Erik Sjoberg  
[esjoberg@cmu.edu](mailto:esjoberg@cmu.edu)

# Assignent 4: Rube Goldberg Machine

## ADT Design

Describe the design of your various ADTs, the data structures you selected, what information and operations you hid and your reasoning for your decisions.

**ADT Choices**

For this assignment, I chose to base all of my list, stack, queue, and binary tree ADT internals on an enhanced version of a doubly linked list.

I made the choice to base all of my ADTs on a doubly linked list to exploit the flexibility of the data structure and enable any length of input in C without the need to check and re-size arrays manually. The consistency of the underlying implementations also enabled a large amount of code reuse between the various ADTs.

**ADT Design**

The most unique feature of my ADT design is the separation of links used for left/right nodes (as in list, stack queue), and left/right descendants, resulting in what is in fact four links per node. The primary motivation was the separation of the two semantically distinct concepts of descendants from the linear concept of a left/right doubly linked list. This separation allows for a single collection of nodes to be stored and represented as a list/queue/stack, and a binary tree simultaneously. This was very useful during process of building the balanced binary tree, since the copying of node elements into a completely separate memory space was not necessary in order to use a queue in during processing.

In addition to left/right and descendent links, each node contains customizable data types for firstname, lastname, birthdate, age which in this case corresponds to 3 60-character strings and an integer. The contents of the node are easy to customize in the linkedlist.h headerfile, enabling easy adaptation to any application

The Queue ADT is implemented as a doubly-linked list where Enqueue() pushes the new element to the back of the list, and Dequeue() pops an element from the front of the list.

The Stack ADT is implemented as a doubly-linked list where Push() pushes the new element onto the front of the list, and Dequeue() pops an element from the front of the list.

The List ADT exposes the base functionality of the doubly-linked list ADT that the rest of the ADTs are based on, including InsertAtTail() and InsertAtBack().

The binary tree ADT is a bit different from the others, in that it uses dedicated bt\_left and bt\_right pointers to its descendants in order to separate the usage of these distinct concepts.

**Information Hiding**

My stack, queue, list, and binary tree ADTs hide all details of the internal operations and memory allocation from the user. Convenience methods such as GetNewNode() and the four Init\_\_\_() functions for each ADT enable the user to create and manipulate data without dealing with the underlying complexity of memory management or list size.

**How my ADT is used**

I provide the user with standard stack functions for initialization of each data type, as well as the following standard ADT functions:

* List:
  + Insert at head
  + Insert at tail
  + Length
  + Print
  + Insert from Binary Tree
* Queue:
  + Enqueue
  + Dequeue
  + Peek
  + Length
* Stack:
  + Push
  + Pop
  + Peek
  + Length
* Balanced Binary Tree:
  + Insert from list
  + Print (Pre-order, ordered, and Post-ordered)

To create each data type, the user simply creates a DLList type variable pointing to an unallocated node, then initialize it with the provided initialization function.

All user-facing ADT-related functions operate using pointers to pointers to nodes, with the exception of the recursive binary tree insertion and print functions which operate directly upon pointers to nodes.

## How to Use the Program

The program can be run at the terminal via the following executable:

goldberg.exe

Typing any key when prompted will cause the program to proceed. When asked for input, please input a comma-separated string of the format <full\_name\_with\_spaces>, <age\_as\_integer>, <birthdate\_in\_mm\_dd\_yyyy\_format>. For example:

Erik Sjoberg, 30, 9-13-1986

To compile the program, go to the directory containing the source code files and use the gcc compiler with the following command:

gcc -Wall goldberg.c linkedlist.h linkedlist.c –o goldberg

## Program Analysis

**Correctness**The key invariants in my system are all contained within the ADT implementations in linkedlist.c and maintained via assertions.

1. Individual data string elements are always < 60 characters
2. List / Queue / Stack length is always >= 1 when element is removed
3. Node left/right links are NULL after being removed from an ADT
4. At the end of binary-tree insertion , the left/right links of all tree elements are NULL
   1. Note that bt\_left and bt\_right binary-tree descendent links are obviously not NULL

In addition, Goldberg.c maintains that individual lines of the input file (and user input) are all less than 1024 characters.

Because the ADTs maintain their characteristics, very little effort is required to show that the main program also completes correctly because the majority of the program is a simple sequence of statements, with a few loops with obvious termination conditions (for n=-; n < length; n++). This shows the power of ADTs to simplify the analysis of what would otherwise be a complex set of nested conditions had the ADT abstraction not been used.

**Complexity: Performance**O(n2) (typical)

The main culprit causing typical O(n2) performance is my simple implementation of last-element access for my core doubly-linked list. To access the last element, the linked list follows every pointer in the list for O(n) performance, and this operation often takes place within the context of another loop operating on the entire set of data, resulting in O(n2) performance.

A significant speed-up could be attained by explicitly storing and maintaining a pointer to the last element of the linked list, at the cost of extra complexity in the implementation.

O(n^3) (worst-case)

However, because the worst-case performance of the QuickSort algorithm is O(n^2), the interaction between with the slow linked-list implementation can add a factor of O(n) resulting in O(n^3) absolute worst-case performance. This is the strongest argument for taking the time to implement the direct end-of-list access routine.

**Complexity: Memory**   
O(n)

Because of the in-place sort and reasonable data structures, worst-case memory usage is simply O(n). That said, an important task yet to be done is to implement recursive memory-freeing routines which would allow for quick and easy recovery of the memory used by list / stack / queue / binary-tree ADTs.

## Build / Run Instructions

1. Change path to the folder containing the source code
   1. cd /path/to/sourcecode/folder/
2. Compile with gcc (use MinGW if on Windows)
   1. gcc -Wall goldberg.c linkedlist.h linkedlist.c –o goldberg
3. Run goldberg.exe (from console, or double click)
4. Press any key as prompted
5. Input new data as prompted
6. Press any key as prompted