Final Exam:

Repository is at https://classroom.github.com/a/qW9mHUaJ

Please place the answers to each question in the appropriate folder.

Final Exam Question #1 Animals

Introduction

In this question we will play a guessing game (somewhat like 20 questions), with the computer doing the guessing—and learning at the same time. In the sample below, the human's responses are shown in red.

Welcome to the Animals game!

Shall we play a game? y
Were you thinking of a elephant? n
Doh! What was the animal? cow
What question separates cow from elephant? Does it moo?
What is the correct answer for this animal? y

Shall we play a game? y
Does it moo? y
Were you thinking of a cow? y
Great!

Shall we play a game? y
Does it moo? n
Were you thinking of a elephant? n
Doh! What was the animal? gnat
What question separates gnat from elephant? Is it bigger than a breadbox?
What is the correct answer for this animal? n

Shall we play a game? y
Does it moo? n
Is it bigger than a breadbox? y
Were you thinking of a elephant? n

Doh! What was the animal? whale
What question separates whale from elephant? Does it live in the water?
What is the correct answer for this animal? y

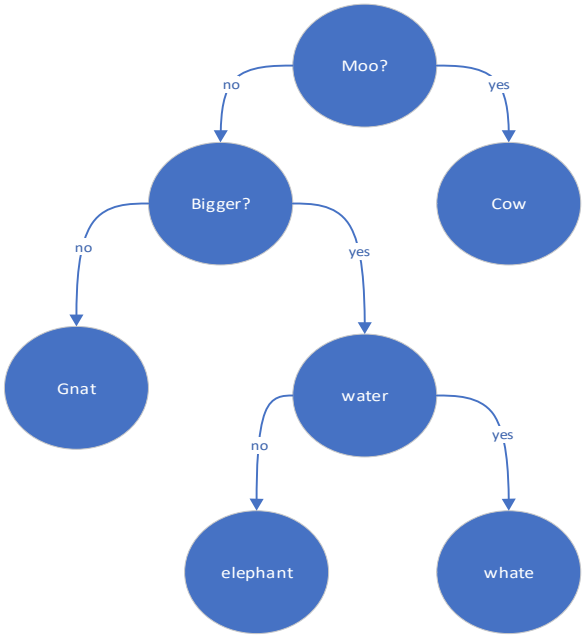
Shall we play a game? n
Bye!

Strategy

The program maintains a binary tree whose internal nodes contain questions and whose leaves contain the names of animals. The left and right children of an internal node correspond to the responses "no" and "yes" (left being no, and right be yes).

When the program makes a wrong guess, it collects enough information to create a new node. The original leaf (the one with a wrong answer) is replaced by a new internal node that contains a new question and whose children are the old wrong answer and the new right answer.

Notice that the tree has the property that a node is either a leaf or has two children. That property makes it easy to distinguish answers from questions. Here is a picture and the corresponding textual representation of a simple tree.



Q: Does it moo?

Q: Is it bigger than a breadbox?
A: gnat
Q: Does it live in water?
A: elephant
A: whale
A: cow

Be sure to test this adequately.

Final Exam Question #2 Doubly linked list

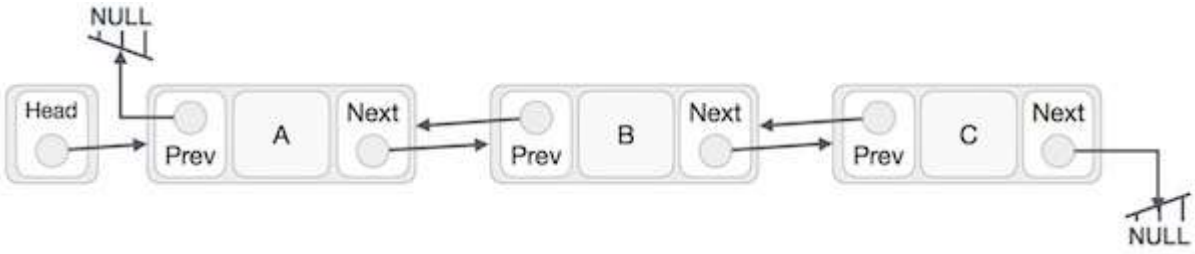
Introduction

In this question you will implement a doubly linked list of integers with the following operations:

Following are the basic operations supported by a list.

- **Insertion** – Adds an element at the beginning of the list.
- **Deletion** – Deletes an element at the beginning of the list.
- **Insert Last** – Adds an element at the end of the list.
- **Insert First** – Adds an element at the start of the list
- **Delete Last** – Deletes an element from the end of the list.
- **Delete First**– Deletes an element from the start of the list.
- **Insert After** – Adds an element after an item of the list.
- **Delete** – Deletes an element from the list using the key.
- **Display forward** – Displays the complete list in a forward manner.
- **Display backward** – Displays the complete list in a backward manner.

Be sure to build a test driver



Final Exam Question #3 Adding Machine

Introduction

It used to be common to use adding machines which did things in RPN (reverse polish notation). We will build a simple one, but it will have the added feature of "Undo".

The adding machine has a single accumulator in it. It will have a current value. All operations will act upon that value plus an additional operand.

Example:

On initial run, the machine has a value of 0. "=>" indicates the application output. The rest is input:

```
=> 0
1
+
=>1
2
+
=>3
3
*
=>9
U
=>3
U
=>1
U
=>0
5
+
=>5
C
=>0
```

The operations it must support are:
+, -, *, /
Along with U for undo and C for clear (reset to 0)

Use the appropriate data structure and build this application.

Final Exam Question #4 Build a word occurrence counter

Given a text file, write a program the counts the number of times each word occurs in the file.

1. It should use hashing (open or closed, up to you) that YOU implement
2. It should take a file name as input
3. It should read the file and print out each word and the number of times it occurs.
4. The output list should be sorted. You can use stl for this.

So, for example:
"This is a test of your skills using data structures. It is quite tricky. Are you ready?"

Would produce:
This 1
Is 2
A 1
Test 1
And so on…

Remember, this needs to properly account for punctuation (removal)

Final Exam Question 5: Simple Graph

Write a C++ program to implement several graph algorithms for simple graphs. These graphs are to be implemented using the adjacency list representation.

The `graph` class is already defined in the header file `question5.h`
This also contains all other files related to this question

The graph is created from a text file that contains the adjacency matrix representation of a graph. The first line of the file is an integer `n` indicating the number of vertices in the graph. Next there is a `(n+1)×(n+1)` table where the first row and the first column are names of vertices. The table records edges of the graph. Integer `1` indicates an edge exists between a pair of vertices. Integer `0` indicates no edge.

For a given graph `size`, each vertex of a graph can be referenced by index values `0, 1, …, ( size – 1 );` however, vertices of a graph are labeled consecutively by capital letters, starting from `A`, which corresponds to the index value `0`. Use index values when you refer to a vertex in a graph in implementing the algorithms; use vertex labels only in printing. The edges are recorded in the data member `adj_list`, which is a `vector` object. The vertex labels are recorded in the data member `labels`, which is also a `vector` object. For example, you can use `labels[0]` to get the name of the first vertex, and `adj_list[0]` to get the `list` of edges (recorded as integer indexes of destination vertices) from the first vertex as the source.

The source file question5.cc already contains the complete main function. Implement **ALL** member functions of the `graph` class and put your implementations of these functions in this source file. **Several (but not all)** of the functions are described in more details below.

- `graph :: graph ( const char* filename )` : This is the constructor. It reads from the input file of the graph with adjacency matrix representation and builds the graph with adjacency list representation. This method sets the value of `size`, builds the vectors `labels` and `adj_list`. For example, for the following line of input:

  D 0 1 0 0 1 0 0

  Add edges to `adj_list[3]`, which records edges starting from vertex `D`, by adding values 1 and 4, which are indexes for vertices `B` and `E`.

- `void graph :: depth_first ( int v ) const` : This private function is used to traverse a graph in the *depth-first traversal/search* algorithm starting at the vertex with the index value of `v`. To implement this method (and together with the `traverse` method

below), you may need several global variable and objects. E.g. container objects to record the visiting order of all vertices, the container object to record the paths of traversing edges, and an integer indicating the current order in traversing.

1

- `void graph :: traverse ( ) const` : This public function is used to traverse a graph and invokes the above `depth_first` method. You will also need to display traverse result: the list of vertices in the order of their visit and the list of edges showing the path(s) of the traversal. At beginning of this method, you need to initialize the global variable(s) and object(s) used in `depth_first`.

- `void graph :: print ( ) const` : This function prints the adjacency list for the graph. The following line is an example from an output.

  D: B, E,

  It indicates there are edges from vertex D to vertices B and E.

Programming Note:

- Include any necessary headers and add necessary global variables and objects.

- You are not allowed to use any I/O functions from the C library, such as `scanf` or `printf`. Instead, use the I/O functions from the C++ library, such as `cin` or `cout`.