Week 04 Lab Exercise

Balanced Binary Search Trees

Objectives

- to implement some core operations of AVL trees
- to get some more practice with binary search tree problems
- to get some more experience with using ADTs
- to develop test data

Admin

Marks	5=outstanding, 4=very good, 3=adequate, 2=sub-standard, 1=hopeless
Demo	in the Week 5 or Week 6 or Week 7 Lab
Submit	`give cs2521 lab04 Tree.c Schedule.c commands.txt` or via WebCMS
Deadline	submit by **11:59pm Sunday 28 June**

Aim

In lectures, we examined AVL trees and the benefits of using AVL trees as opposed to regular binary search trees (with insertion at the leaves). In this week's lab, we will implement some core operations of AVL trees (specifically, rotating and rebalancing), and use the AVL tree to implement a simple scheduler.

Background

The Problem

You work for a small but busy airport, and are tasked with scheduling airplane landings for the airport's one landing strip. There are a couple of rules concerning scheduling landing times:

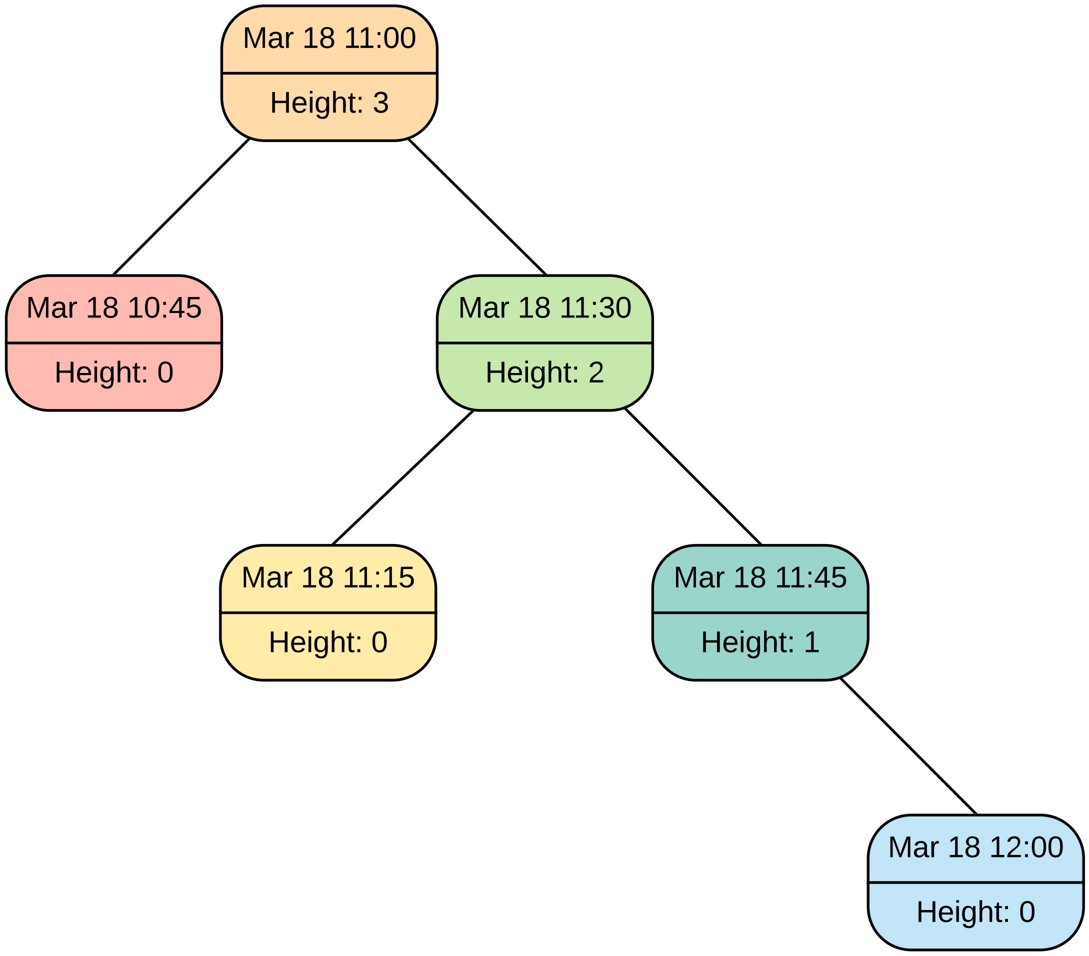
- Bookings are first-come, first-served - If an airline schedules a landing on your runway at a certain time, as long as they are the first to do so, no other airline can schedule a landing at this time.
- For safety reasons, a landing cannot be scheduled within 10 minutes of any other landing. For example, if a landing is scheduled at 11:00 on the 18th of March, no other landings can be scheduled between 10:50 and 11:10 on the 18th of March (inclusive).
- Airlines can schedule landings as far into the future as they want. So they can schedule a landing in 10 years time or even 50 years time if they wanted.
- For simplicity, we'll assume that times in the past can also be scheduled.

How will you manage these landing times? Your colleague suggests using a linked list to store all the landing times, but you argue that this will be too inefficient. Since airlines can schedule landings far into the future, you may need to store hundreds of thousands, if not millions of landing times! Being the expert on data structures and algorithms that you are, you suggest that a balanced binary search tree (such as an AVL tree) would be more suitable.

AVL Trees

AVL trees are binary search trees which rebalance themselves on every insertion and deletion to ensure that they remain height-balanced. (A binary tree is **height-balanced** if, for every node, the absolute difference in height between its left and right subtrees is no greater than one.) A height-balanced BST has the advantage of efficient searches, insertions and deletions (O(log n) complexity), as well as being ordered, which is useful for our problem.

However, there is a *slight* drawback to using an AVL tree as opposed to a regular binary search tree. An AVL tree will frequently need to know the height of its subtrees for balance checking, so we will need a way to compute the height of a subtree that is more efficient than simply traversing the entire subtree. The typical solution is to store an additional field in each tree node that contains the height of the node's subtree. For example:



Note that above tree is *not* an AVL tree, because it is not height-balanced; the height of the right subtree is two more than the height of the left subtree.

Note also that, since each node now has a `height` field, we need to remember to update these whenever we manipulate the tree to ensure they remain correct.

Getting Started

Setting Up

Set up a directory for this lab, change into that directory, and run the following command:

```
$ unzip /web/cs2521/20T2/labs/week04/lab.zip
```

If you're working at home, download `lab.zip`, unzip it, and then work on your local machine.

If you've done the above correctly, you should now find the following files in the directory:

`Makefile`	a set of dependencies used to control compilation
`runway.c`	a program that provides a command-line interface to the Schedule ADT
`runtests.sh`	a script that runs a few tests
`Schedule.c`	an incomplete implementation of the Schedule ADT
`Schedule.h`	the interface for the Schedule ADT
`testTree.c`	a program that tests the Tree ADT implementation
`Time.c`	an implementation of the Time ADT
`Time.h`	the interface for the Time ADT
`Tree.c`	an incomplete implementation of the Tree ADT
`Tree.h`	the interface for the Tree ADT
`tests/`	a sub-directory containing test data, including expected output

Once you've got these files, the first thing to do is to run the command

```
$ make
```

This will compile the initial version of the files, and produce two executables: `./runway` and `./testList`.

File Walkthrough

`runway.c`

`runway.c` provides a command-line interface to the Schedule ADT. It creates a Schedule object, and then accepts commands to interact with it, including adding landing times to the schedule and listing all of the times in the schedule. Here is an example session with the program:

```
$ ./runway
Runway Landing Scheduler v1.0
Enter ? to see the list of commands.
> ?
Commands:
    + <month> <day> <hhmm>     add a new landing time to the schedule
    n                          get the number of times in the schedule
    s <mode (1 or 2)>          show the schedule
    ?                          show this message
    q                          quit

> n
0 landing times have been added to the schedule.
> + 3 18 1100
Successfully added Mar 18 11:00 to the schedule!
> + 3 18 1115
Successfully added Mar 18 11:15 to the schedule!
> s 1
Mar 18 11:00
Mar 18 11:15
> s 2
Mar 18 11:00 (height: 1)
└─R: Mar 18 11:15 (height: 0)
> + 3 18 1110
Successfully added Mar 18 11:10 to the schedule!
> n
3 landing times have been added to the schedule.
> s 1
Mar 18 11:00
Mar 18 11:10
Mar 18 11:15
> s 2
Mar 18 11:00 (height: 2)
└─R: Mar 18 11:15 (height: 1)
      └─L: Mar 18 11:10 (height 0)
> q
$
```

Note that the program doesn't correctly reject landing times. In the above example, we were able to add Mar 18 11:10, even though there were already two landing times within 10 minutes of it (11:00 and 11:15). Also, the tree produced was not balanced. We will add code to fix this later.

`Schedule.c`

`Schedule.c` implements the Schedule ADT, which manages landing times. A client program can attempt to schedule a landing time by calling the `ScheduleAdd()` function, and the Schedule ADT will either accept the landing time and store it, or reject it. The Schedule ADT also provides a few other useful functions, such as getting the number of landing times added to the schedule, and listing the times in the schedule.

At the moment, the Schedule ADT accepts all proposed landing times. Later in the lab, we will fix this so that it only accepts landing times that don't clash with other landing times.

`Tree.c`

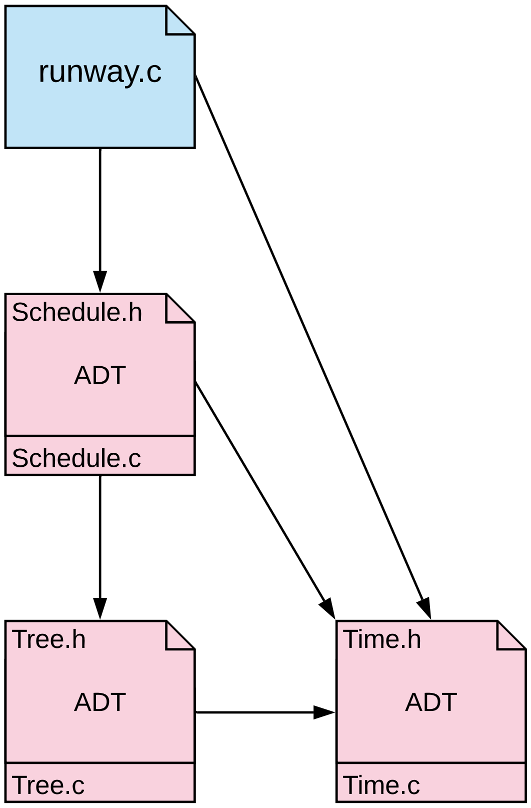
`Tree.c` implements the Tree ADT using an AVL tree, and is used to store the landing times. However, the Tree ADT does not currently behave like an AVL tree, as some of the functions are incomplete. For example, the `doInsert()` function inserts new times, but does not perform any rebalancing. Furthermore, the tree rotation functions `rotateLeft()` and `rotateRight()` don't do anything.

`Time.c`

`Time.c` implements the Time ADT. The landing times are represented by instances of this ADT. You will need to use this ADT throughout the lab, so make sure you read `Time.h` to see what operations are provided. (Hint: A particularly important operation is `TimeCmp()` - it's like `strcmp()`, except it compares Times.) This ADT is fully implemented, so you do not need to modify it.

Dependency Structure

The diagram below shows the dependency structure of the `runway` program. An arrow leading from one file to another indicates that the first file *uses* the second.



For example, the `runway.c` program *uses* the Schedule ADT. But what does it mean for a program to *use* an ADT? If you have a look at the top of the `runway.c` program, you'll see that it #includes `Schedule.h`, which means that the code in `runway.c` has access to all the functions declared in `Schedule.h`. However, note that this does **not** mean that `runway.c` has access to the *implementation* of the Schedule ADT, which is hidden away in `Schedule.c`.

As another example, the Schedule ADT uses the Tree ADT. Interestingly, note that there is no line from `runway.c` to the Tree ADT, which means that `runway.c` doesn't know about (and doesn't need to know about) the existence of the Tree ADT.

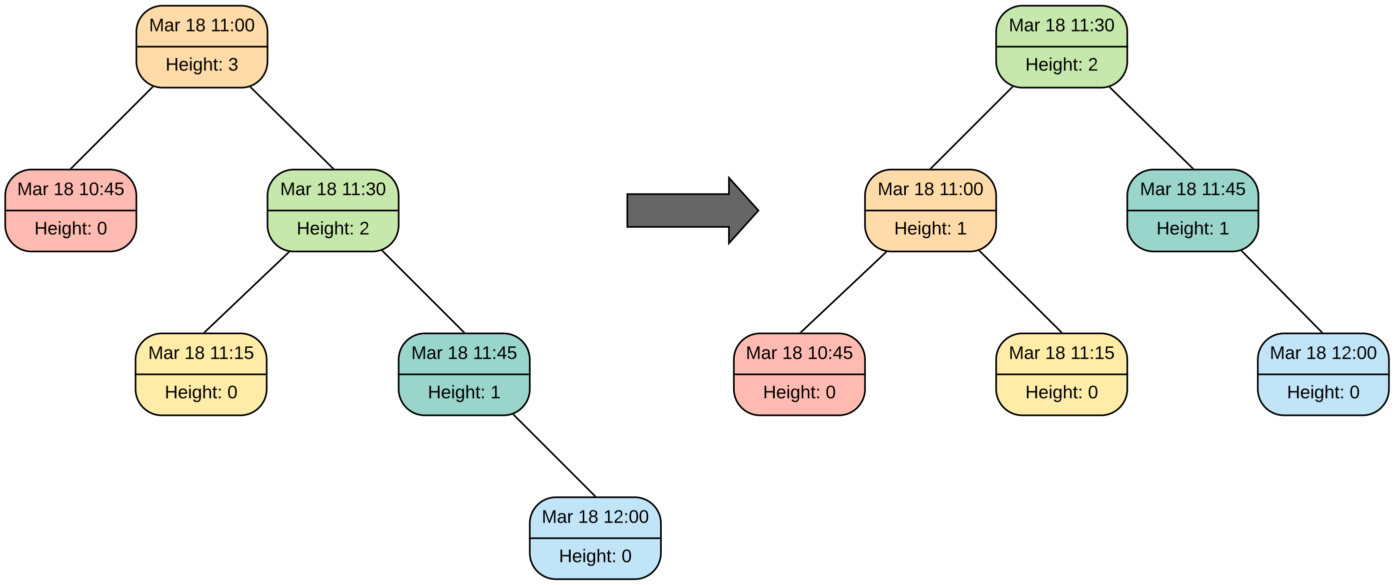
Meanwhile, `runway.c`, the Schedule ADT and the Tree ADT all use the Time ADT. This makes sense since all of the files need to know about the Time ADT:

- `runway.c` needs to be able to create times to add them to the schedule
- The Schedule ADT needs to determine whether a time can be added to the schedule and then pass it to the Tree ADT to be stored
- The Tree ADT needs to store times and arrange them into a binary search tree

Keep this structure in mind as you write your code. Remember that programs that use an ADT don't have access to its implementation, only its interface!

Task 1

Implement the `rotateLeft()` and `rotateRight()` functions in `Tree.c`. `rotateLeft()` performs a left rotation on the given tree and returns the root of the updated tree, while `rotateRight()` performs a right rotation and returns the root of the updated tree. The following diagram shows an example left rotation at the node with `Mar 18 11:00`:



Remember that each node contains a height field which stores the height of the node's subtree. Your functions will need to update this field in all nodes that require it to ensure that the height fields remain correct.

When you think you're done, run the following commands to test your code:

```
$ ./runtests.sh 1
$ ./runtests.sh 2
```

The first command will test `rotateLeft()`, while the second command will test `rotateRight()`. Make sure you pass these tests before moving on to the next task, or your next function won't work properly!

Task 2

Complete the implementation of the `doInsert()` function in `Tree.c`. `doInsert()` is a recursive helper function that inserts a `Time` object into the tree. Currently, the function inserts the `Time` object, but doesn't do any balance checking or rebalancing. Your task is to add this rebalancing.

Hint 1: If you're not entirely sure how an AVL tree rebalances itself, you may find these lecture slides on AVL trees helpful.

Hint 2: You will need to use the Time ADT in this task. If you haven't already, read `Time.h` and identify operations that you might need to use.

When you think you've completed the function, run the following command to test it:

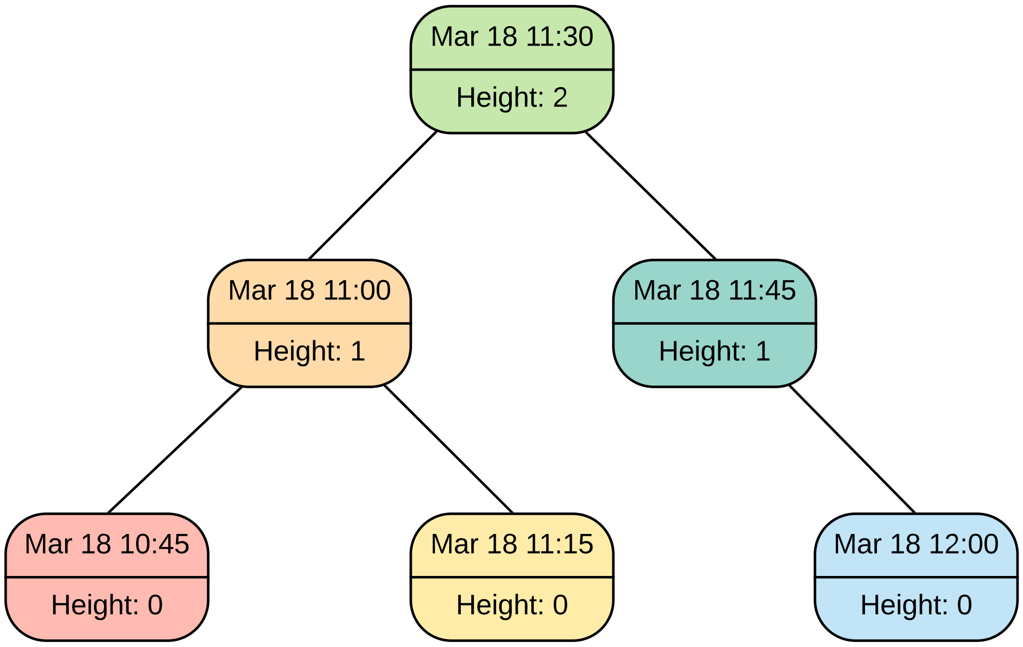
```
$ ./runtests.sh 3
```

If you manage to pass the tests, then congratulations! You've completed the implementation of the AVL tree. Now, we need to add some specialised functions to solve the scheduling problem.

Task 3

Implement the `TreeFloor()` and `TreeCeiling()` functions in `Tree.c`. `TreeFloor()` takes a `Time` object, and returns the latest time in the tree that is earlier than or equal to the given time, or `NULL` if no such time exists. Meanwhile, `TreeCeiling()` takes a `Time`, and returns the *earliest* time in the tree that is later than or equal to the given time, or `NULL` if no such time exists.

Let's clarify this with a few examples. Suppose the tree contained the following times:



- `TreeFloor(t, Mar 18 10:59)` should return `Mar 18 10:45`, since `Mar 18 10:45` is the latest time in the tree that is earlier than or equal to `Mar 18 10:59`.
- `TreeFloor(t, Mar 18 11:45)` should return `Mar 18 11:45`, since `Mar 18 11:45` is the latest time in the tree that is earlier than or equal to `Mar 18 11:45`.

- `TreeFloor(t, Dec 31 23:59)` should return `Mar 18 12:00`, since `Mar 18 12:00` is the latest time in the tree that is earlier than or equal to `Dec 31 23:59`.
- `TreeFloor(t, Mar 18 10:44)` should return `NULL`, since there are no times in the tree that are earlier than or equal to `Mar 18 10:44`.
- `TreeCeiling(t, Mar 18 11:01)` should return `Mar 18 11:15`, since `Mar 18 11:15` is the earliest time in the tree that is later than or equal to `Mar 18 11:01`.
- `TreeCeiling(t, Mar 18 12:05)` should return `NULL`, since there are no times in the tree that are later than or equal to `Mar 18 12:05`.

Hint: If you choose to use recursion, you likely won't be able to perform recursion in `TreeFloor()` and `TreeCeiling()`, as these functions take a `Tree` as input. Instead, create recursive helper functions that take a `Node` as input (as well as the given `Time` object). For an example of this, see the `TreeInsert()` function, and notice that it does almost nothing, except call `doInsert()`.

Note: Aim to visit as few nodes as possible in your `TreeFloor()` and `TreeCeiling()` functions. If you achieve this, then your functions will have a time complexity of O(log *n*), where *n* is the number of nodes in the tree.

When you think you're done, run the following commands to test your code:

```
$ ./runtests.sh 4
$ ./runtests.sh 5
```

The first command will test `TreeFloor()`, while the second command will test `TreeCeiling()`. Make sure you pass these tests before continuing, or you won't be able to complete the next task!

Task 4

We finally have all the pieces required to fix our scheduler. Implement the `ScheduleAdd()` function in `Schedule.c`, which takes a landing time and determines if it can be added to the schedule. If a landing can be scheduled at that time, add it to the schedule and return `true`. Otherwise, don't add it to the schedule, and return `false`. As with the previous tasks, you will need to make use of the `Time` ADT, so carefully read `Time.h` and identify potentially useful operations.

Reminder: A landing can be scheduled if it is not within 10 minutes of any other landing.

Once you are done, recompile the `runway` program and run it. Try scheduling a variety of landing times. Think of as many edge cases as you can and test your program with them. Does your program correctly accept and reject landing times?

Task 5

To make testing a little less tedious, we can create a file containing commands and make the program read in commands from the file, as we did in Lab 2. Create a file called `commands.txt` that contains commands to test several interesting cases. Here is an example `commands.txt` (yours should have more interesting cases):

```
+ 3 18 1200
+ 3 18 1200
s 1
```

Now run the following command:

```
$ ./runway -e < commands.txt
```

This causes the program to read in and execute commands from `commands.txt`. The `-e` option stands for "echo mode" - it echoes all commands read in so that we can see what commands are being performed in our output. Check the output. Does it appear to be correct?

After you have created the `commands.txt` file, you can also run the following command:

```
$ ./runtests.sh 6
```

This will apply the commands in your `commands.txt` file to your implementation and our reference implementation and compare the results. (Note that this command won't work unless you're on the CSE system.) If there are any differences, inspect the output and fix your program.

Submission

You need to submit three files: `Tree.c`, `Schedule.c` and `commands.txt`. Submit them from the command line via `give`, and then show your tutor, who will give you feedback on your coding style and your test quality, and award a mark.

Have fun, *Kevin*