CS251 - Project 3: Hashing and Heaps

Out: February 8, 2016 @ 9:00 am
Due: February 29, 2016 @ 9:00 am

Overview

In this project you will code slightly different but practical implementations of hashing and heaps: Cuckoo Hashing and sorting using Binary Heaps. Both implementations must be coded in the same project according to the specifications below. Read the handout thoroughly since details for each implementation are given. As on previous projects, it must be coded in C++.

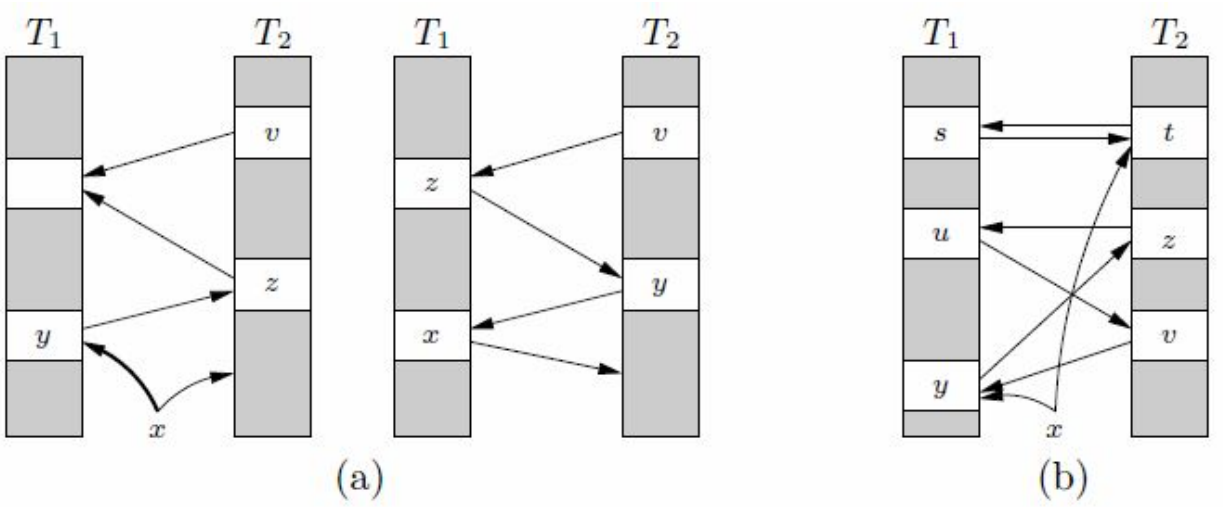
1. Cuckoo Hashing (50 pts)

Cuckoo Hashing is a variation of the regular hashing concept. It was presented by R. Pagh and F. Rodler on the Proceedings of European Symposium on Algorithms in 2001 (original paper here). We are interested in this hashing mechanism since it is straightforward to implement and it has worst case constant lookup time. The name is received from the European cuckoo who throws out the eggs from other bird's nests in order to make room for its own eggs!

This method uses two hash tables T_1 and T_2, each of length r, and two hash functions $h_1(x)$ and $h_2(x)$. Every key x is stored in cell $h_1(x)$ of T_1 or $h_2(x)$ of T_2, but never in both. The lookup function is defined as follows:

```
bool function lookup(x)
  return T1[h1(x)] == x or T2[h2(x)] == x
end
```

For insertion, as described in the original paper, it turns out that the "cuckoo approach" (kicking other keys away until every key has its own nest) works very well. Specifically, if x is to be inserted we first see if cell $h_1(x)$ of T_1 is occupied. If not occupied, we set $T_1[h_1(x)]$ = x and we are done. Otherwise, we set $y = T_1[h_1(x)]$ and then set $T_1[h_1(x)]$ = x anyway, thus making y "nestless". The key y is then inserted in T_2. If its nest in T_2 is occupied, we proceed in a similar way and so forth iteratively. Check the example in Figure 1a. It may happen that this process loops (see Figure 1b). Therefore the number of iterations is bounded by a *MaxLoop* value. If this number of iterations is reached, everything is rehashed (which is described later) and we try to accommodate the nestless key.



[1]**Figure 1.** (a) Key *x* is successfully inserted by moving keys *y* and *z* from one table to the other. (b) Key *x* cannot be accommodated and a rehash is necessary.

The insert procedure is defined as follows (notation $x \leftrightarrow y$ means the values of variables *x* and *y* are swapped):

```
procedure insert(x)
  if lookup(x) then return
  loop MaxLoop times
    if T1[h1(x)] = empty then {T1[h1(x)] = x; return}
    x ↔ T1[h1(x)]
    if T2[h2(x)] = empty then {T2[h2(x)] = x; return}
    x ↔ T2[h2(x)]
  end loop
  rehash(); insert(x)
end
```

Removing a key is of course simple to perform. If lookup(*x*) is true then remove *x* from $T_1[h_1(x)]$ or $T_2[h_2(x)]$. Remember an element in T_1 cannot be in T_2 and vice versa.

For rehashing, we will perform two operations: 1) double the size of both tables, and 2) add the elements of T_1 and T_2 into the doubled-size new tables. Keep in mind that you need to add such values in the same way as inserting any x values (i.e., use the insert operation).

[1] R. Pagh and F. Rodler, Cuckoo Hashing, Proceedings of European Symposium on Algorithms, 2001.

In http://www.lkozma.net/cuckoo_hashing_visualization, you can visualize how Cuckoo Hashing works by inserting the values of your choosing. Here is a suggestion: delete TEST1 and TEST2 and insert numbers in increasing order, one by one. Observe how the keys are swapped between tables. Keep doing this until you fall in a loop.

You will implement the Cuckoo Hashing mechanism as described before. The hashing functions to be implemented are:

$$h_1(x) = x \bmod r \text{ and } h_2(x) = x \bmod (r-1)$$

The input for the program is:

program < inputfile.txt > outputfile.txt

The format of the **inputfile.txt** has the following structure:

1
r m
a b
a b
a b
a b
...

Where **1** indicates that the Cuckoo Hashing part will be executed, ***r*** is a positive integer that indicates the length of the tables and ***m*** is a positive integer that indicates the maximum number of loops. For each tuple (*a b*), ***a*** is the instruction to perform and ***b*** is the number for the required instruction. The possible values for *a* are "*i*" for insert, "*r*" for remove and "*l*" for lookup. The values of *b* are positive integer numbers.

The program has to write the following outputs:

- Every time an element *x* is **inserted** in $T_i[h_i(x)]$ (during insertion, rehashing, and element swapping) then write the element and where it is in which table; i.e., if *x = 2* and it was inserted in table T_1 at position $h_1(2) = 11$ then the output is "2 in T1[11]". If x was inserted into an empty cell then add an exclamation mark "!" at the end of the output line. If *x* is already on either T_1 or T_2 then the output is "2 already in T1[11]".
- When x is **removed** using the remove function then write "2 out T1[11]"; i.e., if *x = 2* and is was removed from table T_1 and position $h_1(2) = 11$. If *x* is not removed because it isn't on neither T1 or T2 then write "no out 2".
- When *x* is **looked up** and it was found then write where; i.e., if *x = 2* and it is in T_1 at position $h_1(2) = 11$ then the output is "2 at T1[11]". If *x* is not found then write "no 2". i.e., if *x = 2* is neither in T_1 nor T_2 then write "no 2".

- If the MaxLoop value is reached then write “maxloop reached”.

The following is an example of the output given a set of tuples with instructions and values:

input file	output
1 5 2 i 21 i 38 i 49 i 81 i 40 i 11 i 55 l 38 l 24 r 66 i 40 i 30 i 73 r 49 r 56 i 21 l 40	21 in T1[1]! 38 in T1[3]! 49 in T1[4]! 81 in T1[1] 21 in T2[1]! 40 in T1[0]! 11 in T1[1] 81 in T2[1] 21 in T1[1] 11 in T2[3]! 55 in T1[0] 40 in T2[0]! 38 at T1[3] no 24 no out 66 40 already in T2[0] 30 in T1[0] 55 in T2[3] 11 in T1[1] 21 in T2[1] maxloop reached 30 in T1[0]! 40 in T1[0] 30 in T2[3]! 11 in T1[1]! 21 in T1[1] 11 in T2[2]! 38 in T1[8]! 55 in T1[5]! 49 in T1[9]! 81 in T1[1] 21 in T2[3] 30 in T1[0] 40 in T2[4]! 73 in T1[3]! 49 out T1[9] no out 56 21 already in T2[3] 40 at T2[4]

Additional test cases are given in the attached files.

2. Binary Heaps and Sorting (50 pts)
In this section you need to implement a Binary Heap. Then, using this heap we are going to implement a sorting algorithm.

First you are going to implement a binary heap. One should be able to use it as either a max-heap or a min-heap --- it should be configurable during heap initialization. Once the heap is initialized, it continues to behave the same way until its deletion. Your implementation should use a binary tree based approach, not an array based approach. Refer to your lecture slides to understand how heaps can be implemented. The elements in the heap should follow this structure:

```
struct element {
  int key;
  int value;
} ;
```

When you execute the insert() or extract() functions on the heap, the comparison should be on 'key', not 'value'. You can also use a class to define the above structure, or use your own names.

Your heap should implement the following functionalities:

1. **Peek:** basically find-max or find-min. If it is configured as a max-heap, find the element with maximum 'key'. [If it is configured as a min-heap, find the element with minimum 'key'.]
2. **Insert:** add a new element to the heap.
3. **Extract/delete:** return the element with minimum 'key' if the heap is a min-heap [or maximum 'key' if the heap is a max-heap] after removing it from the heap.
4. **Create:** create an empty heap.
5. **Heapify:** create the heap out of a given array of elements. Assume here that an empty heap is already initialized (as max-heap or min-heap).
6. **Size:** return the number of items in the heap.

For the sorting implementation, given a list of <key, value> pairs you are going to sort the list based on the keys. This should be straightforward. You just need to call heapify() and then extract one by one the elements on the heap. The sort order will be specified (ascending or descending key values), and depending on that you should use max-heap or min-heap.

The input for the program is:

program < inputfile.txt > outputfile.txt

The first line of the inputfile.txt could be the numbers 2 or 3 declaring which part of the project is being tested (2 for the heap part and 3 for sorting using heaps).

For heap part the following lines of the input file are as follows:

- An integer n that represents the number of lines to follow
- n lines of the following format:
    - The character ‘c’ (for create) followed by an integer. If the integer value is ‘1’, create an empty max-heap. If the integer value is ‘2’ create an empty min-heap.
    - The character ‘i’ (for insert) followed by 2 integers. Insert these two values into the heap as <key, value> pair.
    - The character ‘p’ (for peek). Call peek() and print the key and value of the returned element separated by a space. Print “empty” in case the heap is empty.
    - The character ‘e’ (for extract/delete). Call extract and print the key and value separated by a space. Print “empty” in case the heap is empty.
    - The character ‘h’ (for heapify) followed by many integers separated by a space. Consider the first integer as the size of the array, and the rest of the integers as the keys for the elements. Consider the value as 0 for all the elements of the heap. Here you can safely assume that the heap is empty when heapify is called.
    - The character ‘s’ (for size). Print the current size of the heap.

For the sorting part the following lines of the input file are as follows:

- An integer that represents the order of the sorting. The value 1 represents that you have to sort in increasing order. The value 2 represents decreasing order.
- An integer n representing the number of elements that you have to sort.
- n lines each containing two integers. First integer is the key and the second one is the value.
- The character ‘q’ represents that now you need to sort the elements and print the sorted list. While printing the sorted list you will print each pair at each line. The key and value will be separated by one space character.

Sample test cases:

Input	Expected output
2 6 c 1 i 5 2 i 4 3 e p s	5 2 4 3 1
2 8 c 2 p i 5 2 i 4 3 i 12 13 e p s	empty 4 3 5 2 2
2 4 c 1 h 4 12 15 7 9 e s	15 0 3

3 1 5 56 78 32 45 24 66 49 20 79 1 q	24 66 32 45 49 20 56 78 79 1
3 2 5 56 78 32 45 24 66 49 20 79 1 q	79 1 56 78 49 20 32 45 24 66

Programming Environment and Grading

Assignments will be tested in a Linux environment. You will be able to work on the assignments using the Linux workstations in HAAS and LAWSON (use your username and password). Compilation will be done using g++ and makefiles. You must submit all the source code as well as the Makefile that compiles your provided source code into an executable named “program”.

Your project must compile using the standard g++ compiler (v 4.9.2) on data.cs.purdue.edu. For convenience, you are provided with a template Makefile and C++ source file. You are allowed to modify such file at your convenience as long as it follows the I/O specification. Note some latest features from C++14 are not available in g++ 4.9.

The grading process consists of:

1. Compiling and building your program using your supplied makefile.
2. The name of produced executable program must be “program” (must be lowercase)
3. Running your program automatically with several test input files we have pre-made according to the strict input file format of the project and verifying correct output files thus follow the above instruction for output precisely – do not “embellish” the output with additional characters or formatting – if your program produces different output such as extra prompt and space, points will be deducted.

4. Inspecting your source code.

Input to the programming projects will be via the command line:

program < input-test1.txt > output-test1.txt

The file output-test1.txt will be tested for proper output.

Important:
1. If your program does not compile, your maximum grade will be 50% of the grade (e.g., 5 out of 10) -- no exceptions.
2. Plagiarism and any other form of academic dishonesty will be graded with 0 points as definitive score for the project and will be reported to the corresponding office.

Submit Instructions

The project must be turned in by the due date and time using the turnin command. Follow the next steps:

1. Login to data.cs.purdue.edu (you can use the labs or a ssh remote connection).
2. Create a directory named with your **username** and copy your solution (makefile, all your source code files including headers and any other required additional file) there.
3. Go to the upper level directory and execute the following command:
   turnin -c cs251 -p project3 your_username
   (Important: previous submissions are overwritten with the new ones. Your last submission will be the official and therefore graded).
4. Verify what you have turned in by typing **turnin -v -c cs251 -p project3**
   (Important: Do not forget the -v flag, otherwise your submission would be replaced with an empty one). If you submit the wrong file you will not receive credit.