Simbulan, John Rian Von M. BSIS – II

Assignment in DSA

Difference between ArrayList and LinkedList in Java

ArrayList and LinkedList both implements List interface and their methods and results are almost identical. However there are few differences between them which make one better over another depending on the requirement.

ArrayList Vs LinkedList

1) Search: ArrayList search operation is pretty fast compared to the LinkedList search operation. get(int index) in ArrayList gives the performance of O(1) while LinkedList performance is O(n).

Reason: ArrayList maintains index based system for its elements as it uses array data structure implicitly which makes it faster for searching an element in the list. On the other side LinkedList implements doubly linked list which requires the traversal through all the elements for searching an element.

2) Deletion: LinkedList remove operation gives O(1) performance while ArrayList gives variable performance: O(n) in worst case (while removing first element) and O(1) in best case (While removing last element).

Conclusion: LinkedList element deletion is faster compared to ArrayList.

Reason: LinkedList's each element maintains two pointers (addresses) which points to the both neighbor elements in the list. Hence removal only requires change in the pointer location in the two neighbor nodes (elements) of the node which is going to be removed. While In ArrayList all the elements need to be shifted to fill out the space created by removed element.

3) Inserts Performance: LinkedList add method gives O(1) performance while ArrayList gives O(n) in worst case. Reason is same as explained for remove.

4) Memory Overhead: ArrayList maintains indexes and element data while LinkedList maintains element data and two pointers for neighbor nodes hence the memory consumption is high in LinkedList comparatively.

There are few similarities between these classes which are as follows:

Both ArrayList and LinkedList are implementation of List interface.

They both maintain the elements insertion order which means while displaying ArrayList and LinkedList elements the result set would be having the same order in which the elements got inserted into the List.

Both these classes are non-synchronized and can be made synchronized explicitly by using Collections.synchronizedList method.

The iterator and listIterator returned by these classes are fail-fast (if list is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove or add methods, the iterator will throw a ConcurrentModificationException).

When to use LinkedList and when to use ArrayList?

1) As explained above the insert and remove operations give good performance (O(1)) in LinkedList compared to ArrayList(O(n)). Hence if there is a requirement of frequent addition and deletion in application then LinkedList is a best choice.

2) Search (get method) operations are fast in Arraylist (O(1)) but not in LinkedList (O(n)) so If there are less add and remove operations and more search operations requirement, ArrayList would be your best bet.

QUEUES

```
createQueue() // create an empty queue

destroyQueue() // destroys a queue

isEmpty() // determines whether a queue is empty

enqueue (newItem) // insert new item at back of queue

dequeue() // remove the front item of the queue

dequeue (frontItem) // retrieve the front item and then

                    //   remove it from the queue

getFront (frontItem) // retrieve the front item
```

Simple Application for a Queue: Reading a String of Characters

```
// read a string of characters from a single line

// and store them in a queue

aQueue.createQueue()

while (not end of line) {

  Read a new characer ch

  aQueue.enqueue(ch)

  } // end while
```

STACKS

A stack is used to provide temporary storage space for values. It is defined as a data structure which operates on a first in, last out basis. Its uses a single pointer (index) to keep track of the information in the stack.

The basic operations associated with a stack are, insert (push) an item onto the stack

remove (pop) an item from the stack

The following diagram shows an empty stack of four locations. It looks just like an array, and it has an index pointer pointing to the beginning of the stack (called the TOP of the stack).

```
+--------+
|        |  <------- Stack Pointer
+--------+
|        |
+--------+
|        |
+--------+
|        |
+--------+
```

Pushing items onto the stack

The stack pointer is considered to be pointing to a free (empty) slot in the stack. A push operation thus involves copying data into the empty slot of the stack, then adjusting the stack pointer to point to the next free slot.

```
Module Push
    stack[stack_pointer] = data;
    stack_pointer = stack_pointer + 1;
Module End
```

Lets push the value 45 onto the stack. [Note: The stack could hold any data type, not necessarily decimal numbers. We have used numbers for simplicity]. The stack now looks like

```
+--------+
|  45  |
+--------+
|      |  <------- Stack Pointer
+--------+
|      |
+--------+
|      |
+--------+
```

Note how the stack pointer has been adjusted to point to the next free location in the stack. [Note: for the time being we are ignoring certain problems. We will address these shortly!!!].

Removing items from the stack

To remove an item, first decrement (subtract 1 from) the stack pointer, then extract the data from that position in the stack.

```
Module Remove
    stack_pointer = stack_pointer - 1;
    data = stack[stack_pointer];
Module End
```

Time now to address the problems of the above implementation

There are a number of problems associated with the above routines for pushing and removing items.

the push module does not check to see if the stack is full

the remove module does not check to see if the stack is empty

There are a number of solutions to this problem. We shall present a simplified solution. We do not argue that it is the best, just that it is one of a possible number of solutions.

```
Comment: Assume that the array elements begin at 1
Comment: Assume that the maximum elements of the stack is MAX

Var: stack[MAX] : Integer;

Module Initialize
    stack_pointer = 1;
Module End

Module Push
    if stack_pointer >= MAX then
      return error
    else begin
      stack[stack_pointer] = data;
      stack_pointer = stack_pointer + 1;
    end
Module End

Module Remove
    if stack_pointer <= 1 then
      return error
    else begin
      stack_pointer = stack_pointer - 1;
      data = stack[stack_pointer];
    end
Module End
```