ECS 60 **Programming Assignment #1 (50 points)** **Fall 2015**

Due: Wednesday, October 7th, Write-ups: 4:00pm; Programs: 11:59pm.
Filenames: timetest.cpp, deque.cpp, deque.h, and authors.csv.

Format of authors.csv: author1_email,author1_last_name,author1_first_name
author2_email,author2_last_name,author2_first_name
For example: simpson@ucdavis.edu,Simpson,Homer
potter@ucdavis.edu,Potter,Harry

Use handin to turn in just your files to cs60. Do not turn in any Weiss files, object files, makefiles, or executable files! You will find copies of my own executables in ~ssdavis/60/p1. All programs will be compiled and tested on Linux PCs. All programs must match the output of mine, except that the CPU times may be different. Do not get inventive in your format! Programs are graded with shell scripts. If your program asks different questions, or has a different wording for its output, then it may receive a zero!

#1. Timetest: (30 points with 25 points for write-up and 5 points for timetest.cpp, 15 minutes)

Write a driver program, timetest.cpp, that will ask for a filename and repeatedly asks the user for the ADT to which he/she wishes to apply the commands from the specified file. You will then run your program and note the performance of ADT in a two or three page typed, double-spaced write-up. **Hand written reports will receive no points!** We will be storing only integers for this assignment. The format of each file will be:

First Line: A string describing the contents of the file.
Second Line: {<command (char)> [values associated with command]}+

The two commands in the files are: 1) insert: 'i' followed by an integer and a space character; and 2) delete 'd' followed by an integer and a space. Since only the list ADTs can delete a specific value, you need delete the specific value for only those three implementations. For stack, simply pop the next integer, and queue simply dequeue the next integer, no matter what the value is specified by the delete command. Some ADT constructors require a maximum size parameter. You should hard code this to 250,000.

Your driver program will need all of the following files from ~ssdavis/60/p1: CPUTimer.h, LinkedList.cpp, StackAr.cpp, dsexceptions.h, CursorList.cpp, LinkedList.h, StackAr.h, CursorList.h, QueueAr.cpp, StackLi.cpp, vector.cpp, QueueAr.h, StackLi.h, vector.h, SkipList.h, SkipList.cpp, File1.dat, File2.dat, File3.dat, and File4.dat. You may copy the .h and .cpp files, but you should simply link to the .dat files using the UNIX ln command.: `ln -s ~ssdavis/60/p1/*.dat .` (Note the period to indicate your current directory.)

To make CursorList compile you should add the following line below your #includes, and pass cursorSpace in the CursorList constructor:

```
vector<CursorNode <int> > cursorSpace(250000);
```

After you've completed your program, apply File1.dat, File2.dat, File3.dat, and File4.dat three times to each ADT and record the values returned. You will find the shell script run3.sh in ~ssdavis/60/p1 that will do that for you, assuming the name of your executable is a.out. Just type "run3.sh", and then look at the file "results" to see the times for all eighteen runs.

In your write-up, have a table that contains the values for each run, and the average for each ADT for each file. Another table should contain the time complexity for each ADT for each file; this should include five big-O values: 1) individual insertion; 2) individual deletion; 3) entire series of insertions (usually N times that of an individual insertion); 4) entire series of deletions (usually N times that of an individual deletion); and 5) entire file. These two tables are not counted as part of the required two or three pages need to complete the assignment. For each ADT, you should explain how the structure of each file determined the big-0. Concentrate on <u>why</u> ADTs took longer or shorter with different files. Do not waste space reiterating what is already in the tables. For example, you could say "Stacks perform the same on the three files containing deletions because they could ignored the actual value specified to be deleted." The last section of the paper should compare the ADTs with each other. Most of the differences can be directly explained by their complexity differences. The lion's share of the last section should explain why some ADTs with the same complexities have different times. In particular, why is the CursorList slower than the normal list? You should step through Weiss' code with gdb for the answer.

The members of a team may run the program together, but each student must write their own report. Turn in this write-up to the appropriate slot in 2131 Kemper. If you declare ct as a CPUTimer then the essential loop will be:

```
do
{
     choice = getChoice();
     ct.reset();
     switch (choice)
     {
       case 1: RunList(filename); break;
       case 2: RunCursorList(filename); break;
       case 3: RunStackAr(filename); break;
       case 4: RunStackLi(filename); break;
       case 5: RunQueueAr(filename); break;
       case 6: RunSkipList(filename); break;
     }

     cout << "CPU time: " << ct.cur_CPUTime() << endl;
} while(choice > 0);
```

A sample run of the program follows:

```
% timetest.out
Filename >> File2.dat

      ADT Menu
0. Quit
1. LinkedList
2. CursorList
3. StackAr
4. StackLi
5. QueueAr
6. SkipList
Your choice >> 1
CPU time: 15.66

      ADT Menu
0. Quit
1. LinkedList
2. CursorList
3. StackAr
4. StackLi
5. QueueAr
6. SkipList
Your choice >> 0
CPU time: 0
%
```

#2. Deque: (20 points, 1 hour) Filenames: deque.cpp, deque.h

According to the C++ STL standard a deque, a double ended queue, has a constant time complexity based on the number of elements stored for all of the following operations: push_front(), pop_front(), push_back(), pop_back(), and positional accesses using the [] overloaded operator. There are no operations to insert or remove within a deque. The guarantee of constant time is over the long run no matter what operations are called, and doesn't speak to the underlying data structure(s). One source on their implementation is http://stackoverflow.com/questions/8627373/what-data-structure-exactly-are-deques-in-c/8632350#8632350

You are to implement a deque template class that can process a series of these operations as quickly as possible while being space efficient. Your program will be graded based on its speed, and the amount of RAM dynamically allocated after the last operation.

The grading script will copy Makefile, dequeRunner.cpp, mynew.cpp, mynew.h, and CPUTimer.h into your directory, and then calls make to create your executable. You will find those four files, barebones deque.cpp and deque.h files, data files, Makefile, my deque.out, and CreateDeque.out in ~ssdavis/60/p1.

Here are the specifications:

1. DequeData files
   - 1.1. These files are read by the DequeRunner code, so you do not have to worry about their format. Nonetheless, if you are curious, here is the format.
     - 1.1.1. The first line of the files has the number of operations listed in the file.
     - 1.1.2. Each succeeding line describes one operation.
       - 1.1.2.1. The operation type is specified with a single character: F = push_front, f = pop_front, B = push_back, b = pop_back, A = set value using operator[], a = read value using operator[].
       - 1.1.2.2. F, B, A, and a, are all followed by an unsigned short used by their respective operations.
       - 1.1.2.3. A, and a, both have an integer after their short which is the index to be accessed.
   - 1.2. The names of the files have the following format: deque-<maximum_size>-<final_size>-<seed for random number generator>.txt
   - 1.3. During the first phase of each file, the percentages of these operations are approximately: 35% push_front, 35% push_back, 10% operator[] setting, 10% operator[] reading, 5% pop_back, and 5% pop_front until it reaches its maximum size.
   - 1.4. During the second phase of each file, the percentages of these operations are approximately: 5% push_front, 5% push_back, 10% operator[] setting, 10% operator[] reading, 35% pop_back, and 35% pop_front until it reaches its final size.
   - 1.5. While maximum size is set by the user, final size is set randomly and will be between 25% and 50% of the maximum size.
   - 1.6. You may assume all operations are valid, e.g., indices are less than the current size of the deque.
2. Dynamic Memory Allocation.
   - 2.1. You must use "new," and "delete" to manipulate dynamic memory allocations. You may not use malloc(), free(), nor currentRAM anywhere in your source code. No entity larger the 80 bytes may be created without using new. currentRAM keeps track of the total amount of dynamic RAM allocated at that moment.
3. Grading
   - 3.1. Performance will be tested with three DequeData files having maximum sizes of approximately 1,000, 100,000, and 5,000,000 elements.
   - 3.2. The program must work properly to receive any points. This is assessed by checking the values returned by the "a" operation that reads from the operator[]. If a program has any printed errors, then it will receive zero for the entire program.
   - 3.3. (10 points) Total of the CPU time of the three runs: min (15, 10 * Sean's CPU Time / Your CPU Time);
     - 3.3.1. CPU time may not exceed 10.
     - 3.3.2. Programs must be compiled without any optimization options. You may not use any precompiled code, including the STL and assembly.
   - 3.4. (10 points) Total of final currentRAM when the program terminates from each of the three runs: min(15, 10 * Sean's RAM / Your RAM)
4. Suggestion
   - 4.1. I found adding a print() method to the Deque class that printed out the indices and contents of the entire deque very useful for debugging.

```
[ssdavis@pc15 p1]$ deque.out deque-1000-460-2.txt
CPU Time: 6.8e-05 RAM: 42050
[ssdavis@pc15 p1]$ deque.out deque-100000-45254-3.txt
CPU Time: 0.008597 RAM: 132816
[ssdavis@pc15 p1]$ deque.out deque-5000000-1921699-4.txt
CPU Time: 0.250829 RAM: 3919759
[ssdavis@pc15 p1]$
```

```
int main(int argc, char** argv)
{
  int opCount, index, retval;
  Operation *operations = readFile(argv[1], &opCount);
  currentRAM = 0;
  CPUTimer ct;
  Deque <unsigned short> *deque = new Deque<unsigned short>();

  for(int i = 0; i < opCount; i++)
  {
    switch(operations[i].op)
    {
      case 'F' : deque->push_front(operations[i].value); break;
      case 'B' : deque->push_back(operations[i].value); break;
      case 'f' : deque->pop_front(); break;
      case 'b' : deque->pop_back(); break;
      case 'A' : (*deque)[operations[i].index] = operations[i].value; break;
      case 'a' :
        index = operations[i].index;
        retval = (*deque)[index];

        if(retval != operations[i].value)
          cout << "Error: Operation #" << i << " index #" << index
            << " returned " << retval << " instead of " << operations[i].value
            << endl;
        break;
      default: cout << "Should never get here\n";
    }  // switch
    // deque->print();
  }  // for
  cout << "CPU Time: " << ct.cur_CPUTime() << " RAM: " << currentRAM << endl;
  return 0;
} // main()
```

```
[ssdavis@pc15 p1]$ CreateDeque.out
Maximum number of elements (10 - 5000000): 10
Seed (any integer): 1
[ssdavis@pc15 p1]$ cat deque-10-4-1.txt
23
F20738
B48946
f
B50101
B10282
f
B9158
F20955
F6772
B9521
B57207
F9092
F2507
F16912
f
a2507 0
a20955 3
b
b
b
f
A15382 4
b
[ssdavis@pc15 p1]$
```