Assignment 1

Efficient Set ADT

Changelog

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

- `[21/06 13:00]` Assignment released
- `[25/06 17:20]` Clarified the array constraint and added more clarifications from the forum thread

Aims

- To implement a set 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
Deadline	8pm on Monday of Week 7
Late penalty	0.2% per hour or part thereof deducted from 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

Sets

A **set** is a collection, or ensemble, of distinct items. We call these items **elements**, or **members** of the set. We can also say that they *belong to* that set. In this assignment, we are concerned with sets of integers.

Usually, we write a set by displaying its elements between two *curly braces*. For example, $\{1, 4, 6, 9\}$ is a set whose elements are 1, 4, 6, 9. Note that the order in which the elements are written does not make the set different, so for example, $\{1, 4, 6, 9\}$ and $\{6, 9, 1, 4\}$ and $\{1, 6, 9, 4\}$ all represent the same set.

Notation

Conventionally, sets are denoted by capital Roman letters and their elements denoted by small Roman letters.

Now let A and B be sets and v be an item.

- v is said to be **in** A, written as $v \in A$, if v is an element of A.
- Set A is said to be **equal** to set B, written as $A = B$, if both sets have the same elements.
- Set A is said to be a **subset** of set B, written as $A \subseteq B$, if all elements of A are also elements of B. For example, we have that $\{1, 2, 3\} \subseteq \{1, 2, 3, 5\}$ and $\{1, 5, 8\} \nsubseteq \{1, 5, 3, 9\}$.

Operations on Sets

In **Lecture 2.2 - Abstract Data Types (ADTs)**, we introduced some fundamental set operations:

- **insert** an item into a set
- check if an item is a **member** of a set
- check the **size**, or **cardinality**, of a set
- **display** a set using standard set notation
- get the **union** of two sets
- get the **intersection** of two sets

For those unfamiliar with set theory, we define union and intersection as follows:

Union

Let A and B be two sets. The **union** of A and B, denoted by $A \cup B$, is the set of all elements that are contained in A, or B, or both A and B. Formally,

$$A \cup B = \{x : x \in A \text{ or } x \in B\}$$

For example:

$$\{1, 2, 3\} \cup \{2, 4\} = \{1, 2, 3, 4\}$$
$$\{1, 5\} \cup \{2, 6, 8\} = \{1, 2, 5, 6, 8\}$$

Intersection

Let A and B be two sets. The **intersection** of A and B, denoted by $A \cap B$, is the set of all elements that are contained in both A and B. Formally,

$$A \cap B = \{x : x \in A \text{ and } x \in B\}$$

For example:

$$\{1, 2, 3\} \cap \{2, 4\} = \{2\}$$
$$\{1, 5\} \cap \{2, 6, 8\} = \emptyset$$

Here, we define a few more set operations: **difference**, **floor** and **ceiling**.

Difference

Let A and B be two sets. The **difference** of A and B, denoted by $A - B$, is the set of elements that are in A, but not in B. Formally,

$$A - B = \{x : x \in A \text{ and } x \notin B\}$$

For example:

$$\{1, 2, 3\} - \{2, 4\} = \{1, 3\}$$
$$\{1, 5\} - \{2, 6, 8\} = \{1, 5\}$$

Floor

Let A be a set and v be an item (integer). The **floor** of v in A is the largest element in A that is less than or equal to v. Formally,

$$\text{floor}(A, v) = \max\{x : x \in A \text{ and } x \leq v\}$$

For example:

$$\text{floor}(\{1, 2, 3\}, 5) = 3$$
$$\text{floor}(\{1, 5, 8\}, 5) = 5$$

Note that the **floor** of an item may not be defined if there are no elements in the set which are less than or equal to it. We discuss how to handle such cases below.

Ceiling

Let A be a set and v be an item (integer). The **ceiling** of v in A is the smallest element in A that is greater than or equal to v. Formally,

$$\text{ceiling}(A, v) = \min\{x : x \in A \text{ and } x \geq v\}$$

For example:

$$\text{ceiling}(\{1, 5, 8\}, 6) = 8$$

$$\text{ceiling}(\{1, 5, 8\}, 5) = 5$$

Note that the **ceiling** of an item may not be defined if there are no elements in the set which are greater than or equal to it. We discuss how to handle such cases below.

Set ADT

A set is an example of an abstract data type: there is a collection of operations that can be performed on them, but the exact details of the implementation are unimportant, as long as they produce the desired behaviour from the user's perspective.

In lectures, we considered several different implementations of the Set ADT: unordered/ordered arrays and unordered/ordered linked lists. Although these were relatively simple to implement, each of them were flawed/inefficient in some way:

	Unordered	**Ordered**
Array	Insertion requires $O(n)$ membership query Search requires $O(n)$ linear scan	Insertion requires $O(n)$ array shift to maintain order
Linked List	Insertion/search requires $O(n)$ traversal of the list	Insertion/search requires $O(n)$ traversal of the list

We then introduced binary search trees as a data structure that could have improved search and insertion costs. But these operations are only guaranteed to be efficient if the tree is relatively balanced. Thus, your task in this assignment will be to implement a Set ADT using a **balanced binary search tree**.

Cursors

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 set? With only the basic operations listed above, the only way the user could do this is to check every possible item for membership in the set, like so:

```
i = smallest possible integer
while i <= largest possible integer:
    check if i is in the set
    i = i + 1
```

However, this is not feasible given how many possible integers there are. Therefore, many collection data types provide access to items via cursors (or iterators). Cursors provide a relatively simple way for users to iterate over the items in a collection:

```
let s be a set
c = create a cursor for s
while thereIsStillANextItem(c):
    i = getNextItem(c)
```

Setting Up

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

```
$ unzip /web/cs2521/22T2/ass/ass1/downloads/files.zip
```

If you're working at home, download `files.zip` by clicking on the above link and then run the `unzip` command on the downloaded file.

You should now have the following files:

`Makefile`	a set of dependencies used to control compilation
`Set.h`	interface to the Set ADT - **cannot be modified**

`Set.c`	implementation of the Set ADT (incomplete)
`SetStructs.h`	definition of structs used in the Set ADT (incomplete)
`testSet.c`	a main program containing some basic tests for the Set ADT
`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, `SetStructs.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 binary search tree nodes
- The elements of the set must be stored in the `item` fields of the `struct node`
- The `left` and `right` pointers should be used to connect a tree node to its left and right subtrees respectively
- The `tree` field must be used to point to the binary search tree that stores all the elements of the set

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

Task 1: Implementation

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

Note

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.

Important Constraint

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, as such solutions go against the spirit of using binary search trees.

Group 1: Basic Operations

Operation/Function	Description
`SetNew`	Creates a new empty set
`SetFree`	Frees all memory associated with the given set
`SetInsert`	Inserts an item into the given set. Any integer may be inserted (including negative integers) except for the value `UNDEFINED`. **Note:** The worst-case time complexity of this function must be $O(\log n)$. Inefficient solutions will be heavily penalised.
`SetSize`	Returns the number of elements in the given set
`SetContains`	Checks if an item is in the given set. Returns true if it is, and false otherwise.
`SetShow`	Prints the elements in the given set in ascending order between curly braces, with items separated by a comma and space. **Do not print a newline.** Examples: `{}` `{2}` `{2, 4}` `{2, 4, 7, 8}`

Group 2: Further Operations

Operation/Function	Description
`SetUnion`	Given two sets A and B, returns a new set $A \cup B$ (defined above)

`SetIntersection`	Given two sets A and B, returns a new set $A \cap B$ (defined above)
`SetDifference`	Given two sets A and B, returns a new set $A - B$ (defined above)
`SetEquals`	Returns true if the two given sets contain the same elements, and false otherwise.
`SetSubset`	Given two sets, returns true if the first set is a subset of the second set, and false otherwise.
`SetFloor`	Given a set A and an item v, returns $\text{floor}(A, v)$ (defined above) or `UNDEFINED` if the result is undefined. `UNDEFINED` is `#define`d in `Set.h`, and you may assume that this value is never inserted into the set via `SetInsert`. You may also assume that this function is never given `UNDEFINED` as the item.
`SetCeiling`	Given a set A and an item v, returns $\text{ceiling}(A, v)$ (defined above) or `UNDEFINED` if the result is undefined. You may assume that this function is never given `UNDEFINED` as the item.

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 described above, cursors provide a way for users to iterate over elements of the set without accessing the internal representation of the set. There are three cursor operations:

Operation/Function	**Description**
`SetCursorNew`	Creates a cursor for the given set, initially positioned at the smallest element of the set
`SetCursorFree`	Frees all memory associated with the given cursor
`SetCursorNext`	Returns the element at the cursor's current position, and then moves the cursor to the next greatest element. If there is no next greatest element, then all subsequent calls to `SetCursorNext` on this cursor should return `UNDEFINED`.

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

```c
int main(void) {
    Set s = SetNew();
    SetInsert(s, 7);
    SetInsert(s, 4);
    SetInsert(s, 8);
    SetInsert(s, 1);

    SetCursor cur = SetCursorNew(s);
    int item;
    while ((item = SetCursorNext(cur)) != UNDEFINED) {
        printf("%d ", item);
    }
    SetCursorFree(cur);

    SetFree(s);
}
```

This would produce the output:

```
1 4 7 8
```

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

- A user should be able to create multiple cursors for a given set and iterate over the set independently with each cursor.
- `SetCursorNext` should continue to work as described above if elements are inserted after the cursor has been created.
  - For example, suppose a set contains the elements 2, 5 and 8, and a cursor is currently positioned at element 5. If 7 is now inserted into the set, then calling `SetCursorNext` should return 5, then 7, then 8, then `UNDEFINED`. But if 3 was inserted instead, then `SetCursorNext` will not return 3, as cursors only move forwards, not backwards.

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 the cursor operations have a worst-case time complexity of $O(\log n)$ is worth half the marks. Solutions that are slower than this will not receive marks for this part.

You must also explain the design and implementation of your solution and how it met the time complexity requirement of $O(1)$ or $O(\log n)$ in `analysis.txt`.

Task 2: Complexity Analysis

You are required 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.

- SetUnion
- SetIntersection
- SetDifference
- SetEquals
- SetSubset
- SetFloor
- SetCeiling

Your answers should be in big-O notation. If an operation involves two sets, then the time complexity should be in terms of n and m, the sizes of the two sets respectively, otherwise it should just be in terms of n, the size of the one set.

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 `testSet.c` containing basic test cases. You should examine `testSet.c` to see what the tests do and how the tests call your functions.

To run these tests, first run `make`, which compiles your code and creates an executable called `testSet`, and then run `./testSet`.

All of the given tests (except for `SetShow`) 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 `testSet.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.
- **Are we allowed to modify `Set.h` in any way?** No.
- **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 set. 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 `Set.c`, `SetStructs.h` and `analysis.txt`. You can submit via the command line using the `give` command:

```
$ give cs2521 ass1 Set.c SetStructs.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 submission does not appear under Last Submission or the timestamp is not correct, then resubmit.

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

76% of the marks for this assignment will be based on the correctness of your code, and will be based on autotesting. Marks for correctness will be distributed as follows:

Group 1	SetFree	See **memory errors/leaks** section below
	SetInsert	18%
	SetSize	2%
	SetContains	2%
	SetShow	4%
Group 2	SetUnion	6%
	SetIntersection	6%
	SetDifference	6%
	SetEquals	6%
	SetSubset	6%
	SetFloor	6%
	SetCeiling	6%
Group 3	SetCursorNew, SetCursorFree, SetCursorNext	8%

Please note that we expect you to prioritise completing functions correctly over implementing as many functions as possible (potentially incorrectly), so partial marks will not be awarded. 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 `SetEquals` operation will be 0.6%. Note that our tests will always call `SetFree` at the end of the test (and `SetCursorFree` if appropriate) to free all memory associated with the set.

memory associated with the set.

Complexity analysis

14% 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):

- Consistent and sensible indentation and spacing
- Using blank lines and whitespace
- 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.

Relevant scholarship authorities will be informed if students holding scholarships are involved in an incident of plagiarism or other misconduct. If you knowingly 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.