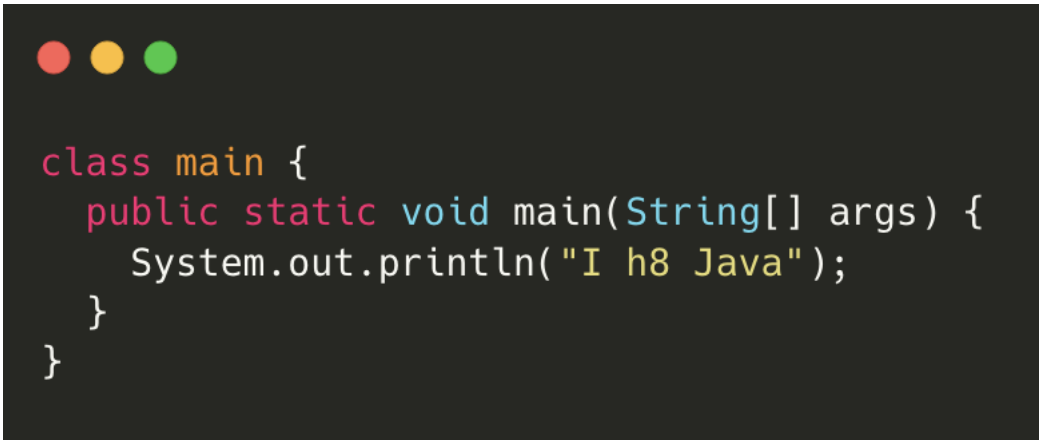
Introduction to Computer Science



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.

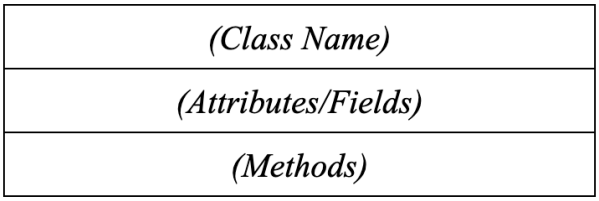


Figure 1: UML diagram layout

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

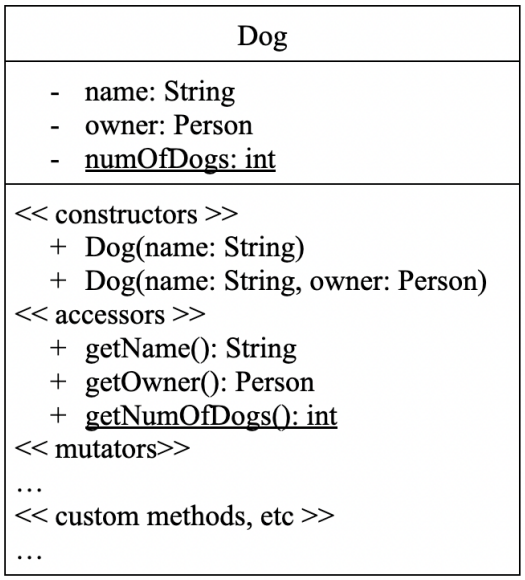


Figure 2: Example UML diagram

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.

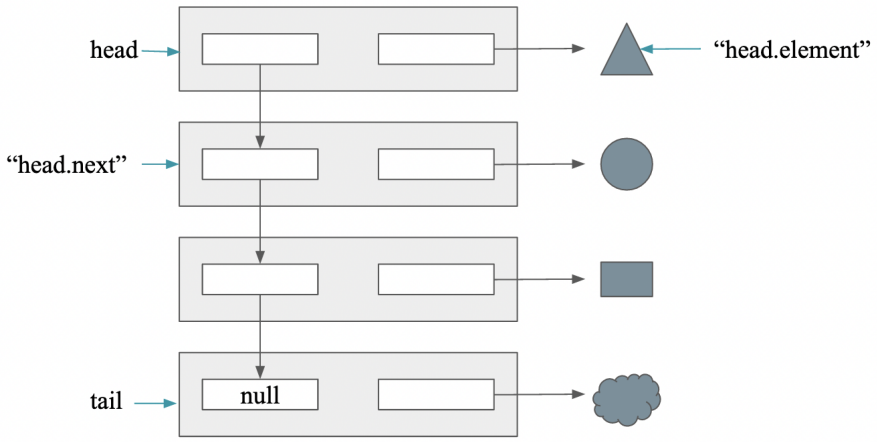


Figure 3: Singly Linked List

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).

2.3 Stacks

Stacks, as the name implies, are a data structure that follows the *LIFO* (Last In, First Out) principle. This means that the last element added to the stack is the first element removed from the stack.

2.4 Queues

Somewhat inverse to stacks, queues act like lines: the first element added to the queue is the first element removed from the queue. This is known as the *FIFO* (First In, First Out) principle. This can

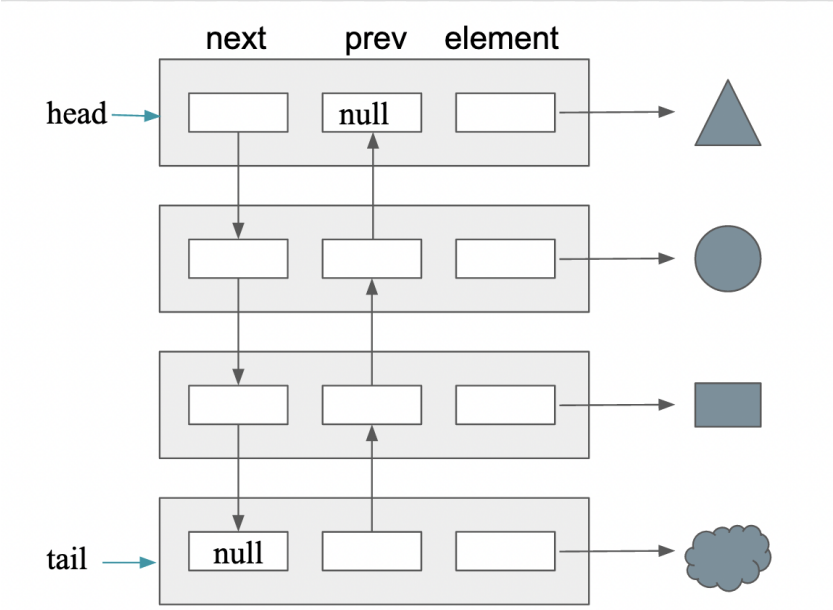


Figure 4: Doubly Linked List

be implemented in a number of ways, but one popular method is to use a circular array, which, as more elements are added, automatically expands.

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..

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 array
    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 the sorted part of the list and make space for it
    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|]

4 Asymptotic Notation

This section will cover a variety of notations used to analyze a particular function's efficiency.

4.1 Big-O Notation

Formally, given a function $g(n)$, $O(g(n))$ is defined as:

$$O(g(n)) = \{f(n) : \exists c, n_0 \in \mathbb{N} \text{ s.t. } 0 \leq f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0\} \tag{1}$$

In other words, $O(g(n))$ is the set of functions that are bounded above by a constant multiple of $g(n)$. *This is where the term asymptotic comes from, as the function is asymptotically bounded by* $g(n)$.

Graphically, see figure **??**.

To make more sense of this, take the function $5n + 70$.

Proof. Our goal is to find some c such that cn upper bounds $5n + 70$ when $n \geq n_0$; in others words, $5n + 70 \leq cn, n \geq n_0$.

Let's pick some n_0 that makes the upper bound easy to compute: $n_0 = 1$.

We can use our n_0 to see if c exists:

$$\begin{aligned} 5n + 70 &\leq cn, n \geq 1 \\ 5n + 70 &\leq 75n, n \geq 1 \\ \therefore c &= 75 \end{aligned} \tag{2}$$

Thus, there exists a n_0 and corresponding c such that $5n+70 \leq cn, n \geq n_0$, and therefore $5n+70 \in O(n)$. Clearly, we could have picked any n_0 and calculated the corresponding c, and this would have held. □

Note that because $O(n)$ is an upper bound, it therefore represents the worst case scenario for a function. Specifically, it represents a tight upper bound, since if a function is upper bounded by $O(n^2)$, it is also upper bounded by $O(n^3), O(n^4)$, etc.

4.2 Properties of Asymptotic Functions

- **Scaling:** For all constant $a > 0$, if $f(n)$ is $O(g(n))$, then $af(n)$ is also $O(g(n))$.
- **Sum Rule:** If $f_1(n)$ is $O(g(n))$ and $f_2(n)$ is $O(h(n))$, then $f_1(n) + f_2(n)$ is $O(g(n) + h(n))$.
- **Product Rule:** If $f_1(n)$ is $O(g(n))$ and $f_2(n)$ is $O(h(n))$, then $f_1(n) \cdot f_2(n)$ is $O(g(n) \cdot h(n))$.
- **Transitivity Rule:** If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$.

4.3 Big-Ω Notation

While Big-O notation is concerned with the *upper* bound of a function, Big-Ω notation is concerned with the *lower* bound of a function. More specifically, we can say that some function $f(n)$ is asymptotically lower bounded by $g(n)$ if there exists some n_0 such that for all n greater than or equal to n_0, then $f(n) \geq g(n)$.

More formally, we can say:

$$\Omega(g(n)) = \{f(n) : \exists c, n_0 \in \mathbb{N}, c, n_0 > 0, \text{ s.t. } f(n) \geq c \cdot g(n) \text{ for all } n \geq n_0\} \tag{3}$$

Notice that this is almost identical to the definition of Big-O notation, except that the inequality is flipped.

Say we want to prove that the lower bound of $f(n) = \frac{n(n-1)}{2}$ is lower bounded by $g(n) = n^2$. We can take $c = \frac{1}{4}$, and say:

$$\begin{aligned} \frac{n(n-1)}{2} &\geq cn^2 \\ \frac{n(n-1)}{2} &\geq \frac{n^2}{4} \\ \frac{n^2 - n}{2} &\geq \frac{n^2}{4} \\ 2n^2 - 2n &\geq n^2 \\ n^2 &\geq 2n \\ n &\geq 2 \\ \therefore n_0 &= 2 \end{aligned} \tag{4}$$

Thus, an n_0 and c exist for this inequality to be true, so $f(n) \geq cn^2, n \geq n_0$, and therefore $f(n) \in \Omega(n^2)$.

4.4 Big-Θ Notation

While Big-O described worst performance (upper bound) and Big-Ω described best performance (lower bound), Big-Θ describes a tight bound; in other words, it describes both of the previous sets. More formally, we can say that some function $f(n)$ is $\Theta(g(n))$ if there exist three positive constants n_0, c_1, c_2 such that for all $n \geq n_0$:

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \tag{5}$$

Notice that this definition is simply combining Big-O and Big-Ω.

Also note that Big-Θ does not always exist; we can say:

$$f(n) = \Theta(g(n)) \iff f(n) \in O(g(n)) \cap \Omega(g(n)) \tag{6}$$

5 Induction

Induction, while more generally used in math, is helpful in a variety of computer science applications (such as proving run times, etc.). Induction is a method of proving a statement about a set of numbers, and is based on the idea that if a statement is true for a number, it is true for all numbers greater than that number.

Generally, you are given a statement to prove (say, $f(n) \geq g(n)$), and a base condition that said statement must be proven for (say, $n \geq 1$). You can then prove the statement by showing that it is true for the base condition, and then showing that if it is true for n, it is true for $n + 1$. We can break this down into three steps:

- **Base Clause**
- **Inductive Clause**
- **Final Clause**

Say, for instance, you'd like to prove that $1 + 2 + 3 + ... + (n - 1) + n = \frac{n(n+1)}{2}$ for all natural numbers $n \geq 1$. To prove this inductively, we follow the outline above and say:

- **Base Clause:**

  Does this hold for the base clause, ie $n = 1$? We can check:

  $$1 = \frac{1 * (1 + 1)}{2} = 1$$

  Since this holds true for our base case (note that this came from the original question...) we can then move on to. . .

- **Inductive Clause:**

  Assume that our statement holds true for k; can we prove it holds for $k + 1$? To consider why this step makes logical sense, imagine $k = 1$, our base clause. We've proved this is true, so we can then see if $k + 1$ is true. Technically, we could continue checking our statement this way until infinity, but instead, if we simply prove it generally using an arbitrary k, we can then say that it holds for all k.

  For this specific situation, we can say (using the assumption that k satisfies our statement):

  $$1 + 2 + ... + k = \frac{k(k + 1)}{2}$$

  We can check if $k + 1$ holds using this assumption:

$$1 + 2 + \ldots + k + (k+1) = \frac{(k+1)(k+1+1)}{2}$$

The left side of this operation is simply our original assumption (in red) plus $k + 1$, so we can rewrite:

$$\frac{k(k+1)}{2} + k + 1 = \frac{(k+1)(k+2)}{2}$$

This simplifies to the same equation on either side, and thus we can say that our statement is true inductively

6 Recursion