Map með tætitöflum (hash tables)

Gefinn er abstract klasinn Map ásamt main forriti sem smíðar HashMap og kallar á föllin á Map. Verkefnið er að búa til klasann *HashMap* sem útfærir *Map* með tætitöflum (*hash tables*). *Einnig þá hjálparklasa sem þarf.* Gögn eru sett inn í töflurnar og sótt úr þeim með *lykli* sem er af taginu ***K*** en hver færsla geymir einnig *gögn* af taginu ***T***.

Í *Map.h* er einnig lýst yfir frávikskl̈sunum *NotFoundException* og *ItemExistsException*.

HashMap þarf að yfirskrifa öll pure virtual föllin á Map. Þau eru eftirfarandi:

- **void insert(K key, T data)**
  Bætir gögnunum *data* inn með lyklinum *key*.
  Ef þegar er stak með lykilinn *key* í gagnagrindinni er kastað ***ItemExistsException***.
- **void update(K key, T data)**
  Uppfærir gögnin tengd lyklinum *key* þannig að þau séu núna *data*.
  Ef ekkert stak með lykilinn *key* er í gagnagrindinni er kastað ***NotFoundException***.
- **T get(K key)**
  Skilar gögnunum sem tengd eru lyklinum *key*.
  Ef ekkert stak með lykilinn *key* er í gagnagrindinni er kastað ***NotFoundException***.
- **void remove(K key)**
  Fjarlægir lykilinn *key* og gögnin honum tengd úr gagnagrindinni.
  Ef ekkert stak með lykilinn *key* er í gagnagrindinni er kastað ***NotFoundException***.
- **bool contains(K key)**
  Skilar *true* ef gagnagrindin inniheldur lykilinn *key* og tengd gögn, annars *false*.
- **int size() const**
  Skilar fjölda staka sem gagnagrindin inniheldur.
- **bool empty() const**
  Skilar *true* ef gagnagrindin inniheldur engin stök, annars *false*.
- **void clear()**
  Tæmir gagnagrindina.
- **void print(ostream& out) const**
  Prentar út innihald gagnagrindarinnar.
  Þetta fall er bara fyrir ykkur að gera prófanir. Mooshak kallar ekki á það.

Nokkur atriði sem þið verðið einnig að útfæra til að allt virki:

HashMap þarf að hafa smið sem tekur inn færibreytu sem er fallabendir (function pointer) til þess að hægt sé að senda inn í klasann bendi á það tætifall (hash function) sem tekur inn rétt tag, tætir gildið og skilar integer. Þessi fallabendir getur verið skilgreindur svona:

```
int (*hash_func)(K);
```

Þetta er bendir á fall sem er þá héðan í frá kallað *hash_func*, tekur inn færibreytu af taginu ***K*** og skilar gildi af taginu ***int***.

Þessu falli er síðan einfaldlega hægt að gefa gildi:

```
HashMap(int (*hash_func)(K)){
      this->hash_func = hash_func;
};
```

Þaðan í frá er hægt að nota nafnið hash_func til að kalla í þetta fall:

```
… hash_func(key) …
```

Skoðið main() til að sjá hvernig fall er sent inn í þessa breytu. Þá er nafn falls notað (t.d. string_hash) án þess að hafa sviga fyrir aftan. Með svigum er verið að kalla í fallið. Án sviga er verið að senda bendi á fallið.

Þið þurfið síðan að útfæra fallið string_hash*, sem finna má í hashfunctions.cpp, á mun betri hátt en nú er gert. Skoðið útfærslur á string hash functions online og athugið hvað gæti verið gott að nota.*

Að lokum verðið þið að útfæra einhvers konar rebuild virkni fyrir tætitöfluna. Mooshak getur samþykkt verkefnið án þess, þar sem það er hægt að setja óendanlega mörg stök í listana sem tætitaflan á fylki af. Hins vegar viljum við bara hafa fá stök í hverjum lista. Virknin verður því eftirfarandi:
Þegar fjöldi staka er orðinn 120% eða meira af fjölda lista í tætitöflunni (lengd fylkisins) þarf að tvöfalda fjölda lista (tvöfalda lengd fylkisins) og re-inserta öllum stökunum í töfluna til að hún sé með rétt tætigildi.

Tillaga að högun verkefnisins:
Tætitafla er fylki af listum þar sem hver listi inniheldur öll stök sem fá sama tætigildi (hash value). Þess vegna þarf fyrst að útfæra slíkan lista í sérklasa. Best er að útfæra eintengdan lista, og þar með þarf líka að útfæra klasa fyrir hnútinn.

Í minni lausn útfæri ég klasana KeyDataList og KeyDataListNode þannig að KeyDataList útfæri eftirfarandi föll:

- **bool add(K key, T data)**
  Skilar true ef eitthvað bættist við, annars false (ef stakið var þegar í listanum).
- **bool update(K key, T data)**
  Skilar true ef eitthvað var uppfært, annars false (ef stakið var ekki í listanum).
- **T find(K key)**
  Skilar stakinu ef það finnst í listanum (hér er bara gengið á listann), kastar annars fráviki.

- **bool remove(K key)**
  Skilar true ef eitthvað var fjarlægt, annars false (ef stakið var ekki í listanum).
- **bool contains(K key)**
  Skilar true ef stakið finnst í listanum, annars false.
- **void clear()**
  Tæmir og hreinsar listann.
- **bool pop(K &key, T &data)**
  Setur key og data af fremsta staki í þessar reference breytur, hendir síðan stakinu og skilar true. Skilar false ef listinn er tómur.
- **void print(ostream& out) const**
  Bara fyrir ykkar eigin test.

Þessi föll notar HashMap síðan sem hjálparföll í sínum föllum.
Ykkar útfærsla þarf ekkert að vera nákvæmlega eins, en þið þurfið að útfæra þessa einföldu gagnagrind. Tætitaflan á síðan fylki af þeirri gagnagrind, eina fyrir hvert mögulegt tætigildi, sem hún setur stök með því tætigildi inn í. Í góðu jafnvægi ætti samt aldrei að vera meira en eitt eða tvö stök í hverjum svona lista, þar sem listarnir eru dreifðir um tætitöfluna.

HashMap á þá klasabreytu sem er kviklegt fylki af KeyDataList.

Verkefninu er skilað á Mooshak.

- Þið fáið ***þrjár tilraunir*** til að senda verkefnið inn í Mooshak. Þriðja tilraunin er lokaskil.
- Sendið ZIP skrá inn í Mooshak sem inniheldur allan forritstextann ykkar, .cpp og .h skrár.

Þið verðið að gera virkilega góðar prófanir í main fallinu.

- Núverandi main() forrit inniheldur hjálparföll og dæmi um notkun þeirra.
- Þið verðið að sjá til þess að jaðartilvik séu prófuð.
  - Bætið í gagnagrindina og takið úr henni í mismunandi röð.
  - Bætið í og takið úr tómri gagnagrind.
  - Athugið hvort stök eru í gagnagrindinni fyrir og eftir eyðingu.
- Verið alveg viss um að ykkar forritstexti virki og að þið skiljið hvernig og hvers vegna hann virkar áður en þið byrjið að reyna að skilja villuskilaboðin frá Mooshak.
- Verið viss um að forritið leki ekki minni.
  - Fyrir hvert new ætti að vera delete.
  - Eyðirinn ætti að eyða öllu minni sem hefur við úthlutað.

English:
You are given the abstract class Map as well as a main program that constructs a HasjMap and calls the functions defined in Map.
The assignment is to make the class *HashMap* that implements *Map* with Hash Tables. *Also any helper classes as needed.* Data is put into the tables and fetched from them using a *key* of the type ***K*** but each node also contains *data* of the type ***T***.

In *Map.h* you will also find definitions of the exception classes *NotFoundException* and *ItemExistsException*.

BSTMap must override all the pure virtual functions on Map. They are the following:

- **void insert(K key, T data)**
  Add the data *data* with the key *key*.
  If there is already an item with the key *key* in the map, throw ***ItemExistsException***.
- **void update(K key, T data)**
  Updates the data connected to the key *key* so they are now *data*.
  If no item with the key *key* is found in the map, throw ***NotFoundException***.
- **T get(K key)**
  Returns the data connected to the key *key*.
  If no item with the key *key* is found in the map, throw ***NotFoundException***.
- **void remove(K key)**
  Removes the key *key* and its connected data from the data structure.
  If no item with the key *key* is found in the map, throw ***NotFoundException***.
- **bool contains(K key)**
  Returns *true* the map contains the key *key* and connected data, otherwise *false*.
- **int size() const**
  Returns the number of items currently in the map.
- **bool empty() const**
  Returns *true* if the map contains no items, otherwise *false*.
- **void clear()**
  Empties the data structure.
- **void print(ostream& out) const**
  Prints the contents of the data structure.
  This function is only for your own testing. Mooshak does not call it.

Things that you must implement in order for everything to work:

HashMap must have a constructor that takes a parameter that is a ***function pointer*** so that a pointer to a ***hash function*** can be sent into the class. This *hash function* must take in the type of the key, hash the value, and return an integer. This *function pointer* can be defined like this:

```
int (*hash_func)(K);
```

This is a pointer to a function that from now on is called *hash_func*, takes in a value of the type ***K*** and returns a value of the type ***int***.

This function can then simply be given a value:

```
HashMap(int (*hash_func)(K)){
      this->hash_func = hash_func;
};
```

From then on the name *hash_func* can be used to call this function:

```
… hash_func(key) …
```

Look at main() to see how a function is sent into this variable. Then the name of a function is used (e.g. string_hash) without the parantheses after it. With parantheses the function is being called. Without parantheses a pointer to the function is being sent.

You must then implement the function string_hash*, that can be found in hashfunctions.cpp, in a superior way to what is already there. Look at implementations of string hash functions online and see what could be good or interesting to use.*

Finally you must implement some sort of rebuild functionality for the hash table. Mooshak can accept your solution without it, since endless items can be put into the lists that the hash table stores an array of. On the other hand we wish only to have few items in each list. The functionality must therefore be as follows:
When the number of items has become 120% or more of the number of lists (buckets) in the hash table (length of the array) the number of lists (buckets) must be doubled (double the length of the array) and re-insert all the items back into the hash table so that all items have the correct hash value indices.

Suggestion for program design:
A hash table is an array of lists where each list contains all the items that get the same hash value (index). Therefore we must first implement such a list in a separate class. It is best to implement a singly linked list and thus also implement a class for the node.

In my solution I implement the classes KeyDataList and KeyDataListNode such that KeyDataList implements the following functions:

- **bool add(K key, T data)**
  Returns true if something was added to the list, otherwise false (item already there).
- **bool update(K key, T data)**
  Returns true if something was updated, otherwise false (item not in list).
- **T find(K key)**
  Returns the item if it is found in the list (just check all items), otherwise throws exception.

- **bool remove(K key)**
  Returns true if something was removed, otherwise false (item not in list).
- **bool contains(K key)**
  Returns true if item is in list, otherwise false.
- **void clear()**
  Empties and cleans up the list.
- **bool pop(K &key, T &data)**
  Puts key and data into these reference variables, then removes the item and returns true. Returns false if list is empty..
- **void print(ostream& out) const**
  Just for your own testing.

HashMap then uses these functions as helper functions in its own operations.
Your implementation does not need to be exactly the same, but you must implement this type of simple data structure. The hash table then has an array of this data structure, one such structure for each possible hash value, that it puts the items for each hash value into. With good balance there should never really be more than one or two items in each such list, where the lists are distributed throughout the hash table.

HashMap then has a class variable that is a dynamic arrau of *KeyDataList*.

The assignment will be returned through Mooshak.

- You get ***three attempts*** to enter it into Mooshak. The third one is final.
- Send a ZIP folder into Mooshak containing all your code, .cpp and .h files.

You must make really good tests in your own main function.

- The current main function contains helper functions and examples of their use.
- You must make sure that you test all edge cases.
  - Add to the data structure and remove from it in different orders
  - Add to and remove from an empty data structure
- Be absolutely sure that your code works and that you understand why it works before trying to understand Mooshak's error messages.
- Make sure your program doesn't leak memory.
  - For every new there should be a delete.
  - Deconstructor should delete everything that was allocated.