CS 162
Intro to Computer Science II

Lecture 23

STL

Linked List

3/11/24

Oregon State University

Odds and Ends

- Lab 10 + Worksheet 10 posted
- Assignment 5 rubrics posted

Today’s topic(s)

- Standard Template Library (STL)
- Linked List

Template Classes

- Work the same way as templated functions
- All functions within the class will operate on the provided types
- Scope with `ClassName<T>::functionname()`
- Each function needs the Template prefix

Today's topic(s)

- Template
- Standard Template Library (STL)

Standard Template Library (STL)

- C++ STL can be broken down into:
  - **Containers** – general purpose data structures (templates) for holding things
  - **Iterators** – special classes for traversing containers
  - **Algorithms** – sorting, searching, etc.

- Iterators make it possible to run the algorithm on the containers

- The STL is a great resource:
  - It contains a wide variety of very useful structures and algorithms
  - It is well-implemented, which means the structures and algorithms perform very efficiently
  - In general, it allows us to avoid re-inventing the wheel

Introducing STL Containers

- Predefined templates that can store any type of data
- The appropriate container will be dictated by the application requirements
- Example considerations:
  - Does the data need to be stored?
  - How will the data be accessed?
    - Front to back
    - Randomly?
  - Will additional data ever need to be added or removed?
- Careful planning will allow you to write clean, efficient code

Types of Containers

- Sequential containers (vector, deque, list)
  - Programmer controls the order of the elements
- Associative containers (map, set, multimap, multiset)
  - Position of elements is controlled by container
  - Elements are generally accessed by using a “key”
- Adapters (stack, queue)
  - Use an existing type of container to build other types
    - In this context, we call these “Abstract Data Types”

Examples of C++ Containers

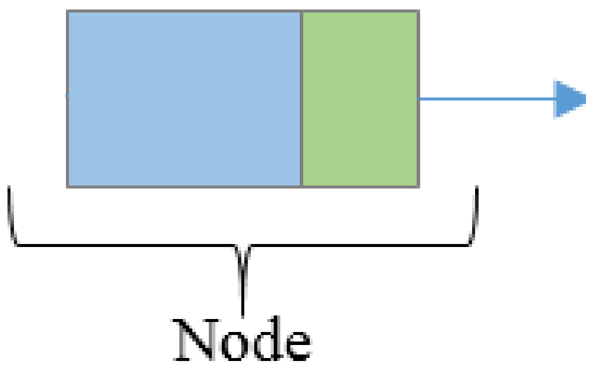
- <array> - stores a constant amount of data in contiguous memory
- <vector> - An array that can be resized
- <list> - Linked list that stores data in non-contiguous memory
- <set> - An ordered collection of items
- <queue> - Stores data & returns it in the order it was received
  - First in, first out
- <stack> - Stores data & returns it in the opposite order that it was received
  - First in, last out

- Generally, it is a good idea to refer to the STL documentation before starting a project

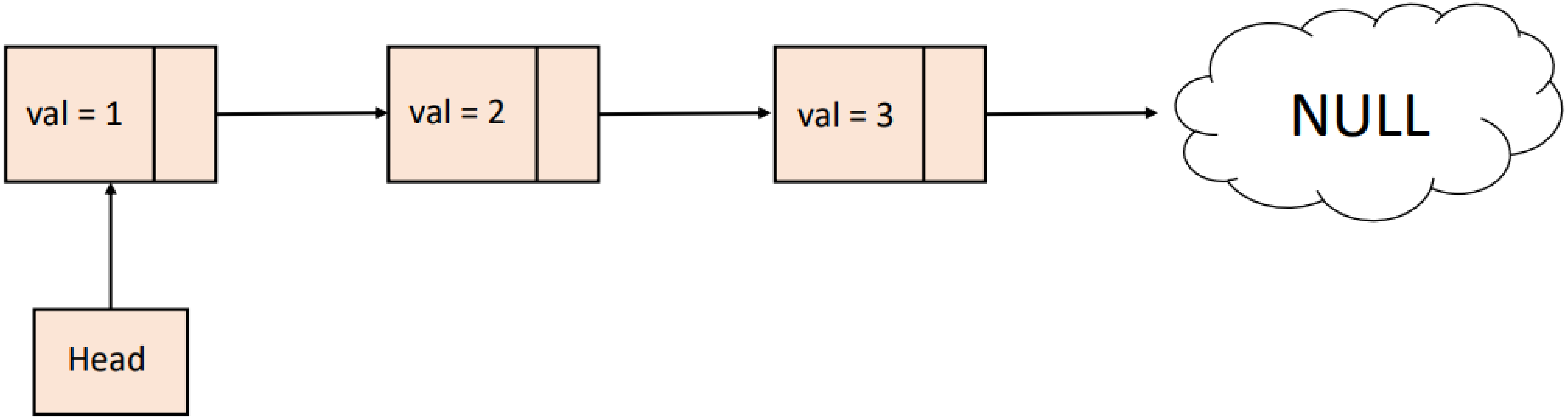
Linked List

- A list constructed using pointers
- Can grow and shrink easily while the program is running
- Not stored contiguously in memory
- Use structs to create

```
struct Node {
        int val;
        Node* next;
};
```



Singly Linked List



In class activity

- Use the code provided on Canvas, complete the following tasks:
  - Task 1: What does the code do? (Hint: Trace through the code by drawing the picture out)
  - Task 2: Write code to print the list you just created. Trace the code you wrote to verify
    - Hint: Use while loop and Node* current
  - Task 3: Delete the list you just created. Trace the code you wrote to verify
    - Hint: You might need another Node*

Pros and Cons of Singly Linked List

- Pros
  - Easy to implement
  - Insertion and deletion of elements can be done easily and doesn’t requires movement of all elements compared to an array
  - Can allocate or deallocate memory easily during its execution
- Cons
  - Uses more memory when compared to an array
  - No random access
  - Traversing in reverse is not possible for singly linked list

Today’s topic(s)

- Begin Complexity Analysis

How to compare/describe algorithms

- We have different data structures and sorting algorithms, how to compare them?

- We want a way to characterize runtime or memory usage that is completely **platform-independent**
  - i.e. does not depend on hardware, operating system, programming language, etc.

Complexity Analysis

- Use Complexity Analysis to help make platform-independent comparisons of data structures
  - Refer to as Big O

- Allow us to assess a data structure or algorithm's resource usage (i.e., runtime and memory consumption) in an abstract way

- To do this, we describe how a data structure's or algorithm's runtime or memory usage changes relative to a change in the input size (**n**)

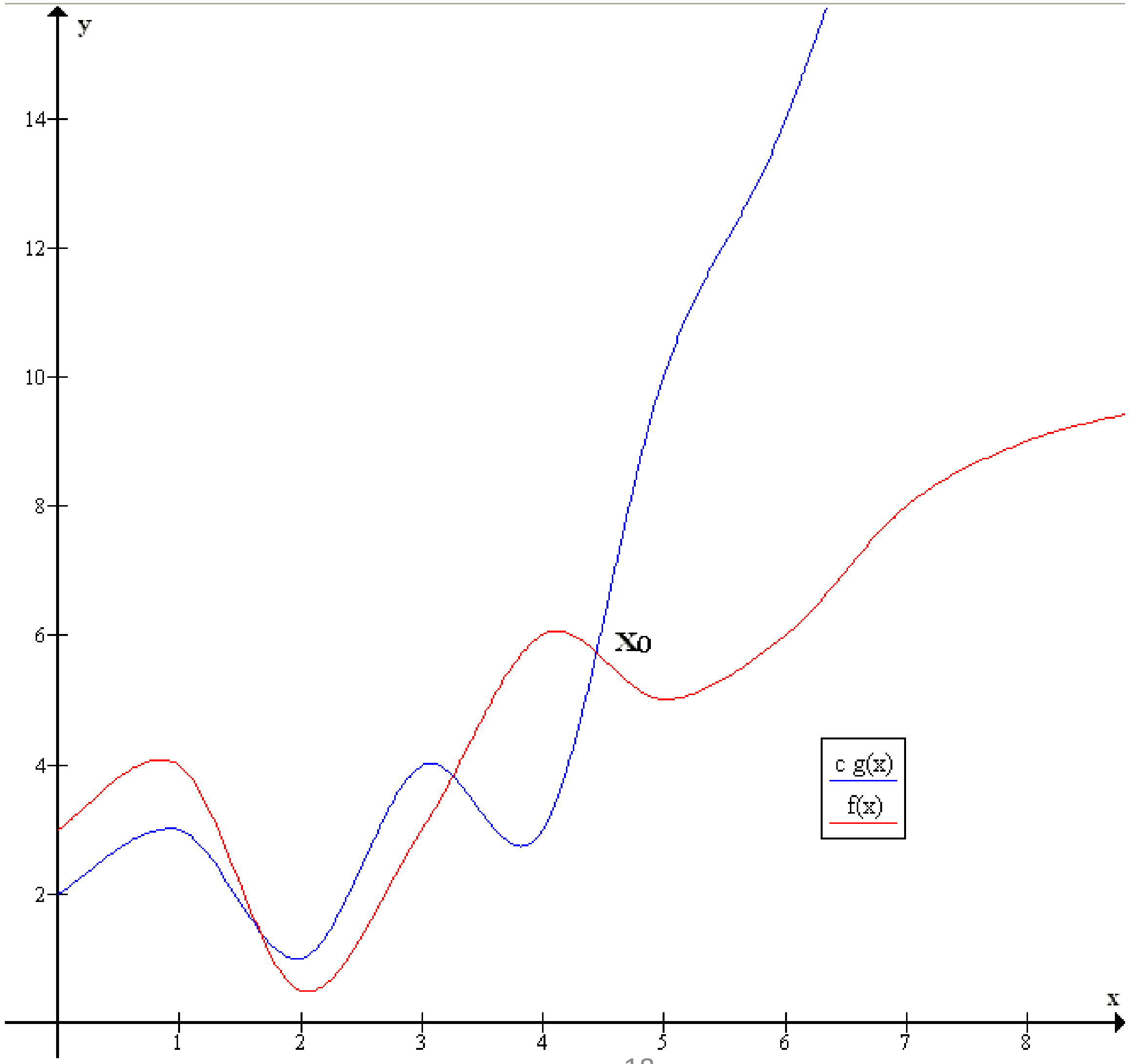
Big O

- We use Big O notation to assess a data structure or algorithm’s performance.

- Big O notation: a tool for characterizing a function in terms of its growth rate
  - Indicate an upper bound on the function’s growth rate, known as growth order

Big O

g(x) provides an upper bound on *f(x)*



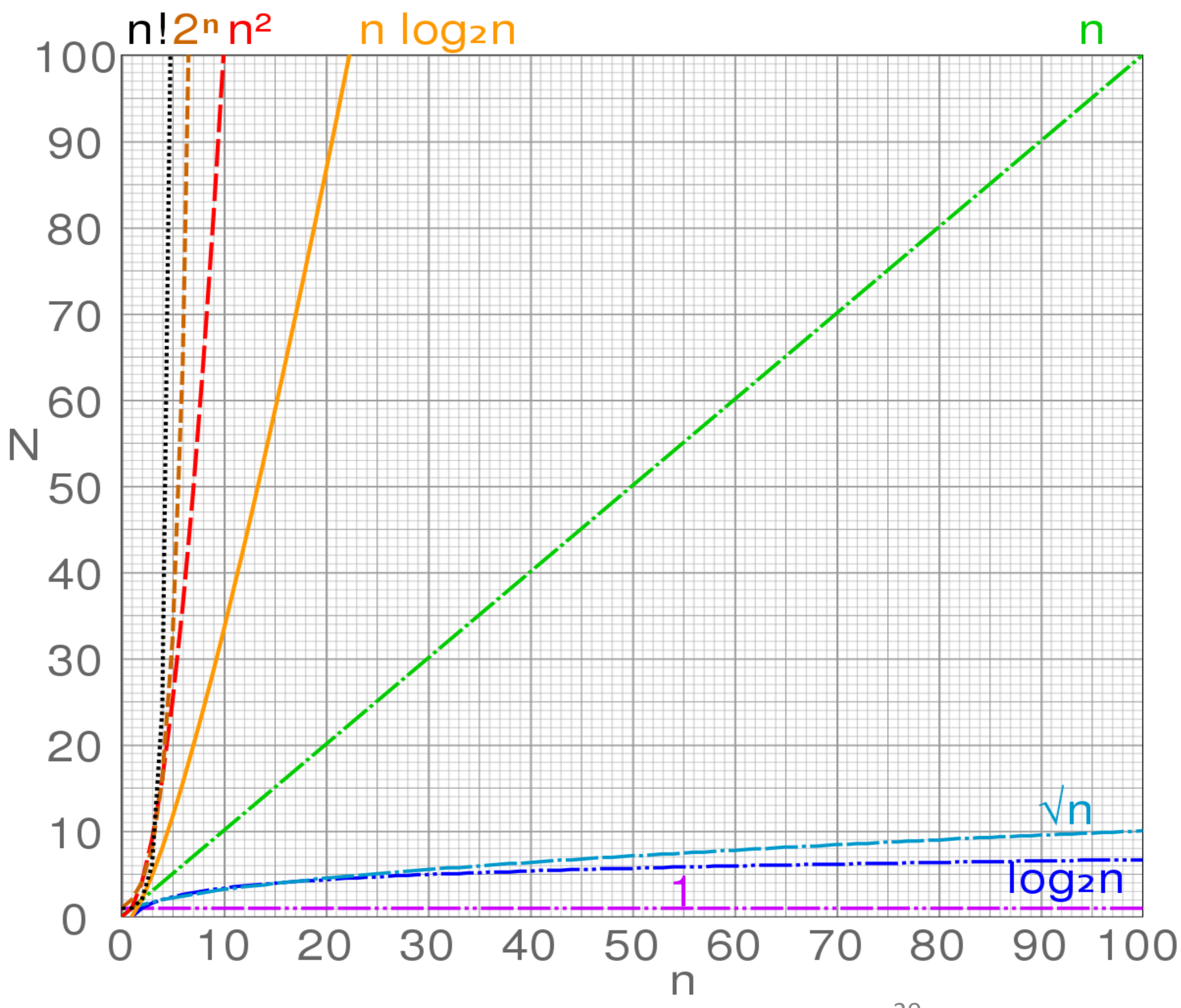
g(x) is *O(f(x))*

Big O

- To assess a data structure or algorithm's complexity, we will compute a growth order for its runtime (or memory usage) as a function of the input size n

- Importantly, we want to describe how data structures behave in the limit, as n approaches ∞ (infinity)

Common growth order functions



Common growth order functions

- O(1) – constant complexity
- O(log n) – log-n complexity
- O(√n) – root-n complexity
- O(n) – linear complexity
- O(n log n) – n-log-n complexity
- O(n^2) – quadratic complexity
- O(n^3) – cubic complexity
- O(2^n) – exponential complexity
- O(n!) – factorial complexity

Big O

- Consider this example…

```
int sum = 0;
for (i = 0; i < n; i++) {
     sum += array[i];
}
return sum;
```

- This function is summing an array of n integers
- What’s the run-time complexity of the function?

Big O example

```
int sum = 0;
for (i = 0; i < n; i++) {
        sum += array[i];
}
return sum;
```

- The instruction `int sum = 0;` executes in some constant time c1 independent of n
- Each iteration of the loop executes in some constant time c2, and this happens n times
- The return statement executes in some constant time c3 independent of n
- So runtime is c1 + c2*n + c3
- c1, c2, and c3 depend on the particular computer running this function, so we ignore them to figure out run-time complexity
- Thus, this function grows on the order of n, a.k.a. its run-time complexity is **O(n)**

Determining a program’s complexity

```
node* push (node * head, int val) {
        node *temp = new node;
        temp->val =val;
        temp->next = head;
        head = temp;
        return head;
}
```

- Every instruction in this function executes in some constant time, independent of n
- Thus we ignore them to figure out runtime complexity.
- Complexity: O(c1+c2+c3+c4+c5) = **O(1)**

Dominant components

- When a growth order function has additive terms, one of those will dominate the others
  - Specifically, function *f(n)* dominates *g(n)* if n0:n>n0, *f(n)* > *g(n)*

- In these cases, we simply ignore the non-dominant terms
  - i.e. $n^2 - n$, n^2 dominates n, so we ignore n, and we say this complexity is $O(n^2)$

More examples

- Loops are one of the main determinants of a program's complexity

- ```
  for (int i = 0; i < n; i++) {
        ...
  }
  ```

- ```
  for (int i = n; i > 0; i/=2) {
        ...
  }
  ```

- ```
  for (int i = 0; i*i < n; i++) {
        ...
  }
  ```
  ```

More examples

- ```
  for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
                ...
      }
  }
  ```

- ```
  for (int i = n; i > 0; i/=2) {
      for (int j = 0; j < n; j++) {
                ...
      }
  }
  ```

Real-world Consideration

- Your program will only perform as well as your design
  - Constant factors can still play a part
- Suppose you have two algorithms…
  - Algorithm A) 1,000,000n → O(n)
  - Algorithm B) 2 n^2 → O(n^2)
  - Which one is better?
    - It depends