# Project 4 - Heaps and C++

Handed out: **November 2, 2012**
Due: **November 16, 2012, 11:59pm**

## Description

This project focuses on the heap data structure and its application to a computational linear algebra problem. This is also the only time this semester that you are asked to implement your solution in C++ instead of Java. Doing so will help you familiarize yourself with important features of C++ programming such as standard library and template programming, proper use of pointers and debugging memory leaks.

## Problem

Finding the sum of N **sparse vectors** (an array of numbers with only a few non-zero elements) is a basic operation in linear algebra and must be done efficiently. In this problem, we only look at a simple aspect of this problem which is to **find the sparsity structure of the sum vector**. Knowning this sparsity structure is very useful as it dramatically improves the performance of the sum, especially if we need to compute the sum of vectors multiple times with fixed sparsity structure.

As an illustration, consider the following sample problem: we need to find the sum of 7 sparse vectors

$$\begin{pmatrix}17\\10\\0\\17\\0\\0\\0\\0\\14\end{pmatrix} = \begin{pmatrix}1\\0\\0\\4\\0\\0\\0\\0\\0\end{pmatrix} + \begin{pmatrix}7\\2\\0\\0\\0\\0\\0\\0\\2\end{pmatrix} + \begin{pmatrix}4\\1\\0\\3\\0\\0\\0\\0\\0\end{pmatrix} + \begin{pmatrix}5\\0\\0\\0\\0\\0\\0\\0\\0\end{pmatrix} + \begin{pmatrix}0\\0\\0\\2\\0\\0\\0\\0\\9\end{pmatrix} + \begin{pmatrix}0\\1\\0\\8\\0\\0\\0\\0\\0\end{pmatrix} + \begin{pmatrix}0\\6\\0\\0\\0\\0\\0\\0\\3\end{pmatrix}$$

The sparsity of this sum is as follows:

$$\begin{pmatrix}X\\X\\0\\X\\0\\0\\0\\0\\X\end{pmatrix} = \begin{pmatrix}X\\0\\0\\X\\0\\0\\0\\0\\0\end{pmatrix} + \begin{pmatrix}X\\X\\0\\0\\0\\0\\0\\0\\X\end{pmatrix} + \begin{pmatrix}X\\X\\0\\X\\0\\0\\0\\0\\0\end{pmatrix} + \begin{pmatrix}X\\0\\0\\0\\0\\0\\0\\0\\0\end{pmatrix} + \begin{pmatrix}0\\0\\0\\X\\0\\0\\0\\0\\X\end{pmatrix} + \begin{pmatrix}0\\X\\0\\X\\0\\0\\0\\0\\0\end{pmatrix} + \begin{pmatrix}0\\X\\0\\0\\0\\0\\0\\0\\X\end{pmatrix}$$

where 'X' simply designates the non-zero entries that must be processed to compute the sum. The sparsity representation that we will use in this assignment is the following.

$$\begin{pmatrix}0\\1\\3\\8\end{pmatrix} = \begin{pmatrix}0\\3\end{pmatrix} + \begin{pmatrix}0\\1\\3\end{pmatrix} + \begin{pmatrix}0\\1\end{pmatrix} + \begin{pmatrix}0\end{pmatrix} + \begin{pmatrix}3\\8\end{pmatrix} + \begin{pmatrix}1\\3\end{pmatrix} + \begin{pmatrix}1\\8\end{pmatrix}$$

Here the sparsity structure of the sum vector is [0; 1; 3; 8] because the vector has non-zero elements in the positions with index 0, 1, 3, and 8. The goal of this assignment is to implement an algorithm in C++ that can efficiently produce this

sparsity representation for a given set of sparse vectors. Additional formatting details are provided below.

## Task 1 - Heap Implementation

The approach that you will follow in this project consists in using a heap to solve this problem efficiently. The first step of this assignment is therefore to implement a heap. To familiarize yourself with the concept of template in C++ (which can be seen as the C++ counterpart of Java's generics), you will implement a maxPQ heap template with following API and code skeleton.

```
template<typename Key, typename Value, typename Comp=std::less<Key> > // (1)
class Heap {
public:
    typedef Key                                 key_type;
    typedef Value                               value_type;
    typedef std::pair<key_type, value_type>     node_type;

    Heap();  // constructor
    ~Heap(); // destructor (2)
    void push(const key_type& k, const value_type& v);
    node_type pop(); // (3)
    const node_type* find(const key_type& k) const; // (4)
    size_t getNumberOfOperations() const; // (5)
    size_t getHeight(); // (6)
    size_t getSize(); // (7)

private:
    node_type* pq; // (8)
};
```

**Comment:** As we saw in class, your heap should implement a priority queue with O(log N) push and pop.

**Notes:**

(1) The code above corresponds to a heap of keys of type **Key**, whereby each key is associated with a corresponding value of type **Value**. The priority among keys is determined by the comparator **Comp**, which by default resorts to the less_than operator defined for the type **Key**. Note that this default choice amounts to creating a **Max Priority Queue**. This comparator can be changed however to yield a **Min Priority Queue**. Observe also that this comparator must implement the operator shown below.

```
template<typename Key>
struct MyComparator {
    bool operator()(const Key& k1, const Key& k2) const {
       // ... return a boolean indicating whether k2 has higher priority than k1.
       // for instance, for Key=int returning (k1 < k2) yields a MaxPQ
       // while returning (k1 > k2) yields a MinPQ.
    }
}
```

(2) the destructor must free all the memory that has been allocated by the heap.

(3) **pop()** should **throw an exception** if the heap is empty.

(4) **find()** must return a pointer to a **pair** (key=k, value) stored in the heap, if such a pair exists, a NULL pointer otherwise.

(5) **getNumberOfOperations**() must return the number of comparisons that have been performed by the heap since its creation.

(6) **getHeight**() must return the height of the heap, whereby a heap with a single element (root) should have height 0.

(7) **getSize**() must indicate how many <key, value> pairs are stored in the heap.

(8) the internal representation of the heap of pairs <key, value> is an array that must be dynamically resized when necessary. In particular, the constructor does not require an initial size for the array.

Practically, your implementation should be contained in the header file **heap.hpp**, for which we are

providing a skeleton. You are free to add functions and variables to this code but you are not allowed to remove anything. The correctness of your heap implementation will be checked automatically by running the program contained in test_heap.cpp.

## Task 2 - Efficient K-Way Merge

Using the heap that you implemented in Part 1, you must now implement a function **merge** that takes as input an arbitrary number of sparse vector descriptions and computes efficiently the sparse representation of the resulting sum vector. Note that this function is called '*merge*' and not '*sum*' because the sparsity structure it identifies can be applied to compute other quantities, e.g., the difference of these vectors. Practically, your function should have following API.

```
typedef std::vector<int>                    SparseVector;
typedef std::list<int>                      IList;
typedef std::vector<std::pair<int, IList> > SparseMerge;

void merge(const std::vector<SparseVector>& input, SparseMerge& output);
```

Here, **input** contains the sparsity description of each input vector, encoded as a **SparseVector**. Specifically, you can access the contents of a SparseVector as shown below.

```
SparseVector v;
// ...
size_t n = v.size(); // number of non-zero entries in v
int coef = v[i];     // index of i-th non-zero coefficient in v (with 0 <= i < N)
```

Note that you can assume that the indices stored in each SparseVector are always sorted in increasing order and that no sparse vector will be empty. This will be enforced by the I/O routine in the main function, see below.

The **output** type is a bit more complicated. It consists of a vector that stores, for each non-zero coefficient of the merge vector, the index of that coefficient as well as the indices of all the input vectors that contributed non-zero coefficients to this index. Practically, each entry of the output vector is a pair, whereby the first part of the the **pair** is the index of the non-zero coefficient and the second part is a **list** of contributing vectors (named by their index in the input). No particular order is assumed for the contents of each list. You are simply expected to return for each non-zero index a list that contains **all** the sparse vectors involved. A skeleton for this function is provided in merge.hpp. As stated previously, you are expected to add your implementation to this file (and this file only) and you should not remove anything from it.

The expected performance of the solution described above is $N_{total} \log M$, where $M$ is the number of input sparse vectors and $N_{total}$ is the total number of non-zero coefficients in all input vectors. To test the correctness of your solution and verify that you are achieving the expected performance, you should use test_merge.cpp that we will be using during the grading phase. Note that you need to make sure that your merge() operation is reasonably efficient. For an input of $M$ queues of size $N$ ($Ntotal$ < 1,000,000), your merge() will be considered as failed if it takes more than 1 minute on the standard desktop PC used for grading.

## Resources

For those of you who are new to C++, following tutorials may prove useful: **tutorial1**, **tutorial2**, and **tutorial3**. A complete overview of the standard library (to which std::vector, std::list, std::pair, and std::set, and std::less belong, among many other) is available on the **SGI web page**. To debug this code, you will find gdb to be a valuable ally. An **introductory tutorial** is available for this debugger. Note that you will need to compile your code in debug mode to use gdb (see compiling instructions below). To find memory leak, you can use **Valgrind** to test your code. To use Valgrind, you just need to compile your code with -g (debug flag), and then run valgrind

```
valgrind --leak-check=full  <your executable>
```

Your code is considered to be free of memory leak if this line can be found at the end of valgrind output

```
"All heap blocks were freed -- no leaks are possible"
```

Note: if valgrind failed to run on your machine, you can use one of the mc cluster machines (mc01.cs.purdue.edu through mc18.cs.purdue.edu)

## Compiler aspect

The use of header files for your code makes the compilation of your code straightforward. Specifically, you will need to compile test_heap.cpp and test_heap.cpp, which will include your own header files heap.hpp and merge.hpp. In addition, test_heap.cpp will include the header file stl_heap.hpp, which is used to test correctness of your solution. Practically, you can compile each file (assuming that all the files needed are in the same directory will following instruction:

```
g++ -O2 -o test_heap test_heap.cpp
g++ -O2 -o test_merge test_merge.cpp
```

Note that to run your code through gdb (debugger) you will need to select the debug option when compiling your code. Practically your should replace the optimization "-O2" by "-g".

```
g++ -g -o test_heap test_heap.cpp
g++ -g -o test_merge test_merge.cpp
```

You can then run these files by typing ./test_heap or ./test_merge. The files will indicate the parameters that they accept in input from the command line.

## Deliverables

Use the submission guidelines below to submit the following files for this part of the assignment:

- heap.hpp
- merge.hpp
- readme.txt (if needed)

## GRADING

1. Compiled code: 50%
2. Correct output: 30%
- 15% for correct output from heap's API
- 15% for correct merge() implementation
3. Efficient algorithm (O(NlogM) complexity): 15%
4. No memory leak: 5%

Note: you will only get credit for part 3 and 4 if you get full credit for part 2, i.e. your APIs must work correctly first.

## Submission

Submit your solution on or before **November 16, 2012**. turnin will be used to submit this assignment. The submission procedure is the same as for the first project. Inside your working directory for this project on data (e.g., ~/cs251/project4), create a folder in which you will include all source code used and libraries needed to compile and run your code. DO NOT use absolute paths in your files since they will become invalid once submitted. Optionally, you can include a README file to let us know about any known issues with your code (like errors, special conditions, etc).
After logging into data.cs.purdue.edu, please follow these steps to submit your assignment:

- Enter the working directory for this project

```
% cd ~/cs251/project4
```

- Make a directory named <your_first_name>_<your_last_name> and copy all the files needed to compile and run your code there.
- While still in the working directory of your project (e.g., ~/cs251/project4) execute the following turnin command

```
% turnin -c cs251 -p project4 <your_first_name>_<your_last_name>
```

Keep in mind that old submissions are overwritten with new ones whenever you execute this command. You can verify the contents of your submission by executing the following command:

```
% turnin -v -c cs251 -p project4
```

Do not forget the -v flag here, as otherwise your submission would be replaced with an empty one.