The starting point is problem 10.21 of OSCJ, page 418. Instead of Java, I recommend you use C or C++ because it takes less programming expertise to get best performance from C/C++ than it does from Java.

Instead of generating a random page-reference string, your program should successively process pairs of hexadecimal numbers from the standard input. The first number is 0, 1, or 2 to signify whether the reference is a read, write or instruction fetch respectively. The second number is the virtual address. Hint: To read hex in C++:

```c++
#include <iomanip>
...
cin >> hex >> reftype >> hex >> vaddr;
...
```

You will first test your program on short input files to verify correctness and to possibly debug because you will be able to figure out the correct output by hand. You will then use your program to verify the phenomenon called Belady’s anomaly, and to observe paging behavior on large address trace files that were captured from some old benchmarks which were used for research and advanced teaching.

For simplicity, and in deference to Intel, fix the page size to be 4K. Define the number of page frames parameter with a global `const int` so that you can easily change it and recompile your program to simulate different size memories. An even better way, harder to design though, is to make all the data structures dynamic so that the parameter(s) can be set at run time. The number of page frames should range from 1 or 2 up to thousands.

Instead of merely reporting the number of page faults, your program should accumulate all relevant statistics from the simulation: number of page frames in the simulated memory, number of references processed, number of simulated page faults, number of dirty pages written, number of pages evicted, etc. Each statistic should be printed on in decimal on line with an explanation; eg. “Number of page frames=1000”. It should also calculate and report the “hit ratio” and “miss ratio” for entire simulation.

The statistics should of course be printed at the end of the simulation.

For each memory reference processed, your program should print one line similar to the each column in figures 10.10, 10.12 or 10.13 of OSCJ. However, instead of printing each column vertically, the page number, a colon, a space, and the sequence of which page numbers are in each page frame should be printed on one line, separated by spaces. Unused page frames should not produce output characters. Example from FIFO in figure 10.10:

```
7:  7
0:  7 0
1:  7 0 1
2:  2 0 1
0:  2 0 1
etc.
```
The program should test for command line argument “-q” and if it is given, omit the reference-by-reference listing. That way, your screen will not be swamped by millions of output lines!

The program should (ideally) accommodate a million or so references. Test it first with just a few, and reproduce the examples from AOSC.

The trace files are available on itsuix as .din.Z files under the pathname 
ACSICOM/Traces, and from the web page for the project linked from the class page. Do not keep copies of them in your itsunix account because they will use up your quota. To supply them to your tracer program for testing, use the pipeline: (What does zcat do?)

```
zcat ~acsic00/public/Tracer/cc1.din.Z | tracer -q
```

Your program should handle each full trace in less than 2 minutes (That’s my guess but it might be modified as people begin to try this out on itsunix. Students who did this project in Java had found it took much longer.)

Investigate how each algorithm behaves on some of the supplied traces (or shorter versions of them) plus how it behaves on a random sequence of addresses. (Use a random number generator subroutine.)

For the grading, write a report that explains to the TA and I how we should compile and test your program, and reports what you did and what you found out. The last 25% of the credit for each of the 2 algorithms will be granted if your programs handle the full million addresses.

Turn in your work to the project named “tracing”

**Comments**

A more sophisticated data structure would be needed to obtain reasonable performance on the much larger traces. One idea is to simulate a TLB too. But that is not necessary for the purpose of the current project.

Another is to use a hash table (in addition to the linked list or other data structure for simulating the algorithm) to quickly locate the record for each page or determine it doesn’t exist.