CS 202 Spring 2022 - Assignment 8

Linked Lists

Overview

Oh no, someone has left a pile of garbage around and we're the only ones who can sort it. This may not sound like the most enticing job, but hey, someone's gotta do it.
In this assignment, you will be given a "garbage" text file containing a piece of data on each line. However, the type of each datum is arbitrary. That is, on any line, you could receive any piece of garbage, here being any primitive type. Any line could contain an int, a double, a bool, a char, or a string. As an example from the provided garbage file:

```
j
t
3.0000
57
true
man
```

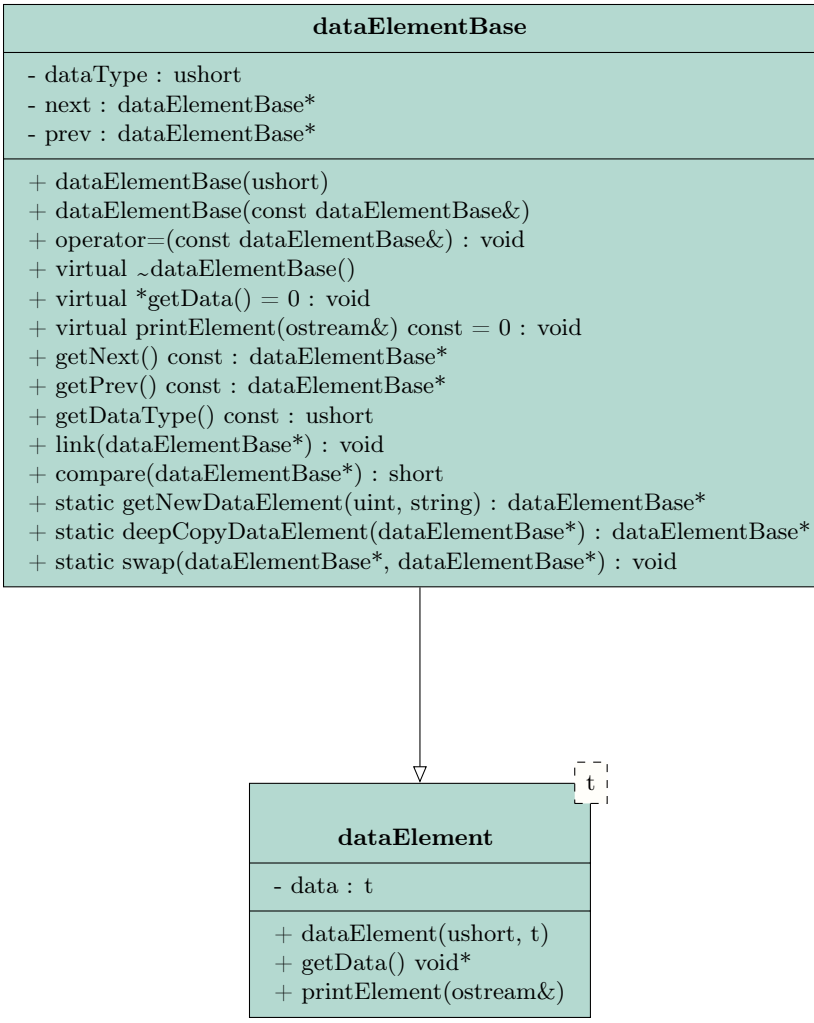
Contains two chars, a double, an int, a bool, and then a string. It is possible that each of these lines could be interpreted as several different data types, so we will need to establish a precedence for the types. Each line will be given the following priority with regards to types: **int** > **char** > **double** > **bool** > **string**, where int has the highest priority and string has the lowest priority. This means something like **3** will be treated as an int first, rather than a double, char, or string, which might all also make sense.
To organize these various pieces of trash, we will place them into a big garbage pile and then sort our way through it. This will involve reading the data, determining the type, and then placing it into our pile. The pile will be represented here as a series of doubly linked nodes, with each node containing a single piece of garbage. After throwing everything into the pile, we will sort all of the garbage first by its type, and then by the standard sorting implementation for that type. See the **Sample Output** section for an example of how the sorting will look.

Global Helper Functions

Below is a list of global helpers independent of any classes. Aside from these, please also be aware of the preprocessor definitions at the top of dataProcessing.h which can be used. These are described by the functions in which they are relevant. You have to implement one of the functions, which is in red.

- **bool isInteger(string s)** - Tells whether a given string can be interpreted as an integers. Returns true if so, false if not.
- **bool isChar(string s)** - Tells whether a given string can be interpreted as an integers.
- **bool isDouble(string s)** - Tells whether a given string can be interpreted as an integers.
- **bool isBool(string s)** - Tells whether a given string can be interpreted as an integers.
- **stob(string s)** - Converts the given string to a boolean value. Assumes the string is either "true" or "false", else gives an error.
- **unsigned int getDataType(string garbage)** - This function takes in a string and determines what kind of data the string is. Using all the "is" functions above, check the string to see if any are true. If it is an integer, return INT. Next, check for char and return CHAR if true. The next checks are for bool and then for double. If none of the checks return true, return STRING.

UML Diagrams



Important Classes and Functions

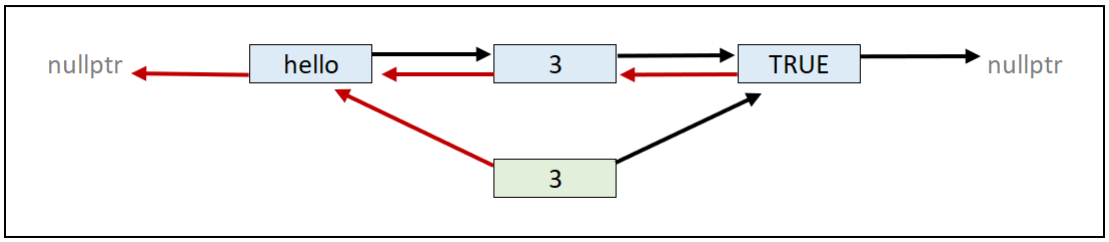
Below is a list of functions and variables that are important for this assignment. ***Variables are in green***, **functions that you will need to write are in red**, **functions implemented for you already are in**

blue, and **functions that are abstract that will be implemented later are in magenta**.

dataElementBase

This class serves as an abstract base for our **dataElement** class. A **dataElementBase** can be thought of as similar to a doubly linked node, just with a few extra functions added in. These will be linked together later in our **garbagePile** class to form a full pile, with each node holding a single piece of garbage of some arbitrary type along with pointers to the next and previous node. Since this acts as the base class for all variations of **dataElement**, we will use pointers of this type to point to any single **dataElement**, regardless of the actual type of data it holds.

- ***unsigned short dataType*** - An enumeration-esque variable that stores an integer representing the type this object is storing. Use this to do easy checking on the contents of a node. Use the preprocessor definitions at the top of the **dataProcess.h** when using this variable. This should be limited to INT, CHAR, DOUBLE, BOOL, and STRING.
- ***dataElementBase* next*** - Points to the next **dataElementBase** object that comes after this one in our pile. If nothing is after it, this will be nullptr.
- ***dataElementBase* prev*** - Points to the previous **dataElementBase** object that comes before this one in our pile. If nothing is before it, this will be nullptr.
- **dataElementBase(unsigned short dt)** - A constructor to initialize a brand new item for the linked list. The parameter *dt* is the data type that the item will be {INT, CHAR, DOUBLE, BOOL, or STRING}. Note that this class is abstract, so this constructor will only be called for objects of the derived class. The actual data value that will be stored is initialized in the derived class constructor. The derived class constructor must specify a call to this one.
- **dataElementBase(const dataElementBase& copy)** - This constructor will initialize the **this** object to be a copy of the parameter object. The dataType is the only thing that should be copied. Do not copy the *next* and *prev* pointers, as this will link the new item into the original list (*seen below*).



The green item is the new copy of the second element in the original list (*in blue*). The problem here is that the original list doesn't know that the new item is linked in, therefore, we cannot expect this list to properly deallocate. Because of this problem, we will treat the copy constructor as a method to create a new item with the same dataType that will be linked in a completely different list. So, just initialize *next* and *prev* to be nullptr.

- **void link(dataElementBase* node)** - This function is used to link a node into the pile. The parameter *node* to be linked will always be linked to the right of *this* **dataElementBase** object. That is, it will always be linked through the *this->next* pointer. Assign the parameter *node* to the *next* pointer. The *node* parameter has a *prev* pointer that needs to be linked as well. This should only be done if *node* is not **nullptr**.
- **unsigned short getDataType() const** - Simple getter for the ***dataType*** variable.
- **virtual void* getData() = 0** - This will grab the data being stored in our node. This act as a getter for the **dataElement** class, but will return the data generically as a void*.

- **virtual void printElement(ostream& out) = 0** - This will print the data stored in the node to the given stream. We will implement this later in the **dataElement** class.
- **virtual ~dataElementBase()** - This destructor has to exist in order to properly deallocate dataElements that contains objects of a class.
- **short compare(dataElementBase* deb)** - Tells how two **dataElementBase** compare to one another by comparing this to the other object, *deb*, passed. Returns an enumeration-esque short integer on the status of the comparison defined by preprocessor directives at the top of dataProcessing.h. Can return LESSTHAN, GREATERTHAN, or EQUAL. This will compare the two objects based first on their type, followed by comparing the actual contents if the types match. The priority can be seen in the other preprocessor directives or earlier in the **Overview** section.
- **void operator =(const dataElementBase& copy)** - = overload that makes a deep copy of the given dataElementBase object to **this** one. Should copy only the dataType of the other over to this one. Similar to the copy constructor, we do not want to link the **this** object into the list that the data is being copied from, therefore, leave the *next* and *prev* pointers as they are or just set them to a default of nullptr.
- **dataElementBase* getNext() const** - Getter for the ***next*** member.
- **dataElementBase* getPrev() const** - Getter for the ***prev*** member.
- **static dataElementBase* getNewDataElement(unsigned short dataType, string data)** - This is where the code can get kind of funky. The job of this function is to return a reference to a brand new data element for a linked list. The *dataType* parameter is the type of data being stored and the *data* parameter is the data to be stored. The pointer that is returned will contain a value given by **new**. An example of this can be seen below, in which we return a reference to a new data element that is of *dataType* type:

```
1    return new dataElementBase(dataType);
```

  Unfortunately, this does not work because we cannot instantiate an abstract class. Actually, a reference to a derived class object must be returned. Note here that derived class objects are of template type, thus they need to be allocated appropriately. For example, if the data type of our data element is INT, the code may look like:

```
1    return new dataElement<int>(dataType , stoi(data));
```

  The "?" is where the *data* parameter will go, though it will need to be converted to the proper datatype. You have access to **stoi()**, **stod()**, and a helper function called **stob()** (*for bools*). This function should use the given dataType to figure out the type, construct a **dataElement** object of the corresponding type as seen above, and then set its data field after converting it using the appropriate conversion function.
- **static dataElementBase* deepCopyDataElement(dataElementBase *deb)** - This function operators similar to the one above. The only difference is the parameter, in which a deep copy of the parameter must be made. So, a reference to a derived object allocated with new is required. The main problem is getting the data being stored in the parameter. The dataType is easily accessible as shown below:

```
1    unsigned short dt = deb->dataType;
```

  Getting the data is different. You must use the virtual function **void* getData()**, which returns an address to *deb*'s data. It is stored as a void pointer though, so it needs to be appropriately casted per the dataType. For example:

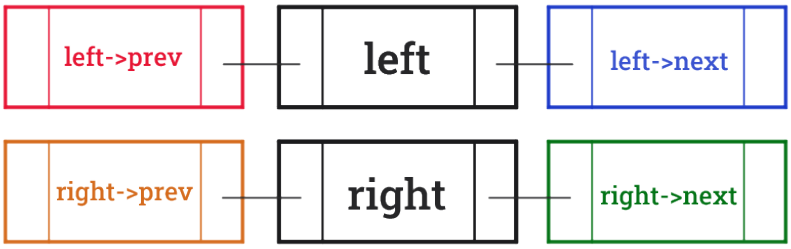
```
1    unsigned short dt = deb->dataType;
2    if(dt == INT){
3        void *ref = deb->getData();
4        int result = *(static_cast<int*>(ref));
5    }
```

The code above shows how the void pointer can be casted to an int pointer and then dereferenced to deep copy the data. Once you have this, you have the ability to return a proper deep copy.

```
1     return new dataElement<int>(dt,result);
```

- **void swap(dataElementBase* left, dataElementBase* right)** - Swaps the left and right dataElementBases. This needs to consider the elements to the left and right of each given node, in other words, their prev and next, respectively. The prev and next of each should be swapped, and the prev and next **dataElementBase** nodes of each will also need their next and prev pointers updated, respectively. As a safety measure, it is recommended to check that prev and next are each not null before setting them. This cannot modify the head or tail since it is not a **garbagePile** operator, so it is recommended to do any checks for these alongside any calls to this function you may do elsewhere.

  Below is a diagram of what the nodes would look like prior to the swap. Afterwards, the prev and next of the left and right should swap places. So, the original left->prev and left->next must now point to the right and the right->prev and right->next must point to the left. As a starting point, the **left->prev**->next should = the right afterwards.There should be a total of six assignments. It is recommended to use 4 locals to represent the 4 colored nodes in the diagram below to make readability and coding on your part easier. Be careful not to leave a memory leak.



  Note: The above case only works if the left and right nodes are **not adjacent**. If the nodes are adjacent, you must consider that the left->next is equal to the right node, and the right->prev is equal to the left. This swap will be slightly different than the case where the nodes are not adjacent.

dataElement

The fully implemented class that will make up a single element in our pile. This inherits from the **dataElementBase** class and implements all of its abstract functions. This class is also templated, meaning that several versions of this class will be generated by the compiler with the type of data able to be stored determined by what is filled in for the generic type, ***t***.

- ***t data*** - Generically typed variable that will store the data to be held in this **dataElement**. Equivalent to the data field of a linked node.
- **void* getData()** - Implementation of base class's abstract function. This just gets a pointer to/address of the data stored in this node.
- **void printElement(ostream& out) const** - Prints this **dataElement**'s data to the given stream.
- **dataElement(unsigned short dt, t d)** - This constructor inlines a call the base class constructor and passed on the dataType parameter. The data within the node is also intitialized with the template parameter.

garbagePile

The **garbagePile** class will act as a place to store all of our garbage. This is an expanded implementation of a doubly linked list, where each of our nodes is a **dataElement** object. This implementation will store both a head pointer pointing to the start of the list and a tail pointer pointing to the end of the list, as well as contain some extra functions to sort our pile and make copies.

- **garbagePile()** - Default constructor for the garbage pile. This should just set the dataElement pointers to nullptr and initialize the count to 0 by default.
- **garbagePile(const garbagePile& copy)** - This should make a deep of the given copy list and copy over all of the contents to *this* pile. This means that you should not copy the head and tail directly, but instead go through the copy list, make a copy of the data each **dataElementBase** node, and put it into a new node in *this* pile.
- **void operator=(const garbagePile& copy)** - The = operator overload should make a deep copy of the given garbagePile to this one in much the same way as the copy constructor. It is suggested to also delete any existing nodes that *this* may contain beforehand using **clearGarbagePile()** to avoid memory leaks.
- **~garbagePile()** - The destructor of our garbage. For simplicity, you can just call the **clearGarbagePile()** function.
- **void addItem(string garbage)** - Adds an item to the end of the pile. This will be given the data to add as a string. The type of the data will need to be determined, converted, and a **dataElement** of that type should be added to the end at the tail. It is recommended to use the static **dataElementBase getNewDataElement** function, link it to the tail using the **link** function, and then update the tail. If the head is uninitialized and the list is empty, this should also set the head to the new node.
- **void clearGarbagePile()** - **This function has to be recursive for full points.** This should delete all of the **dataElement**s in our pile. Start at the head of the list, then delete each node until the tail is reached. Remember to keep a temp variable when deleting to avoid memory leaks. Try storing a reference to the node you want to delete in the temp, moving to the next node, and then deleting the node being pointed to.
- **void sort()** - Sorts the **garbagePile dataElement**s first by the type contained in the **dataElement**, and then by the contents if relevant. The **compare()** function should do this comparison for you and return a resulting int about the result as described for that function. Sorting in a linked list can be tricky and inefficient, in this program we'll settle for a slow algorithm to make the programming a little less challenging. For the sorting, implement a BubbleSort like approach. Use two loops to iterate over the pile, access a given node using the [] overload for *this* object and the index, and then swap in the event that the elements are out of order. Since there is no > overload, consider using the **compare** function to compare two adjacent elements and **swap** as needed.
- **dataElementBase *operator[](unsigned int index)** - Gets the data stored int the **dataElement** node at the given index. This should start at the head pointer and traverse by following each ***next*** pointer as many times as the index. It is recommended to do this with a for loop.
- **friend ostream& operator«(ostream& out, garbagePile gp)** - Prints the entire **garbagePile** to the given stream, then returns the stream. This should start at the head, traverse, and then move to the next node until the tail is reached.
- **friend istream& operator»(istream& in, garbagePile &gp)** - This operator should read a string from the **in** parameter using getline and add the new string into the garbage pile using **addItem**.

TO-DO

It is suggested to start with the **dataElementBase** and **dataElement** classes and then try to move on to the **garbagePile** afterwards. Take it slowly and try to check that functions work as you go. CodeGrade has many unit tests that can be used to verify many of the smaller parts, so it is recommended to upload your code periodically and check which parts are working. You will need to implement code whenever you see **YOUR CODE HERE**. All the smaller unit tests are also uploaded. They are not labeled, so you can use CodeGrade or your own code reading skills to figure out what each unit program does.

Sample Run

```
./processor garbage.txt
Printing the garbage pile:
-------------------------
p , garbage , u ,
4756.46 , r , 1 ,
0 , 1 , 212 ,
5.78 , twitch.tv , 36.345 ,
a , 1 , 0 ,
i , m , 31 ,
-1.22345 , 72.88 , float ,
1 , u.s.a , 1 ,
4 , 0 , 4.22 ,
5 , 3.3.3 , 5.99 ,
1.23 , walmart , 19 ,
0 , 86 , k ,
-22 , acura rsx , g ,
1 , q , h ,
5.79 , 3 , -3.012 ,
red3 , 99 , 947 ,
d , 107 , is this thing on? ,
I don't know , 1 , f ,
c , 9 , m ,
james , 13 , b ,
1 , l , s ,
httpsHalo , general kenobi , 15 ,
integer , -37 , double ,
1 , 17 , 1 ,
big mess , register , help ,
1 , j , t ,
3 , 57 , 1 ,
man , 96 , life.time ,
7 , green5 , my brand! ,
1.22 , 1.24 , buying rs gf ,
blue4 , e , 0 ,
60 , 1 , 11 ,
61
*************
*  Sorting  *
*************
Printing the garbage pile:
-------------------------
-37 , -22 , 1 ,
3 , 5 , 7 ,
9 , 11 , 13 ,
15 , 17 , 19 ,
31 , 57 , 60 ,
61 , 86 , 96 ,
99 , 107 , 212 ,
947 , a , b ,
c , d , e ,
f , g , h ,
i , j , k ,
```

```
l , m , m ,
p , q , r ,
s , t , u ,
-3.012 , -1.22345 , 1.22 ,
1.23 , 1.24 , 3 ,
4 , 4.22 , 5.78 ,
5.79 , 5.99 , 36.345 ,
72.88 , 4756.46 , 0 ,
0 , 0 , 0 ,
0 , 1 , 1 ,
1 , 1 , 1 ,
1 , 1 , 1 ,
1 , 1 , 1 ,
1 , 3.3.3 , I don't know ,
acura rsx , big mess , blue4 ,
buying rs gf , double , float ,
garbage , general kenobi , green5 ,
help , httpsHalo , integer ,
is this thing on? , james , life.time ,
man , my brand! , red3 ,
register , twitch.tv , u.s.a ,
walmart
```