only requires the name of the library. This `loadLibrary()` assumes that the library is inside some system directory, other than that, the idea is the same. And now, for a paragraph of preferences:

I found it simpler to use the `System.load()` as opposed to `System.loadLibrary()`. In the former one, you specify the full path, and you're done with. With `System.loadLibrary()` however, it's not that simple. For one, you have to specify just the name of the `DLL` under Windows, without the actual `.DLL` extension. This technique obviously doesn't work under UNIX, where there are no `DLL` files. Under both systems, the library loaded by `System.loadLibrary()` has to be inside some system library directory. Under Microsoft Windows, it is supposedly the `\Windows\System`, and under UNIX, it is supposedly the `/lib` (among other things). Everything will still work under Windows if the `DLL` is not in the system directory, but not under UNIX. One could argue that you can modify the `Properties` directly from within the Java application, and make it think that the current directory is a system's lib directory, but that's a pain in the neck. Because of these reasons, we will use `System.load()` throughout this document.

Modifying the original Java test application gives us:

```
import java.lang.*;
import java.io.*;

public class pQuicksortNative{

    static{
        System.load("c:/projects/javadata/qsort.dll");
    }

    public static native void qsort(int[] c);

    public static void main(String[] args){
        int i;
        int[] arr = new int[20];
        System.out.println("inserting: ");
        for(i=0;i<arr.length;i++){
            arr[i] = (int)(Math.random()*99);
            System.out.print(arr[i]+" ");
        }
        qsort(arr);
        System.out.println("\nsorted: ");
        for(i=0;i<arr.length;i++)
            System.out.print(arr[i]+" ");
        System.out.println("\nDone ;-)");
    }
}
```

Not much change from the original one, huh? This is obviously a Microsoft Windows version (so much for code portability...) We can just replace that absolute path with a UNIX path to that of UNIX lib, and it will work there as well. To make it more portable, we will move the library loading code into the `main()` method, and will accept the path to the lib from command line. So, under UNIX, you'll pass the path to UNIX lib, and under Windows, you'll pass the path to that `DLL`. For example:

```
import java.lang.*;
import java.io.*;

public class pQuicksortNative{

    public static native void qsort(int[] c);

    public static void main(String[] args){

        if(args.length > 0)
            try{
                System.load(args[0]);
            }catch(java.lang.UnsatisfiedLinkError e){
                System.out.println("bad lib name: "+args[0]);
                return;
            }
        else{
            System.out.println("include lib name as parameter");
            return;
        }

        int i;
        int[] arr = new int[20];
        System.out.println("inserting: ");
        for(i=0;i<arr.length;i++){
            arr[i] = (int)(Math.random()*99);
            System.out.print(arr[i]+" ");
        }
        qsort(arr);
        System.out.println("\nsorted: ");
        for(i=0;i<arr.length;i++)
            System.out.print(arr[i]+" ");
        System.out.println("\nDone ;-)");
    }
}
```

The output of such a program follows:

```
C:\projects\javadata>java pQuicksortNative c:\projects\javadata\qsort.dll
inserting:
55 3 46 54 18 89 0 71 89 45 31 81 67 76 57 4 97 48 59 60
sorted:
0 3 4 18 31 45 46 48 54 55 57 59 60 67 71 76 81 89 89 97
Done ;-)
```

Alternatively, you might have a configuration file which stores this kind of information (what system it is running on, which lib files to load, etc.) An ugly thing to do could be to name all libraries the same name, so that the code always knows what to load. This situation can lead to wrong libraries being loaded, and other horrible things.

I am guessing that's enough of drilling this stuff into your mind. If you want to learn more about `native` methods, you can pick up a lot from the Java documentation found at Sun's web-site. If you don't want to learn more, then what you've learned so far, should be more than enough for the purposes of general knowledge.

Bibliography...

I have gotten enough responses from people asking me to list good references for all this info, so, here I am, trying to list everything that is relevant. It is by no means a complete list, but it should provide the reader with adequate reference. I will supply the name, author(s), and a very brief opinionated description. Most book stores will let you search for the title, so, I don't think ISBN is relevant in most cases.

Most relevant:

Introduction to Algorithms, by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Riverst. It is a rather large book, which not only covers all the common data structures, but most common algorithms as well. A really nice book to have as a reference.

The Art of Computer Programming Volume 1, Fundamental Algorithms, Donald E. Knuth. This is *the* reference for all your computer programming needs. I suggest getting all 3 volumes, in addition to most of Knuth's publications.

Data Structures Using C and C++, Second Edition, by Yedidyah Langsam, Moshe J. Augenstain, Aaron M. Tenenbaum. A very nice book, covering most things I could think of about data structures. The code is not very readable though, and could have been a bit clearer. Some concepts lack concrete working examples. Incidentally, I had most of these authors as my professors for some class or another.

The Art of Computer Programming Volume 3, Sorting and Searching, Second Edition, by Donald E. Knuth. Without an argument, this is the BEST reference to get on sortint and searching. I don't think I used anything out of other volumes of the series, but ALL 3 volumes are definitely worth getting. It's a classic!

Computer Algorithms, Introduction to Design and Analysis, Second Edition, by Sara Baase. A nice book with lots of information on various algorithms. I still think The Art of Computer Programming by Donald Knuth is much better.

Java Distributed Objects, The Authoritative Solution by Bill McCarty and Luke Cassady-Dorion. This book is the ultimate book on OOP concepts and distributed systems using Java. It covers CORBA, RMI, DCOM, etc. It even implements the same program using ALL those approaches, including the plain sockets implementations!

Database System Concepts by Abraham Silberschatz, Henry F. Korth, S. Sudarshan. The usual database book. You'll need one of these, since in the real world, most companies require a whole lot of DB experience.

C/C++ Annotated Archives By Art Friedman, Lars Klander, and Mark Michaelis. This book is kind of a source book. It has source for binary trees and other interesting stuff.

Discrete Mathematics, Forth Edition, by Richard Johnsonbaugh. A very readable book on math as it relates to computers. Covers some typical data structures, including trees, while illustrating some interesting algorithms that use them. I like this book mostly for it's simple illustration of some algorithms. If all books seem too high level, this is the one to get.

Data Structures and Algorithms in Java Michael T. Goodrich, Roberto Tamassia, 1998. "The text focuses on applications, with numerous Java code examples and object-oriented software design patterns. Animations illustrate data structures and algorithms in a clear visual manner without the need for lenghty mathematical derivations. Website devoted to the book: **http://www.wiley.com/college/cs2java**."

Less relevant:

Borland C++ v4.5 Object-Oriented Programming, Forth Edition, by Ted Faison. Don't get me wrong, I know that this book is outdated, and that nobody in their right mind would use Borland v4.5 now, however, this book is in my opinion the best Object Oriented Programming book I've seen. It covers C++ and most related topics. This is not how to use Borland C++ type of a book, it's really full of Object Oriented theory and pure C++ examples to back it up. I'd personally look for future editions by the same author.

Artificial Intelligence, Structures and Strategies for Complex Problem Solving, Second Edition, by George F. Luger, William A. Stubblefield. Most of this book is fairly high level stuff, but it does show some interesting ways to play around with tree based structures, and how they relate to games. It has lots of different methods for tree traversal.

Practical UNIX Programming, A Guide to Concurrency, Communication, and Multithreading, by Kay A. Robbins, Steven Robbins. This is the book to get for system programming. The way in which it relates to this document is that it has lots of multithreading stuff, it describes it inside out. It's also a pretty good reference on basic networking. (well, maybe this document didn't have all that, but it's still a good reference...)

Java 1.1, Developer's Guide, Second Edition, by Jamie Jaworski. This is my second, and probably the last Java book. It's a good reference for all those classes, etc., but it really has nothing new for somebody who already knows Java. No interesting algorithms, just a systematic coverage of the language, and it's supporting API. (hey, I needed to put in at least one Java reference into this list ;-)

Code Complete, by Steve McConnell. This books talks about code design. How to structure code, how to plan, layout, write, test, and intergrate your code. It is definitely a "*Practical Handbook of Software Construction.*"

Irrelevant, but could be interesting (graphics):

Gardens of Imagination by Christopher Lampton This book is pretty good for a total beginner in graphics. It is a bit dated though, since it only has DOS code, which most of the time, simply doesn't run under current operating systems. BEGINNER LEVEL

Tricks of the Game Programming Gurus, by LaMothe, Ratcliff, Seminatore & Tyler Best Startup book I've seen. It is interesting to read, and offers quite a bit of inspiration. However, practically speaking, this book is outdated (some chapters could be interesting, but...) BEGINNER LEVEL

3D Game Programming With C++ by John De Goes That's the book that taught me how to create a window, under Windows, set palette, etc., however, other than that, it has nothing new. It does cover quite a bit of material, but for some strange reason, I didn't find it useful. You do need to know C++ before getting this book though. LOWER INTERMEDIATE LEVEL.

Computer Graphics, Principles and Practice, Second Edition in C, Foley, van Dam, Feiner, Hughes. This is the definitive book on Computer graphics. If you need a book that has everything, then get this one!

Image Processing In Java by Douglas A. Lyon. Includes lots of cool algorithms for all kinds of fun image effects. Very readable and easy to follow. (I actually read the whole thing in an evening.)

Algorithmic Geometry by J-D Boissonnat and M. Yvinec. A very technical book on algorithms like generating convex hulls, triangulating, and other fun things. Note that this is not a general type of Graphics book, it's a very technical math-like book.

Matrix Computations, Third Edition, by Gene H. Golub and Charles F. Van Loan. Another very techy book. Anything you ever wanted to know about Matrix manipulation on a computer. It tries to be a programming book, but in my eyes, it's a math book.

Computer Graphics by Roy A. Plastock and Gordon Kalley. A Schaum's Outline series book. Covers all the required graphics concepts. A bit brief for my taste though. The book is very cheap though (23 Canadian Dollars), so, if you're on a tight budged, this might be the one to get. I always tend to get techy Schaum's Outlines, they're cheap, concise, and attempt cover the subject more or less completely.

ZEN of Gaphics Programming, by Michael Abrash Very interesting, and the most useful book I read at the time (when it came out). It has code, and is quite inspirational. HIGHER INTERMEDIATE LEVEL

Michael Abrash's Graphics Programming Black Book Special Edition This book contains full text of several other books, including most of the text from ZEN of Graphics Programming. It is a bit more interesting book. It's final chapters talk quite a bit about the theory behind Quake engine & BSP trees. HIGHER INTERMEDIATE LEVEL

Computer Graphics, C version, Second Edition, by Donald Hearn, M. Pauline Baker One of those text-book type books. It covers everything you could possibly think up, but at times is quite brief. It's a good reference to have though (in case you forget how to do gouraud shading or something). ADVANCED LEVEL

OpenGL Programming Guide, Third Edition, The Official Guide To Learning OpenGL. Best OpenGL reference I've seen. Has most of the stuff anybody would need. Nothing system specific though, which is good.

Digital Typography, by Donald E. Knuth. A nice introduction and description of the field of making pretty text. Briefly describes TeX, Metafont, and other Knuth's creations, along with a few algorithms on aligning text, and other fun stuff found in TeX.

Pre-calculus Mathematics, 3rd Edition, by Hungerford, Mercer Most of the problems that come up in graphics are mathematical ones. This book might seem like a joke to some math major, but that's the book which has most of the equations for graphics programming. (like finding the determinant of an NxN matrix ;-) BEGINNER LEVEL

Totally Irrelevant, but Might be Interesting (theory):

Introduction to Languages and the Theory of Computation, Second Edition, by John C. Martin. This book goes through quite a bit of theory behind scanners, parsers, and other interesting topics. This is the type of book which programmers wanting to design languages and write compilers use. ADVANCED LEVEL

Crafting a Compiler with C, by Charles N. Fischer, Richard J. LeBlanc Jr. From this book you'll learn the basics of compiler design from the ground up. No previous knowledge necessary. However, I'd recommend reading the above book before, since that will make it a LOT easier for you to comprehend some of the ideas. Note that it has a few bugs in the LR parsing example, nothing wrong with algorithms as far as I know. ADVANCED LEVEL

Advanced Compiler Design and Implementation, by Steven S. Muchnick. A lot more detailed & more practical book than the above one. However, that *advanced* word in the title is not a joke. The book does jump right into things like optimization, without fully covering compiler basics. Basically, if you've written a compiler and want to make it run faster, or port it, etc., that's the book to get. REALLY ADVANCED LEVEL

Lex & Yacc, by John R. Levine, Tony Mason, and Doug Brown. This O'Reilly's reference is one of the best descriptions (with examples) of Lex and Yacc I have yet to come across.

Programming Languages, Concepts & Constructs, by Ravi Sethi. This book is too simple for anybody who read at least one of the books mentioned above. It could be interesting as a history book; like which computer language evolved out of which, and the general historical significance of events which led to the development of particular languages, etc.

The Data Compression Book, Second Edition, by Mark Nelson and Jean-Loup Gailly. A rather nice, slow paced introduction to data compression using all kinds of algorithms.

Numerical Recipes in C, The Art of Scientific Computing, Second Edition, by William H. Press, William T. Vetterling, Saul A. Teukolsky, Brian P. Flannery. This book, along with **The Art of Computer Programming Volume 2, by Donald E. Knuth.** has just about every single numerical algorithm most people can think of. For a *very simple* introduction to cryptography, try getting **Applied Cryptography by Bruce Schneier**. Note that some people claim Numerical Recipes has a few bugs in it; see **http://math.jpl.nasa.gov/nr/nr.html** for more information. ADVANCED LEVEL

An Introduction to The Theory of Numbers, Fifth Edition, by Ivan Niven, Herbert S. Zuckerman, Hugh L. Montgomery. This book has everything you'll ever need to know about numbers. So far, this book has the best description of RSA I've ever seen. Very technical, but that's exactly what makes this book good.

Simulation, Second Edition, by Sheldon M. Ross. A general book on simulations. Covers algorithms on generating random numbers, sampling various distribution functions, etc.

Irrelevant (General):

Data And Computer Communications, Fifth Edition, by William Stallings. A nice book on general Networking concepts. Very technical when it comes to hardware etc., but still fun. Gives the basics of lots of network protocols, including a fairly good description of SMTP.

Operating System Concepts, Fifth Edition, by Abraham Silberschatz, and Peter Baer Galvin. General purpose Operating Systems book. Has most of the relevant algorithms, etc. Offers nice descriptions of various actual existing operating systems, including UNIX, Linux, and Windows NT.

UNIX System Administration Handbook, Second Edition, by Evi Nemeth, Garth Snyder, Scott Seebass, Trent R. Hein. An indispensable book on System Administration; best that I've seen!

UNIX System V, A practical Guide, Third Edition, by Mark G. Sobell. A fairly descriptive book of the basics of UNIX. Simple and easy to follow, with nice examples of most of the interesting stuff. If you want to learn shell programming, this is the book to get.

Mastering Perl 5, by Eric C. Herrmann. A complete reference for Perl. I think that said it all.

COBOL, From Micro to Mainframe, Third Edition, by Robert T. Grauer, Carol Vazquez Villar, Arthur R. Buss. Nicely covers this arcane language, with more or less real world examples. Lots of code, lots of pages, but COBOL always tended to be wordy.

Visual Basic 5, by Alan Eliason and Ryan Malarkey. Since I put a COBOL book on this list, it's only fair that I put a Visual Basic book as well.

Programming and Customizing the PIC Microcontroller by Mike Predko. A simple description of what is involved in programming and experimenting with microcontrollers. Truthfully, when it came to actually writing code for one of these, the PIC data sheet proved a lot more useful than this book. This book is still a nice kick-start if you really have no idea what's going on.

This is just about one shelf worth of books (I'm not gonna go through another one...) One thing I would like to mention is that you should always also get a system programming book as well. For example, if you are doing programming under WindowsNT, you should get something like *Windows NT4 Programming from the Ground Up by Herbert Schildt*. Or something similar. It does help to know how the underlying operating system works.

Special Thanks...

Due to the enormous amounts of corrections I get from people, I'd like to thank those that make significant contributions. If your name is not on this list (and you believe it should be), just e-mail me, and I'll put it there. Anyway, special thanks goes to:

CheezHankrn for some constructive criticism.
Shalva S. Landy for lots of spelling/grammar corrections.
Oliver Neuberger for fixing the `postrav()` method.
Dan Perl for noticing some remnants of the postrav() problem.
Roosevelt Victor for thoroughly testing a lot of this code.
Andrey Salaev for fixing the peek() method in pLinkedList (no one has apparently noticed the bug for many long years...)
Bjørn Harald Olsen for noticing some more remnants of the postrav() problem.

and many others...

Contact Info...

The only sure way to find me, is to use my e-mail **@theparticle.com**. Yep, it's my domain which will hopefully stay with me.

I encourage feedback!

`Particle`
`particle at NO SPAM the particle dot com`
http://www.theparticle.com/