Sequential/Contiguously Allocated List

© 2012

David S. Burris, Ph.D., CCP, CSP

Certified Computing Professional

Certified Computer Programmer

(Systems Programming)

Certified Systems Professional

(Software Engineering)

Computer Science

P.O. Box 2206

Sam Houston State University

Huntsville Texas, 77341

[csc\_dsb@shsu.edu](mailto:csc_dsb@shsu.edu)

Permission is granted to students in my classes to use any or all of the material in this document for their personnel education. All other rights reserved.

Basic Sequentially (contiguously) Allocated List Operations

**Bounded Linearly Allocated List**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  | Sam | Mary | Joe |  |  |  |  |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |  |
|  |  |  | F |  | L |  |  | M |  |

Assume a sequentially allocated list occupies memory locations F(first item) through M (maximum location). Further assume the last item in the list currently occupies location L (current last item). The following are basic list operations:

**Insert a new item as the last entry in the list:**

1. Prompt user for “y” to insert in the list.
2. If L < M

L 🡨 L + 1

List[L] 🡨 y

Else

Report the list is full, overflow.

End if

**Randomly Insert a new item in the list at location J, F<= J<= L:**

1. Prompt the user for a valid value of J.
2. If (L < M) // there is space left in the list

For K := L down to J loop

List[ K+1 ] <-- List[ K ];

End Loop;

List[ J ] <-- y;

L <-- L + 1;

Else

Overflow;

End if;

**Randomly Delete the Jth item from the list, J >= F and J<= L:**

1. Prompt the user for a valid value of J.
2. For K := J to (L-1) loop

List[ K ] <-- List[ K+1 ];

End loop;

L <-- L – 1;

**Bounded Linearly Allocated List**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  | Joe | Mary | Sam |  |  |  |  |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |  |
|  |  |  | F |  | L |  |  | M |  |

Assume a sequentially allocated list occupies memory locations F(first item) through M (maximum location). Further assume the last item in the list currently occupies location L (current last item) and there is at least one free place in the list (L < M). The following are basic list operations:

**Insert a new item in lexicographic (ascending sorted) order:**

1. Prompt the user for the value to insert y.
2. K := L;
3. If (L<M){

While ( K >= F and List[ K] > y) loop

List[ K+1] <-- List[K];

K <-- K – 1;

End loop;

If (K = L) // New last item in list

List[ L+1 ] <-- y

Else If ( K < F) // New first item in list

List[ F ] <-- y;

Else // New interior item in list

List[ K + 1 ] <-- y;

End if;

End if;

L <-- L + 1;

}

Else

Overflow;

End if:

New values of “y” are inserted behind all existing values of “y.” We assume the Boolean short circuit for the test “While ( K >= F and List[ K] > y) loop”, i.e., if “K >= F” is false, the second test “List[ K ] > y” is not performed. Try inserting “Betty,” “Marty,” and “Zoro.”

**Stacks**

Stacks are the most widely used data structure in computing. They are frequently referred to as LIFO list, as the last item placed in the stack it the first to be removed. Typically stacks support two operation, “push” and “pop.” Other stack methods such as “how many items are currently in the stack,” “is the stack empty,” etcetera are convenient.

Consider the problem of assembling a space station in space. To reduce risk, it has been decided astronauts should be tethered to the incomplete station or ship as much as possible. Assume we have been contracted to build Robby Robot. Robby is supposed to find tools and parts required for the assembly process and bring them to the astronaut. This was an actual NASA project eventually dropped for weight, power, and cost restrictions. The actual search would have been in 3-D. A partial solution to the problem in 2 dimensions follows utilizing a stack. If the goals was not in site, the robot would move to the next grid coordinate counter clockwise (starting below the current location} from its current location as long as it had not previously visited that location after pushing the location into the stack. Once the item was found, the robot returned to the astronaut by popping the stack. **The stack is the data structure of choice any time you must backup!**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  | GOAL |  |  |  |  |
|  | L |  |  |  |  |
|  | K |  |  |  |  |
|  | J |  | I |  |  |
|  | D | E | F | H |  |
|  | C |  | G |  |  |
|  | B |  |  |  |  |
|  | A |  |  |  |  |
|  | START |  |  |  |  |
|  |  |  |  |  |  |

The initial trip would be A, B, C, D, E, F, G, H, I, J, K, L, GOAL with the stack appearing as follows where position with lines through them(E, F, G, H, and I) were popped out of the stack during the search:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| START | A | B | C | D | ~~E~~ | ~~F~~ | ~~G~~ | ~~H~~ | ~~I~~ | J | K | L | GOAL |  |

The “return trip” would be accomplished by simply popping the remaining grid coordinates from the stack: GOAL, L, K, J, D, C, B, A, START. This represents the most direct path back to the astronaut.

**Sequentially Allocated Stack**

Assume we implement the stack as an array of 0 to (Max-1) items. The variable “top” points to the top element of the stack. Top = -1 will be the boundary condition to indicate an empty stack.

(pseudo code):

|  |
| --- |
| Class Stack |
| Private:  Data structures |
| Public  Procedure Push( X: in MyType);  Function Pop( ) return MyType; |

max: integer := 10;

type MyType is record

F1: array(1..max) of character;

F2: integer;

End record;

stack: array( 0.. Max-1) of MyType;

top: integer range -1..Max - 1 := -1; // Set empty.

**Method Push**

Procedure Push( X: in MyType) is

begin

If( top < max-1) then

top 🡨 top + 1; // top 🡨 top + c;

Stack( top ) 🡨 X; // Insert item X into stack.

else

report overflow; // This is normally an error condition.

end if;

end Push;

**Method POP:**

function Pop( ) return MyType is

begin

If( top > -1 ) then

top 🡨 top – 1; //top 🡨 top – c;

return stack( top + 1); // Pop the stack. => return stack( top + c);

else

report underflow; // This is normally a desired condition.

End if;

end Pop;

**//SimpleStack.java We fired the employee who wrote this code!**

**import java.util.Scanner;**

**public class StackSimple**

**{**

**public static void main( String args[ ])**

**{**

**Scanner input = new Scanner( System.in );**

**String name[ ] = { "kites", "pencils", "paper",**

**"pens", "kites" };**

**String Stack [ ] = new String[ 10 ];**

**int top = 0;**

**for(top = 0; top < 4; top++) {**

**System.out.printf("Push %s\n", name[top]);**

**Stack[ top ] = name[ top ];**

**}**

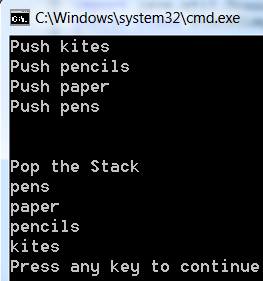
**System.out.printf("\n\n%s\n", "Pop the Stack");**

**for( top = 3; top >=0; top--)**

**System.out.printf("%s\n", Stack[top]);**

**} // end main**

**}// end SimpleStack**



//StackClass.java Kiddie Grade

import java.util.Scanner;

public class StackClass

{

public static void main( String args[ ])

{

Scanner input = new Scanner( System.in );

Stack aStack = new Stack( );

String item[ ] = {"kites", "pencils", "chairs", "toy", "car"};

for(int K = 0; K < 4; K++) {

System.out.printf("Push %s\n", item[ K ]);

aStack.push( item[ K ]);

}

System.out.printf("\n\n%s\n", "Pop the Stack");

for( int K = 0; K < 4; K++)

System.out.printf("%s\n", aStack.pop( ) );

} // end main

**static class Stack // In same file as main program.**

**{**

**String nameStack[ ] = new String[ 10 ];**

**int top = -1;** // Compile Time initialization.

**int Max = 10;**

**public Stack( ) { top = -1;} //Constructor.**

**public void push( String strIn){**

**if (top < (Max-1) )**

**nameStack[++top] = strIn;**

**else**

**System.out.printf("%s\n", "Overflow");**

**//end if;**

**} // end push**

**public String pop( ){**

**if( top == -1){**

**System.out.printf("%s\n", "Underflow");**

**return "";**

**}**

**else**

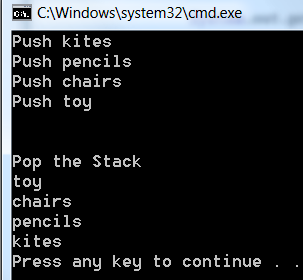
**return nameStack[ top-- ];**

**//end if;**

**}//end pop**

**}//end Stack**

**}// end StackClass**



//StackClassSemiPro1.java Semi Professional Grade

// Required by StackClassSemiPro1Use.java

import java.util.Scanner;

class StackClassSemiPro1

{

String nameStack[ ] = new String[ 10 ];

int top = -1;

int Max = 10;

public StackClassSemiPro1( ) { top = -1;} //constructor

public void push( String strIn){

if (top < (Max-1) )

nameStack[++top] = strIn;

else

System.out.printf("%s\n", "Overflow");

//end if;

} // end push

public String pop( ){

if( top == -1){

System.out.printf("%s\n", "Underflow");

return ""; //signal failure with null string

}

else

return nameStack[ top-- ];

//end if;

}//end pop

}//end StackSemiPro1

/\*

Potential additional methods:

public Boolean stackFull( );

public Boolean stackEmpty();

public int sizeOfStack( );

public void setStackEmpty( );

\*/

//StackClassSemiProUse.java Low Industrial Grade

import java.util.Scanner;

//import StackClassSemiPro1;

public class StackClassSemiProUse

{

public static void main( String args[ ])

{

Scanner input = new Scanner( System.in );

**StackClassSemiPro1 aStack = new StackClassSemiPro1( );**

**StackClassSemiPro1 Stack2 = new StackClassSemiPro1( );**

String item[ ] = {"kites", "pencils", "chairs", "toy", "car"};

for(int K = 0; K < 4; K++) {

System.out.printf("Push %s\n", item[ K ]);

**aStack.push( item[ K ]);**

}

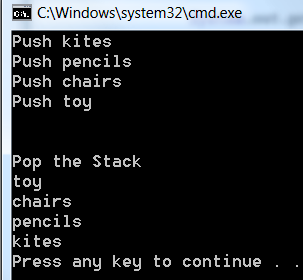
System.out.printf("\n\n%s\n", "Pop the Stack");

for( int K = 0; K < 4; K++)

System.out.printf("%s\n", **aStack.pop( )** );

} // end main

}// end StackClass



//StackClassProfessional.java Professional Grade using "templates."

// Supports intrinsic scalar types but not programmer defined types.

// Required by StackClassProfessionalUse.java

import java.util.Scanner;

**class StackClassProfessional<DataType>**

{

private final int size;

private int top = -1;

private DataType[ ] genericStack;

**public StackClassProfessional( ) {**

**this(10);**  //default stack size or size = 10; Now allocate?

}

**public StackClassProfessional( int s) // User determines size.**

**{**

**size = s > 0 ? s : 10;**

**top = -1;**

**genericStack = ( DataType[ ] ) new Object[ size ];**

**} //Will generate –Xlint notice at compile time. It is okay here!**

**public void push( DataType pushValue)**

{

if ( top == size - 1 )

throw new FullStackException ( String.format(

"The stack is full, method fails for %s", pushValue));

genericStack[ ++top ] = pushValue;

} // end push

**public DataType pop( )**

{

if( top == -1)

throw new EmptyStackException("The stack is empty");

return genericStack[ top-- ];

//end if;

}//end pop

// Programmer defined exceptions.

public class FullStackException extends RuntimeException

{

public FullStackException( )

{

this( "The stack is full!\n");

}

public FullStackException( String exception )

{

super( exception );

}

}

public class EmptyStackException extends RuntimeException

{

public EmptyStackException( )

{

this("The stack is empty!\n");

}

public EmptyStackException( String exception )

{

super( exception );

}

}

}//end StackClassProfessional<DataType>

//StackClassProfessionalUse.java Professional Grade, but not Super Professional.

// It works for intrinsic scalar data types but not all programmer defined objects.

//import StackClassProfessional;

import java.util.Scanner;

public class StackClassProfessionalUse

{

private double[ ] dblData = { 1.0, 2.5, 3.6, 4.8, 5.7};

private int[ ] intData = { 1, 2, 3, 4, 5 };

**private StackClassProfessional< Double > dblStack;**

**private StackClassProfessional< Integer > intStack;**

**private StackClassProfessional< Float > floatStack;**

public void testStack( )

{

**dblStack = new StackClassProfessional< Double >(5);**

**intStack = new StackClassProfessional< Integer >( 15 );**

**floatStack = new StackClassProfessional< Integer >( );**

try

{

for( int K = 0; K < 5; K++) {

System.out.printf("Push %f\n", dblData[ K ]);

**dblStack.push( dblData[ K ] );**

}

System.out.printf("\n\n%s\n", "Pop the double Stack");

for( int K = 0; K < 5; K++) {

**double temp = dblStack.pop( );**

System.out.printf("Pop %f\n", temp );

}

}

catch ( StackClassProfessional.FullStackException fullStackException )

{

System.err.println( );

fullStackException.printStackTrace();

}

System.out.printf("\n\n%s\n", "Start of integer Stack");

try

{

for( int K = 0; K < 5; K++) {

System.out.printf("Push %d\n", intData[ K ]);

**intStack.push( intData[ K ] );**

}

System.out.printf("\n\n%s\n", "Pop the integer Stack");

for( int K = 0; K < 5; K++) {

System.out.printf("Pop %d\n", **intStack.pop( )** );

}

}

catch ( StackClassProfessional.FullStackException fullStackException )

{

System.err.println( );

fullStackException.printStackTrace();

}

} // end test stack

public static void main( String args[ ] )

{

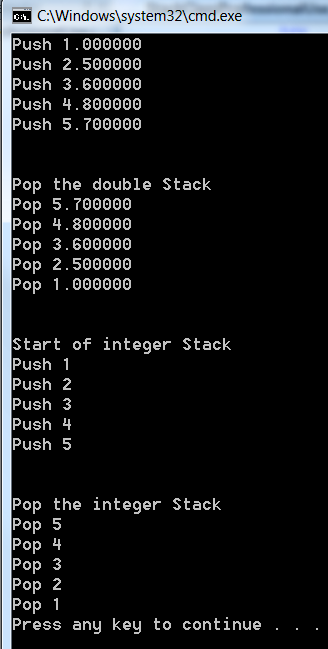
StackClassProfessionalUse app =

new StackClassProfessionalUse( );

app.testStack( );

}

}// end StackClassProfessionalUse



**Sequentially Allocated Queue**

Assume we implement the queue as an array of Max items. The variable “front” points to the next available element in the queue. “Rear” points to the last item in the queue. We assume all insertions at the rear and all removals from the front. Front = rear will be the boundary condition to indicate an empty queue. For initialization, we set front = rear = -1. We implement the queue under the assumption that the queue empties on a regular basis, i.e., the front = rear when the last item is removed from the queue. When the queue becomes empty, we reset front = rear = -1 to reclaim previously used space (prevent the queue from constantly growing in the same direction). The following queue is only practical if all data in the queue can be processed on a regular basis. For convenience, we assume a high level language so that C = 1 (the distance from one queue element to the next). Throughout this algorithm the pointer front is usually one less than the actual location of the first element of the queue. The pointer rear actually points to the last item in the queue if the queue is not empty.

|  |
| --- |
| Class Queue |
| Private:  Data structures |
| Public  Procedure Insert( X: in MyType);  Function Remove( ) return MyType; |

max: integer := 10;

type MyType is record

F1: array(0..9) of character;

F2: integer;

End record;

Queue: array(0..9) of MyType;

front, rear: integer range 0..9 := -1;

**Insert in rear of the queue:**

If ( rear < max - 1) then

rear 🡨 rear + 1;

Queue( rear) 🡨 X; // Insert X in rear of queue.

else

Report overflow; // Normally an error condition.

end if;

**Remove from the front of the queue:**

If( front /= rear) then

front 🡨 front + 1;

X 🡨 Queue( front ); // Set X to item at front of queue removing it.

If( front = rear ) then front 🡨 rear 🡨 -1; // Reset, queue is empty.

else

report underflow; -- Normally a desired condition.

end if;

MAX = 6

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| -1 | 0 | 1 | 2 | 3 | 4 | 5 |
|  | Joe | Tom | Sue |  |  |  |
| F = -1 |  |  | R = 2 |  |  |  |

This implementation of a queue is most useful when the queue empties on a periodic basis, e.g., an earth satellite dumping over the line of sight home antenna, polling gas stations, ATM,s etceteras for all transactions.**Three Approaches Sequential Search Logic**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **get(desired)** |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| **k = 0**  **count = 0** |  |  | **Search Method 1: The desired item can appear more than once in the table.** |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| **k < ATS** |  |  | **Yes** |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| **No** |  |  | **invTab[K].item == desired** |  | **Yes** |  |
|  |  |  |  |  |  |  |
|  |  |  | **No** |  | **count++** |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  | **K++** |  |  |  |
|  |  |  |  |  |  |  |
| **cout << count** |  |  |  |  | **Item** | **Price** |
|  |  |  |  | **0** | Kites | 2.39 |
|  |  |  |  | **1** | Pencils | 16.78 |
|  |  |  |  | **2** | Paper | 0.59 |
|  |  |  |  | **3** | Pens | 10.00 |
|  |  |  |  | **4** | Kites | 2.39 |
|  |  |  |  | **5** |  |  |
| **Legend:**  **Blue rectangle – diamond for decisions**  **Black – rectangle for processing**  **Green – trapezoid I/O** | | | |  | **TS = 6**  **ATS = 5** |  |
|  |  |  |  |  |  |  |

**Require = (ATS compares in table? + ATS compares is desired? + ATS increments) = 3\*ATS on the maximum.**

**You must look in every table entry to determine the item is not in the table.**

**Sequential Search Method 2: The desired item can only appear once in the table.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **get( Desired )** |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  | **k = 0** |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  | **Item** | **Price** |
|  | **K < ATS** |  |  | **0** | Kites | 2.39 |
|  |  |  |  | **1** | Pencils | 16.78 |
| **no** | **yes** |  |  | **2** | Paper | 0.59 |
|  |  |  |  | **3** | Pens | 10.00 |
|  |  | **no** |  | **4** | Tops | 25.10 |
|  | **invTab[K] == desired** |  | **K++** | **5** |  |  |
|  |  |  |  |  |  |  |
|  | **yes** |  |  |  | **TS = 6**  **ATS = 5** | |
|  |  |  |  |  |
|  |  |  |  |  |  |  |
|  | **K < ATS** |  |  |  |  |  |
|  |  |  |  |  |  |  |
| **no** |  | **yes** |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| **Not in**  **table** |  | **Found**  **at K** |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

**Analysis:**

**Average Logic = (ATS compares in table? + ATS compares is desired? + ATS increments) / 2 = 3\*ATS/2 on the average. The theoretical value for the average is (ATS+1)/2 or (N+1)/2 hence:**

**Average = (N+1)/2 average checks if in table? + (N+1)/2 average check desired? + average (N+1)/2 increment K + final check = approximately 3\*(N+1)/2 + final.**

**Max to find = TS checks if in table? + TS checks desired? + TS increments K**

**+ final check = 3\*TS + final**

**The minimum probes to find is an item is 1.**

**Average to find not in the table: search about half the table or (ATS+1)/2.**

**Search Method 3: Fast search if the desired item can only appear once in the table.**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | | **get( desired )** | | |  |  |  |  |
|  |  | |  |  | |  |  |  |  |
|  | **1) placed desired item invTab[ ATS ] (bottom list)**  **2) K = 0** | | | | | | |  |  |
|  |  |  | | |  |  |  |  |  |
|  |  |  | | |  |  |  |  |  |
|  |  |  | | |  |  |  | **Item** | **Price** |
|  |  | **invTab[K] == desired** | | |  |  | **0** | Kites | 2.39 |
|  |  |  | | |  |  | **1** | Pencils | 16.78 |
|  | **yes** | **no** | | |  |  | **2** | Paper | 0.59 |
|  |  |  | | |  |  | **3** | Pens | 10.00 |
|  |  | **K++** | | |  |  | **4** | Tops | 25.10 |
|  |  |  | | |  |  | **5** | Desired! |  |
|  |  |  | | |  |  |  |  |  |
|  |  |  | | |  |  |  | **TS = 6**  **ATS = 5** |  |
|  |  | **K < ATS** | | |  |  |  |  |  |
|  | **No** |  | | |  | **Yes** |  |  |  |
|  |  |  | | |  |  |  |  |  |
|  |  |  | | |  |  |  |  |  |
|  | **Not Found** |  | | |  | **Found**  **at K** |  |  |  |
|  |  |  | | |  |  |  |  |  |
|  |  |  | | |  |  |  |  |  |
|  |  |  | | |  |  |  |  |  |
|  |  |  | | |  |  |  |  |  |
|  |  |  | | |  |  |  |  |  |

**Analysis:**

**Average:**

**Requires = Insertion at bottom + (N+1)/2 average check desired? + (N+1)/2 average increment K + final check = insert + 2\*(N+1)/2 + final = insert + (N+1) + final.**

**Max = Insertion at bottom + TS check desired? + TS increment K + final check**

**= insertion + 2\*TS + final**

**The larger the table, the greater the savings over search method 2. Search Method 3 is about 1/3 faster than Search Method 2. Allowing for fixed overhead, about 25% to 30% faster.**

**//SequentialSearchMethod1.java**

**// Implements Search Method 1 where the desired key may appear more than one**

**// time in the table. Non-professional implementation of inventory record.**

**import java.util.Scanner;**

**public class SequentialSearchMethod1**

**{**

**public static void main( String args[ ])**

**{**

**Scanner input = new Scanner( System.in );**

**String name[ ] = { "kites", "pencils", "paper", "pens", "kites" };**

**double unitPrice [ ] = { 2.39, 16.78, 0.59, 10.00, 2.39 };**

**System.out.printf("%-10s%12s\n", "Item", "Price");**

**for( int bin = 0; bin < name.length; bin++)**

**System.out.printf("%-10s %10.2f\n", name[bin], unitPrice[bin]);**

**System.out.printf("/n%s", "Enter search key or 'Stop': ");**

**String schKey = input.next();**

**while ( !(schKey.equals("Stop")) ){**

**for( int bin = 0; bin < name.length; bin++)**

**if ( schKey.equalsIgnoreCase( name[bin] ) )**

**System.out.printf("%2d %-10s %10.2f\n",**

**bin, name[bin], unitPrice[bin]);**

**//end if**

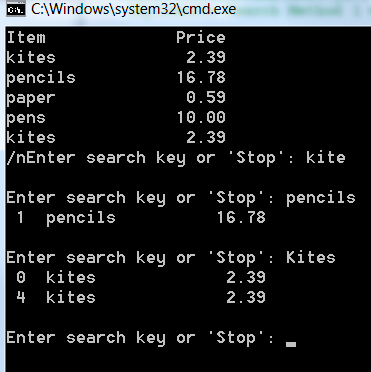
**System.out.printf("\n%s", "Enter search key or 'Stop': ");**

**schKey = input.next();**

**} //end while**

**} // end main**

**} //end SequentialSearchMethod1**



//InventoryRecord for use with SequentialSearchMethod1B.java

// In file InventoryRecord.java

**public class InventoryRecord**

**{**

**private String name = "";**

**private double unitPrice = 0.0;**

**public InventoryRecord( String itemName, double price){**

**name = itemName;**

**unitPrice = price;**

**}**

**public String getItemName( ){ return name; }**

**public double getUnitPrice( ){ return unitPrice;}**

**public String getInventoryRecord( ){ return name + unitPrice; } // toString( );**

**}**

// SequentialSearchMethod1B.java

// Implements Search Method 1 where the desired key may appear more than one time in the table.

// Better but still not professional grade. Uses class InventoryRecord.java

import java.util.Scanner;

public class SequentialSearchMethod1B

{

public static void main( String args[ ])

{

Scanner input = new Scanner( System.in );

**String nameType[ ] = { "kites", "pencils", "paper", "pens", "kites" };**

**double unitPriceValues [ ] = { 2.39, 16.78, 0.59, 10.00, 2.39 };**

String schKey = new String("go");

**InventoryRecord inventory[ ] = new InventoryRecord[ nameType.length ];**

**for( int K = 0; K < inventory.length; K++)**

**inventory[K] = new InventoryRecord( nameType[K], unitPriceValues[K] );**

System.out.printf("%-10s%12s\n", "Item", "Price");

for( int bin = 0; bin < **inventory.length**; bin++)

System.out.printf("%-10s %10.2f\n", **inventory[bin].getItemName( )**,

**inventory[bin].getUnitPrice( )** );

System.out.printf("\n%s", "Enter search key or 'Stop': ");

**schKey = input.next( );**

**while ( !(schKey.equals("Stop")) ){**

**for( int bin = 0; bin < inventory.length; bin++)**

**if ( schKey.equalsIgnoreCase( inventory[bin].getItemName( ) ) )**

**System.out.printf("%2d %-10s %10.2f\n", bin,**

**inventory[bin].getItemName( ), inventory[bin].getUnitPrice( ));**

**//end if**

**//end for**

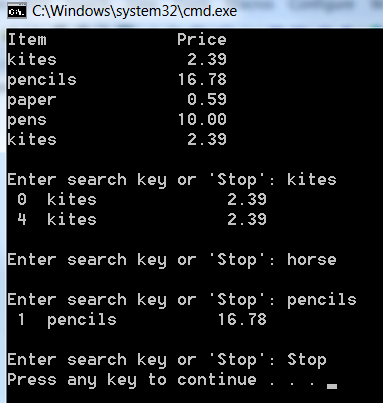
**System.out.printf("\n%s", "Enter search key or 'Stop': ");**

**schKey = input.next();**

**}** //end while

} // end main

}// end SequentialSearchMethod1B



The primary problem with this implementation is the inventory table is initialized with fixed values in the code. During the work day one would expect some items to be dropped from inventory, additional items to be added to inventory, and possibly price changes. Most commercial programs must reflect this dynamic behavior from one day to the next. This ability to remember dynamic behavior from one execution of the program to the next is called “persistence.” Persistence is normally accomplished by reading starting values from a file at the start of the work period. When the work period is over, the current values are saved to a file to initialize the program for the next work period.

Persistence may be achieved with sequential files, random access files, or a database. Databases and random access files are typically utilize to update values on a per transaction basis. This represents substantial overhead compared to sequential files where the inventory table is updated multiple times during the day but only written out occasionally or at the end of the work period. The downside is loss of dynamic data in the program upon a power, hardware, or program error.

**Lists Ordered by Frequency of Use**

Assume a list of N items is maintained in random order and searched sequentially. The minimum number of probes to find a random item in the list is 1 and the maximum number of probes is N. If all items in the list have an equal probability of reference, then the expected number of probes to find a random item is:

Eprobes = 1p1 + 2p2 + 3p3 + °°° + NpN

where pi is the probability that the ith object is the item desired. We have assumed equal probabilities, hence pi = 1 / N for all i. This implies

Eprobes = (1 + 2 + 3 + °°°+ N) \* (1/N) = (N\*(N+1)/2) \* (1/N) = (N + 1) / 2.

For large values of N, (N+1) is approximately N, hence this reduces to: Eprobes = N / 2.

W. P. Hesing reported in IBM Systems Journal {2(1963), p114-115} that many commercial applications obey an “80-20” rule. Eighty percent of the transactions are restricted to twenty percent of the stock, and the same rule may be applied recursively to transactions on the most active twenty percent of stock. In other words 64% of the transactions are restricted to the most active 4% of stock. (A Pareto distribution is also used to explain this phoneme.) If a list is ordered in descending order based on frequency of use and searched sequentially, then the expected number of probes reduces to approximately Eprobes = 0.122N, which represents about a factor of four improvement over the (N+1)/2 probes required for random permutations.

List ordered by frequency of use may exhibit better search performance than the binary search and other search techniques. Note that while the list order by frequency of reference may require more average probes to locate an item than a binary search it may still be faster. If the current item is not the desired item in a sequential search, you only need to look in the next position. In a binary search you must decide which half of the list to eliminate then calculate a new mid point in the remaining list. Hence the overhead per probe in a binary search is higher than the ovehead per probe in a sequential search.

Example: Assume 8 items with frequencies of reference = 80%, 14%, 1%, 1%, 1%, 1%, 1%, and 1%. Then Eprobes = 1(0.8) + 2(0.14) + 3(0.01) + 4(0.1) + 5(0.01) + 6(0.01) + 7(0.01) + 8(0.01) = 0.8 + 0.28 + 0.03 + 0.04 + 0.05 + 0.06 + 0.07 + 0.08 = 1.41. Hence the expected number of probes for a random item in the list is 1.41. Eighty percent of request are satisfied in one probe and 94% of requests in two probes. A binary search would require 3 probes and sequential search 4.5 probes.

**“C/C++” implementation of search methods for comparison.**

//file SequentialSearch.cpp

#include<iostream.h>

#include<string.h>

struct inventoryItem {

char item[20];

double price;

};

int seqSchMethod1( struct inventoryItem p1[ ], char key[ ], int numItems );

int seqSchMethod2( struct inventoryItem p1[ ], char key[ ], int numItems );

int seqSchMethod3( struct inventoryItem p1[ ], char key[ ], int numItems );

void main() {

struct inventoryItem invTable[ 6 ] = {

{ "kites", 2.39},

{ "pencils", 16.78},

{ "paper", 0.59},

{ "pens", 10.00},

{ "tops", 25.10}

};

char itemName[20];

int pos;

// Number of items in table allowing for slack bytes minus 1

// reserved for search method #3.

int tableSize = sizeof(invTable)/sizeof(struct inventoryItem) - 1;

do {

cout << "Enter the item name, '.' to stop: " ; cin >> itemName;

if(\*itemName != '.') {

pos = seqSchMethod1( invTable, itemName, tableSize );

cout << itemName << " appears in the table " << pos << " times.\n\n";

pos = seqSchMethod2( invTable, itemName, tableSize );

if (pos < tableSize) { //case found

cout << invTable[pos].item << " found using search method #2 at location "

<< pos << ".\n\n";

}

else // not found

cout << "\nThe item code " << itemName << " does not exist.\n\n\n";

pos = seqSchMethod3( invTable, itemName, tableSize );

if(pos < tableSize)

cout << "Found using method 3 in location " << pos << ".\n\n";

else

cout << "Not found by method 3\n\n";

} //end if

} //end do

while(\*itemName != '.'); // loop until the user enters a '.'

cout << "Thats all folks!\n" ;

}

// We assume the item may appear in the table more than once. This

// functions counts (returns) how often the key appears in the table.

int seqSchMethod1( struct inventoryItem p1[ ], char key[ ], int numItems ){

int count = 0;

for(int i = 0; i < numItems; i++)

if( !strcmp( p1[ i ].item, key) ) count++;

return count;

};

// Search table p1 for "key" and report its position if found.

// Return table size if the key is not in the table. We assume

// the item does not appear in the table more than once. The expected

// number of probes is (N+1) / 2, hence overhead is approximately

// 2 \* [(N+1) / 2] = N+1 for the number of compares to prevent searching

// past the end of the table and compare to see if desired item.

int seqSchMethod2( struct inventoryItem p1[ ] , char key[ ], int numItems ){

for(int i = 0;

i < numItems // Test for end of table

&& strcmp( p1[ i ].item, key); // Desired key?

i++

);

return i;

}

// Fastest sequential search is item appears at most one time. Note the

// expected numbe of compare to see if this is the desired item is (N+1)/2

// plus 1 compare to see if we went past the end of the table (done back

// in the main program) plus copying the item we are looking for into the

// last entry of the table prior to the search. Search method 3 makes

// approximately 50% fewer compares than search method 2 and runs about 20%

// faster when all loop control overhead is taken into consideration.

int seqSchMethod3( struct inventoryItem p1[ ], char key[ ], int numItems ){

strcpy(p1[numItems].item, key);

for(int i = 0; strcmp( p1[ i ].item, key); i++) ;

return i;

}

Search Method Summary

We assume a sequentially allocated table containing N records each having an equal probability of being the desired record unless otherwise stated.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Search Method | Min Probes | Expected Probes | Maximum Probes | Not in the Table | Comment |
| Sequential Search Method 1 | 1 | N | N | N | The item may appear more than once in the table, list need not be ordered. |
| Sequential Search Method 2 | 1 | (N+1)/2 | N | N – less if table sorted | Does not require additional space, list need not be ordered.. |
| Sequential Search Method 3 | 1 | (N+1)/2 | N | N | 20% to 30% faster that search method 2, prefers larger tables, waste one unit of space in the table, list need not be ordered.. |
| Sequential search of list in descending order of frequency of use. |  |  |  |  | Obeys Hesing 80/20 rule: 80% of sales are restricted to 20% of inventory, apply recursively. |
| Binary Search | 1 | ≈ log2 N,  Approaches maximum | ≈ log2 N | ≈ log2 N | Table must be stable and in sorted order, doubling the size of the table only increase the maximum number of probes by 1 probe. |
| Interpolation Search | 1 | LogLog N | LogLog N |  | Search keys evenly (uniformly) distributed. |
| Binary Search Tree |  | ≈ log2 N,  Approaches maximum | ≈ log2 N,  Approaches maximum | ≈ log2 N,  Approaches maximum | Search depend on the trees balanced, random insertions and deletions improve balance, fast insertions and deletions, allows random search and sorted listings. |
| B-Tree |  |  | Levels searched <= 1 + log⎡m/2⎤ ((N+1)/2). |  | Better for auxiliary storage than most methods, works in main memory also. |
| Fibonacci |  |  |  |  | Slightly faster than binary under right circumstances but more complex.. |
| Hashing | 1 perfect hash | 1 perfect hash | 1 perfect hash | 1 perfect hash | Great for main memory and auxiliary storage. |

**Binary Search**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  | **Table** |
|  |  |  |  | **32** |  |  |  |  |  |  |  | **4** | **1** | **8** |
|  |  |  |  | **4** |  |  |  |  |  |  |  |  | **2** | **12** |
|  |  |  |  |  |  |  |  |  |  |  |  |  | **3** | **15** |
|  | **12** |  |  |  |  |  | **47** |  |  |  |  | **+/-2** | **4** | **32** |
|  | **2** |  |  |  |  |  | **6** |  |  |  |  |  | **5** | **33** |
|  |  |  |  |  |  |  |  |  |  |  |  |  | **6** | **47** |
| **8** |  | **15** |  |  |  | **33** |  | **56** |  |  |  | **+/-1** | **7** | **56** |
| **1** |  | **3** |  |  |  | **5** |  | **7** |  |  |  |  | **8** | **72** |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  | **72** |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  | **8** |  |  |  |  |  |

**Algorithm B1 (Binary Search):**

Assume an array of N records R1, R2, ..., Rn whose keys are in increasing order K1 < K2 < ... < Kn. This algorithm searches for a record with key Key. Let LB mean lower bound of the current table and UB mean the upper bound of the current table.

Set LB := 1; UB := N;

loop

exit when UB < LB //Terminate, the search is unsuccessful.

I := ⎣ (LB + UB) / 2 ⎦; // Use of floor operator “ ⎣ (LB + UB) / 2 ⎦”

if Key < KI then

UB := I - 1;

elsif Key > KI then

LB := I + 1;

else -- note: Key = KI  then

Process the record;

Exit;

end if;

end loop;

Note that each time you move right of left in a balanced tree you eliminate ½ the remaining possibilities. Hence the maximum number of levels for a tree containing N keys is approximately log2 N. Assume all keys have an equal probability of being the next desired key on a search. The minimum number of probes to find a random key is 1. The maximum number of probes to locate a key is the number of level is in the tree, approximately log2 N. The average must be between the minimum and maximum. Looking at a few trees we quickly discover most of the keys are located either on the bottom level or preceding level. **This implies the average (expected) number of probes is almost identical to the maximum, i.e., O(log2 N). The log2 1024 = 10. The log2 2048 = 11. If you double the size of the table the maximum number of probes to find a random key increases by 1. Unfortunately the average (expected) number of probes is almost the same as the maximum number of probes. Look at the symmetry of the tree. The lowest level has the most nodes in a balance tree, the level above it the next most nodes, etcetera.**

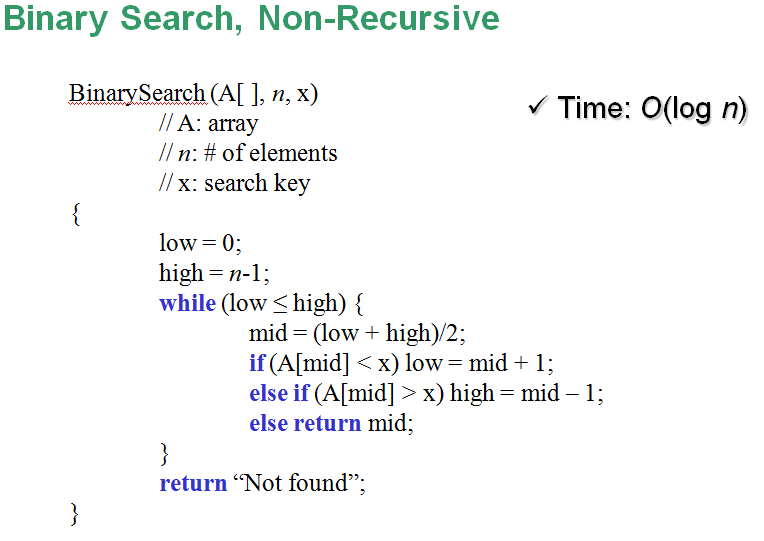
**Note a sequential search of a table consisting of about 64 items may be faster than a binary search. The sequential search will make approximately (N+1)/2 probes or 32.5 probes. A binary search would only make log2 N or 8 probes. But the cost per probe is higher.** In a sequential search, we need only add a constant offset to reach the next search location. For the binary tree we must first decide which half to throw away. Then we must move boundaries and calculate a new mid point. The cost per probe for the binary search is substantially higher than for the sequential search. Put another way, as the expected number of probes in a sequential search increases, it will eventually exceed the overhead experienced by the binary search which makes far fewer probes in a large table. As a rule of thumb, a sequential search is faster for tables with fewer than 50 items and a binary search faster for larger tables.

**To make the binary search run faster we must reduce the overhead.** Consider 8 / 2. The divide instruction on most computers is expensive. Even with a hardware assist it is up to 30 times slower than shift and logical instructions (“and” and “or”) on most computers. Note if we shift the binary 8 in base 10 right by one bit in base 2, the result is 4, e.g. 810 = 10002 shifted right one bit is 01002 = 410. The expression A/256 can be accomplished by a right shift of 8 bits as 256 = 28. Hence dividing (right shift) and multiply (left shift) by numbers that are a power of two using shift operations is up to 30 times faster that traditional divide and multiply operations. Using a shift instruction to replace the divide operation in the binary search will make it run substantially faster.

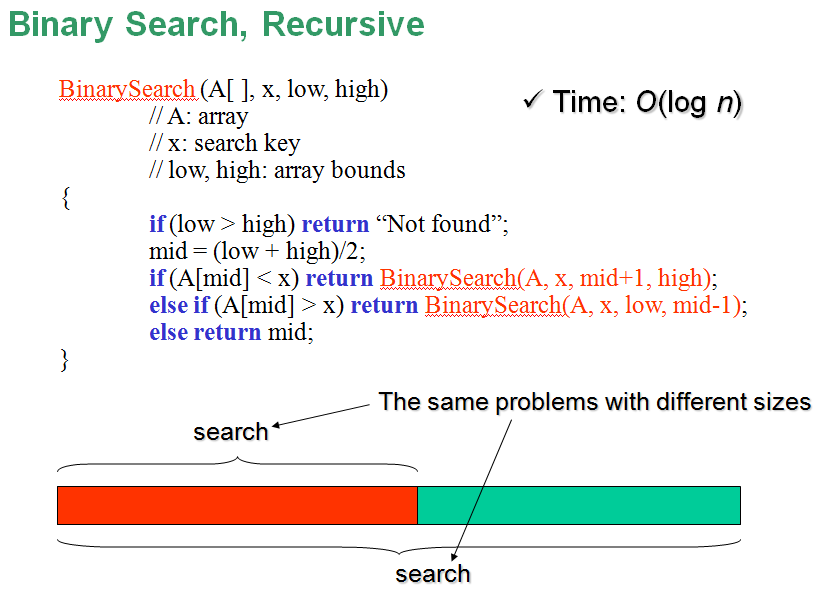
Over head can be further reduce eliminating tracking of the exact position of the base and bounds as shown below in algorithm B2.

Binary searches are most effective when the contents of the table is static as in the names of state. Consider the overhead associated with frequent random insertions and deletions and the restriction of keeping the table in sorted order.

Multiple approaches exist to implement a binary search. The method shown is the most widely used and also the slowest, least professional method to implement the search method. As an example, on most computers logical instructions, shit, add, and subtract require one machine cycle. Multiply typically requires closer to 25 machine cycles and divide typically more than 30 machine cycles. It is possible to replace the divide by 2 in the above algorithm with a shift-right-arithmatic requiring a single machine cycle. The savings are on the order of 35:1. Additional optimizations are possible.



Complements of Dr. Cho.



Complements of Dr. Cho.

**Sample Java Code to Manipulate Tables**

import java.awt.Graphics;

import java.applet.Applet;

public class Array1 extends Applet{

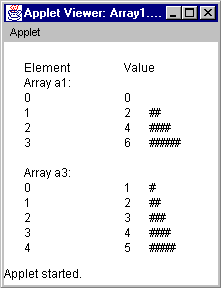
**int a1[ ], a2[ ];** // Declare pointers to array objects.

**final** int **ArraySize = 10;** // Declare a constant, must give value.

**int a3[ ] = {1, 2, 3, 4, 5};** //Declare an array of 5 elements

public void init( ){

**a1 = new int[ 4 ];** // Allocate a1 to hold 4 integers.



**a2 = new int[ ArraySize ];**

// Allocate 10 integers.

for(int j = 0; j < **a1.length**; j++) **a1[ j ] = 2 \* j;**

}

public void paint( Graphics gc ){

int xPos = 20, yPos = 30;

gc.drawString("Element", xPos, yPos);

gc.drawString("Value", xPos + 100, yPos);

gc.drawString("Array a1:", xPos, yPos += 15);

for(int j = 0; j < a1.length; j++) {

yPos += 15;

c.drawString(String.valueOf( j) , xPos, yPos);

gc.drawString(**String.valueOf( a1[ j ] ),** xPos + 100, yPos);

**for(int k = 1; k <= a1[ j ]; k++)**

**gc.drawString("#", xPos + 120 + 5 \* k, yPos);**

}

gc.drawString("Array a3:", xPos, yPos += 30);

for(int j = 0; j < **a3.length**; j++) {

yPos += 15;

gc.drawString(String.valueOf(j), xPos, yPos);

gc.drawString(String.valueOf(a3[j]), xPos + 100, yPos);

for(int k = 1; k <= **a3[ j ];** k++)

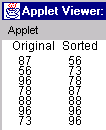
gc.drawString("#", xPos + 120 + 5 \* k, yPos);

}

}

} // Numeric array elements are automatically set to zero, Boolean to false, and references to null.

import java.applet.Applet;



import java.awt.Graphics;

**public class simpleBubbleSort extends Applet {**

**public void paint( Graphics g ){**

**int grades[ ]** = { 87, 56, 96, 78, 88, 96, 73 };

print( **g**, "Original", **grades**, 10, 10 );

bubbleSort( **grades** );

print( **g**, "Sorted", **grades**, 60, 10 );

**}**

**public void print( Graphics g, String message, int array[ ],**

**int row, int col ){**

g.drawString( message, row, col );

row += 6;

col += 15;

for (int pt = 0; pt < **array.length**; pt++ ) {

g.drawString( String.valueOf( **array[ pt ]** ), row, col );

col += 10;

}

**}**

**private void bubbleSort( int grades[ ] ){** *// Ascending order.*

boolean switched = true; *// Stop if nothing is switched during a pass.*

int temp; *// Maximum length – 1 passes.*

for( int pass = grades.length; ( pass > 0 ) && switched; pass--){

switched = false; *// At top nothing switched.*

for(int k = 0; k < (pass - 1); k++){

if(grades[ k ] > grades[ k+1 ] ){

temp = grades[ k ];

grades[ k ] = grades [k+1 ];

grades[ k+1 ] = temp;

switched = true;

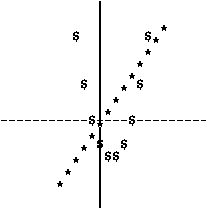
}

}

}

**}**

**}**



import java.applet.Applet;

import java.awt.Graphics;

import java.awt.Font;

**public class Graph1 extends Applet{**

**public void paint( Graphics g ) {**

int x1 = 0, x2 = 0;

**char paper[ ][ ] = new char[41][41];**

initializeGraphPaper( **paper** );

insertAxis( **paper** );

*// Draw parabola from x1 = -3 through x2 = 6.*

drawParabola( **paper** , -3, 6 );

// Draw line from x1 = -5 through x2 = 8.

drawLine( **paper** , -5, 8 );

printGraph( **paper** , **g** );

**}**

**void initializeGraphPaper( char p[ ][ ] ){**

for(int y = 0; y < **p.length**; y++)

for(int x = 0; x < **p[ y ].length**; x++) **p[ y ][ x ]** = ' ';

**}**

**void insertAxis( char p[ ][ ] ){ //**Translate axis center to (21, 21)

int x = 0, y = 0;

for(y = 0; y < **p.length**; y++) **p[ y ][ 21 ]** = '|';

y = 0;

for(x = 0; x < **p[ y ].length**; x++) **p[ 21 ] [ x ]** = '-';

**}**

**void drawParabola( char p[ ][ ] , int x1, int x2 ){**

int y;

// translate axis at (0,0) to (21, 21)

for(int x = x1; x <= x2; x++){

y = x \* x - 3 \* x - 4; **p[ 21 – y ][ x + 21 ]** = '$';

}

**}**

**void drawLine( char p[ ][ ] , int x1, int x2 ){**

int y; // translate axis at (0,0) to (21, 21)

for(int x = x1; x <= x2; x++){

y = 2 \* x - 1;

**p[ y + 21 ][ x + 21 ]** = '\*';

}

**}**

**void printGraph( char p[ ][ ] , Graphics g ){**

String aString;

int x = 0, y = 0, row = 8;

Font font1;

font1 = new Font( "Monospaced", Font.BOLD, 14 );

**g**.setFont( font1 );

for(y = 0, y < **p.length**; y >= 0; y++){

aString = new String(**p[ y ]** , 0, **p[ y ].length** );

**g**.drawString( aString, 6, row += 6 );

}

**}**

**}**

**import javax.swing.\*;**

**import java.awt.\*;**

**import java.awt.Graphics;**

**public class MyClass extends JApplet{**

**student aStudent = new student( "Joe", "123-45-6789", 84, 97, 76);**

**student gradeBook[ ] = new student[ 50 ]; //Array of pointers to students.**

**public void init( ){**

**System.out.println( aStudent.toString( ) );**

**gradeBook[0] = new student( ); // Create the students.**

**for(int k = 1; k < 50; k++) gradeBook[k] = new student( );**

**gradeBook[0].setName( "Tammy" ) ;**

**gradeBook[0].setSSAN( "987-65-4321" );**

**gradeBook[0].setGrade1( 89 );**

**gradeBook[0].setGrade2( 78 );**

**System.out.println( gradeBook[0].toString( ) );**

**}**

**public void paint( Graphics gc ){**

**gc.drawString( " " + gradeBook[0].toString( ), 20, 20 );**

**}**

**}**

**class student{**

**private String name, ssan;**

**private int g1, g2, g3;**

**//consturctors**

**public student(String n, String s, int t1, int t2, int t3){**

**name = n; ssan = s; g1 = t1; g2 = t2; g3 = t3;**

**}**

**public student(String n, String s){**

**name = n; ssan = s; g1 = 0; g2 = 0; g3 = 0;**

**}**

**public student( ){**

**name = ""; ssan = ""; g1 = 0; g2 = 0; g3 = 0;**

**}**

**//Mehtods**

**public void setName( String n ){ name = n; }**

**public void setSSAN( String s ){ ssan = s; }**

**public void setGrade1( int g ){ g1 = g; };**

**public void setGrade2( int g ){ g2 = g; }**

**public void setGrade3( int g ){ g3 = g; }**

**public String toString( ){**

**return name + " " + ssan + " " + g1 + " " + g2 + " " + g3;**

**}**

**public String getName( ){ return name; }**

**public String getSSAN( ){ return ssan; }**

**public int getGrade1( ){ return g1; }**

**public int getGrade2( ){ return g2; }**

**public int getGrade3( ){ return g3; }**

**}**

import javax.swing.\*;

import java.awt.\*;

import java.awt.Graphics;

public class MyClass extends JApplet{

student aStudent = new student("Joe", "123-45-6789", 84, 97, 76);

student gradeBook[ ] = new student[ 50 ]; //Array of pointers to students.

public void init( ){

System.out.println( aStudent.toString( ) );

gradeBook[0] = new student( ); // Create the students.

for(int k = 1; k < 50; k++) gradeBook[k] = new student( );

gradeBook[0].setName("Tammy");

gradeBook[0].setSSAN("987-65-4321");

gradeBook[0].setGrade1(89);

gradeBook[0].setGrade2(78);

System.out.println( gradeBook[0].toString( ) );

}

public void paint( Graphics gc ){

gc.drawString( " " + gradeBook[0].toString( ), 20, 20 );

}

}

class student{

private String name, ssan;

private int g1, g2, g3;

//consturctors

public student(String n, String s, int t1, int t2, int t3){

name = n; ssan = s; g1 = t1; g2 = t2; g3 = t3;

}

public student(String n, String s){

name = n; ssan = s; g1 = 0; g2 = 0; g3 = 0;

}

public student( ){

name = ""; ssan = ""; g1 = 0; g2 = 0; g3 = 0;

}

//Mehtods

public void setName( String n ){ name = n; }

public void setSSAN( String s ){ ssan = s; }

public void setGrade1( int g ){ g1 = g; };

public void setGrade2( int g ){ g2 = g; }

public void setGrade3( int g ){ g3 = g; }

public String toString( ){

return name + " " + ssan + " " + g1 + " " + g2 + " " + g3;

}

public String getName( ){ return name; }

public String getSSAN( ){ return ssan; }

public int getGrade1( ){ return g1; }

public int getGrade2( ){ return g2; }

public int getGrade3( ){ return g3; }

}