# Week Lecture Notes

## O-notation Examples

In-class activity: O-notation quiz

**Rule of thumb**: The highest order term in an expression tells you it’s tightest O-notation expression. For example:

* n^2 – 5n + 100 is O(n^2)
* 1 + 20n^2 + 4n^3 + 20n is O(n^3)
* n(n + 4) + 60 is O(n^2)
* n + n log 2n + 5 is O(n log n)

**Fact**: Degree k polynomials are O(n^k), e.g.

* n^2 + n + 100 is O(n^2)
* n^3 – n^2 + 100n is O(n^3)

We will **not** be covering big omega or big theta notation.

|  |  |  |  |
| --- | --- | --- | --- |
| O-notation | Name | Example | Example Algorithm |
| O(1) | Constant | 1, 50, 483, … | **Finding the max** of a sorted array |
| O(log n) | Logarithmic | 4 + log 2n | **Binary search** on an array of n items in sorted order |
| O(n) | Linear | 10n+4 | **Linear search** on an array of n items in any order |
| O(n log n) | n-log-n | 2n log (n + 4) | Worst-case performance of **mergesort** to sort n items |
| O(n^2) | Quadratic | 3n^2 + 3n - 5 | Worst-case performance of **insertion sort** to sort n items |
| O(n^3) | Cubic | n^3 + n + n log n - 1 | **Multiplying two n-by-n matrices** in the standard way |
| O(2^n) | Exponential | 2^n, 3 \* 2^(n+5), … | **Printing all subsets** of {1, 2, …, n}. |

## ADTs: Abstract Data Types

An **abstract data type (ADT)** is a mathematical model of a data structure that specifies the type of the data stored, the operations supported on them, and the types of the parameters of the operations. An ADT specifies **what** each operation does, but **not** how it does. In other words, an ADT carefully specifies the interface between it and the rest of the program it’s operating in.

**Example**. In C++, ADTs are typically represented as abstract base classes, such as **Wordlist\_base** from assignment 1. The base class defines **how** the methods behave, but does **not** say how to implement those methods.

**Example**. In C++, we can print strings like this:

cout << "hot dog";

You don’t need to know how cout or << work. You only need to know *what* they do.

**Example.** In mathematics, integers can be treated as an ADT. Numbers, plus the operations on them like + and \*, are described to work in very precise way. But math doesn’t tell you how you implement, say, \*. That’s up to you, and you can do it any way that makes sense and gives the correct answer.

For instance, 23 \* 41 can be calculated in a variety of ways:

* Elementary method taught in North America
* Cross multiplication
* Using a calculator

These are all very different methods, yet give the same answer when done correctly.

**Example**. An intuitive example of an ADT is an electrical outlet on a wall. It has 3 precisely defined holes that let you can insert any plug into. It “promises” to provide electricity through the plug, but it says nothing about how the electricity is generated. The electricity could come from solar energy, a gas engine, geo-thermal energy, or even from a hamster running in a wheel.

ADTs also help with:

* **Encapsulation**: low-level details specific to the ADT are hidden, “encapsulated”, inside the ADT. In C++, we can declare such details private, to prevent code outside the class from seeing/changing these details.  
  **Example**. C++ strings typically store a pointer to an underlying array of characters, plus a length and capacity variable. These variables are private in the string class, as we don’t want programmers changing them. A regular programmer using strings doesn’t need to worry about these details, or even know that they exist. Compared to C-strings, C++ strings are much easier to use since you can ignore the implementation details.
* **Modularity**: Since ADTs have a clearly defined interface, they can be treated as independent modules. We built and test ADTs on their own separate from the rest of the program, and then “plug” them in when they are ready. Making programs modular is an important part of good software engineering.  
  **Example**. There are multiple ways that C++ strings could be implemented. For example, a string might use the **rope** data structure, which stores strings as a binary tree and works well for large strings. Or it could implement the **small string optimization** (**SSO**) that can store strings of about 20 or fewer characters without allocating any memory. If you are currently using one string implementation, then you can “plug in” in a new implementation without needing to change any of the code that calls the string.

## Templates

In our code for stacks, queues, and deques, we will use C++ templates (also called generics). Templates let us write classes and functions that work with any type of value **T**.

For example, if we want a stack of ints and a stack of strings, templates let us write **one** class that can represent both. Without templates, we would need to write a string stack class and an int stack class.

## Stacks

The basic **stack** ADT:

* **push(x)**: puts x on top of the stack; error if the stack if full (stack overflow)
* **pop()**: removes the item on top of the stack; error if the stack is empty (stack underflow)
* **top()**: returns a copy of the top of the stack without removing it; error if the stack is empty
* **size()**: returns the number of items in the stack
* **empty()**: returns true if the stack is empty, and false otherwise

We usually assume all these are O(1) operations, i.e. the time it takes them to run is a small fixed amount of time that doesn’t depend upon the number of items in the stack.

An array/vector implementation of a stack.

A linked list implementation of a stack.

## Queues

The basic **queue** ADT:

* **enqueue(x)**: add x to the end of the queue; error if queue is full
* **dequeue()**: remove the item at the front of the queue; error if queue is empty
* **front()**: return a copy of the front queue item without removing it; error if queue is empty
* **size()**: return number of items in the queue
* **empty()**: return true if queue is empty, and false otherwise

An array/vector implementation of a queue.

A linked list implementation of a queue.

## Deques

The **deque** ADT:

* **insertFront(x)**: add x to the front of the deque
* **insertBack(x)**: add x to the end of the deque
* **eraseFront()**: remove the front item of the deque; error if it’s empty
* **eraseBack()**: remove the back item of the deque; error if it’s empty
* **size()**: returns number of items in the deque
* **empty()**: returns true if empty, false otherwise

An array/vector implementation of a deque.

A linked list implementation of a deque.

Getting O(1) insert/erase operations at both ends of the deque is a bit tricky:

* For an array-based implementation, use a circular array.
* For a linked list based implementation, use a doubly-linked list.

**Adaptors**: A deque implements the functionality of both a stack and a queue, and so you can use a deque to implement both a stack and a queue.