CS 507 (Fall 2022)

Programming Assignment #1 (100 points), Due on 12/15/2022, 23:59

Application of Linked List to Simulate Cache

Introduction:
This programming assignment asks you to design a **cache** implementation using a linked list data structure. Create a file "doublelinkedlist.py", which contains two classes "Node" and "DoublyLinkedList". The DoublyLinkedList class should have at least the following functions as instructed by its interface:

- Constructor.
- addFirst(E element): Inserts the specified element at the beginning of this list.
- addLast(E element): Appends the specified element to the end of this list.
- add(int index, E element): Inserts the specified element at the specified position in this list.
- clear(): Removes all of the elements from this list.
- get(int index): Returns the element at the specified position in this list.
- getFirst(): Returns the first element in this list.
- getLast(): Returns the last element in this list.
- remove(int index): Removes the element at the specified position in this list.
- removeFirst(): Removes and returns the first element from this list.
- removeLast(): Removes and returns the last element from this list.
- set(int index, E element): Replaces the element at the specified position in this list with the specified element.
- size(): Returns the number of elements in this list.
- printlist(): print the entire list.

Next, write a `Cache` class (cache.py) that will make use of the above DoublyLinkedList class. That means, the Cache's underlying container is the DoublyLinkedList and should have major functionalities of the DoublyLinkedList. The Cache should have at least the following functions– constructor, get, add, remove, clear, and some others. Also, write a test program to test your cache implementation.

Description:
A **cache** is a hardware (GPU, CPU) or software (disk cache, web cache) component (managed by the operating system kernel) that stores data so that future requests for that data can be served faster. If a data item has a copy in cache, application can read this data item from cache directly. To be cost-effective and to enable efficient use of data, caches must be relatively **small**, so it may not hold the entire data for the application but has to read the data chunk by chunk.

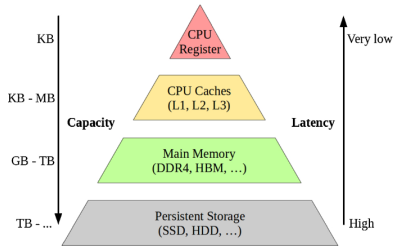

Figure 1: Example: CPU Cache in a memory system. It operates faster than the much larger main memory.

The usage of cache is as follows. Whenever an application requires a data item, It searches the cache first. When a **cache hit** occurs, the cache returns the data item to the application and the data item will be move to the first position in the cache (we call it the Most Recently Used MRU scheme). On the other hand, when a **cache miss** occurs, then the application needs to read the data item from disk and then add the data to the first position of the cache. Note that if the cache is full, the last entry (oldest one) in the cache will be removed before a new entry can be added.

Similarly, whenever an application writes a data item to disk, the system will perform the same write operation to the cache copy of the data item (if any) and then move it to the first position in cache. Note that the `write` operation is equivalent to a `remove` operation followed by an `add` operation.

It is possible to have a cache in cascaded levels (e.g., CPU cache). Below we are going to simulate such multiple level cache using a small sized linked list.

One-level Cache:
A single-level cache and it works as described above.

Two-level Cache:
A 2nd-level cache sits behind the 1st-level cache. Usually, the 2nd-level cache is much bigger than the 1st-level cache. Assume the 2nd-level cache contains all data in the 1st level cache, which is called (`multilevel inclusion property`). Two-level cache works as follows:

```
Search cache1 (Increment the # of cache1 references);
if cache1 hits
   then Increment the # of cache1 hits;
        Move the hit data item to the top of both cache.
   else Search cache2 (Increment the # of cache2 references);
        if cache2 hits
           then Increment the # of cache2 hits;
                Move the hit data item to the top of cache2;
                Add the data item to the top of cache1;
           else Add the data item to the top of both cache;
```

Hit Ratio:
Some terms used to define hit ratio are:
HR_1: 1st-level cache hit ratio
HR_2: 2nd-level cache hit ratio
HR: (global) cache hit ratio

NH_1: number of 1st-level cache hits
NH_2: number of 2nd-level cache hits
NH: total number of cache hits (= $NH_1 + NH_2$, in case of two-level cache simulation)

NR_1: number of references to 1st-level cache
NR_2: number of references to 2nd-level cache (= number of 1st-level cache misses)
NR: total references to cache (= NR_1, in case of two-level cache simulation)

- One-level cache: $HR = \frac{NH}{NR}$
- Two-level cache: $HR1 = \frac{NH_1}{NR_1}$ $\quad HR_2 = \frac{NH_2}{NR_2}$ $\quad HR = \frac{NH}{NR} = \frac{NH_1+NH_2}{NR_1}$

What you need to do:

1. Create a directory ~/cs507/lab1 for this assignment.
2. Write the aforementioned classes.
3. Execute the following command (assuming you are in the directory ~/cs507/lab1) to make a link to the text file. Don't copy this file as it is quite large!

   ln -s /home/MINLONG/repos/cs507/lab1files/Encyclopedia.txt

4. Write a test program `cacheclient.py`. It's usage should be

   python cacheclient 1 <cache size> <input textfile name> or
   python cacheclient 2 <1st-level cache size> <2nd-level cache size> <input textfile name>

   The cache size(s) and the text file should be input as command line arguments. Your program should create a cache (option 1) or two cache (option 2) with the specified size(s) and read in the input text file word by word. For each word, search the cache(s) to see whether there is a cache hit and update the cache accordingly. We will use the file `Encyclopedia.txt` and a small text file `small.txt` in the same directory to test your program.

5. Your program should output the cache hit ratio(s) after reading all words from the input text file. You can find the sample outputs, `result1k2k`, `result1k5k` and `result.small`, in

   ln -s /home/MINLONG/repos/lab1files/

Submission:

Submit your program(s) from `onyx` by copying all of your files to an empty directory (with no subdirectories) and typing the following FROM WITHIN this directory (note, CS507 are case sensitive):

submit minlong CS507 Lab1

Grading

Points will be awarded according to the functionality, quality of the code and other factors as follows.

- (30 points) Functionality of DoublyLinkedList (compile and run, correct outputs for various tests, handling exceptions and so on).
- (30 points) Functionality of Cache (compile and run, correct outputs for various tests, handling exceptions and so on).
- (10 points) Functionality of the drive code of cacheclient (compile and run, correct outputs for various tests, handling exceptions and so on).
- (20 points) Quality (inconsistent indentation, formatting, naming convention, design, efficiency and so on).
- (10 points) Documentation (comments) and README (Analysis, and rest of README).

Here is an overview of the README file:

https://cs.boisestate.edu/~mlong/teaching/references/README.overview

and an example of README:

https://cs.boisestate.edu/~mlong/teaching/references/README.example