CSCI-1200 Data Structures — Spring 2021
Homework 3 — Jagged Arrays

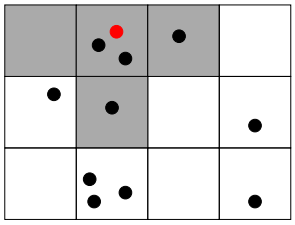
In this assignment you will build a custom data structure named `JaggedArray`. A `JaggedArray` is a 1D array of n bins, and each bin stores a collection of elements of templated type `T`. Building this data structure will give you practice with pointers, dynamic array allocation and deallocation, and writing templated classes. The implementation of this data structure will involve writing one new class. You are not allowed to use any of the STL container classes in your implementation or use any additional classes or structs. Please read the entire handout before beginning your implementation. Also, be sure to review the Lecture 8 notes.

Motivation

The `JaggedArray` class is found in many real-world applications where a variable number of elements are stored in a collection of bins. This data structure is helpful to ensure high performance for applications needing frequent read and write access to this organizational information.

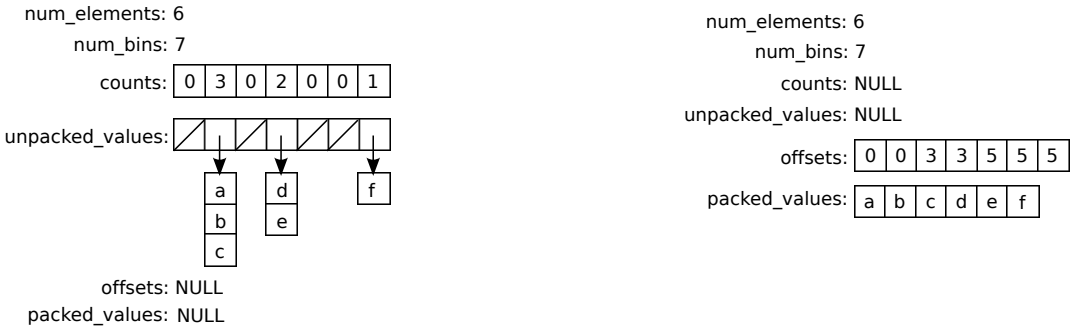
For example, let's consider a physical simulation with hundreds of ping pong balls bouncing around inside a box. We will need to calculate the collisions of the balls with the sides of the box, but also the collisions of the balls with each other. A naive, brute-force implementation will require consideration of all pairwise collisions – for n balls, this is n^2!

A relatively straightforward optimization will instead consider the interaction of each ball with only its nearest k neighbors $= k*n$ total potential collisions (a great improvement for large n!). How do we efficiently find each ball's nearest neighbors? If we divide space into a large grid of bins and sort all the balls into the bins, we can find the nearest neighbors of a particular ball by collecting the balls in the same bin (and the adjacent bins). How many balls are in each bin? Some bins are empty. Some bins have several or many balls. As the simulation progresses and the balls move around, balls move in and out of bins and the number of balls in each bin changes. A `JaggedArray` is an obvious choice for storing this data for efficient collision detection.



The Data Structure

The `JaggedArray` class you will implement has two fundamental representation modes: *unpacked* and *packed*. In the unpacked representation, the `JaggedArray` stores an array of integers containing the counts of how many elements are in each bin and an array of pointers to arrays that hold these elements. In the packed representation, the `JaggedArray` stores an array of offsets for the start of each bin. These offsets refer to a single large array that stores all of the elements grouped by bin but all packed into the same large array. We can convert a `JaggedArray` from unpacked to packed mode by calling the `pack` member function and similarly convert from packed to unpacked by calling `unpack`. Below is an illustration of both representations for a small example with 7 bins storing 6 total elements (unpacked on the left, packed on the right):

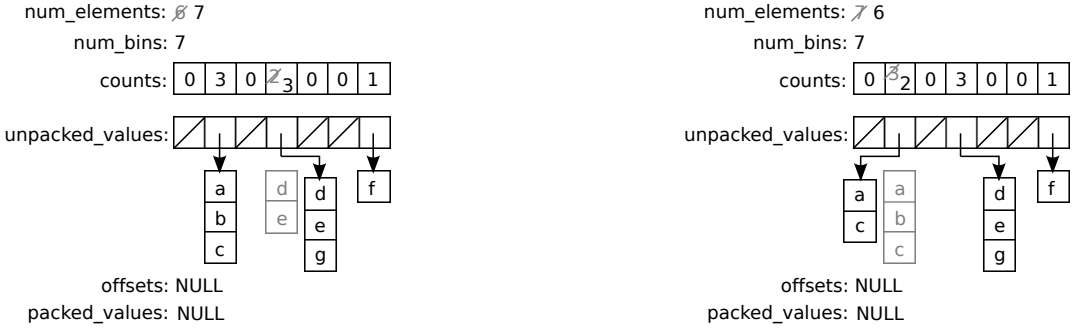


The `JaggedArray` stores elements of template type `T`. In the examples in this handout, `T` is type `char`. The representation consists of the six member variables presented in these diagrams and you should exactly follow this representation (variable names, types, and data). Depending on the mode (`unpacked` or `packed`) of the `JaggedArray`, two of the variables are set to NULL. The `JaggedArray` contains the expected accessor functions `numElements`, `numBins`, `numElementsInBin`, `getElement`, and `isPacked` and modifiers `addElement`, `removeElement`, and `clear`. See the provided `main.cpp` code for example usage and details on the arguments and return values for these functions.

In the packed version of the data structure, the first element in bin i is stored at the index stored in `offsets[i]`. The number of elements in the bin is equal to `offsets[i+1] - offsets[i]`. Except the last bin, which contains `num_elements - offsets[i]` elements.

Modifying the Data Structure

The `JaggedArray` can only be edited while in *unpacked mode*. The left diagram illustrates the necessary changes to the representation for the call `addElement(3,'g')`, which adds the element `'g'` to bin 3 (the bins are indexed starting with 0). And the diagram on the right illustrates the changes to the representation for the subsequent call to `removeElement(1,1)` which removes the element in slot 1 (element `'b'`) from bin 1.



Note how the variables holding the total number of elements in the `JaggedArray` and the count of the elements for the specific bin are updated appropriately.

For our implementation we will always size each of the arrays to exactly contain the required information. This will ensure you get plenty of practice with proper allocation and de-allocation. When the new array is allocated, the necessary data from the old array is copied and then the old array is de-allocated.

Attempting to access the non-existent bins or elements (out-of-bounds indices) is an error. Similarly, attempting to edit a `JaggedArray` while the array is in `packed` mode is an error. Your program should check for these situations, and if a problem is detected, your program should print an appropriate warning message to `std::cerr` and call `exit(1)`.

Testing, Debugging, and Printing

We provide a main.cpp file with a wide variety of tests of your data structure. Some of these tests are initially commented out. We recommend you get your class working on the basic tests, and then uncomment the additional tests as you implement and debug the key functionality of the `JaggedArray` class. Study the provided test cases to understand what code triggers calls to your `JaggedArray` copy constructor, assignment operator, and destructor and verify that these functions are working correctly.

It is your responsibility to add additional test cases, including examples where the template class type `T` is something other than `char`. You must also implement a simple `print` function to help as you debug your class. Include examples of the use of this function in your new test cases. Your function should work for `JaggedArray`s containing `char`, `int`, and *reasonably short* `string`s. The `print` function does not need to work for more complex types. Please use the example output as a guide (we will grade this output by hand).

Performance Analysis

The data structure for this assignment (intentionally) involves a lot of memory allocation & deallocation. Certainly it is possible to revise this design for improved performance and efficiency or adapt the data structure to specific applications. For this assignment, please implement the data structure *exactly* as described.

In your `README.txt` file include the order notation for each of the `JaggedArray` member functions described above: `numElements`, `numBins`, `numElementsInBin`, `getElement`, `isPacked`, `clear`, `addElement`, `removeElement`, `pack`, `unpack`, and `print` *and* don't forget the constructors, destructor, and assignment operator. For each function determine the performance for each of the representation modes (*packed* and *unpacked*), as appropriate. You will be graded on the order notation efficiency of your implementation, so carefully consider any significant implementation choices.

You should assume that calling `new []` or `delete []` on an array will take time proportional to the number of elements in the array. In your answers use the variables b = the number of bins, e = the number of elements, and k = the number of elements in the largest bin.

Looking for Memory Leaks

To help verify that your data structure does not contain any memory leaks and that your destructor is correctly deleting everything, we include a batch test function that repeatedly allocates a `JaggedArray`, performs many operations, and then deallocates the data structure. To run the batch test case, specify 2 command line arguments, a file name (`small.txt`, `medium.txt`, or `large.txt`) and the number of times to process that file. If you don't have any bugs or memory leaks, this code can be repeated indefinitely with no problems.

On Unix/Linux/OSX, open another shell and run the `top` command. While your program is running, watch the value of "RES" or "RPRVT" (resident memory) for your program. If your program is leaking memory, that number will continuously increase and your program will eventually crash. Alternately, on Windows, open the Task Manager (Ctrl-Shift-Esc). Select "View" → "Select Columns" and check the box next to "Memory Usage". View the "Processes" tab. Now when your program is running, watch the value of "Mem Usage" for your program (it may help to sort the programs alphabetically by clicking on the "Image Name" column). If your program is leaking memory, that number will continuously increase.

Memory Debuggers

We also recommend using a memory debugging tool to find errors. Information on installation and use of the memory debuggers "Dr. Memory" (available for Linux/MacOSX/Windows) and "Valgrind" (available for Linux/OSX) is presented on the course webpage:

http://www.cs.rpi.edu/academics/courses/spring21/csci1200/memory_debugging.php

The homework submission server is configured to run your code with Dr. Memory to search for memory problems. Your program must be memory error free to receive full credit.

Submission

Do all of your work in a new folder named `hw3` inside of your Data Structures homeworks directory. Use good coding style when you design and implement your program. Be sure to make up new test cases and don't forget to comment your code! Please use the provided template README.txt file for any notes you want the grader to read. You must do this assignment on your own, as described in the "Collaboration Policy & Academic Integrity" handout. If you did discuss the problem or error messages, etc. with anyone, please list their names in your README.txt file. Details about the Wednesday night extension will be available on the Submitty submission page.