FitBit Challenge Documentation

Release

Jul 06, 2016

CHAPTER

ONE

12 BIT DATA HANDLING LIBRARY

This project produces an executable which has been designed to for the FitBit interview question, as worded below.

To run it, simply run:

```
make
./fitbit test-cases/test1.bin
./fitbit test-cases/test2.bin
./fitbit test-cases/test3.bin
```

To run the unit tests for the heap, run:

```
make test
```

This will require CppUTest to be installed, and discoverable with pkg-config.

To read this documentation in a HTML or PDF, see docs/_build. To re-build the documentation, you will need sphinx, breathe and doxygen. To build them:

```
cd docs
doxygen
make html
make latexpdf
```

This project can also be used as a library that provides functions to handle common tasks associated with reading 12 bit numbers from sensors and doing some pre-filtering. It also includes a lightly tested Min-heap implementation used to solve the challenge

1. Read and emit an 12 bit integers from a stream of 8 bit integers
2. Keep a running list of the N largest numbers encountered so far
3. Keep a list of the last N numbers encountered so far.

1.1 Notes

I've written this code in an object oriented style that may be excessive for something as simple as this problem. However, it should be more maintainable and understandable than 'simpler' implementations. Real world requirements and design pressure could result in a very different implementation.

Completing this in total took about 12 hours.

Here is the breakdown:

- Initial fully working implementation: 6 hrs

- Re-writing to use Min-heap: 4 hours
- Adding fancy documentation: 2 hours

The initial implementation used a brute-force linear time algorithm to implement the priority queue. I was already ware of priority queues, but wanted to get something working fast.

1.1.1 Performance

There are a few areas that could be optimized for performance, depending on the target architecture and compiler.

- **Object oriented style** There are a few extra pointer de-references that could be minimized by making more state static, and not using function pointers. Gcc does a good job of optimizing these
- **Max Value Algorithm** The current implementation uses a priority queue based on a Min-heap. A Red-black tree, or heap variant is potentially faster.

  A brute-force linear search algorithm may actually be faster for keeping track of 32 max values. I haven't benchmarked it.
- **size of number variable** Depending on the native word size, choosing a specific size could improve performance. Usually int has the best performance on a particular platform

1.2 FitBit Problem Statement

Write an ANSI C program which takes as arguments the name of a binary input file and a text output file. The binary input file will contain 12 bit unsigned values. The output file should contain the 32 largest values from the input file. It should also contain the last 32 values read from the input file. Try to write the portions of the code that are not related to IO if you would port it to a memory and speed limited device. Attached you will find three sample input (.bin) and output (.out) files which you can use for testing.

Notes:

1. If there are an odd number of 12bit readings in the file, then the last nibble in the file will be zero and can be ignored.
2. The file format should be as follows
   - Start with "–Sorted Max 32 Values–"
   - The 32 largest values in the file, one value per line. Duplicates are allowed if a large value appears multiple times. The list should be sorted smallest to largest.
   - Output "–Last 32 Values–" on its own line
   - The last 32 values read, one per line. They should appear in the order they were read. The last value read from the file will appear last.
3. If there are fewer than 32 values in the file then whatever is read should be output
4. Your output files should match the output files provided with the given input.
5. If your program is passed bogus values, it should not core dump.
6. test#.bin is the binary file that corresponds with test#.out Three test inputs and outputs have been provided.
7. Bonus point for providing a make file or some other build script.
8. Please provide comments in the top of your code describing design decisions / tradeoffs and if you did any optimizations for speed, memory, etc. Most of our projects are on MCUs with limited memory powered by a small battery. Optimizations are not required but a good way to show off IF they work :)

CHAPTER

TWO

IMPLEMENTATION

2.1 Parser

The 'Parser' consumes one byte at a time, and emits unsigned ints when a complete 12 bit number is completed. It uses three states to make the logic very clear. See the source code to understand the reasoning.

For numbers other than 12 bits, the algorithm could be generalized. A few observations:

- The number of states would be the least common multiple of 8, and the number of bits, divided by 8.
- You might be able to reduce the number of states. I haven't thought about this much

2.2 Number Statistics

The number statistics module consists of two data structures, the RecentValuesList, and the LargestValuesList.

2.2.1 RecentValuesList

The recent values list is implemented using a simple ring buffer. There is not much to it.

2.2.2 LargestValueList

The largest value list is implemented using a Min-heap. A min heap is a data structure of O(log(N)) insertion and deletion. It is considered a maximally efficient implementation of a priority queue.

You can keep adding items to the Queue until there are LIST_SIZE items. After that, every new item with a value larger than the smallest value in the queue will displace that small value.

Items can be removed from the queue smallest to largest.

CHAPTER

THREE

API DOCUMENTATION

The main library code is grouped into two sections, the Parser and Number Statistics.

The parser is responsible for consuming the input binary stream and breaking apart into numbers.

The Number Statics module has data structures and functions for keeping a list of most recently stored numbers, and largest numbers.

3.1 Parser

See parser.c

bool **parser_init** (Parser **parser*, NewNumberHandler *h*)
Initalize a binary number parser that parses 12 bit numbers from a stream of bytes

Return true on success sets errno on failure and returns false

Parameters

- `parser` - uninitialized parser object
- `h` - function to process each number parsed

bool **parser_parse_byte** (Parser **parser*, uint8_t *byte*)
Parse one byte of the input stream. The parser will call the NewNumberHandler specified during initialization if enough bits to form a number are received in this call.

Return true on succes sets errno on failure and returns false

Parameters

- `parser` - initalized parser instance
- `byte` - byte to parse

3.2 Number Statistics

3.2.1 Keeping Track of Recent Values

void **recent_values_list_init** (RecentValuesList **recent_values*)
Initialize the list of recent items, preparing it for use, and erasing all items

Parameters

- `recent_values` - RecentValuesList to initialize

void **recent_values_list_add** (RecentValuesList **recent_values*, unsigned int *number*)
Add a number to the list.

NOTE: Concurrency issues have not been considered, so adding items from multiple execution contexts is not supported.

Parameters

- `recent_values` - Initalized RecentValuesList to append to

void **recent_values_list_print** (RecentValuesList **recent_values*)
Print the most recent LIST_SIZE values, starting with the text "–Last LIST_SIZE valeus–\r\n"

NOTES

•The printed header intentionally mispells values to conform with the test cases.

•The output will contain windows style line endings i.e. CR LF

•Concurrency issues have not been considered, so printing while adding to the list may result in strange behaviour, and is not supported.

Parameters

- `recent_values` - Initalized RecentValuesList to print

Example Code

```
RecentValuesList recent_values;

recent_values_list_init(&recent_values);

recent_values_list_add(&recent_values, 1);
recent_values_list_add(&recent_values, 5);
recent_values_list_add(&recent_values, 3);

recent_values_list_print(&state.recent_values);
```

3.2.2 Keeping Track of Largest Values

void **largest_values_list_init** (LargestValueList **largest_values*)
Initialize the list of largest numbers, preparing it for use, and erasing all existing items.

Parameters

- `largest_values` - RecentValuesList to initialize

void **largest_values_list_add** (LargestValueList **largest_values*, unsigned int *number*)
Add a value to the list of largest values. If there are fewer than LIST_SIZE items stored, it will be added, regardless of whether it is a duplicate

In the case where LIST_SIZE items are stored, this will replace the smallest item in the list. If there are duplicate smaller items, this will replace on of them arbitrarily.

NOTE

•Concurrency issues have not been considered, so adding items from multiple execution contexts is not supported.

Parameters

- `largest_values` - Initialized LargestValueList to add to
- `number` - number to add

void **largest_values_list_print** (LargestValueList **largest_values*)
Print the LIST_SIZE largest values, starting with the text "–Largest LIST_SIZE values–\r\n"

NOTES

•The output will contain windows style line endings i.e. CR LF

•Concurrency issues have not been considered, so printing while adding to the list may result in strange behaviour, and is not supported.

Parameters

- `largest_values` - Initialized LargestValueList to print

Example Code

```
largestValuesList largest_values;

largest_values_list_init(&largest_values);

largest_values_list_add(&largest_values, 1);
largest_values_list_add(&largest_values, 5);
largest_values_list_add(&largest_values, 3);

largest_values_list_print(&state.recent_values);
```

CHAPTER

FOUR

HEAP API DOCUMENTATION

> In computer science, a heap is a specialized tree-based data structure that satisfies the heap property: If A is a parent node of B then the key (the value) of node A is ordered with respect to the key of node B with the same ordering applying across the heap. A heap can be classified further as either a "max heap" or a "min heap". In a max heap, the keys of parent nodes are always greater than or equal to those of the children and the highest key is in the root node. In a min heap, the keys of parent nodes are less than or equal to those of the children and the lowest key is in the root node. Heaps are crucial in several efficient graph algorithms such as Dijkstra's algorithm, and in the sorting algorithm heapsort. A common implementation of a heap is the binary heap, in which the tree is a complete binary tree (see figure).'
>
> —Wikipedia - Heap

The heap below is designed to be used with integer data types. It can fairly easily be adapted to store aribtrary pointers, by augementing the data structure to use a compare function provided during initialization, and also providing the size of each arbitrary data structure to be stored, to allow the library to copy items inside the heap.

Functions

bool **heap_init** (*Heap* **heap*, int **storage*, size_t *size*)

Initialize a heap ojbect

Pass in a pointer to a heap object to be initialized and a pointer to an int array of size + 1 to store the data for the heap.

NOTE

- This function has this awkward to initalization to allow static memory allocation for embedded systems where using malloc is not a good idea. If you want a simpler initalizer, use heap_allocate_and_init

Return true on success

Parameters

- `heap` - pointer to heap structure to initalize
- `storage` - pointer to space to store size + 1 ints
- `size` - number of elements this heap can store

Heap ***heap_allocate_and_init** (size_t *size*)

Allocate and initalize a heap data structure to store at most size ints.

Return pointer to a *Heap* or NULL on failure

Parameters

- `size` - number of elements this heap can store

void **heap_free** (*Heap* *heap)
Free all memory associated with this *Heap*

Parameters

- `heap` - pointer to a *Heap* to free

bool **heap_insert** (*Heap* *heap, int *element*)
Insert an item into the heap. If the heap is full return false, and don't add the item.

Return true on success

Parameters

- `heap` - pointer to allocated *Heap*
- `element` - int to insert

bool **heap_insert_replace** (*Heap* *heap, int *element*)
Insert an item into a heap. If the heap is full insert only if element is larger than the smallest item in the heap.

This is effecively a size constrained priority queue. If

Return true on success

Parameters

- `heap` - pointer to allocated *Heap*
- `element` - int to insert

bool **heap_extract_min** (*Heap* *heap, int **element*)
Remove and return the smallest number from the heap.

Return true on success

Parameters

- `heap` - pointer to allocated *Heap*
- `element` - pointer to store extracted value

size_t **heap_size** (*Heap* *heap)
Retun the size of the heap (current number of stored elements)

Parameters

- `heap` - pointer to allocated *Heap*

void **heap_print** (*Heap* *heap)
Print the heap for debugging purposes, as layed out in memory

Parameters

- `heap` - pointer to allocated *Heap*

struct Heap
#include <heap.h> *Heap* data structure

Please treat as opaque, and don't access any data memberes directly as the implementatio may change

Public Members

int ***storage**

size_t **max_size**

size_t **size**

size_t **elem_size**

H

L

P

R