Project 4 – DNA

CS 251, Fall 2020, Reckinger

Collaboration Policy: By submitting this assignment, you are acknowledging you have read the collaboration policy in the syllabus for projects. This project should be done individually. You may not receive any assistance from anyone outside of the CS 251 teaching staff.

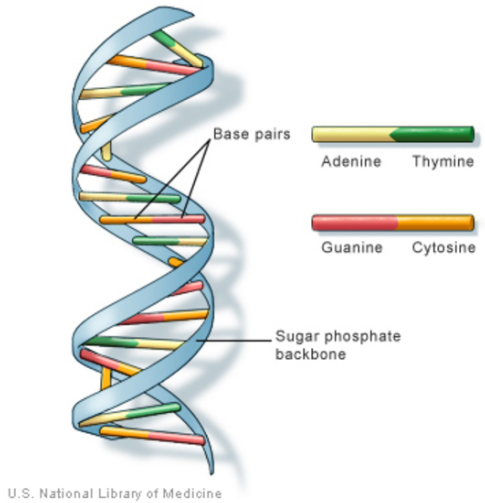
Late Policy: You are given a total of 5 late days to use at your discretion (for all projects, not each project). You may use the late days in 24-hour chunks (either 1 day at a time or all five at once). You do not need to alert the instructor to ask permission to use late days. You can manage your use of late days on Mimir directly.

Test cases/Submission Limit: You have a maximum of 15 submissions. You need to test your code own your own before submitting to the test cases. Most of the test cases are hidden.

What to submit: (1) dna.h; (2) tests.cpp; (3) application.cpp.

.pdf starter code solution.exe

Project Background



Source: https://medlineplus.gov/genetics/understanding/basics/dna/

DNA, the carrier of genetic information in living things, has been used in criminal justice for decades. But how, exactly, does DNA profiling work? Given a sequence of DNA, how can forensic investigators identify to whom it belongs?

Well, DNA is really just a sequence of molecules called nucleotides, arranged into a particular shape (a double helix). Each nucleotide of DNA contains one of four different bases: adenine (A), cytosine (C), guanine (G), or thymine (T). Every human cell has billions of these nucleotides arranged in sequence. Some portions of this sequence (i.e. genome) are the same, or at least very similar, across almost all humans, but other portions of the sequence have a higher genetic diversity and thus vary more across the population.

One place where DNA tends to have high genetic diversity is in Short Tandem Repeats (STRs). An STR is a short sequence of DNA bases that tends to be repeated back-to-back numerous times at specific locations in DNA. The number of times any particular STR repeats varies a lot among different people. In the DNA samples below, for example, Alice has the STR `AGAT` repeated four times in her DNA, while Bob has the same STR repeated five times.

Alice: CTAGATAGATAGATAGATGACTA

Bob: CTAGATAGATAGATAGATAGATT

Using multiple STRs, rather than just one, can improve the accuracy of DNA profiling. If the probability that two people have the same number of repeats for a single STR is 5%, and the analyst looks at 10 different STRs, then the probability that two DNA samples match purely by chance is about 1 in 1 quadrillion (assuming all STRs are independent of each other). So if two DNA samples match in the number of repeats for each of the STRs, the analyst can be pretty confident they came from the same person. CODIS, The FBI's DNA database, uses 20 different STRs as part of its DNA profiling process. More on this later.

Utimately, you are going to write a DNA profiling app. The app will be able to take a dna strand and determine who the DNA belongs to (using a provided database). We will be making some approximations to the above discussion to simplify our code writing. And there are lots of ways you could write an app like this, but for this project you will be required to solve it using a custom DNA abstraction that we will write.

Project Summary

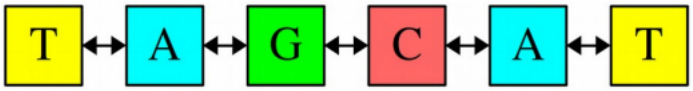
For this project, you are going to design a custom abstraction for DNA (in the file, dna.h). Unlike your last project which had a c-array under the hood, this implementation is a linked structure. As you work through the design, you will notice that implementing an abstraction using a linked structure is very different than implementing an abstraction using a c-array. Managing memory is different and also any updates to the implementation is very different. You MUST draw out what you are doing when you have a linked structure. There is absolutely no way to write this code unless you are diagramming as you go. To test your dna abstraction class, you must write tests using the Google Test framework, which you will write in tests.cpp. The starter code for tests.cpp sets up 10 tests cases for you, categorized by public member function. You can add more tests or just put your assertions in each test. You may/should have 100s of assertions (calls to EXPECT_EQ) in ***each*** of the TESTs. Think loops. Once you have finished testing your dna abstraction class, you can move on to the application.cpp which will implement the DNA profiling described above.

DNA Abstraction Class – dna.h and tests.cpp

DNA strands are made up units called nucleotides. There are four different nucleotides present in DNA, as discussed above: A, C, G, and T. We are going to model any DNA strand as a linked structure consisting of nucleotides that make up the strand, in order. To do this, we are going to set up a doubly-linked list made up of nucleotide:

```
struct Nucleotide {
    char value;
    Nucleotide *next;
    Nucleotide *prev;
};
```

If we wanted to store the DNA sequence for TAGCAT, it would look like this:



The prev pointer of the first T and the next pointer of the last T would each be set to nullptr, but will not be shown in the digrams (for simplicity). However, you probably will find it useful to draw them in when you are diagramming.
Check out the dna.h file which describes the functionality of each member function you must write. If you would like a step-by-step guide of how to proceed, please read the rest of this section.
STEP 1: Write the **toString** function first. It convert's the current object's linked structure into a string. With the example above, it will return the string "TAGCAT". It is one of the more straight forward functions to write, but it also very useful to use in your tests so it is best to get it implemented right away! You will not be able to test it quite yet.
STEP 2: Write the **default constructor** and **size**. For this constructor, you will allocate memory for two Nucleotide nodes and link them together properly. You can set the value of these two nodes to 'X'. Make sure to set all private member variables when writing the default constructor. Now, go ahead and implement the size() function. Now, you can write some tests to test everything you have implemented so far. Make sure you have Google Tests installed before testing (see section below). NOTE: there are a few tests in the starter code, make sure to comment out any of the public function assertions that you have not yet implemented before testing.
TIPS:

- Before you start, draw it out! And continue to diagram as you write.
- If you leave a pointer uninitialized in C++, it does not default to nullptr and instead points somewhere random. Uninitialized pointers like this will mostly likely crash your program immediately. It is a good idea to explicitly set pointers you don't want pointing at anything to nullptr.
- These functions must work on large inputs, therefore it likely means you won't able to implement recursively since it could lead to stack overflows on large strands.
- You must not use any additional C++ container types in your dna.h file.
- The isLinked() function is implemented for you. You can use it to test that your linked structure is linked properly. You should call it everytime you create or change the linked structure.

STEP 3: Write the **at** function. This at function is both a getter and setter into the dna abstraction. We are primarily implementing it as it helps us with testing. Note the at function does not insert elements, but just is able to access them or change the character stored in the value. Once you have this implemented, time to add tests! You should be able to test it in both the setting and getting mode on the default constructor.
STEP 4: Now, it is time to write the **second constructor**. The constructor has one parameter which is a string that represents the DNA strand. It constructs the dna object, which is a doubly linked structure of Nucleotides. You can think of this as the inverse of the toStrand method in some ways. However, this constructor must allocate memory for each Nucleotide created for the linked structure. Make sure that you set all member variables of each Nucleotide and also make sure you set all private member variables for the dna object. And finally, lots of testing to do. Now, you can test the second constructor with size, at, and toString. Make lots and lots of tests before moving on. Test large strands, empty strands,
TIPS:

- Remember your linked structure is doubly-linked. This will require you to do some extra wiring as you go. Do not forget to diagram!

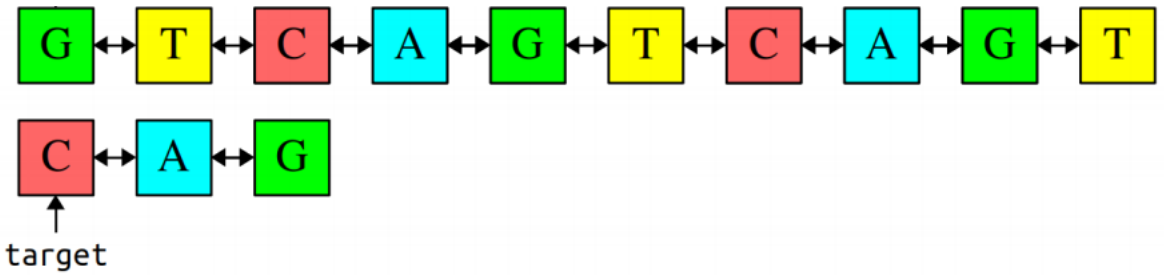
- gdb is a good tool for debugging on linux machines. See section below for more details. As you run into segmentation faults or have other bugs in your code, running your code on gdb can pinpoint where the problem is occurring.
- Your implementation of the second constructor should run in O(n) time.
- Your function must work on large inputs, so do not write your code recursively as it will cause stack overflows.
- You must not use any additional C++ container types in your dna.h file.
- Make sure to test theLinked() function on your newly constructed objects using your ***second constructor.***

STEP 5: Write the **clear** function and the **destructor**. Make sure you do not have duplicated code for these two functions. Since clear is a public member function, you can write tests for it! So, do that. Once you get your destructor implemented, test valgrind often.

STEP 6: Write the **operator==.** This operator is super handy for testing. It should go through two strands and determine if they contain the same values, in the same order. Note, we are not trying to determine if two objects are the same object, just that the two objects

STEP 7: Implement the **copy constructor** and **operator=**. Similar idea as previous classes we have written or dealt with. The testing is also similar. Test, test, test! Don't forget to test on both the default and second constructor and also on lots of different DNA strands. Don't forget to test it on valgrind.

STEP 8: Write **findFirst**, which is a private member function. The findFirst function searches for a copy of the strand given by the dna object target inside the current dna object. It should return a pointer to the first Nucleotide cell in the match. For example, given these DNA strands, findFirst should return a pointer to the third Nucleotide in the dna list. Note in this image target is the object itself not a pointer to first element in the list.

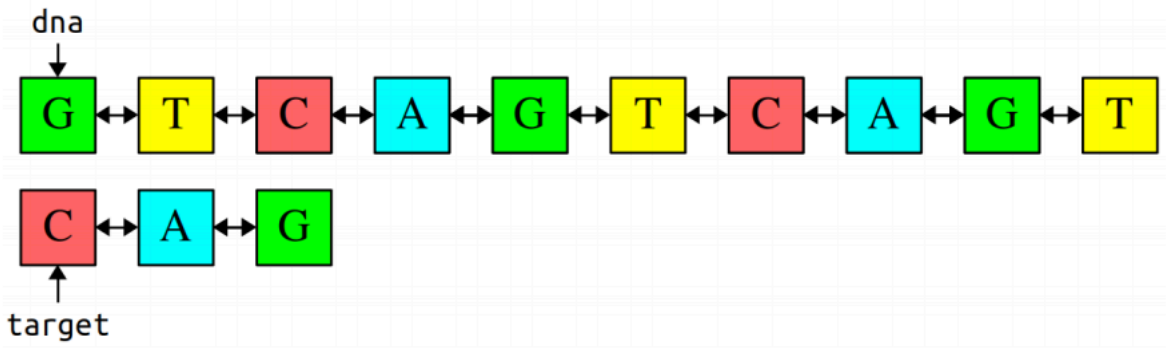


If the sequence given by target doesn't appear in the object you are searching, the function should return nullptr to indicate the sequence wasn't found. You should implement this function but will not be able to test it until you write the next function.

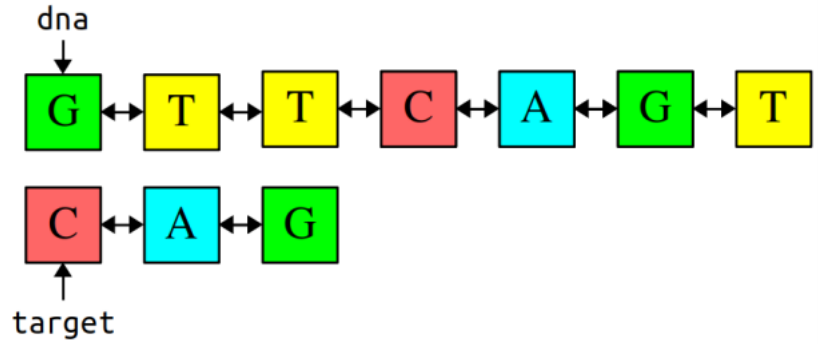
TIPS:

- You can assume target is non-empty.
- You can assume the target object and the current object you are searching are different objects.
- You should not be calling new or allocating any new memory for findFirst.
- There is no specific Big-O target for this function, but it should not time out during tests.
- This function should work very very long sequences, so do not use recursion.
- Your function must work on large inputs, so do not write your code recursively as it will cause stack overflows.
- You must not use any additional C++ container types in your dna.h file.

STEP 9: Write the **splice** function. The splice function takes in an input dna object named target. The current dna object should be searched and the target dna object is a specific sequence to chop out of the current object's DNA strang. Therefore, your function should find the first copy of target inside of the current dna object, then edit the dna strand by removing that sequence in its entirety. If that sequence does not exit, the function should return false to indicate that nothing was removed. For example, if we have these strands:



Calling splice on this target would give:



The first copy of "CAG" was deleted from the dna strand, though the second copy of "CAG" still exists. Splice only removes the first copy of target.

TIPS:

- Your function should leave the original strand unchanged except for the removed section and possibly the Nucleotides that appear just before and just after that section. In other words, you should only make local modifications to the strand rather than, say, allocating a new strand or changing which characters are stored in which Nucleotide objects.
- You can assume that Nucleotide objects that make up the target sequence are different objects than the Nucleotide objects that make up dna.
- Your implementation of splice should not leak any memory. In particular, if you remove any nucleotides from the DNA strand, you should be sure to delete them.
- You should not allocate any memory using new in the course of implementing splice, since you're not actually creating any new nucleotides. Feel free to declare variables of type Nucleotide* (remember that creating a pointer is separate from allocating memory using new), but don't use the new operator.
- If, in the course of coding this one up, you start seeing crashes or bugs in previous functions you have written, it could mean either that there's a bug in your splice function that caused the list to be wired improperly or that there was a lurking issue in one of those earlier functions that accidentally sneaked past the tests from the earlier steps. In other

words, don't immediately assume that one of the earlier functions is the culprit; investigate splice as well to make sure that it didn't leave the wiring between the nucleotides in a bad state.

- Gdb might be super helpful here. Don't hesitate to set breakpoints in test cases or step through your code if you find any issues with your program; this is a great way to see what your code is doing.
- You will probably draw out 10, 20, 30+ diagrams while writing this function. It is not possible to write this without drawing a lot of pictures. There are lots of edge cases and special cases to consider.
- There is no specific Big-O target for this function, but it should not time out during tests.
- Your function must work on large inputs, so do not write your code recursively as it will cause stack overflows.
- You must not use any additional C++ container types in your dna.h file.
- This is the most challenging part of the entire project.

DNA Profiling app – application.cpp

You are going to write an app that is able to build DNA strands using your dna abstraction class and then determine who the DNA matches to in a database. What might such a DNA database look like? Well, in its simplest form, you could imagine formatting a DNA database as a text file, wherein each row corresponds to an individual, and each column corresponds to a particular STR:

```
name,AGAT,AATG,TATC
Alice,28,42,14
Bob,17,22,19
Charlie,36,18,25
```

The data in the above file would suggest that Alice has the sequence AGAT repeated 28 times consecutively somewhere in her DNA, the sequence AATG repeated 42 times, and TATC repeated 14 times. Bob, meanwhile, has those same three STRs repeated 17 times, 22 times, and 19 times, respectively. And Charlie has those same three STRs repeated 36, 18, and 25 times, respectively. To write your app, you will need to do some file reading. You should use an appropriate C++ container to act as a database. Then, you should write the code that saves all the data from the text file and saves it into an appropriate C++ container.

So given a sequence of DNA, how might you identify to whom it belongs? Well, imagine that you looked through the DNA sequence for the longest consecutive sequence of repeated AGATs and found that the longest sequence was 17 repeats long. If you then found that the longest sequence of AATG is 22 repeats long, and the longest sequence of TATC is 19 repeats long, that would provide pretty good evidence that the DNA was Bob's. Of course, it's also possible that once you take the counts for each of the STRs, it doesn't match anyone in your DNA database, in which case you have no match. This is the most rigorous way to identify DNA. However, to keep things simple, we are going to simplify the algorithm a bit. Instead of finding the longest sequence of STRs, we are just going to count up all instances of each STRs in a particular DNA strand. The trick is: you can do this only using the dna abstraction class. You will be given DNA sequences in text files (see 1.txt, 2.txt, etc.). You will read in those files and construct a dna object:

```
fstream inFile(filename);
string dnastr;
inFile >> dnastr;
dna d(dnastr);
```

You will then count up how many str instances in the d object using only member functions of the dna class (basically…using splice). Remember that splice removes parts of the dna strand. Make sure you scope your dna object approrpiately so that each time you splice the dna with a new str, you are starting with the original dna strand (not a version that already has some other strs removed). Are there other and maybe better ways to solve this problem? Yes. But we are practicing working with classes with linked structures. And this application really stress tests your dna abstraction class. You can, no doubt, write up this application with purely string parsing. However, that is not allowed for this project. You also can't use any other C++ containers to repace the functionality that of the dna abstraction class. You will need some C++ containers to help with the database and database lookup, but the dna class member functions must be used to determine how many strs are in each dna strand.
Sample output (red are inputs):

```
Welcome to the DNA Profiling App!
Enter database textfile name: small.txt
Enter dna file: 1.txt

Searching data base...
DNA match: Bob
```

And another:

```
Welcome to the DNA Profiling App!
Enter database textfile name: large.txt
Enter dna file: 5.txt

Searching data base...
DNA match: Lavender
```

```
And another:
Welcome to the DNA Profiling App!
Enter database textfile name: large.txt
Enter dna file: 20.txt

Searching data base...
No match.
```

See solution.exe for more details on sample output and behavior. The small.txt file goes with 1-4.txt and large.txt goes with 5-20.txt.
For testing purposes, if you open up a file storing a dna strand (e.g. "1.txt") and ctrl+f one the strs, this should be the number your dna class determines. If you are thinking that there are cases when removing an str, creates a new instance of an str…you are right! For example if your DNA strand is "CT**CTAG**AGTT". You can see there is one instance of "CTAG". However, if I remove it, I am left with "**CTAG**TT", which is now another str. So are there 1 or 2? Well, that is a limitation to using splice, because it does not work in this case. Therefore, this case is not tested. You can assume that this does not happen.

Using gdb

This is a really nice tutorial on how to use gdb, including a video walkthrough:
https://web.stanford.edu/class/archive/cs/cs107/cs107.1194/resources/gdb

Google Tests

Your tests for this assignment need to be written using the Google Test framework. To write Google Tests, you will need to download some files and build them on Mimr. You will need to run these commands:

```
>>apt-get -y install libgtest-dev
>>apt-get -y install cmake
>>cd /usr/src/gtest/
>>cmake CMakeLists.txt
>>make
>>cp *.a /usr/lib
```

Mimir will clear this every time you are logged off for sometime. Instead of downloading and making each time, you should add these lines to your .bashrc file (located at ~/.bashrc). If you use the file directory gui on the left, you will need to go to the settings and check "show hidden files". Simply copy and paste all lines above into this file. You can either log off and log back on, or open a new terminal and it should automatically run those six commands.

Check out lectures from last week to find lots of Google Test examples. Note that there are two assertion tests: EXPECT_EQ and ASSERT_EQ, but there are lots of other assertions too: https://github.com/google/googletest/blob/master/googletest/docs/primer.md

EXPECT_* will continue to test all assertions listed in the TEST. ASSERT_* will leave the TEST once it reaches the first ASSERT call that is false. You can use either but might prefer one or the other in some cases.

Requirements

1. You may not change the API provided. You must keep all private and public member functions and variables as they are given to you. If you make changes to these, your code will not pass the test cases. You may not add any private member variables but you may add private member helper functions.
2. In dna.h, no additional C++ containers are allowed. No additional header files are allowed to be included.
3. In dna.h, he implementation must use a doubly-linked list structure. You may not add or use any private member variables that are not included in the starter code.
4. In dna.h, the required Big O complexity is indicated in the starter code and described in this handout. In some functions, the Big O is not specified and just needs to be reasonable.
5. In dna.h, do not write recursive functions. We are testing your code on very long DNA strands and this could lead to stack overflow problems.
6. You must have a clean valgrind report when your code is run. Check out the makefile to run valgrind on your code. All memory allocated (call new) must be freed (call delete). The test cases run valgrind on your code.
7. Your tests.cpp should have 100s of assertions for each public member function. You need to put in significant effort into testing. Check out the Mimir rubric for how this will be graded.
8. No global or static variables.
9. In application.cpp, you must use dna.h to implement the application. If you use any other containers or strings to store the dna and count the instances of str, you will receive deductions up to 75 pts. You will need to include other containers to store the data in the data base files (small.txt and large.txt), however, you may only store the the dna from the

dna files (1.txt, 2.txt, etc.) in dna objects. You also may only count str instances by calling public member functions in the dna class.

Citations/Resources

Assignment is inspired by Brian Yu and David J. Malan at Harvard University; Keith Schwarz at Stanford University; Owen Astrachan at Duke University; Richard Pattis at University of California, Irvine.