# Assignment 1

## Efficient Multiset ADT

## Changelog

All important changes to the assignment specification and files will be listed here.

- `[20/06 09:00]` Assignment released
- `[20/06 14:05]` Added the definition of `struct cursor` to `MsetStructs.h` and fixed a typo in `testMset.c`
- `[21/06 23:05]` Added some clarifications about the expected behaviour of `MsetCursorNext` and `MsetCursorPrev`

## Aims

- To implement a multiset ADT using a balanced binary search tree and analyse the time complexity of its operations
- To give you practice with binary search trees and complexity analysis
- To appreciate the importance of using efficient data structures and algorithms

## Admin

| | |
|---|---|
| **Marks** | contributes 15% towards your final mark (see Assessment section for more details) |
| **Submit** | see the Submission section |

the attained mark, submissions later than 5 days not accepted

## Prerequisite Knowledge

- Recursion
- Analysis of Algorithms
- Abstract Data Types
- Binary Search Trees
- Balanced BSTs (including AVL Trees)

## Background

### Multisets

You are probably familiar with the idea of a **set**. In mathematics, a set is a collection of distinct elements.

Meanwhile, a **multiset** is a collection of elements where duplicates are allowed. In this assignment, we are concerned with multisets of integers.

Since duplicates are allowed, each distinct element in a multiset has a **count**, which is the number of times it occurs in the multiset. For example, in the multiset $\{1, 4, 4, 4, 8, 8\}$, the count of $1$ is $1$, the count of $4$ is $3$ and the count of $8$ is $2$. Since the counts of the elements in a multiset can be arbitrarily high, we will write a multiset by listing all the distinct elements with their counts between curly braces. For example, the multiset $\{1, 4, 4, 4, 8, 8\}$ is written as $\{(1, 1), (4, 3), (8, 2)\}$.

### Notation

Let $A$ be a multiset and $x$ be an item.

- $c_A(x)$ is the count of $x$ in $A$. If $x$ does not occur in $A$, then $c_A(x) = 0$.

## Operations

Here are some fundamental operations on multisets:

### Union

Let $A$ and $B$ be two multisets. The **union** of $A$ and $B$, denoted by $A \cup B$, consists of all the elements that are contained in $A$ or $B$, and the count of each element in $A \cup B$ is the maximum of its counts in $A$ and $B$. Formally,

$$c_{A \cup B}(x) = \max(c_A(x), c_B(x))$$

For example:

$$\{(1,4),(2,3),(3,1)\} \cup \{(1,5),(2,1),(4,2)\} = \{(1,5),(2,3),(3,1),(4,2)\}$$

### Intersection

Let $A$ and $B$ be two multisets. The **intersection** of $A$ and $B$, denoted by $A \cap B$, consists of all the elements that are contained in both $A$ and $B$, and the count of each element in $A \cap B$ is the minimum of its counts in $A$ and $B$. Formally,

$$c_{A \cap B}(x) = \min(c_A(x), c_B(x))$$

For example:

$$\{(1,4),(2,3),(3,1)\} \cap \{(1,5),(2,1),(4,2)\} = \{(1,4),(2,1)\}$$

$$\{(1,5),(5,3)\} \cap \{(2,1),(6,4),(8,2)\} = \varnothing$$

### Sum

Let $A$ and $B$ be two multisets. The **sum** of $A$ and $B$, denoted by $A + B$, consists of all the elements that are contained in $A$ or $B$, and the count of each element in $A + B$ is given by the sum of its counts in $A$ and $B$. Formally,

$$c_{A+B}(x) = c_A(x) + c_B(x)$$

For example:

### Difference

Let $A$ and $B$ be two multisets. The **difference** of $A$ and $B$, denoted by $A - B$, consists of all the elements that have a higher count in $A$ than in $B$, and the count of each element in $A - B$ is given by the difference of their counts in $A$ and $B$. Formally,

$$c_{A-B}(x) = \max(c_A(x) - c_B(x), 0)$$

For example:

$$\{(1,4),(2,3),(3,1)\} - \{(1,5),(2,1),(4,2)\} = \{(2,2),(3,1)\}$$

### Inclusion

Let $A$ and $B$ be two multisets. $A$ is said to be **included** in $B$, denoted by $A \subseteq B$, if the count of each element in $A$ is less than or equal to its count in $B$.

For example:

$$\{(1,4),(2,3),(3,1)\} \subseteq \{(1,5),(2,3),(3,4),(4,2)\}$$

$$\{(1,4),(2,3),(3,1)\} \nsubseteq \{(1,5),(2,1),(4,2)\}$$

### Equality

Let $A$ and $B$ be two multisets. $A$ is said to be **equal** to $B$, denoted by $A = B$, if $A$ contains exactly the same elements and counts as $B$.

For example:

$$\{(1,4),(2,3),(3,1)\} = \{(1,4),(2,3),(3,1)\}$$

$$\{(1,4),(2,3),(3,1)\} \neq \{(1,4),(2,1),(3,1)\}$$

## Multiset ADT

unimportant, as long as they produce the desired behaviour from the user's perspective.

It is possible to implement the Multiset ADT in many different ways, such as using arrays or linked lists. Although these are relatively simple to implement, each of them are inefficient in some way:

- An ordered array allows for binary search to be used, but requires elements to be shifted to maintain order when inserting
- A linked list does not require shifting, but requires a linear traversal when inserting and searching for elements

Recently, we introduced binary search trees as a solution to both of these problems (shifting and linear search). A binary search tree (BST) is a linked data structure, and therefore does not require shifting, and the way searching works in a BST is similar to how binary search works. However, a BST is only guaranteed to be efficient if the tree is relatively balanced. Thus, your task in this assignment will be to implement a Multiset ADT using a **balanced binary search tree**.

## Setting Up

Change into the directory you created for the assignment and run the following command:

```
$ unzip /web/cs2521/23T2/ass/ass1/downloads/files.zip
```

If you're working on your own machine, download `files.zip` by clicking on the above link and then unzip the downloaded file.

You should now have the following files:

| | |
|---|---|
| `Makefile` | a set of dependencies used to control compilation |
| `Mset.h` | interface to the Multiset ADT - **cannot be modified** |
| `Mset.c` | implementation of the Multiset ADT (incomplete) |
| `MsetStructs.h` | definition of structs used in the Multiset ADT (incomplete) |

| | |
|---|---|
| **`analysis.txt`** | a template for you to enter your time complexity analysis for selected functions |

Usually, the structs used by an ADT implementation are defined in the `.c` file along with the functions. However, in this assignment, they are defined in a separate file, `MsetStructs.h`, because our tests will need access to these structs in order to make it easier to independently test each function, as well as to check whether your trees are balanced. You may add extra fields to these structs and define additional structs if needed, but you must use the given structs/fields in your implementation as follows:

- You must use `struct node` for your binary search tree nodes
- The elements of a multiset must be stored in the `item` fields of the `struct node`, and their counts must be stored in the `count` fields
- The `left` and `right` pointers should be used to connect a tree node to its left and right subtrees respectively. **These pointers must not be used for any other purpose.**
- The `tree` field must be used to point to the binary search tree that stores all the elements of the multiset

Note that the only files that you are allowed to submit are `Mset.c`, `MsetStructs.h` and `analysis.txt`. This means all your code for implementing the Multiset ADT must be placed in `Mset.c`, *except* the struct definitions which should be in `MsetStructs.h`.

## Task 1: Implementation

Your task is to use a **balanced binary search tree** to implement a Multiset ADT with the aforementioned operations. We have divided the operations into groups - you must fully implement all the operations in each group before moving on to the next group.

When we say you must use a **balanced** binary search tree, we mean a **height-balanced** binary search tree. This means for every node in the tree, the absolute difference in height between its left and right subtrees must be no greater than 1.

## Group 1: Basic Operations

**Note**

Most functions in this section have a time complexity requirement. Solutions that are slower than required will be penalised. Note that $n$ refers to the number of distinct elements in the given multiset.

| Operation/Function | Description | Required worst-case time complexity |
|---|---|---|
| `MsetNew` | Creates a new empty multiset | N/A |
| `MsetFree` | Frees all memory allocated to the given multiset | N/A |
| `MsetInsert` | Inserts one of an item into the given multiset. Does nothing if the item is equal to `UNDEFINED`. | $O(\log n)$ |
| `MsetInsertMany` | Inserts the given amount of an item into the given multiset. Does nothing if the item is equal to `UNDEFINED` or if the given amount is 0 or less. | $O(\log n)$ |
| `MsetSize` | Returns the number of distinct elements in the given multiset. | $O(1)$ |
| `MsetTotalCount` | Returns the sum of counts of all elements in the given multiset | $O(1)$ |

| | | |
|---|---|---|
| | multiset. | |
| MsetShow | Prints the elements in the given multiset in ascending order between curly braces, with items separated by a comma and space. Each element should be printed inside a pair of parentheses with its count. **Do not print a newline.** Here are some examples:<br>`{}`<br>`{(1, 1)}`<br>`{(1, 1), (4, 3)}`<br>`{(1, 1), (4, 3), (8, 2)}` | N/A |

## Group 2: Advanced Operations

**Important Constraint**

For all functions in this group **except** `MsetMostCommon`, the method of converting the given tree(s) into an array or linked list, solving the main problem using the array/linked list and then returning the result or converting the result back into a tree is strictly forbidden.

| **Operation/Function** | **Description** |
|---|---|
| MsetUnion | Given two multisets $A$ and $B$, returns a new multiset $A \cup B$ (defined above) |
| MsetIntersection | Given two multisets $A$ and $B$, returns a new multiset $A \cap B$ (defined above) |
| MsetSum | Given two multisets $A$ and $B$, returns a new multiset $A + B$ (defined above) |
| MsetDifference | Given two multisets $A$ and $B$, returns a new multiset $A - B$ (defined above) |

| | |
|---|---|
| `MsetEquals` | Given two multisets $A$ and $B$, returns true if $A = B$ (defined above), and false otherwise |
| `MsetMostCommon` | Given a multiset $A$, an integer $k$ and an array `items`, stores the $k$ most common elements in $A$ into the `items` array in decreasing order of count and returns the number of elements stored. Elements with the same count should be stored in increasing order. Returns 0 if $k$ is 0 or less. Assumes that the `items` array has size $k$. |

## Group 3: Cursor Operations

This is a challenge! You should not expect to receive hints for this part.
**Note:** You may need to modify some of your existing code to get this working.

As you have learned, an ADT does not provide users access to the internal representation of the data type. But what if a user wanted to know what elements are contained in a multiset? With only the basic operations listed above, the only way the user could achieve this is to check the count of every possible item, like so:

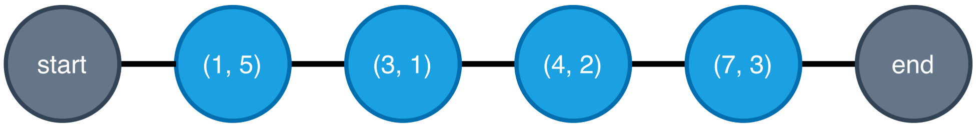
```
i = smallest possible integer
while i <= largest possible integer:
    count = get the count of i in the multiset
    i = i + 1
```

However, this is not practical given how many possible integers there are. To solve this problem, many collection data types provide a simple way for users to iterate over their elements, called iterators. Here is how an iterator would be used:

```
let s be a multiset
it = create an iterator for s
while thereIsStillANextItem(it):
    i = getNextItem(it)
```

Your task is to implement a more flexible iterator that allows users to move forwards and backwards through the elements of a multiset, which we will call a cursor.

elements - one at the start, and one at the end:



A cursor is like a pointer that moves between these elements. When a cursor is created, it will be positioned at the *start* element, and the user can use operations called `next` and `prev` to move the cursor right or left.

There are five cursor operations:

| **Operation/Function** | **Description** |
|---|---|
| `MsetCursorNew` | Creates a new cursor for the given multiset, initially positioned at the start of the multiset |
| `MsetCursorFree` | Frees all memory allocated to the given cursor |
| `MsetCursorGet` | Returns the element at the cursor's position and its count, or `{UNDEFINED, 0}` if the cursor is positioned at the start or end of the multiset. |
| `MsetCursorNext` | Moves the cursor to the next greatest element, or to the end of the multiset if there is no next greatest element. Does not move the cursor if it is already at the *end*. Returns false if the cursor is now at the *end*, and true otherwise. |
| `MsetCursorPrev` | Moves the cursor to the next smallest element, or to the start of the multiset if there is no next smallest element. Does not move the cursor if it is already at the *start*. Returns false if the cursor is now at the *start*, and true otherwise. |

Here is an example of how a cursor could be used:

```c
int main(void) {
    Mset s = MsetNew();
    MsetInsertMany(s, 4, 2);
    MsetInsertMany(s, 7, 3);
    MsetInsertMany(s, 1, 5);
    MsetInsertMany(s, 3, 1);
    struct item item;
    while (MsetCursorNext(cur)) {
        item = MsetCursorGet(cur);
        printf("%d occurs %d time(s)\n", item.item, item.count);
    }
    MsetCursorFree(cur);

    MsetFree(s);
}
```

This would produce the output:

```
1 occurs 5 time(s)
3 occurs 1 time(s)
4 occurs 2 time(s)
7 occurs 3 time(s)
```

The following are some clarifications about the expected behaviour of the cursor:

- If the multiset associated with a cursor is empty, `MsetCursorNext` should move the cursor to the *end*, and `MsetCursorPrev` should move the cursor to the *start*. In both cases, the functions should return false as specified above.

- A user should be able to create multiple cursors for a given multiset and iterate over the multiset independently with each cursor.
- `MsetCursorNext` and `MsetCursorPrev` should continue to work as described above if elements are inserted after the cursor has been created.
    - For example, suppose a multiset contains the elements 2, 5 and 8, and a cursor is currently positioned at 5. If 7 is now inserted into the multiset, then calling `MsetCursorNext` should move the cursor to 7.

In order to obtain full marks for this part, all cursor operations should have a worst-case time complexity of $O(1)$. A solution where one or more of the cursor operations have a worst-case time complexity of $O(\log n)$ is worth half the marks. Solutions where one or more operations are slower than $O(\log n)$ will not receive marks for this part.

## Task 2: Complexity Analysis

In this task, you need to determine the worst-case time complexity of **your implementation** of the following operations, and write your answers in `analysis.txt` along with an explanation of each answer.

- MsetUnion
- MsetIntersection
- MsetIncluded
- MsetEquals
- MsetMostCommon

Your time complexities should be in big-O notation. For `MsetUnion`, `MsetIntersection`, `MsetIncluded` and `MsetEquals`, your time complexities should be in terms of $n$ and $m$, where $n$ and $m$ are the number of distinct elements in each of the multisets respectively. For `MsetMostCommon`, your time complexity should be in terms of $n$ and $k$, where $n$ is the number of distinct elements in the given multiset and $k$ is the given value of $k$.

In your explanations, you may use time complexities established in lectures without proof, as long as you indicate that it was established in lectures. For example, you may use the fact that the worst-case time complexity of searching in an AVL tree is $O(\log n)$, as this was established in lectures.

## Testing

We have provided a main program `testMset.c` which contains basic test cases. You should examine `testMset.c` to see what the tests do and how the tests call your functions.

```
$ make
...
$ ./testMset
...
```

All of the given tests (except for `MsetShow`) are assertion-based, which means that the program will exit as soon as a test fails. If you want to ignore a test for the time being, then you can comment out the corresponding function which runs that test.

We strongly recommend you to add your own tests, as the given tests are very simple. You can easily add your own tests by creating a new function in `testMset.c` and then calling it from the `main` function. You are also free to completely restructure the testing program if you want.

## Debugging

Debugging is an important and inevitable aspect of programming. We expect everyone to have an understanding of basic debugging methods, including using print statements, knowing basic GDB commands and running Valgrind. In the forum and in help sessions, tutors will expect you to have made a reasonable attempt to debug your program yourself using these methods before asking for help.

You can learn about GDB and Valgrind in the Debugging with GDB and Valgrind lab exercise.

## Frequently Asked Questions

- **Are we allowed to create our own functions?** You are always allowed to create your own functions. All additional functions you create should be made `static`.
- **Are we allowed to create our own `#define`s and structs?** Yes.

for this assignment are provided already.

- **Are we allowed to change the signatures of the given functions?** No. If you change these, your code won't compile and we won't be able to test it.
- **What errors do we need to handle?** You should handle common errors such as `NULL` returned from `malloc` by printing an error message to `stderr` and terminating the program. You are not required to handle other errors.
- **What invalid inputs do we need to handle?** You are not required to handle invalid inputs, such as `NULL` being passed in as a multiset. It is the user's responsibility to provide valid inputs.
- **Will we be assessed on our tests?** No. You will not be submitting any test files, and therefore you will not be assessed on your tests.
- **Are we allowed to share tests?** No. Testing is an important part of software development. Students that test their code more will likely have more correct code, so to ensure fairness, each student should independently develop their own tests.

## Submission

You must submit the files `Mset.c`, `MsetStructs.h` and `analysis.txt`. You can submit via the command line using the `give` command:

```
$ give cs2521 ass1 Mset.c MsetStructs.h analysis.txt
```

You can also submit via give's web interface. You can submit multiple times. Only your last submission will be marked. You can check the files you have submitted here.

**WARNING:**

After you submit, you **must** check that your submission was successful by going to your submissions page. Check that the timestamp is correct. If your

## Compilation

You must ensure that your final submission compiles on CSE machines. Submitting non-compiling code leads to extra administrative overhead and will result in a 10% penalty.

Every time you make a submission, a dryrun test will be run on your code to check that it compiles. Please ensure that your final submission successfully compiles, even for parts that you have not completed.

## Assessment Criteria

This assignment will contribute 15% to your final mark.

### Correctness

**75%** of the marks for this assignment will be based on the correctness of your code, and will be mostly based on autotesting. Marks for correctness will be distributed as follows:

| | | |
|---|---|---|
| Group 1 | MsetFree | See **memory errors/leaks** section below |
| | MsetInsert and MsetInsertMany | 16% (4% for correct insertion, 12% for balancing) |
| | MsetSize | 2% |
| | MsetTotalCount | 2% |
| | MsetGetCount | 2% |
| | MsetShow | 3% |
| Group 2 | MsetUnion | 6% |
| | MsetIntersection | 6% |
| | MsetSum | 6% |

| | | |
|---|---|---|
| | MsetEquals | 6% |
| | MsetMostCommon | 6% |
| Group 3 | All cursor operations together | 8% |

Please note that we expect you to prioritise completing functions correctly over implementing as many functions as possible (potentially incorrectly), so we will not manually award partial marks. You are expected to exercise your debugging skills to fix problems with the function you are currently working on, before moving on to the next function.

### Memory errors/leaks

You must ensure that your code does not contain memory errors or leaks, as code that contains memory errors is unsafe and it is bad practice to leave memory leaks in your program.

Our tests will include a memory error/leak test for each operation. If a memory error/leak arises from your code, you will receive a penalty which is 10% of the marks for that operation. For example, the penalty for causing a memory error/leak in the `MsetEquals` operation will be 0.6%. Note that our tests will always call `MsetFree` at the end of the test (and `MsetCursorFree` if appropriate) to free all memory associated with the multiset.

## Complexity analysis

**15%** of the marks for this assignment will be based on the correctness of your complexity analysis in `analysis.txt` and the quality of your explanations.

## Style

**10%** of the marks for this assignment will come from hand marking of the readability of the code you have written. These marks will be awarded on the basis of clarity, commenting, elegance and style. The following is an indicative list of characteristics that will be assessed, though your program will be assessed wholistically so other characteristics may be assessed too (see the style guide for more details):

- Using functions to avoid repeating code
- Decomposing code into functions and not having overly long functions
- Using comments effectively and not leaving planning or debugging comments

The course staff may vary the assessment scheme after inspecting the assignment submissions but it will remain broadly similar to the description above.

## Originality of Work

This is an individual assignment. The work you submit must be your own work and only your work apart from the exceptions below. Joint work is not permitted. At no point should a student read or have a copy of another student's assignment code.

You may use any amount of code from the lectures and labs of the **current iteration** of this course. You must clearly attribute the source of this code in a comment.

You may use small amounts (< 10 lines) of general purpose code (not specific to the assignment) obtained from sites such as Stack Overflow or other publicly available resources. You must clearly attribute the source of this code in a comment.

You are not permitted to request help with the assignment apart from in the course forum, help sessions or from course lecturers or tutors.

Do not provide or show your assignment work to any other person (including by posting it publicly on the forum) apart from the teaching staff of COMP2521. When posting on the course forum, teaching staff will be able to view the assignment code you have recently submitted with give.

The work you submit must otherwise be entirely your own work. Submission of work partially or completely derived from any other person or jointly written with any other person is not permitted. The penalties for such an offence may include negative marks, automatic failure of the course and possibly other academic discipline. Assignment submissions will be examined both automatically and manually for such issues.

provide or show your assignment work to another person for any reason, and work derived from it is submitted, then you may be penalised, even if the work was submitted without your knowledge or consent. This may apply even if your work is submitted by a third party unknown to you.

---