CMPS101-01 2015F Programming Assignment

Note: Two students may collaborate on this assignment.

BestStuffEver – An Online Store

You are going to launch an online store that is going to sell the coolest stuff with the best customer service ever. That of course means the IT part of your business has to be rock solid, and that's why you are going to do implement it yourself.

You will focus on the following functionality:

User id creation for new customers

Here are the *system requirements* in more detail.

User id creation for new customers

When new potential customers come to the BestStuffEver site, they will need to create a *user id* for themselves. Users can propose any string consisting of up to 16 letters (upper and lower case) or digits as their user id. Your system will check whether the proposed user id is still available; if so, your system will add the chosen user id to the current list of user ids. (If the user id is already taken then the user will be prompted for a new user id suggestion.) Also, whenever a new user id is issued, a unique *customer id* is assigned to the new customer to be used for business accounting purposes.

Examples of valid user ids:

SantaClaus73, gobbleTurkey, 1239087204983321, nobodysbusiness, a123b4567, zzzz0000yyyy11

Examples of invalid user ids:

who_me?, why-ever-not, itsme!!!, JohannSebastianBach

System Design

The number of customers is hard to predict, and in any case, it will vary over time. So it is good idea to design the system so it can grow and shrink with the business.

User id creation for new customers

We need an efficient way of checking whether a suggested user id is available or not. Back-of-the-envelope calculation (cf. your next homework) reveals that there are about 10^{93} 16-character user ids even if we allow just the lower and upper case letters (of the English alphabet) and digits. So simply making a list of all possible user ids and checking the used ones of is out of the question.

A hash table is perfect for the job of determining quickly whether a customer proposed user id has already been taken by another customer or is still available. Note that the number of user ids in use is relatively small compared to the number of possible user ids.

So the user id management system (UIMS) will internally use hash table to record all taken user ids.

CMPS101-01 2015F Programming Assignment

For a hash table of size m we use an array of indexed by $\{0, \dots, m-1\}$. Let $String16$ stand for the set of character strings of length at most 16. Then the hash function $hash$ maps the universe of $String16$, the universe of *keys*, to the set of possible hash values, $\{0, \dots, m-1\}$:

$$hash: String16 \longrightarrow \{0, \dots, m-1\}$$

The hash table userT maps each hash value to a set of user ids:

$$userT: \{0, \dots, m-1\} \longrightarrow 2^{String16}$$ Power set of String16

such that for all $uid \in userT(i)$: $h(uid) = i$.

(Here $2^{String16}$ stands for the power set, i.e. the set of subsets, of $String16$. Why does that make sense? See your next homework.)

When a new customer proposes a user id uid, then

$$uid \text{ is already taken} \Leftrightarrow uid \in userT(hash(uid))$$
$$uid \text{ is available} \Leftrightarrow uid \notin userT(hash(uid))$$

If a user desires the user id uid and uid is available then uid is added to $userT\big(hash(uid)\big)$.

As you know from the lectures, the set $userT(i)$ is the collision set for hash value i. Two possible representations for the collision set discussed in the lectures are linked lists (*chaining*) or storing the set in the hash table itself (*linear probing*). We will use singly-linked lists for representing collision sets.

For the hash function we will use multiplicative hashing, either as described by Mehlhorn/Sanders on pages 87-88 (universal hashing), or the near-universal hash function family described by Erickson in Section 12.5.2. For both we will need a conversion module that can translate between character strings (user ids), bit sequences, and numbers.

[A t truly efficient implementation would use the bit-level operations of Java or C++. To avoid some of the technical noise we will roll our own conversion operations.]

Implementation

The implementation will consist of the following classes:

Class UIMS – the user id management system

Attributes and Methods:

$isAvailable(uid: string): Boolean$
uid: proposed user id
$returns\ true\ if\ the\ uid\ is\ available, false\ otherwise$

$add(uid: string)$
precondition: $uid\ is\ available$ and $customerId\ =\ cid$
postcondition: $uid\ is\ in\ hash\ table\ userT\ associated\ with\ cid$
$and\ customerId = cid + 1$

$lookupCustomerId\ (uid : string) : integer$
returns the customer id associated with user id uid;
returns 0 if uid is not an assigned user id

$customerId$: integer
initially: 0
increased by 1 each time a user id is added
// since the value of $customerId$ is always equal to the number user ids issued
// we will use it to the compute the *load factor* of the hash table (below)

$userT$: $array[0..m-1]$of (pointer to) element list

m: $integer$
size of currently allocated hash table;
a prime number or power of 2, depending on hash family used
initially: 2 (if hash table is dynamically growing)

$hash\ (uid: string): integer$
returns the hash value for the user id uid, a value between 0 and m-1
define using helper functions

$salt$: $digitseq[0..k-1]$ if using Mehlhorn method: $k = 1 + n/w$
$Integer$ if using Erickson 12.5.2
// for the term $salt$ see Erickson, Section 12.5

$generateSalt( )$
Generates random $salt$
// If you want a random number in range [0..k-1], use a random number generator

// to generate an integer n and then take n mod k

$load()$: $number$

$load() = customerId/m$

invariant $load() \leq 0.75$

// when adding a user id would put the load factor over 0.75 then reallocate the hash
// table, doubling its size. You may want to experiment with other load factors.

$reallocate()$

allocates a new array of double the current size;

transfers all existing user ids (with their associated customer ids) to the new array **by rehashing all current user ids**

// it is not necessarily the case that two user ids that collided under the hash function
// for table $size\ m$ will also collide under the function for $size\ 2 * m$

Class SLItemList -- Singly Linked Item List

Attributes and methods – only those needed to support UIMS

$header$: $Item$

initially its $next$ attribute points to itself

// can be used to store sentinel when searching list

$pushFront(uid: string, cid: number)$

Inserts new item it at the beginning of list;

$it.userId \coloneqq uid$

$it.customerId \coloneqq cid$

$find(uid: string)$

returns the item with $item.userId = uid$

// if using sentinel in header, returning header means uid was not found

Class SItem

Attributes and methods

userId: string
customerId: number
next: Item

Class Conversion – helper class for hash function

$stringToBitseq(s: string): array\ of\ \{0.1\}$

returns an array of 0s and 1s constructed by representing each character in s by its assigned bit pattern (position in the list below represented as a numeral to base 2)

CMPS101-01 2015F Programming Assignment

Use the following ordering of digits and letters:

```
0000000000111111111122222222223333333333444444444455555555566
0123456789012345678901234567890123456789012345678901234567890
0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Sample translations:

$0 \longmapsto 000000$ (position 00)
$1 \longmapsto 000001$ (position 01)
$9 \longmapsto 001001$ (position 09)
$a \longmapsto 001010$ (position 10)
$k \longmapsto 010100$ (position 20)
$u \longmapsto 011110$ (position 30)
$E \longmapsto 101000$ (position 40)
$O \longmapsto 011110$ (position 50)
$Y \longmapsto 111100$ (position 60)

The string $molly1$ translates to the bit sequence

```
22     24     47     47     60     01          // position as base 10 numeral
010110 011000 101111 101111 111100 000001      <- least significant digit on right
```

$BitseqToDigitseq(bs\colon array\ of\ \{0,1\}, k\colon number)\colon array\ of\ digits\ to\ base\ 2^k$

Returns an array of numbers in the range $[0..2^k - 1]$

Example: let k=4. Divide the bit sequence for $molly1$ into groups of 4 bits (starting from right; the last group on the left may not always be full: complete with 0s):

```
0101 10 0110 00 1011 11 1011 11 1111 00 0000 01
   5    9    8   11   14   15   15    0    1
```

$BitseqToBigNum(bs\colon array\ of\ \{0,1\}, k\colon number)\colon BigNum$

Returns a BigNum bn whose representation to the base 2 is bs

For the bitseq for $molly1$ we get: $bn = \sum_{i=0}^{35} bs[i] \cdot 2^i$ (least significant bit (0) on right)

$NumToBitseq(n\colon integer)\colon array\ of\ \{0,1\}$

Turns the integer n into a sequence of bits

$Bs\colon array\ of\ \{0,1\}$
$i := 0$
$while\ n > 1\ do$
    $n2 := n/2\ //\ integer\ division\colon e.g.\ 5/2 = 2$
    $bs[i] := n - 2 * n2$
    n:= n2

Class TestDrivers

This class will contain helper methods for generating test data sets and running UIMS on the test data sets. You want to be able to (randomly) generate lists of user ids, for instance. More on this soon.

The specification above contains only the core methods and attributes. Introduce additional setter and getter methods as needed.

CMPS101-01 2015F Programming Assignment

Project Plan

First Phase (60%)

Build the top level of the system first: Focus on the UIMS class; use a simple hashing function for a fixed size table; e.g. for a given user id, convert the characters to numbers, and take the sum mod m (m the table size), or use a hash function provided by Java or C++. You may use a language library module for the singly linked list. Ignore customer ids for now.

Build the TestDrivers class. Make sure your system works end to end.

Second Phase (20%)

Implement your own singly linked list class (if you didn't in phase 1).

Third Phase (10%)

For a fixed table size, implement a multiplicative hashing function (using the Conversion class).

(I will provide more detail on this during the weekend.)

Fourth Phase (10%)

Implement a dynamically growing table $userT$ and associated hash function. Include customer ids.

Submission

Submission details will be provided soon.