EECS 560
Lab 2: Doubly Linked List Algorithmic Time Complexity Analysis

Apoorv Ingle, Prof. Shontz
Fall 2017

1 Lab Details

• Maximum Possible Points: 100

• Lab Timings:
  1. Monday Lab: Aug 28, 9:00 AM–10:50 AM
  2. Wednesday Lab: Aug 30, 12:00 PM–1:50 PM
  3. Friday Lab: Sept 1, 12:00 PM–1:50 PM

• Lab Due:
  1. Monday Lab: Sept 10, 5:00 PM
  2. Wednesday Lab: Sept 12, 5:00 PM
  3. Friday Lab: Sept 14, 5:00 PM

2 Assignment Instructions

In this lab we will practically verify whether the functions that were implemented for doubly linked list adhere the algorithmic complexity in reality. You would also have to implement and analyze a new function getPositionList as explained in §3.

In addition to doubly linked list, you would implement a sparse matrix and some simple operations on it. You will then try to analyze the performance of the operations you have implemented (please see §4 for more details).

Note: If you have had problems in implementing the operations on DoublyLinkedList in Lab 1 please fix them before you start this lab.

2.1 Analysis of add element at tail

Generate a doubly linked list with random integers between 0 and INT_MAX with the following sizes: 10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000 elements. Generate a random integer between 0 and INT_MAX. Call the function void DoublyLinkedList::add(int, int) to add the randomly generated element into the linked list’s tail position and time it.

For each of the sizes mentioned above repeat the experiment at-least 10 times and note the timings for each run. Plot a graph of the list size (x-axis) vs. the average time taken in milliseconds (or nanoseconds) (y-axis).
2.2 Analysis of delete elements

Generate a doubly linked list with random integers between 0 and INT_MAX with the following sizes: 10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000 elements. Choose a random integer that exists in the list. Call the function void DoublyLinkedList::deleteAll(int) to delete all the integers with the value equal to the chosen element and time it.

For each of the sizes mentioned above repeat the experiment at-least 10 times and note the timings for each run. Plot a graph of the list size (x-axis) vs. the average time taken in milliseconds (or nanoseconds) (y-axis).

2.3 Analysis of getPositionList

Generate doubly linked list with random integers between 0 and INT_MAX with the following sizes: 10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000 elements. Call the function DoublyLinkedList* DoublyLinkedList::getPositionList(DoublyLinkedList*) with the argument to the function being the the randomly generated linked list and time it. For example:

myList = {3, 1, 9, 6, 1}
myList.getPositionList(myList) //should return a new doubly linked list with values {0, 1, 2, 3, 1}

For each of the sizes mentioned above repeat the experiment at-least 10 times and note the timings for each run. Plot a graph of the list size (x-axis) vs. the average time taken in milliseconds (or nanoseconds) (y-axis).

2.4 Analysis of checking equality for square sparse matrices

Generate 2 square sparse matrix (Say $M_1$ and $M_2$) with less than 10% non-zero elements of the following sizes: 100 x 100, 500 x 500, 1000 x 1000, 5000 x 5000, 10000 x 10000. Ensure the non-zero elements are randomly distributed across your matrix. For each size of matrix make 5 runs where you time 2 equality operations:

1. computing equality between the 2 generated sparse square matrices. i.e. $M_1.equals(M_2)$
2. compute equality of 1st randomly generated matrix on itself. i.e. $M_1.equals(M_1)$

Compute the Average of 1. (say $T_{avg1}$) and 2. (say $T_{avg2}$).

Plot the size of matrix on x-axis vs. $T_{avg1}$ and $T_{avg2}$ on y-axis.

2.5 Analysis of transpose of square sparse matrix

Generate a sparse square matrix with less than 10% non-zero elements of the following sizes: 100 x 100, 500 x 500, 1000 x 1000, 5000 x 5000, 10000 x 10000. Ensure the non-zero elements are randomly distributed across your matrix. For each size make 10 runs where you time the transpose operation. Compute the average time taken to transpose for each size of matrix (Say $T_{avg}$).

Plot a graph of size of the matrix on x-axis vs. $T_{avg}$ on y-axis.

2.6 Makefile changes

Please use and modify the Makefile given with the lab kit to compile and run your analysis code. There are 5 unimplemented Makefile targets:

1. make analyze-add-tail This target will run and print out the 10 runtimes for each of the list sizes as explained in §2.1
2. make analyze-delete This target will run and print out the 10 runtimes for each of the list sizes as explained in §2.2

---

1 You can use rand function with proper scaling to generate positions and values of the elements
3. **make analyze-getPositionList** This target will run and print out the 10 runtimes for each of the list sizes as explained in §2.3.

4. **make analyze-matrix-equality** This target will run and print out the runtimes for each of the matrix sizes as explained in §2.4.

5. **make analyze-matrix-transpose** This target will run and print out the runtimes for each of the matrix sizes as explained in §2.5.

Your `make` targets may print the times in the following manner.

```bash
$ make analyze-add-tail
g++ -Wall -g -pedantic --std=c++11 -c list.cpp -o list.o
g++ -Wall -g -pedantic --std=c++11 -c analyzeAdd.cpp -o analyzeAdd.o
g++ -Wall -g -pedantic --std=c++11 -o analyzeAdd analyzeAdd.o list.o
./analyzeAdd
size of list: 10
times: 0.03 0.04 0.06 0.10 0.05 0.05 0.02 0.04 0.07 0.05
average time: 0.053

size of list: 50
times: 0.03 0.04 0.06 0.10 0.05 0.05 0.02 0.04 0.07 0.05
average time: 0.053
.
.
size of list: 100000
times: 0.03 0.04 0.06 0.10 0.05 0.05 0.02 0.04 0.07 0.05
average time: 0.053

Note: The timings given above do not represent actual times and are only for illustration purposes.

You may want to create 5 different main functions (in 5 different *.cpp files) that compile and execute when you run each of the 5 targets `make analyze-add-tail`, `make analyze-delete` and `make analyze-getPositionList`, `make analyze-matrix-equality`, `make analyze-matrix-transpose`.

You can make use of the pre-existing random number generation helper functions `rand` and `srand` to generate random values for linked list and position and values for sparse matrix.

You can use `Timer` class from `timer.hpp` to time the operation given in the lab-kit.

For this lab you may not use any standard template library that give you the implementation of vectors, lists etc. Please use the principles of object oriented programming to design your program.

### 3 Operations on Doubly Linked List

1. All the operations implemented as part of Lab 1.

2. **DoublyLinkedList**\* DoublyLinkedList::getPositionList(DoublyLinkedList\*) This returns a DoublyLinkedList which stores the first occurrence of all the elements given in the argument linked list. if the element does not exist the node created for the element will be set to -1. Please consider the following as an illustration in pseudo-code

```cpp
myList = {3, 4, 1, 7, 1, 2, 4, 6, 8, 1}
elems = {3, 1, 9, 6}
myList.getPositionlist(elems) //should return a new DoublyLinkedList with values
{0, 2, -1, 7}
```
**Hint:** Can you make use of a list method that you had implemented in Lab 1 for generating the position list?

## 4 Square Sparse Matrix and its Operations

Sparse matrices are regular matrices but with very few non-zero elements. It would be inefficient to store them in a 2D array as most of the elements will be zero. A better way would be to store only non-zero elements in a modified doubly linked list. For simplicity we will assume that we always have a square matrix of integer values only and once the matrix is created we will not modify its dimensions.

### 4.1 Data Structure Design

One possible design would be to modify the nodes to have an extra field to contain the position of the element in the matrix in addition to the value. For example consider matrix (Say $M_1$)

\[
\begin{bmatrix}
0 & 0 & 1 \\
0 & 0 & 0 \\
4 & 0 & 0
\end{bmatrix}
\]

In our sparse matrix representation for $M_1$ we would have a doubly linked list with two nodes. Assuming we use row major format: first node will contain value 1 with position as (0,2) and second node will contain value 4 with position as (2,0)

\[M_1 = \{(2,0), 1\}, \{(0,2), 4\}\]

### 4.2 Operations

The operations that you have to implement for the sparse matrix are

1. **Equal**: With function signature—`bool SparseMatrix::equals(SparseMatrix*)`. This method takes another matrix of as an input and returns true if the two matrices are equal. Please recall that the two matrices are equal if and only if their dimensions are equal and their values at each corresponding positions are equal. For example:

\[
\begin{bmatrix}
0 & 1 \\
0 & 0
\end{bmatrix}.equals\left(\begin{bmatrix}
0 & 1 \\
1 & 0
\end{bmatrix}\right)
\]

should return `false`

\[
\begin{bmatrix}
0 & 1 \\
0 & 0
\end{bmatrix}.equals\left(\begin{bmatrix}
0 & 1 \\
0 & 0
\end{bmatrix}\right)
\]

should return `true`

2. **Transpose**

With function signature—`SparseMatrix* SparseMatrix::transpose()`. This method generates a new instance of a SparseMatrix that is transposed version of the current object matrix. For example:

\[
\begin{bmatrix}
1 & 2 \\
3 & 4
\end{bmatrix}.transpose()
\]

should return

\[
\begin{bmatrix}
1 & 3 \\
2 & 4
\end{bmatrix}
\]

---

*Note: this is just one way of designing your data structure. You are free and encouraged to use your own design, the only condition being, you have to use a (modified form of) doubly linked list to store non-zero elements*


5 Questions

Please answer the following questions in not more than 5 lines each and submit it with your implemented code and graphs with the write up in the PDF format. (20 points)

1. What is the nature—1, n, n², n³—of the graph you have plotted for each of the operations (§2.1, §2.2, §2.3, §2.4 and §2.5) you ran the experiments on? Can you explain the reason for the operation being constant, linear, quadratic, or cubic in nature by identifying the dominating steps?

2. Are there any anomalies that you noticed in the graphs?

3. Which asymptotic complexities have we calculated in the experiments (§2.1, §2.2, §2.3, §2.4 and §2.5)?—O, Ω, Θ, o, ω?

6 Deliverables

1. The tar archive of the correct implementation of:
   - Doubly linked list with all the functions implemented in lab 1, the new function
     DoublyLinkedList DoublyLinkedList::getPositionList(DoublyLinkedList*) (Files: list.hpp and list.cpp),
   - The new data-structure for SparseMatrix and the 2 operations associated with it:
     bool SparseMatrix::equals(SparseMatrix*)
     SparseMatrix* SparseMatrix::transpose()
   as explained in §3 (Files: sparse-matrix.cpp and sparse-matrix.hpp) and the updated Makefile
   with the 5 new targets explained in §2.6

2. Two reports (one for each data structure) containing the following:
   (1) The overall organization of your experiment
   (2) Observation and conclusion
   (3) Answers to questions in §5
   (4) Tabulated timings for operations for all the runs
   (5) The graphs you have generated for all the 5 operations:
   1. Add at tail
      void DoublyLinkedList::add(int, int)
   2. delete elements
      void DoublyLinkedList::deleteAll(int)
   3. generate position list
      DoublyLinkedList DoublyLinkedList::generatePositionList(DoublyLinkedList*);
   4. Matrix equality
      bool SparseMatrix::equals(SparseMatrix*)
   5. Matrix Transpose
      SparseMatrix* SparseMatrix::transpose()

7 Grading Scheme

1. Doubly linked list and related tasks 50 points maximum
2. Sparse matrix and related tasks 50 points maximum.
3. Your submitted code should compile and run on the EECS unix machines.
   (Please use cycle2.eecs.ku.edu/cycle3.eecs.ku.edu/
    EECS lab machines. g++ v6.2.1)
   There should be no memory leaks and compilation warnings. (10% points)
4. 5 Graphs and tabulated generated times for all runs the analysis (60% points).
5. Write up answering questions given in §5 and additional items explained in §6 (30% points)

8 Submission and Miscellaneous Hints

1. Please add the grader’s (Dravid Joseph) email id in the To section of the mail (dravidjoseph@ku.edu) and my (Apoorv Ingle) email id in CC (apoorv.ingle@ku.edu)

2. Your subject line for the submission should be of the form [EECS 560] Lab-<Lab #> <Lab Day> <Your KU username> eg. [EECS 560] Lab-2 M j543h898

3. Your reports should be named as <your KU username>-dll-lab2-report.pdf for doubly linked list and <your KU username>-matrix-lab2-report.pdf

4. Your code tar archive will be automatically named in correct format by running make tar

5. Expand the tar ball: $ tar xvf <filename>.tar.gz

6. Make cheat sheet:
   - compiling a program: make build
   - testing your program: make test
   - clean compile and run your code: make clean build test
   - bundle your code to send to grader/TA: make tar
   Please change XXXXX in first line of the Makefile into your KU username of the format (j052h567) before running make tar

7. Counting starts from 0

9 Further readings and musings for the curious (no credit)

You might want to check an interesting linked list implementation that is used by the Linux operating system:

1. [Linux Kernel Newbies linked list implementation explained](#)
2. [Linux kernel linked list source-code](#)

You can try using the Linux kernel linked list and analyze the time complexities for the 3 operations we performed in the lab. How does the Linux kernel linked list performance times compare with your implementation? What may be the reasons for the differences?