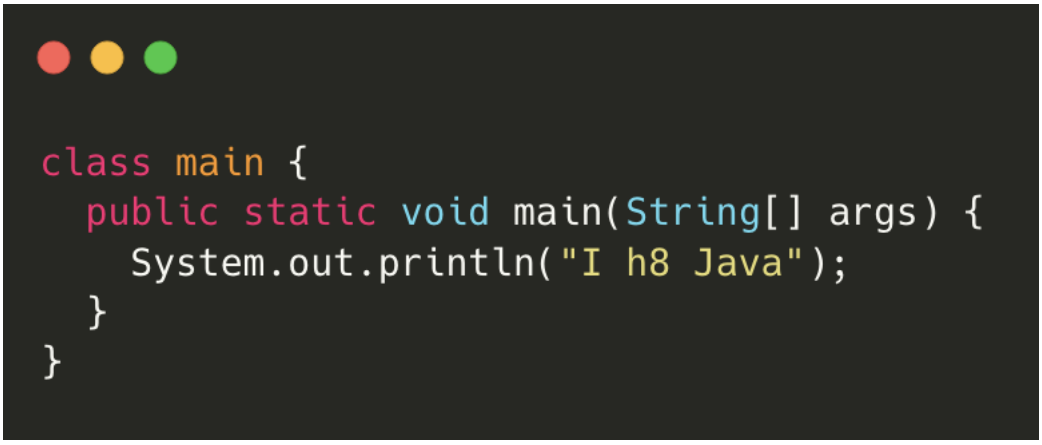
Introduction to Computer Science



by Louis Meunier

notes.louismeunier.net

Contents

Note that basic Java knowledge is assumed for these notes, so the first few chapters of this course are omitted for the sake of being concise.

1 UML Diagrams

UML Diagrams: ”Unified Modeling Language”, a set of standards for creating diagrams to represented object-oriented systems - see figure 1 for the basic layout, and figure 2 for a basic example.

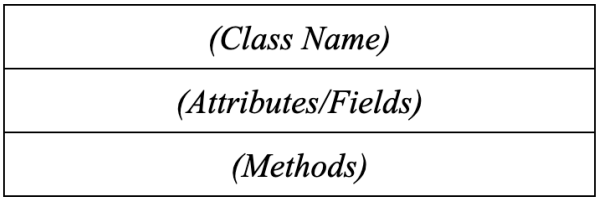


Figure 1: UML diagram layout

- A ”+” before a field indicates public
- A ”-” means private
- An underlined field means static

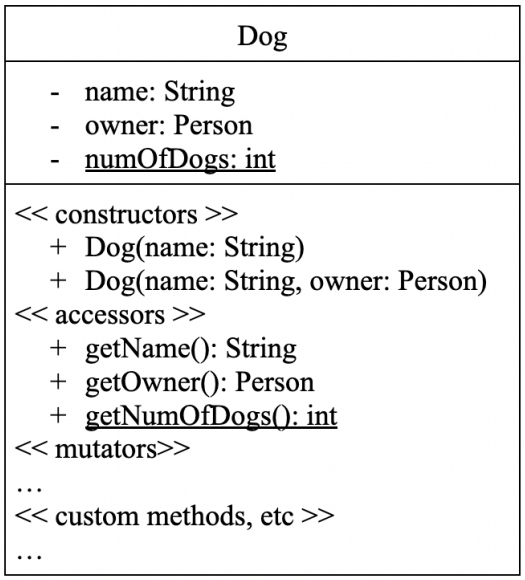


Figure 2: Example UML diagram

2 Basic Data Structures

2.1 Singly Linked Lists

Single Linked List: an object made up nodes with both links to the next item in the list, as well as a reference to an element. Figure 3 demonstrates this well.

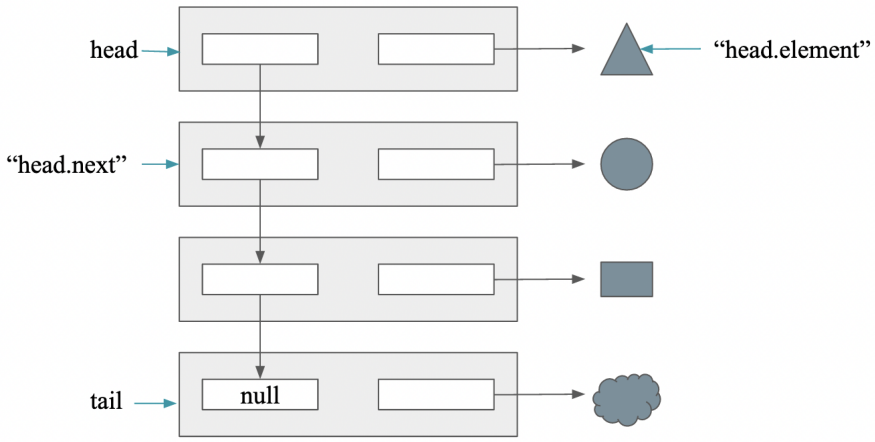


Figure 3: Singly Linked List

An implementation of a Linked List object should include both a head (first) and tail (last) node, as well as a field representing its size. Each node within the list should also include both an "element" field and a reference to the next node in the list.

Appropriate methods that should also be added to the list include *addFirst()*, *removeFirst()*, *addLast()*, and *removeLast()*. It should be noted that all these methods are able to be implemented in O(1) time complexity *except* removeLast(), which is possible in O(N) time. Hopefully this should be intuitive, as for this last method, you must iterate over the entire list until the second-to-last node, which must then be updated to set the last "element.next" to "null".

2.2 Doubly Linked Lists

Doubly Linked Lists: have all the same properties as singly linked lists, but also have a reference to the previous element in the list: see figure 4.

The 4 methods mentioned above for a singly linked list also apply to a doubly linked list, except the method *removeLast()* can be implemented in O(1). This should be intuitive: as you can access the second-to-last node and set its "next" to "null" in constant time by simply accessing "tail.prev".

It should be noted that even though the *time* complexity is better, the *space* complexity is worse; each node in a doubly-linked list actually contains 3 objects, while each node in a singly-linked list contains 2 (plus the object representing the list itself).

3 Sorting

Sorting algorithms are one of the most common algorithms used in programming, and their efficiency is important to understand. While some are clearly better than others in all circumstances, some are better when inputs are of a certain type/etc..

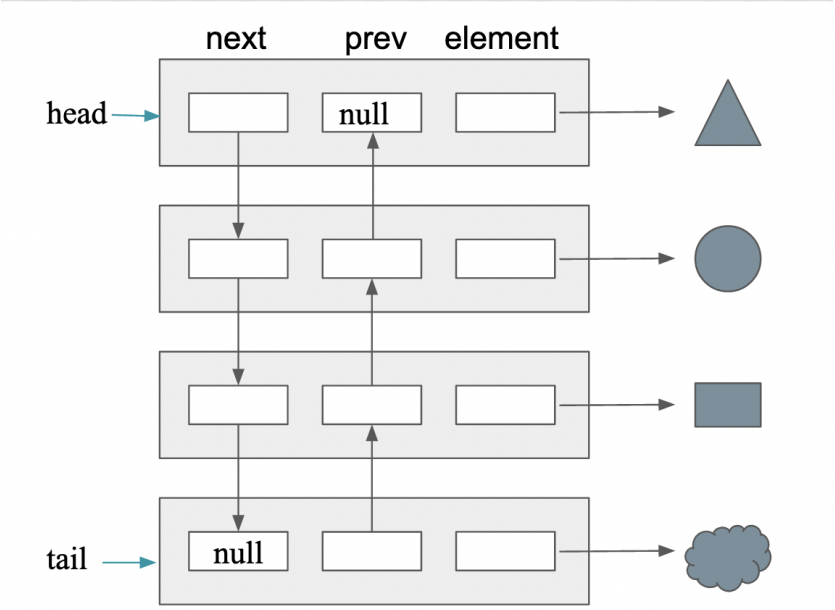


Figure 4: Doubly Linked List

3.1 $O(n^2)$: Quadratic Sorting

All of these algorithms sort elements in $O(n^2)$ time; ie, the time complexity is proportional to the size of the array squared.

3.1.1 Bubble Sort

Iterate through a list, and swap adjacent elements if they are in the wrong order. This is perhaps the "simplest" algorithm.

Pseudocode implementation:

```
sorted = false
i = 0
while (!sorted) {
    sorted = true
    for j from 0 to list.length - i - 2 {
        if (list[j] > list[j+1]) {
            swap(list[j], list[j+1])
            sorted = false
        }
    }
    i++
}
```

For an array $[5, 1, 4, 2, 8]$, the following shows the algorithm during the first iteration:

- $[1, 5, 4, 2, 8]$

- $[1, 4, 5, 2, 8]$
- $[1, 4, 2, 5, 8]$
- $[1, 4, 2, 5, 8]$
- etc..

This continues logically, until the array is sorted. It is important to realize that, when determining how many iterations it takes to sort the array, you have to take into account the last iteration after the array is sorted that the algorithm must take to ensure the array is actually sorted.

3.1.2 Selection Sort

- Consider the list in two parts: one that is sorted at the beginning, and one that is unsorted at the end.
- Select the smallest element in the unsorted part of the list
- Swap this element with the element in the initial position of the unsorted part
- Change the sorted/unsorted division in the array

Pseudocode:

```
// repeat until list is all sorted (~N)
for delim from 0 to N - 2 {
    // find the index of the min element in the unsorted section of the array
    min = delim
    for i from delim + 1 to N - 1 {
        if (list[i] < list[min]) {
            min = i
        }
    }
    // swap the min element with the first element of the unsorted section of the arr
    if (min != delim) {
        swap(list[min], list[delim])
    }
}
```

The following steps represent how this algorithm roughly works, where | represents the delimiter between sorted (toward the beginning) and unsorted (toward the end):

- $[|5, 1, 7, 2]$
- $[1|5, 7, 2]$
- $[1, 2|7, 5]$
- $[1, 2, 5|7]$

- $[1, 2, 5, 7|]$

3.1.3 Insertion Sort

- Consider the list in two parts: one that is sorted at the beginning, and one that is unsorted at the end.
- Select the first element of the unsorted part of the list
- Insert this element into the correct position of the sorted part of the list
- Change where the array is delimited between sorted/unsorted

Pseudocode:

```
// repeat until list is sorted (~N)
for i from 0 to N - 1 {
    // find where the next element in the unsorted portion should be inserted into th
    element = list[i]
    k = i
    while (k > 0 && element < list[k-1]) {
        list[k] = list[k - 1]
        k--
    }
    // insert the element into the sorted part of the list
    list[k] = element
}
```

Example steps:

- $[|5, 1, 7, 2]$
- $[5|1, 7, 2]$
- $[1, 5|7, 2]$
- $[1, 5, 7|2]$
- $[1, 2, 5, 7|]$