CS1007 lecture #22 notes

tue 4 dec 2001

http://www.cs.columbia.edu/~sklar/cs1007

today:
- news
- searching
- data structures (chapter 12)

news.

- Please complete an on-line course evaluation. See the NEWS page for the link.
- Final exam
  - official exam: Tue Dec 18 9am-12noon – LOCATIONS WILL BE POSTED ON THE WEB PAGE
  - makeup exam: Thu Dec 20 1pm-4pm
  - you MUST sign up for the makeup exam — sign up sheet is being passed around in class today
- Homework #7 due tomorrow Wed Dec 5, midnight
  there will be NO extensions!
- DON'T SKIP THE LAST CLASS!!
  I will hand back all graded homeworks, exams, etc.
  I will give out a practice final (which will NOT be posted on the web).

searching.

Often, when you have data stored in an array, you need to locate an element within that array.

This is called searching.

Typically, you search for a *key* value (simply the value you are looking for) and return its *index* (the location of the value in the array)

As with sorting, there are many searching algorithms.

We'll study the following:

- linear search
  - standard linear search, on sorted or unsorted data
  - modified linear search, on sorted data only
- binary search
  - iterative binary search, on sorted data only
  - recursive binary search, on sorted data only

here's some code to start with...

file name = InvItem.java

```
import java.lang.*;

public class InvItem {

  private String name; // this.name
  private int units;
  private float price;

  public InvItem( String name, int units, float price ) {
    this.name = name;
    this.units = units;
    this.price = price;
  } // end of constructor

  public String getName() {
    return( name );
  } // end of getName() method

  public int getUnits() {
    return( units );
  } // end of getUnits() method

  public float getPrice() {
    return( price );
  } // end of getPrice() method
```



```
  public void setPrice( double newprice ) {
    this.price = (float)newprice;
  } // end of setPrice() method

} // end of InvItem class
```


file name = Keyboard.java

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

public class Keyboard {

  public static String read_line() {
    String buf = new String();
    int i = 0;
    char c = (char)i;
    while ( c != '\n' ) {
      try {
        i = System.in.read();
      }
      catch ( IOException iox ) {
      }
      c = (char)i;
      if ( c != '\n' ) {
        buf += c;
      }
    } // end while
    return( buf );
  } // end of read_line()

} /* end of class Keyboard */
```

file name = ex22.java

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

public class ex22 {

  // declare global data
  Vector items = new Vector();
  int iterations;


  public static void main( String[] args ) {

    // instantiate instance of the ex22 class so we can access the
    // non-static portions of the class
    ex22 e = new ex22();

    // read data from the data file (items.dat) into the global items vector
    System.out.println( "reading data from file..." );
    e.readFile();

    // print on the screen the content of the global items vector
    System.out.println( "\nhere's the original data..." );
    e.printData();

    // update the prices of all the items in the global items vector
    System.out.println( "\nupdating prices..." );
    e.updatePrice();
```



```
    // print on the screen the updated contents of the global items vector
    System.out.println( "\nhere's the updated data..." );
    e.printData();

    // write the updated contents of the global items vector to the data file
    System.out.println( "\nwriting data to file..." );
    e.writeFile();

    // now let the user enter the name of an item and we will search the
    // global items vector for that item and print its price on the screen;
    // here, we'll compare the runtime (# of iterations) for the four
    // search methods we've discussed in class
    System.out.print( "\nenter name of item: " );
    String name0 = Keyboard.read_line();

    e.iterations = 0;
    int i = e.linearSearch( name0 );
    System.out.println( "# iterations using Linear Search ............." +
                        e.iterations );

    e.iterations = 0;
    i = e.modifiedLinearSearch( name0 );
    System.out.println( "# iterations using Modified Linear Search ...." +
                        e.iterations );

    e.iterations = 0;
    i = e.binarySearch( name0 );
    System.out.println( "# iterations using Binary Search ............." +
                        e.iterations );
```



```
    e.iterations = 0;
    i = e.recursiveBinarySearch( name0, 0, e.items.size()-1 );
    System.out.println( "# iterations using Recursive Binary Search ..." +
                        e.iterations );

    // all the searches (above) return -1 if the item being searched for is
    // not found; so we need to check the return value -- any of them will
    // do, we'll just use the last one, since it is most convenient
    if ( i == -1 ) {
      System.out.println( "\nitem " + name0 + " not found" );
    }
    else {
      InvItem tmp = (InvItem)e.items.elementAt( i );
      System.out.println( "\nprice of " + tmp.getName() +
                          " = $" + tmp.getPrice() );
    }

    System.exit( 0 );

  } // end of main()
```



```
  public void readFile() {
    String line;
    String name0  = "";
    int    units0 = 0;
    float  price0 = 0;
    try {
      FileReader fr = new FileReader( "items.dat" );
      BufferedReader infile = new BufferedReader( fr );
      while (( line = infile.readLine() ) != null ) {
        StringTokenizer t = new StringTokenizer( line );
        name0 = t.nextToken();
        try {
          units0 = Integer.parseInt( t.nextToken() );
          price0 = Float.parseFloat( t.nextToken() );
          items.addElement( new InvItem( name0,units0,price0 ));
        }
        catch ( NumberFormatException nfx ){
        }
      } // end of while
      infile.close();
    } // end of try
    catch ( FileNotFoundException fnfx ) {
      System.out.println( fnfx );
    }
    catch ( IOException iox ) {
      System.out.println( iox );
    }
  } // end of method readFile()
```



```
  public void writeFile() {
    try {
      FileWriter fw = new FileWriter( "items.dat" );
      PrintWriter outfile =
        new PrintWriter( new BufferedWriter( fw ));
      for ( int i=0; i<items.size(); i++ ) {
        InvItem tmp = (InvItem)items.elementAt( i );
        outfile.println( tmp.getName() + " " +
                         tmp.getUnits() + " " +
                         tmp.getPrice() );
      } // end for i
      outfile.close();
    }
    catch ( IOException iox ) {
      System.out.println( iox );
    }
  } // end of method writeFile()

  public void printData() {
    for ( int i=0; i<items.size(); i++ ) {
      InvItem tmp = (InvItem)items.elementAt( i );
      System.out.println( tmp.getName() + " " +
                          tmp.getUnits() + " " +
                          tmp.getPrice() );
    } // end for i
  } // end of method printData()
```



```
  public void updatePrice() {
    for ( int i=0; i<items.size(); i++ ) {
      InvItem tmp = (InvItem)items.elementAt( i );
      tmp.setPrice( tmp.getPrice() * 1.5 );
    } // end for i
  } // end of method updatePrice()

  public int linearSearch( String key ) {
    //... placeholder for code shown on a subsequent slide...
  } // end of linearSearch() method


  public int modifiedLinearSearch( String key ) {
    //... placeholder for code shown on a subsequent slide...
  } // end of modifiedLinearSearch() method


  public int binarySearch( String key ) {
    //... placeholder for code shown on a subsequent slide...
  } // end of binarySearch() method


  public int recursiveBinarySearch( String key, int lo, int hi ) {
    //... placeholder for code shown on a subsequent slide...
  } // end of recursiveBinarySearch() method


} // end of class ex22
```

linear search on UNSORTED DATA.

Linear search simply looks through all the elements in the array, one at a time, and stops when it finds the key value. This is inefficient, but if the array you are searching is not sorted, then it may be the only practical method.

```
public int linearSearch( String key ) {
  for ( int i=0; i<items.size(); i++ ) {
    iterations++;
    InvItem tmp = (InvItem)items.elementAt( i );
    if ( key.compareTo( tmp.getName() ) == 0 ) {
      return( i );
    }
  } // end for i
  return( -1 );
} // end of linearSearch() method
```


linear search on SORTED data.

If the array you are searching IS sorted, then you can modify the linear search to stop searching if you have looked past the place where the key would be stored if it were in the array. This only helps shorten the run time if the key is not in the array...

```
public int modifiedLinearSearch( String key ) {
  for ( int i=0; i<items.size(); i++ ) {
    iterations++;
    InvItem tmp = (InvItem)items.elementAt( i );
    if ( key.compareTo( tmp.getName() ) == 0 ) {
      return( i );
    }
    else if ( key.compareTo( tmp.getName() ) < 0 ) {
      return( -1 );
    }
  } // end for i
  return( -1 );
} // end of modifiedLinearSearch() method
```


binary search.

Binary search is much more efficient than linear search, ON A SORTED ARRAY. It CAN-NOT be used on an unsorted array!

It takes the strategy of continually dividing the search space into two halves, hence the name *binary*.

Say you are searching something very large, like the phone book. If you are looking for one name (e.g., “Gilligan”), it is extremely slow and inefficient to start with the A’s and look at each name one at a time, stopping only when you find “Gilligan”. But this is what linear search does.

Here’s how binary search works:

Binary search acts much like you’d act if you were looking up “Gilligan” in the phone book.

You’d open the book somewhere in the middle, then determine if “Gilligan” appears before or after the page you have opened to.

If “Gilligan” appears after the page you’ve selected, then you’d open the book to a later page.

If “Gilligan” appears before the page you’ve selected, then you’d open the book to an earlier page.

You’d repeat this process until you found the entry you are looking for.

binary search, 2.

```
public int binarySearch( String key ) {
  int lo = 0, hi = items.size()-1, mid;
  while ( lo <= hi ) {
    iterations++;
    mid = ( lo + hi ) / 2;
    InvItem tmp = (InvItem)items.elementAt( mid );
    if ( key.compareTo( tmp.getName() ) == 0 ) {
      return( mid );
    }
    else if ( key.compareTo( tmp.getName() ) < 0 ) {
      hi = mid - 1;
    }
    else {
      lo = mid + 1;
    }
  } // end while
  return( -1 );
} // end of binarySearch() method
```


recursive binary search.

You might notice that binary search lends itself well to recursion...

```
public int recursiveBinarySearch( String key, int lo, int hi ) {
  if ( lo <= hi ) {
    iterations++;
    int mid = ( lo + hi ) / 2;
    InvItem tmp = (InvItem)items.elementAt( mid );
    if ( key.compareTo( tmp.getName() ) == 0 ) {
      return( mid );
    }
    else if ( key.compareTo( tmp.getName() ) < 0 ) {
      return( recursiveBinarySearch( key, lo, mid-1 ));
    }
    else {
      return( recursiveBinarySearch( key, mid+1, hi ));
    }
  }
  else {
    return( -1 );
  }
} // end of recursiveBinarySearch() method
```


data structures.

A *data structure* is essentially an *abstract data type* which maps a virtual model of data to a real data type, like an array or a Vector or a class.

There are several classic data structures in computer science. We’ll look at a few:

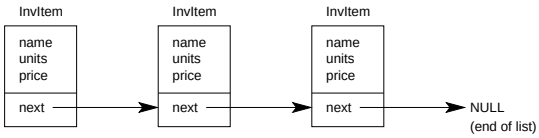
- linked list
- doubly linked list
- queue (next time)
- stack (next time)

Each has rules about how to add and remove elements. For example, a queue is also called FIFO — first in, first out. A stack is also called LIFO — last in, first out.

linked list.

A linked list chains instances of a class together using a field called “next”, which points to the next instance of the class in the chain. (Yes, this looks like another type of array or Vector.)

```
public class InvItem {
  private String name;
  private int units;
  private float price;
  InvItem next; // points to the next InvItem in the chain
}
```



doubly linked list.

A doubly linked list chains instances of a class together using two fields called “next” (which points to the next instance of the class in the chain) and “prev” (which points to the previous instance of the class in the chain).

```
public class InvItem {
  private String name;
  private int units;
  private float price;
  InvItem next; // points to the next InvItem in the chain
  InvItem prev; // points to the previous InvItem in the chain
}
```