Chapter 19: Stacks, Queues, Lists, and Sorting

This chapter introduces a few advanced topics that are commonly covered in the first Computer Science class at the University level. The first three topics (Stack, Queue, and Linked List) are very common ways that information is stored in a computer system. The last two are algorithms for sorting information.

Stack:

A stack is one of the common data structures used by programmers to do many tasks. A stack works like the "discard pile" when you play the card game "crazy-eights". When you add a piece of data to a stack it is done on the top (called a "push") and these items stack upon each other. When you want a piece of information you take the top one off the stack and reveal the next one down (called a "pop"). Illustration 31 shows a graphical example.

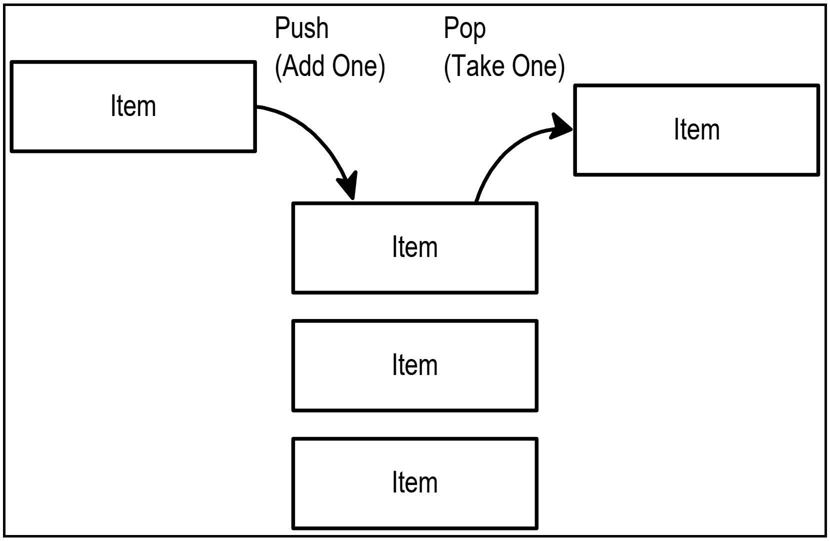


Illustration 31: What is a Stack

The operation of a stack can also be described as "last-in, first-out" or LIFO for short. The most recent item added will be the next item removed. Program 116 implements a stack using an array and a pointer to the most recently added item. In the "`push`" subroutine you will see array logic that will re-dimension the array to make sure there is enough room available in the stack for virtually any number of items to be added.

```
# stack.kbs
# implementing a stack using an array

dim stack(1) # array to hold stack with initial size
nstack = 0 # number of elements on stack
global stack, nstack

call push(1)
call push(2)
call push(3)
call push(4)
call push(5)

```

```
while not empty()
print pop()
end while

end

function empty()
# return true if the start is empty
return nstack=0
end function

function pop()
# get the top number from stack and return it
# or print a message and return -1
if nstack = 0 then
print "stack empty"
return -1
end if
nstack = nstack - 1
value = stack[nstack]
return value
end function

subroutine push(value)
# push the number in the variable value onto the stack
# make the stack larger if it is full
if nstack = stack[?] then redim stack(stack[?] + 5)
stack[nstack] = value
nstack = nstack + 1
end subroutine
```

Program 116: Stack

```
5
4
3
2
1
```

Sample Output 116: Stack


```
global variable
global variable, variable...
```

Global tells BASIC-256 that these variables can be seen by the entire program (both inside and outside the functions/subroutines). Using global variables is typically not encouraged, but when there is the need to share several values or arrays it may be appropriate.

Queue:

The queue (pronounced like the letter Q) is another very common data structure. The queue, in its simplest form, is like the lunch line at school. The first one in the line is the first one to get to eat. Illustration 32 shows a block diagram of a queue.

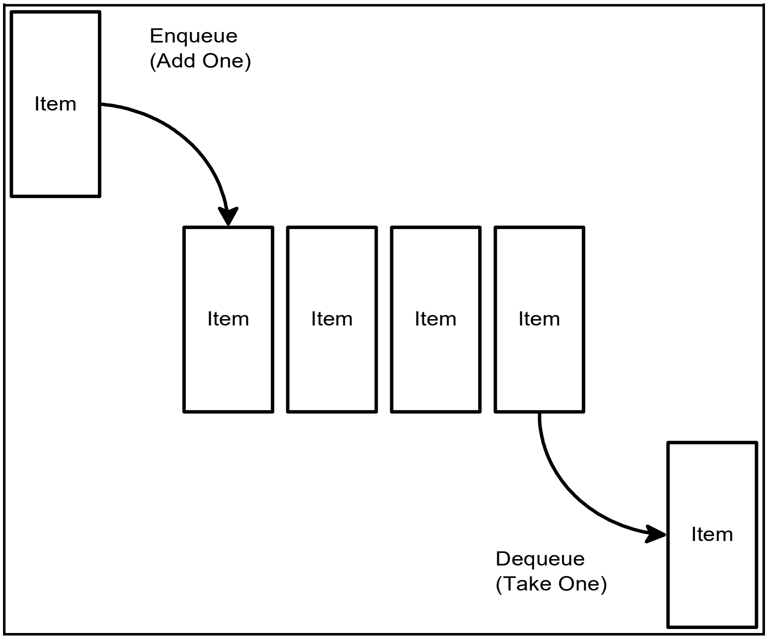


Illustration 32: What is a Queue

The terms enqueue (pronounced in-q) and dequeue (pronounced dee-q) are the names we use to describe adding a new item to the end of the line (tail) or removing an item from the front of the line (head). Sometimes this is described as a "first-in, first-out" or FIFO. The example in Program 117 uses an array and two pointers that keep track of the head of the line and the tail of the line.


```
# queue.kbs
# implementing a queue using an array

global queuesize, queue, queuetail, queuehead, inqueue

call createqueue(5)

call enqueue(1)
call enqueue(2)
```

```

print dequeue()
print

call enqueue(3)
call enqueue(4)

print dequeue()
print dequeue()
print

call enqueue(5)
call enqueue(6)
call enqueue(7)

# empty everybody from the queue
while inqueue > 0
   print dequeue()
end while

end

subroutine createqueue(z)
   # maximum number of entries in the queue at any one time
   queuesize = z
   # array to hold queue with initial size
   dim queue(z)
   # location in queue of next new entry
   queuetail = 0
   # location in queue of next entry to be returned (served)
   queuehead = 0
   # number of entries in queue
   inqueue = 0
end subroutine

function dequeue()
   if inqueue = 0 then
```

```
      print "queue is empty"
      value = -1
   else
      value = queue[queuehead]
      inqueue--
      queuehead++
      if queuehead = queuesize then queuehead = 0
   end if
   return value
end function

subroutine enqueue(value)
   if inqueue = queuesize then
      print "queue is full"
   else
      queue[queuetail] = value
      inqueue++
      queuetail++
      if queuetail = queuesize then queuetail = 0
   end if
end subroutine
```

Program 117: Queue

```
1

2
3

4
5
6
7
```

Sample Output 117: Queue

Linked List:

In most books the discussion of this material starts with the linked list. Because BASIC-256 handles memory differently than many other languages this discussion was saved after introducing stacks and queues.

A linked list is a sequence of nodes that contains data and a pointer or index to the next node in the list. In addition to the nodes with their information we also need a pointer to the first node. We call the first node the "Head". Take a look at Illustration 33 and you will see how each node points to another.

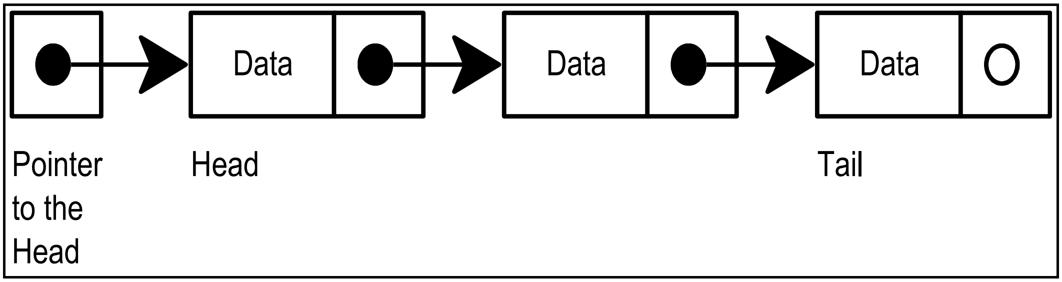


Illustration 33: Linked List

An advantage to the linked list, over an array, is the ease of inserting or deleting a node. To delete a node all you need to do is change the pointer on the previous node (Illustration 34) and release the discarded node so that it may be reused.

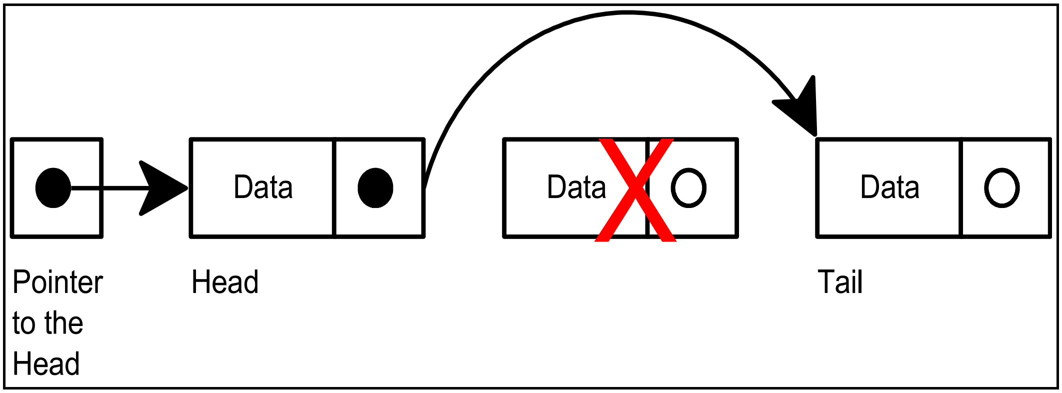


Illustration 34: Deleting an Item from a Linked List

Inserting a new node is also as simple as creating the new node, linking the new node to the next node, and linking the previous node to the first node. Illustration 35 Shows inserting a new node into the second position.

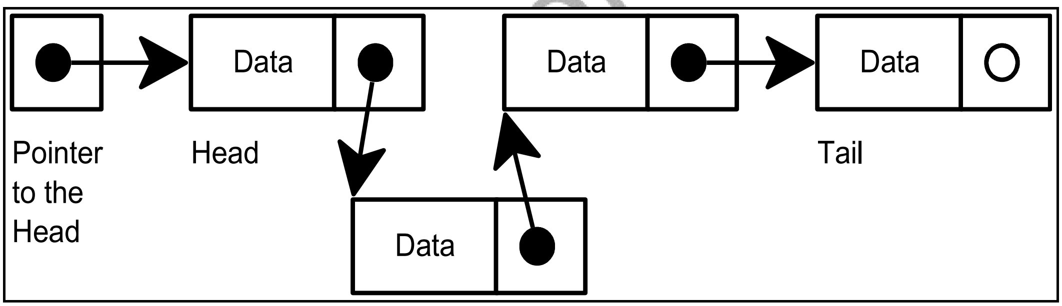


Illustration 35: Inserting an Item into a Linked List

Linked lists are commonly thought of as the simplest data structures. In the BASIC language we can't allocate memory like in most languages so we will simulate this behavior using arrays. In Program 118 we use the data array to store the text in the list, the nextitem array to contain the index to the next node, and the freeitem array to contain a stack of free (unused) array indexes.

```
# linkedlist.kbs

# create a linked list using arrays

# data is an array coitaining the data strings in the list
# nextitem is an array with pointers to the next data item
# if nextitem is -2 it is free or -1 it is the end

global head, data, nextitem
call initialize(6)

# list of 3 people
call append("Bob")
call append("Sue")
call append("Guido")
call displaylist()
call displayarrays()
call wait()

print "delete person 2"
call delete(2)
call displaylist()
call displayarrays()
call wait()

print "insert Mary into the front of the list (#1)"
call insert("Mary",1)
call displaylist()
call displayarrays()
call wait()

print "insert John at position 2"
call insert("John",2)
call displaylist()
call displayarrays()
call wait()

```

```
print "delete person 1"
call delete(1)
call displaylist()
call displayarrays()
call wait()

end

subroutine wait()
   input "press enter to continue> ",foo
   print
end subroutine

subroutine initialize(n)
   head = -1    # start of list (-1 pointer to nowhere)
   dim data(n)
   dim nextitem(n)
   # initialize items as free
   for t = 0 to data[?]-1
      call freeitem(t)
   next t
end subroutine

subroutine freeitem(i)
   # free element at array index i
   data[i] = ""
   nextitem[i] = -2
end subroutine

function findfree()
   # find a free item (an item pointing to -2)
   for t = 0 to data[?]-1
      if nextitem[t] = -2 then return t
   next t
   print 'no free elements to allocate'
   end
end function

```

```
function createitem(text)
   # create a new item on the list
   # and return index to new location
   i = findfree()
   data[i] = text
   nextitem[i] = -1
   return i
end function

subroutine displaylist()
   # showlist by following the linked list
   print "list..."
   k = 0
   i = head
   do
      k = k + 1
      print k + " ";
      print data[i]
      i = nextitem[i]
   until i = -1
end subroutine

subroutine displayarrays()
   # show data actually stored and how
   print "arrays..."
   for i = 0 to data[?]-1
      print i + " " + data[i] + " >" + nextitem[i] ;
      if head = i then print " <<head";
      print
   next i
end subroutine

subroutine insert(text, n)
   # insert text at position n
   index = createitem(text)
   if n = 1 then
      nextitem[index] = head
      head = index
   else
```

```
      k = 2
      i = head
      while i <> -1 and k <> n
         k = k + 1
         i = nextitem[i]
      end while
      if i <> -1 then
         nextitem[index] = nextitem[i]
         nextitem[i] = index
      else
         print "can't insert beyond end of list"
      end if
   end if
end subroutine

subroutine delete(n)
   # delete element n from linked list
   if n = 1 then
      # delete head - make second element the new head
      index = head
      head = nextitem[index]
      call freeitem(index)
   else
      k = 2
      i = head
      while i <> -1 and k <> n
         k = k + 1
         i = nextitem[i]
      end while
      if i <> -1 then
         index = nextitem[i]
         nextitem[i] = nextitem[nextitem[i]]
         call freeitem(index)
      else
         print "can't delete beyond end of list"
      end if
   end if
end subroutine
```

```

subroutine append(text)
   # append  text to end of linked list
   index = createitem(text)
   if head = -1 then
      # no head yet - make item the head
      head = index
   else
      # move to the end of the list and add new item
      i = head
      while nextitem[i] <> -1
         i = nextitem[i]
      end while
      nextitem[i] = index
   endif
end subroutine
```

Program 118: Linked List

Explore	Re-write Program 118 to implement a stack and a queue using a linked list.

Slow and Inefficient Sort - Bubble Sort:

The "Bubble Sort" is probably the worst algorithm ever devised to sort a list of values. It is very slow and inefficient except for small sets of items. This is a classic example of a bad algorithm.

The only real positive thing that can be said about this algorithm is that it is simple to explain and to implement. Illustration 36 shows a flow-chart of the algorithm. The bubble sort goes through the array over and over again swapping the order of adjacent items until the sort is complete,

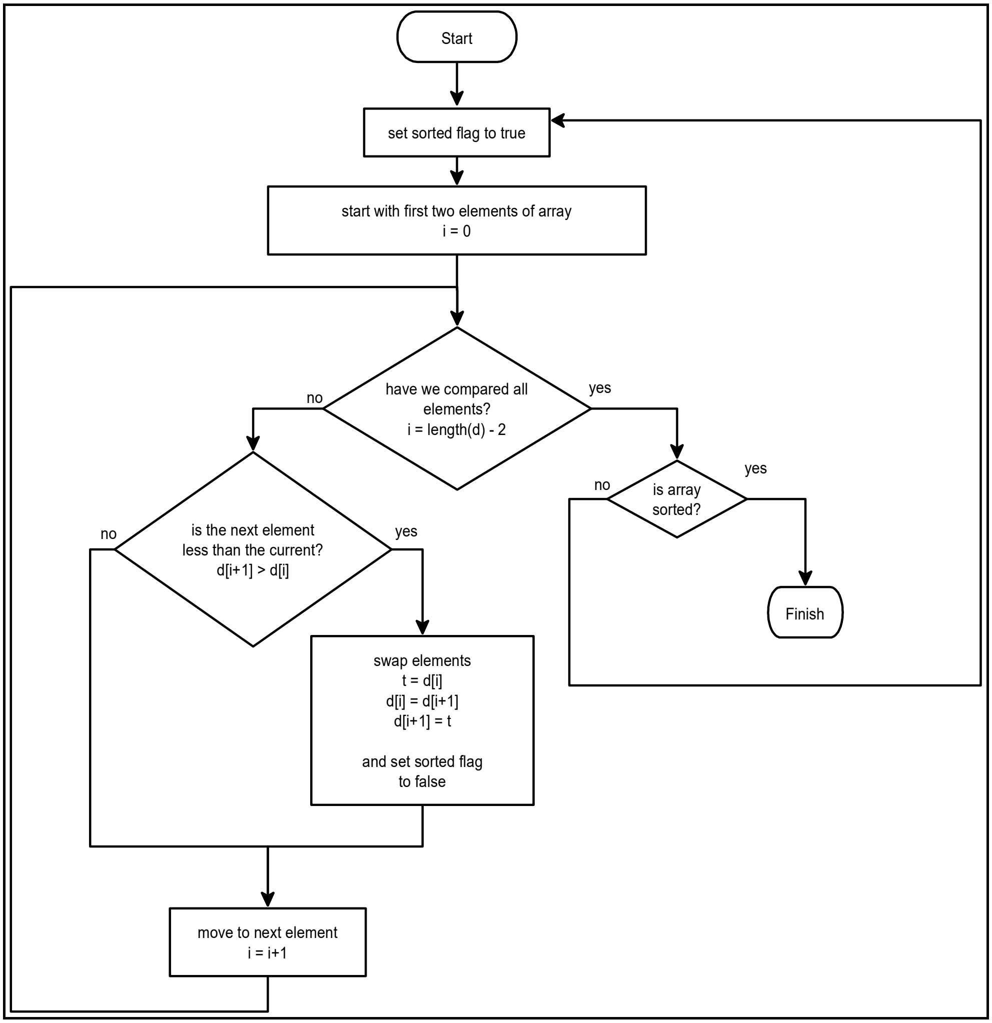


Illustration 36: Bubble Sort - Flowchart

```
# bubblesortf.kbs
# implementing a simple sort

# a bubble sort is one of the SLOWEST algorithms
```

```
# for sorting but it is the easiest to implement
# and understand.
#
# The algorithm for a bubble sort is
# 1. Go through the array swaping adjacent values
#    so that lower value comes first.
# 2. Do step 1 over and over until there have
#    been no swaps (the array is sorted)
#

dim d(20)

# fill array with unsorted numbers
for i = 0 to d[?]-1
   d[i] = int(rand * 1000)
next i

print "*** Un-Sorted ***"

call displayarray(ref(d))
call bubblesort(ref(d))

print "*** Sorted ***"
call displayarray(ref(d))
end

subroutine displayarray(ref(array))
   # print out the array's values
   for i = 0 to array[?]-1
      print array[i] + " ";
   next i
   print
end subroutine

subroutine bubblesort(ref(array))
   do
      sorted = true
      for i = 0 to array[?] - 2
         if array[i] > array[i+1] then
```

```
            sorted = false
            temp = array[i+1]
            array[i+1] = array[i]
            array[i] = temp
         end if
      next i
   until sorted
end subroutine
```

Program 119: Bubble Sort

```
*** Un-Sorted ***
878 95 746 345 750 232 355 472 649 678 758 424
653 698 482 154 91 69 895 414
*** Sorted ***
69 91 95 154 232 345 355 414 424 472 482 649
653 678 698 746 750 758 878 895
```

Sample Output 119: Bubble Sort

Better Sort – Insertion Sort:

The insertion sort is another algorithm for sorting a list of items. It is usually faster than the bubble sort, but in the worst case case could take as long.

The insertion sort gets it's name from how it works. The sort goes through the elements of the array (index = 1 to length -1) and inserts the value in the correct location in the previous array elements. Illustration 37 shows a step-by-step example.

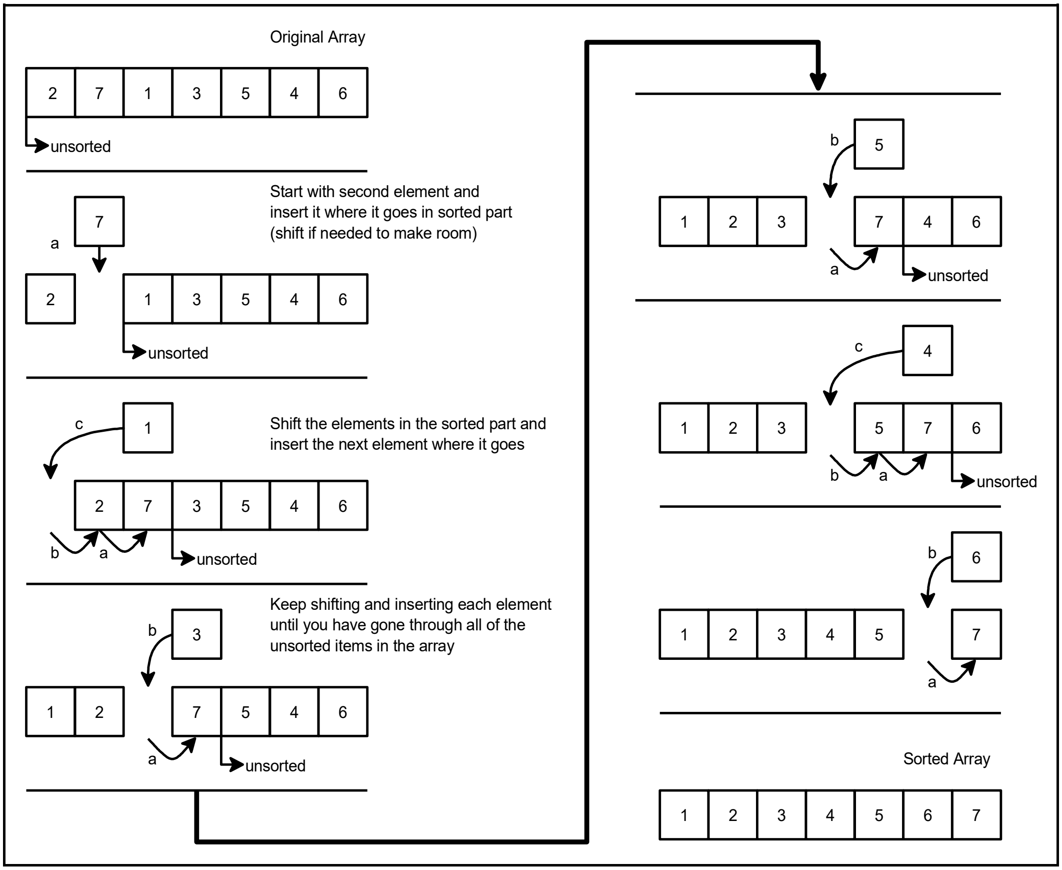


Illustration 37: Insertion Sort - Step-by-step

```
# insertionsort.kbs
# implementing an efficient sort

# The insertion sort loops through the items
# starting at the second element.

# takes current element and inserts it
# in the the correct sorted place in
# the previously sorted elements

# moving from backward from the current
# location and sliding elements with a
```

```
# larger value forward to make room for
# the current value in the correct
# place (in the partially sorted array)

dim d(20)

# fill array with unsorted numbers
for i = 0 to d[?]-1
   d[i] = int(rand * 1000)
next i

print "*** Un-Sorted ***"
call displayarray(ref(d))

call insertionsort(ref(d))

print "*** Sorted ***"
call displayarray(ref(d))
end

subroutine displayarray(ref(a))
   # print out the array's values
   for i = 0 to a[?]-1
      print a[i] + " ";
   next i
   print
end subroutine

subroutine insertionsort(ref(a))
   for i = 1 to a[?] - 1
      currentvalue = a[i]
      j = i - 1
      done = false
      do
         if a[j] > currentvalue then
            a[j+1] = a[j]
            j = j - 1
            if j < 0 then done = true
         else
```

```
                done = true
            endif
        until done
        a[j+1] = currentvalue
    next i
end subroutine
```

Program 120: Insertion Sort

```
*** Un-Sorted ***
913 401 178 844 574 289 583 806 332 835 439 52
140 802 365 972 898 737 297 65
*** Sorted ***
52 65 140 178 289 297 332 365 401 439 574 583
737 802 806 835 844 898 913 972
```

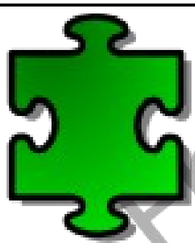
Sample Output 120: Insertion Sort

Exercises:

Word Search

```
k f i f o e q i q h m t o
n o f i l u x q q y e r b
i h p v e o d t q y u o d
l m p u f d s r c t e s e
v o e k x v m o i s u n u
p g f c i l e s a i q o e
q l f a u h m e l l n i u
v o i t q s o l l i e t q
i b c s z u r b o d t r e
z a i v e p y b c s z e d
d l e y d j h u a r o s p
z y n g o v c b t y l n q
m x t s n y i t e i q i b
```

allocate, bubblesort, dequeue, efficient, enqueue, fifo, global, insertionsort, lifo, link, list, memory, node, pop, push, queue, stack



Problems

1. Rewrite the "Bubble Sort" function to sort strings, not numbers. Add a second true/false argument to make the sort case-sensitive/insensitive.

2. Implement the "Insertion Sort" using the linked-list functions so that items are moved logically and not physically moved.

3. Develop a function to do the "Merge Sort" (http://en.wikipedia.org/wiki/Merge_sort) on an array of numbers. Create arrays of random numbers of varying lengths ans sotrt them using the "Bubble Sort", the "Insertion Sort", and your new "Merge Sort". Which is the slowest? Fastest?