Randomized Algorithms

Software Engineering - Algorithms CBC
Department of Computing
Imperial College London

Note: This exercise is designed to be completed under Linux

Exercise Objectives:

The aim of this exercise is to implement (in C++) and compare three data containers:

- Bloom Filters
- Skip Lists
- Randomised Binary Search Trees (RBSTs)

You are required to complete the provided skeletons of the three data structure classes, and then execute a given performance analysis program that plots the running time and complexity for add, delete and find/exists operations. These plots must then be used as the basis for a short report discussing your observed results.

Your submission will therefore be comprised of two parts:

- You must submit the following files electronically via CATE:

  `BloomFilter.hpp`, `SkipList.hpp`, `RBST.hpp`, `BloomFilter.cpp`, `SkipList.cpp`, `RBST.cpp`

  The header files are included so that you can add new private helper methods to them.

- In addition, you must submit a hard-copy report with CATE coversheet, containing the plots generated by `analyse` (see below) and a discussion of the results, to the Student Administration Office.

You must complete the skiplist class first, before attempting to implement the other two containers.

Task 1: Completing the Classes

You are strongly advised to complete one container at a time, making sure that it works and that you can plot the results of the performance analysis for that container before moving on to the next container.

Because of the way in which `analyse` is implemented, you *must* fully implement `SkipList` first before attempting either of the other two containers. Failure to do so will cause `analyse` to hang indefinitely.

You must ensure that no duplicates are placed into the containers.

In addition to the skeletons of the classes that you are required to complete, a number of helper classes are also provided:

- Class `Key` should be used as the keys. It inherits the `String` class and can be treated as a string.
- Classes `SkipListNode` and `RBSTNode` inherit from the `Key` class and should be used as helper classes in the relevant containers. They enable direct comparison between a node and a key.
- Class `DataStructure` is the base class of all the container classes. It stores the counts of calling the `add()`, `del()` and `find()`/`exist()` methods so that they can be plotted against the size of the set. You do not need to do anything to this class as the counting mechanism is already implemented in the derived container classes.

Operations for the `RBST` and `SkipList` classes are recursive and require outer, or portal, methods to interact with the user. These methods are provided in the skeleton classes. Apart from the key, a portal method also requires an argument (`bool v`) that specifies whether there should be some text output indicating the success or failure of the operation. This tag is false by default, which means no text is output. Also, a portal method for membership testing is provided for the `BloomFilter` class `testExist()` so that you do not have to worry about the text output. It is important, however, that you tell the portal methods whether or not the operation is successful by returning the correct value, as will be explained below.

A `dump()` method is provided for each container for debugging purposes. It expects one argument, which tells it which character to use to separate the output of each item stored in the container. This character can be a tab '`\t`', a space '' or a new line '`\n`'. Default values are provided for different classes, but you are free to try other values.

Completing `SkipList`

`SkipList` is defined in `SkipList.hpp` along with an affiliated class, `SkipListNode` which has a vector `m_links` that stores the pointers at each level. The member variable `unsigned int m_max Height` tracks the maximum height of a node in the skip list.

You are required to complete the following four functions:

- **unsigned int** `randHeight();`

  Returns a random height for a node with corresponding probability.

- **int** `add(SkipListNode* target, SkipListNode* newNode,` **unsigned int** `level);`

  Returns 1 if successful, 0 otherwise.

- `SkipListNode* find(SkipListNode* target,` **const** `Key& key,` **unsigned int** `level);`

  Returns pointer to found node, null otherwise.

- `SkipListNode* del(SkipListNode* target,` **const** `Key& key,` **unsigned int** `level);`

  Returns pointer to item, null otherwise; portal will reclaim memory.

Completing `BloomFilter`

Class `BloomFilter` is defined in `BloomFilter.hpp`. It contains a vector of unsigned longs, `m_tickBook`, in which you store the signatures of each key. The `init(unsigned long n)` method specifies the bit-wise length of `m_tickBook`.

It is important to note that you have to use `m_tickBook` in a bit-wise fashion. This can be done by first calculating the required index in the vector and then calculating to which bits in the indexed unsigned long number this corresponds. In C++, you can create a binary signature using decimal numbers in the following ways:

- `Binary number 0000000010000000 is (1 << 7)`

- `Binary number 1111111011111111 is ∼(1 << 8)`

Two hash functions are provided as

- **unsigned long** `hash1(`**const** `string&)` and

- **unsigned long** `hash2(`**const** `string&)`

each returning an unsigned long. You must use them as is.

The portal function `testExist()` calls `exist(...)` to find out whether the Bloom filter contains the specified key.

You are required to complete the following three functions:

- **void** `add(`**const** `Key& key);`

- **void** `del(`**const** `Key& key);`
- **int** `exist(`**const** `Key& key);`

  Returns 0 if it does not exist, 1 if it does.

Please note that you are not required to implement a counting Bloom filter in this exercise. Also note that the delete function in this exercise will violate the bloomfilter rules given in the lectures and thus tolerate false negatives.

Completing `RBST`

Class `RBST` is defined in `RBST.hpp`, along with class `RBSTNode` that has pointers which point to the left and right child nodes. The member variable `unsigned int m_size` records the number of items stored in the tree.

You are required to complete the following six functions:

- `RBSTNode* randomAdd(RBSTNode* target,` **const** `Key& key);`
- `RBSTNode* addRoot(RBSTNode* target,` **const** `Key& key);`
- `RBSTNode* rightRotate(RBSTNode* target);`
- `RBSTNode* leftRotate(RBSTNode* target);`
- `RBSTNode* del(RBSTNode* target,` **const** `Key& key);`
- `RBSTNode* find(RBSTNode* target,` **const** `Key& key);`

All of the above functions must return a pointer to the current subtree, and must modify `m_size` accordingly (e.g. decrementing it in the case of `del()`).

Task 2: Performance Analysis

After you have completed a container, you can compile and link it with the provided `analyse` program to evaluate the performance of the algorithms.

Using `analyse`

Calling `analyse` requires two arguments. The first is the name of a file containing a list of keys (for which the file `dictionary` has been provided to you) and the second is a flag specifying which container(s) to evaluate (`b` for `BloomFilter`, `s` for `SkipList` and `r` for `RBST`). Here are some usage examples:

- To analyse all three data structures: `analyse dictionary bsr`
- To analyse `SkipList` and `RBST`: `analyse dictionary sr`

Since the run time of each method is only a few milliseconds, `analyse` runs the specified tasks a number of times and calculates their mean run time after excluding outliers. The number of repetitions must be specified when the program starts, and it is suggested that you use a number larger than 100.

After specifying the number of repeats, the program asks which functions you want to evaluate. You can test `add()`, `del()` and `find()` by typing `add` or `a`, `del` or `d`, and `find` or `f` respectively. The program will then ask you for a number, `N`, up to which the tests will be run.

To test `add()`, a set of empty sets is created. New keys are added to these sets and the running time is recorded against the size of the sets. Up to `N` keys randomly selected from the dictionary will be added.

For `del()`, a set of containers with `N` random keys chosen from the dictionary will be created. The `del()` method is called on the stored keys randomly, and running times are recorded against the number of entries left in the sets.

For `find()`, a set of empty sets is created. New keys are then added to the sets one-by-one, and `find()` methods are called with the running times recorded against the size of the sets. Up to `N` keys randomly selected from the dictionary will be added.

Once the tests are complete, `analyse` plots results calculated from the recorded running times using `gnuplot`. It also plots the number of counts stored in the `DataStructure` class, representing the number of recursive calls observed in `BloomFilter` and `RBST` and the number of skips observed in `SkipList`. The graphs are displayed on-screen and saved as the following files:

`addcounter.ps`, `deletecounter.ps`, `findcounter.ps`,
`addtimer.ps`, `deletetimer.ps`, `findtimer.ps`

Performing Your Own Analysis For Final Submission

To complete this exercise you must conduct a performance analysis of your implemented container classes. To do this, perform the following steps:

1. Compile and link your data container class files with the provided `analyse.cpp`.

2. Run the resulting program, setting the number of repeats to 500.

3. Perform the following commands:

   ```
   add
   5000
   del
   5000
   find
   5000
   ```

4. Explain the observed trend in your report. Note that when `N` is large the data plot is rather unreadable, and consequently `analyse` will smooth the data points with bins so that only 100 points are plotted.

5. Try different numbers and discuss the results in your short report. In particular, you should compare your observed results with the predictions given by the theoretical complexity analysis discussed in the lectures. Note that each call recreates the sets.

Submission

The submission deadline is **16:00 on Monday 18 February 2013** for **both** the hard-copy **and** electronic submissions. You should submit your hard-copy report to the SAO with a CATE coversheet. You should also submit the following files electronically via CATE:
`BloomFilter.hpp`, `SkipList.hpp`, `RBST.hpp`, `BloomFilter.cpp`, `SkipList.cpp`, `RBST.cpp`

Assessment

Your submission will be marked according to the correctness and style of the program submitted, as well as the results and interpretation presented in your report. The marks for this exercise are allocated as follows:

Bloom Filter	15%
Skip List	35%
RBST	30%
Report	20%

Code that does not compile using the current lab-installed Linux C++ compiler will be limited to a maximum of 60% of the exercise marks.

Feedback for this exercise will be returned by Monday 4 March 2013.