CS 162
Intro to Computer Science II

Lecture 24
Linked List
Assignment 4 Help
Final remarks
3/15/24
Odds and Ends

• Due reminder:
  • Quiz 5 due Sunday midnight via Canvas – open after today’s lecture
  • Assignment 4 due Sunday midnight via TEACH
    • Grace days are allowed

• Today is the last day to demo:
  • Assignment 3 without late demo penalty
  • Assignment 1&2 with 30% late demo penalty

• Final Exam:
  • Wednesday 3/20 at 12pm at LINC 200
Today’s topic(s)

• Linked list
• Final exam review
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)

• Linked list

• Final exam review
Final Exam

• Weight: 15% of course grade
• Time: Wednesday 12:00 – 12:50 pm
• Where: LINC 200
• Close book, close notes, no calculator
• Scratch paper will be provided if needed
• Bring pen/pencil, and your photo ID
• Question types:
  • T/F, multiple choice
  • Similar as the midterm exam 😊
• Question amount: ~40
Coverage

• Non-cumulative

• Emphasis on material covered after Midterm (90%)
  • Lecture 14-24 (start from shallow vs. deep copy)
  • Lab 6 – 10
  • Worksheet 6-10
  • Assigned Reading
  • Assignment 3-4

• General coverage of earlier topics (10%)
Topics

- Shallow vs. Deep copy
- Big 3 and their usage
- Inheritance
- Upcasting vs. downcasting
- Polymorphism
- Virtual vs. pure virtual
- Abstract class
- Function/class templates
- Standard Template Class (STL)
  - vector
- Containers
- Linked List (singly)
- Recursion
Study Guide

• Take the practice exam and time yourself
• Lecture slides 14-24
• Quiz 3-5
• Worksheet 6-10
• Lab 6-10
• Assignment 3-5
• Assigned readings
Winter 2023 Exam Review
You have learned MANY things from CS 162

• Pointers
• Memory model
  • Stack vs. heap
• Dynamic Arrays
• Structs
• File separation
  • .h .cpp
  • Header guards
• Makefile
  • Compilation process
• File I/O
• Object Oriented Programming
  • Encapsulation
• Struct vs. Class

• C++ Classes
  • Access specifiers: private, public, protected
  • Accessor and Mutator functions
  • this keyword
  • Constructors: default vs. non-default
  • const
  • Big Three
    • Copy constructor
    • Assignment operator overload
    • Destructor
  • Class composition vs. class inheritance
  • Polymorphism
You have learned MANY things from CS 162

• Template
  • STL
• Containers
  • Linked list vs. array
  • Vector
• Recursion
Be Confident...

Now you are able to...

• Design and implement programs that require:
  • multiple classes and structures
  • hierarchies of classes that use inheritance and polymorphism
  • an understanding of abstraction, modularity and separation of concerns

• Construct and use basic linear structures (arrays, stacks, queues, and various linked lists) in programs, and be able to describe instances appropriate for their use.

• Develop test-data sets and testing plans for programming projects.

• Produce recursive algorithms, and choose appropriately between iterative and recursive algorithms.
Final Remarks...

• Thank you so much for your commitment to this course

• What’s next?
  • CS 261: Data Structure
  • ECE/CS 271: Computer Architecture and Assembly Language
  • CS 290: Web Development

• Future improvements?
  • Canvas SLE

• ULA position
  • Contact me! And apply through: https://jobs.oregonstate.edu/postings/140560
Final Remarks...

• Submit all your work by the deadline
  • Assignment 4, Quiz 5

• Take the Final Exam on Wednesday
  • Bring your photo ID

• Grade disputation:
  • By 3/23 6pm
Assignment 4 Q&A
*Additional topic(s)*

- Complexity Analysis

*Note: this will not be in the final*
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 $O(f(x))$
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 $\infty$ (infinity).
Common growth order functions
Common growth order functions

- $O(1)$ – constant complexity
- $O(\log n)$ – log-$n$ complexity
- $O(\sqrt{n})$ – 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...

  ```java
  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

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

- The instruction `int sum = 0;` executes in some constant time $c_1$ independent of $n$
- Each iteration of the loop executes in some constant time $c_2$, and this happens $n$ times
- The return statement executes in some constant time $c_3$ independent of $n$
- So runtime is $c_1 + c_2 \times n + c_3$
- $c_1$, $c_2$, and $c_3$ 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

```c
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: \( O(c_1 + c_2 + c_3 + c_4 + c_5) = O(1) \)
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 \( n_0: n > n_0, f(n) > g(n) \)

• 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) $1,000,000n \rightarrow O(n)$
  • Algorithm B) $2n^2 \rightarrow O(n^2)$
  • Which one is better?
    • It depends