Laboratory 4
Data Structures Representation

Computer Science 240

One-dimensional arrays of numerical values

Different languages use different implementations at the machine level to represent data structures.

In Java, arrays are actually implemented as arrays of addresses (pointers) to the elements, which are stored elsewhere in memory (not necessarily in contiguous locations).

In C, the elements of the array are stored in a contiguous block, starting at the base address of the array (this is probably how you coded the lab assignment for today).

In the C model,

address of element in array = base address + element size * index

If the size of the element is limited to 1, 2, or 4 bytes, what is another more efficient way to accomplish the multiplication?

Using the C model, you could use the following MIPS declarations to define an array of 7 bytes:

```
          .data
elements: .word     7  # number of elements in the array
          .word     1  # size of element in the array
          .byte     1,5,19,22,4,7,3
```

A procedure for accessing an element of the array would simply implement the calculation given above.

One-dimensional arrays of strings

For lab assignment, strings were same length (so access elements just like a one-dimensional array of numbers)

How to represent strings of variable length, such as (in Java):

```
String[] words = {'I','do','not','like','green','eggs','and','ham'}
```

1. Variable-length strings prefixed by bytes describing length, stored contiguously in memory.

2. Variable-length null-terminated strings stored (not necessarily contiguously) in memory, where the array contains pointers to (addresses of) the strings.

You will try both of these techniques in lab today.

Two-dimensional arrays

In Java, array of addresses of arrays

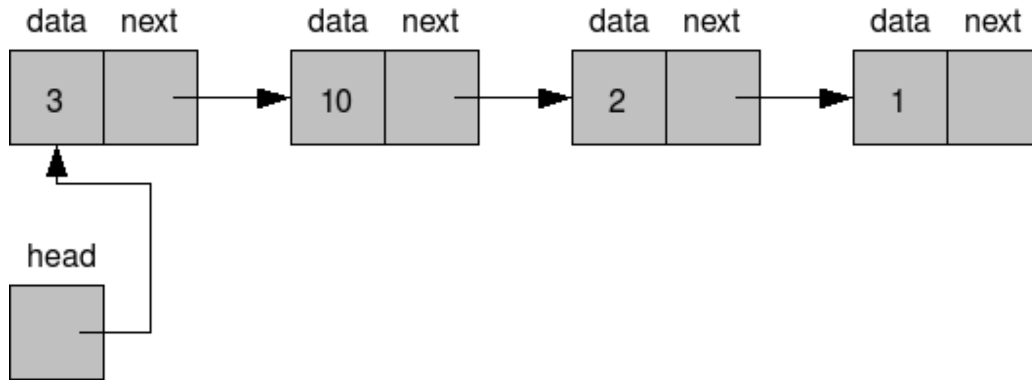
In C, nested array of arrays are used, where each row is stored contiguously in memory (*row-major* format), and the address of an element can be calculated by the following formula:

```
address of element[x][y] =
               base address of array +
               (x * number of columns * size of element) +
               (y * size of element)

          -or-

               base address of array +
               (x*columns + y)*size of element
```

Linked Lists



In MIPS, heap used for dynamic data,starts at address

Allocated with a syscall 9

Lab today: given Java code and the corresponding MIPS code (with one procedure missing) for a List Node

List Node is a word-aligned 8-byte memory block that holds:
- at offset 0: the address of the next node in the list
- at offset 4: an integer element

In MIPS:

```
(head)  0x10040000: .word 0x10040008
                    .word 00 (not used, head node has no value)
                    .
                    .
                    .
(node1) 0x10040008: .word 0x10040030 (address of next node)
                    .word     03 (value)
                    .
                    .
                    .
(node2) 0x10040030: .word 10040084 (address of next node)
                    .word     10 (value)
                    .
                    .
                    .
(node3) 0x10040084: .word 10040220 (address of next node)
                    .word     2 (value)
                    .
                    .
                    .
(node4) 0x10040220: .word 10040300 (address of next node)
                    .word     1 (value)
```

LinkedList.java contains:

```
//add value to the end of the list starting with node head
add(ListNode head, int value)

//insert value at position index in the list starting with node head
insertIterative(ListNode head, int index, int value)

// print the elements of a list in order, separated by spaces.
show(ListNode head)
```

LinkedList.asm contains he MIPS implementations of *add(head, value)* and *show(head)*

In lab, you will Implement the `insert` operation by translating LinkedList.java's `insertIterative` method to MIPS.