Week 04 Lab Exercise

Binary Search Trees and Student Records

Objectives

- To learn how structs can be stored in binary search trees
- To get some practice with binary search trees
- To explore a use of function pointers
- To get familiar with some ideas used in Assignment 1

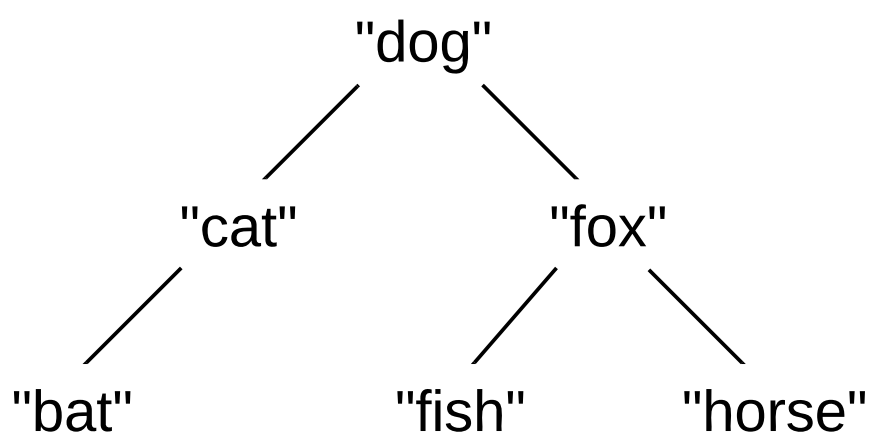
Admin

Marks	5 (see Assessment section for more details)
Demo	in the Week 04 or Week 05 lab
Submit	see the Submission section
Deadline	submit by 5pm Sunday of Week 04
Late penalty	0.5 marks per day

Background

Comparison Functions

So far, most of the examples you have seen of binary search trees (including last week's lab) have been of integers. But binary search trees don't have to contain integers - any data type whose values can be ordered from smallest to largest can be stored in a binary search tree. For example, here is an example of a binary search tree that contains strings:



With integers, the < and > operators (which are available in virtually every programming language) can be used to easily determine the ordering of values. In higher-level programming languages (such as Python), the same operators can be used to compare strings, but in C, string comparison is usually performed using the `strcmp` function. This function takes in the two strings to be compared and returns:

- a negative integer if the first string is lexicographically (alphabetically) less than the second string
- zero if the two strings are equal (i.e., they contain the exact same sequence of characters)
- a positive integer if the first string is lexicographically greater than the second string

This kind of function that takes in two values of the same data type and "compares" them by returning a negative, zero or positive integer is typically called a *comparison function* or comparator. `strcmp`, mentioned above, is a comparison function for strings.

In the context of binary search trees, the return value of a comparison function is very useful, as it determines how we should proceed when inserting into or searching a binary search tree. Suppose we have a binary search tree of strings. If we call `strcmp(s, n->value)`, where `s` is the string being inserted/searched for and `n->value` is the string value in the current node, a negative return value indicates

that we should go left (as `s` is "less than" the string in the current node), a positive return value indicates that we should go right, and a return value of zero indicates that `s` is already in the tree. Here is what a search function for a tree of strings might look like:

```
bool doTreeSearch(Node n, char *s) {
    // not found
    if (n == NULL) {
        return false;
    }

    int cmp = strcmp(s, n->value);
    if (cmp < 0) {
        return doTreeSearch(n->left, s);
    } else if (cmp > 0) {
        return doTreeSearch(n->right, s);
    } else { // (cmp == 0)
        return true;
    }
}
```

We can write comparison functions for any data type. We can even write one for integers, even though integers can already be easily compared with the < and > operators.

```
int compareInts(int a, int b) {
    if (a < b) {
        return -1;
    } else if (a > b) {
        return  1;
    } else {
        return  0;
    }
}
```

Why not just return `a - b`? Because `a - b` is susceptible to integer overflow/underflows!

Comparison functions are especially useful when we have defined our own data type (by defining our own struct), because they allow us to isolate the logic for comparisons (which could be complex) into a separate function, which improves readability. It also allows us to take advantage of function pointers if we have defined multiple comparison functions for the same data type, which we will see later.

A Custom Data Type - The Student Record

Suppose we have defined our own student record data type that consists of a zid and name. Its definition looks like:

```
struct record {
    int  zid;      // must be between 1 and 9999999
    char name[16]; // must be at most 15 chars in length
};
```

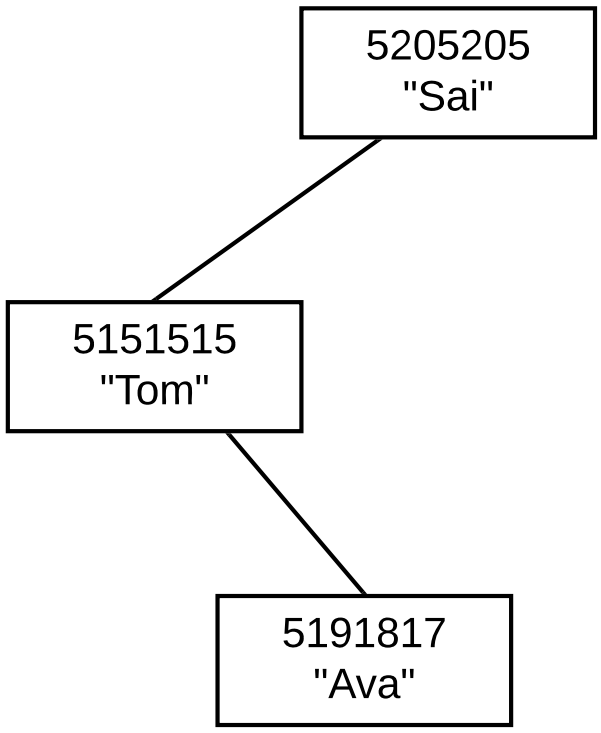
Suppose we now want to implement a binary search tree of records, ordered on zid, for efficient searching. How do we define the comparison function for this tree? Firstly, since the binary search tree is ordered on zid and zids are unique (we'll never have two students with the same zid), the comparison function will never need to be concerned with names. Furthermore, since zids are integers, our comparison function should be similar to a comparison function for integers, except that it takes in records, rather than integers. Here is one possible implementation, based on the `compareInts` function above:

```
int compareByZid(struct record a, struct record b) {
    if (a.zid < b.zid) {
        return -1;
    } else if (a.zid > b.zid) {
        return  1;
    } else {
        return  0;
    }
}
```

Since zids are restricted to be between 1 and 9999999, there is no risk of an integer overflow/underflow, and so we can simplify the function down to the following:

```
int compareByZid(struct record a, struct record b) {
    return a.zid - b.zid;
}
```

With this comparison function, we can now easily implement the search, insertion and deletion algorithms for our tree.



Records ordered on zid

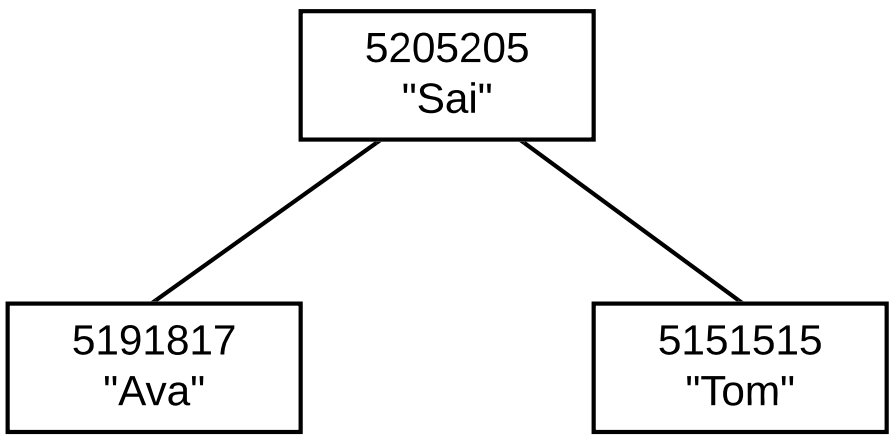
We can now search efficiently by zid. But what if we want to be able to search by name? The first tree won't be able to help us, as it is ordered on zid. We would need another tree ordered on the name field, which means we need another comparison function! Think about how you would implement this function. It should have the same interface as `compareByZid`:

```
int compareByName(struct record a, struct record b);
```

If you thought of this comparison function:

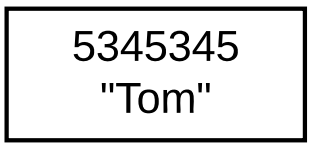
```
int compareByName(struct record a, struct record b) {
    return strcmp(a.name, b.name);
}
```

You're on the right track, but there is a problem. Zids are unique, so if `compareByZid` returns 0, then we *know* we have found the right record, as there can't be another record with the same zid. On the other hand, names are *not* unique, so it is possible for `compareByName` to return 0 for two records that have the same name, but different zids. Suppose we had the following binary search tree, which uses `compareByName` as its comparison function:



Records ordered on name

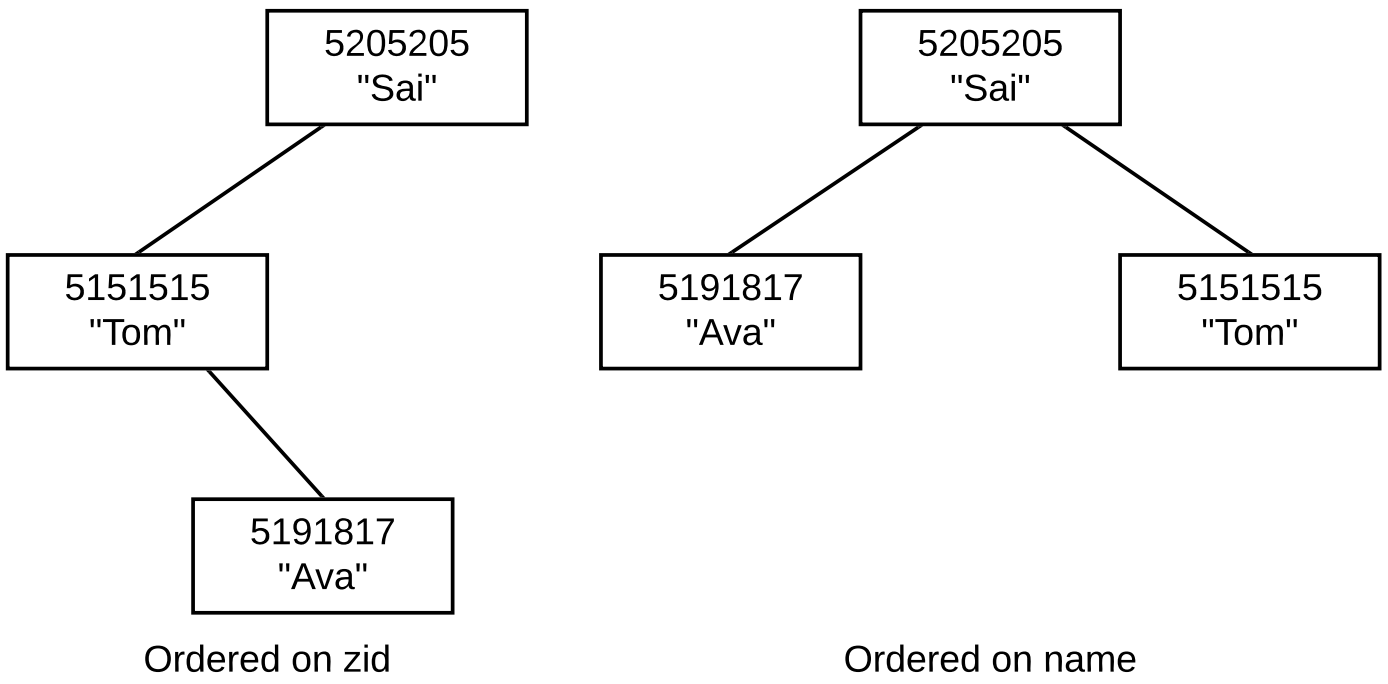
If we wanted to insert the record of another student named "Tom", our insertion function (which uses `compareByName`) would not insert the record, as `compareByName` would say that the new record is equivalent to the existing record with the name "Tom".



Thus, any comparison function must use a combination of fields/attributes that is guaranteed to be unique. `compareByZid`, which we implemented above, only needs to use the zid field, as zids are unique. For `compareByName`, this means also comparing the records' zids in case the names happened to match. Here is the improved `compareByName`:

```c
int compareByName(struct record a, struct record b) {
    int cmp = strcmp(a.name, b.name);

    // if names are not the same, return the result of strcmp
    if (cmp != 0) {
        return cmp;

    // names are the same, so compare zids
    } else {
        return compareByZid(a, b);
    }
}
```

Now that we have comparison functions for zid and name, we can implement two binary search trees - one ordered on zid and another ordered on name.



Since we now have two trees containing the same data, it is important to note that any operation that modifies one of the trees must be performed on both trees, otherwise there would be inconsistencies. For example, if a record is inserted into the zid-ordered tree, it must also be inserted into the name-ordered tree. The same goes for deletion.

Function Pointers

We can now search records efficiently by zid and by name, but let's have a closer look at the implementation of these trees. Here are the insertion functions:

```c
Node doTreeZidInsert(Node n, struct record r) {
    if (n == NULL) {
        return newNode(r);
    }

    int cmp = compareByZid(r, n->value);
    if (cmp < 0) {
        n->left = doTreeZidInsert(n->left, r);
    } else if (cmp > 0) {
        n->right = doTreeZidInsert(n->right, r);
    }
    return n;
}
```

```c
Node doTreeNameInsert(Node n, struct record r) {
    if (n == NULL) {
        return newNode(r);
    }

    int cmp = compareByName(r, n->value);
    if (cmp < 0) {
        n->left = doTreeNameInsert(n->left, r);
    } else if (cmp > 0) {
        n->right = doTreeNameInsert(n->right, r);
    }
    return n;
}
```

Can you see a problem with the above code? If you guessed code duplication, you'd be right! Other than the comparison function used, these functions are virtually identical. This problem only becomes worse when we consider that we also need functions for search and deletion.

This is where function pointers come in! Instead of hard-coding calls to the various comparison functions (which would require multiple tree implementations and lots of code duplication), we can let the user decide what comparison function to use by passing in a pointer to the function when they create the tree. To create a tree ordered on zid, the user can pass in a pointer to `compareByZid`, and to create a tree ordered on name, the user can pass in a pointer to `compareByName`.

In this new implementation, each tree struct now contains a pointer to the root node, as well as a pointer to the comparison function that should be used in that tree.

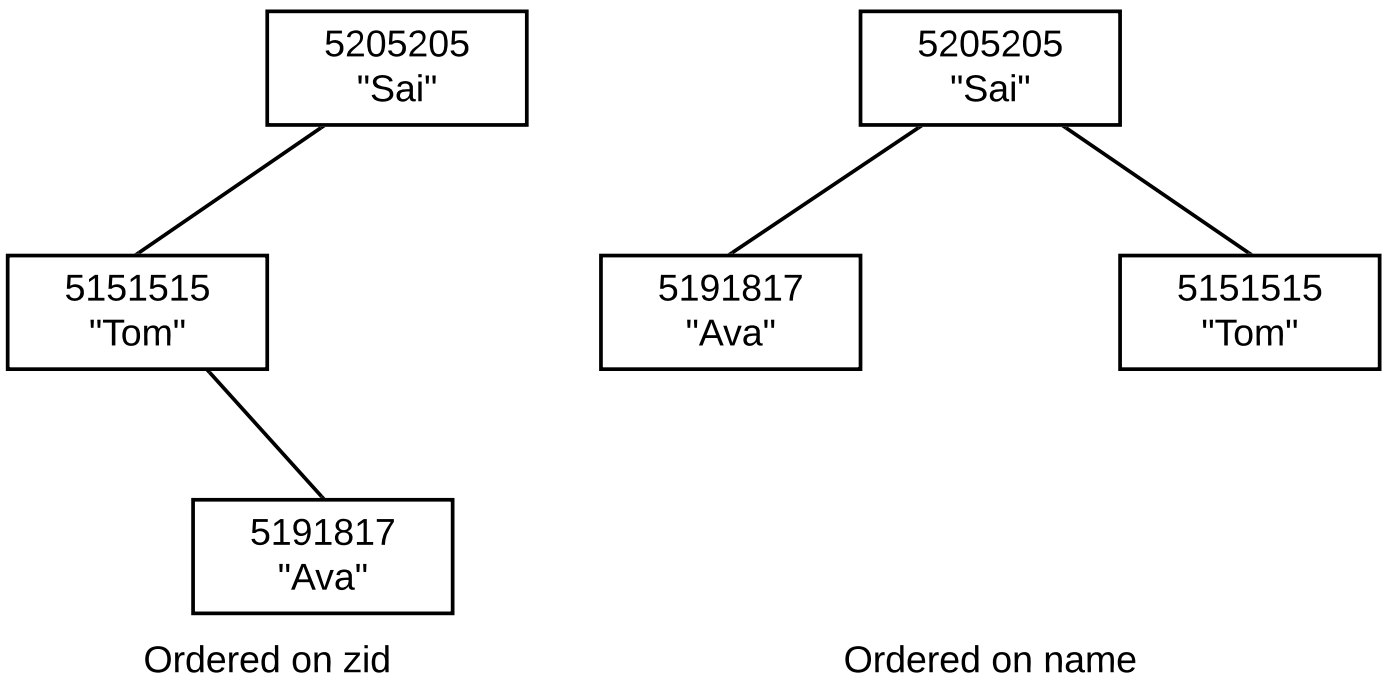
```
// the tree struct
struct tree {
    Node root;
    int (*compare)(struct record, struct record);
};

Tree TreeNew(int (*compare)(struct record, struct record)) {
    Tree t = malloc(sizeof(*t));
    // malloc error checking here

    t->root = NULL;
    t->compare = compare;
    return t;
}
```

Now that each tree has a configurable comparison function, both of our trees can use the same tree implementation. Here is the new insertion function. Take notice of how the comparison function is called.

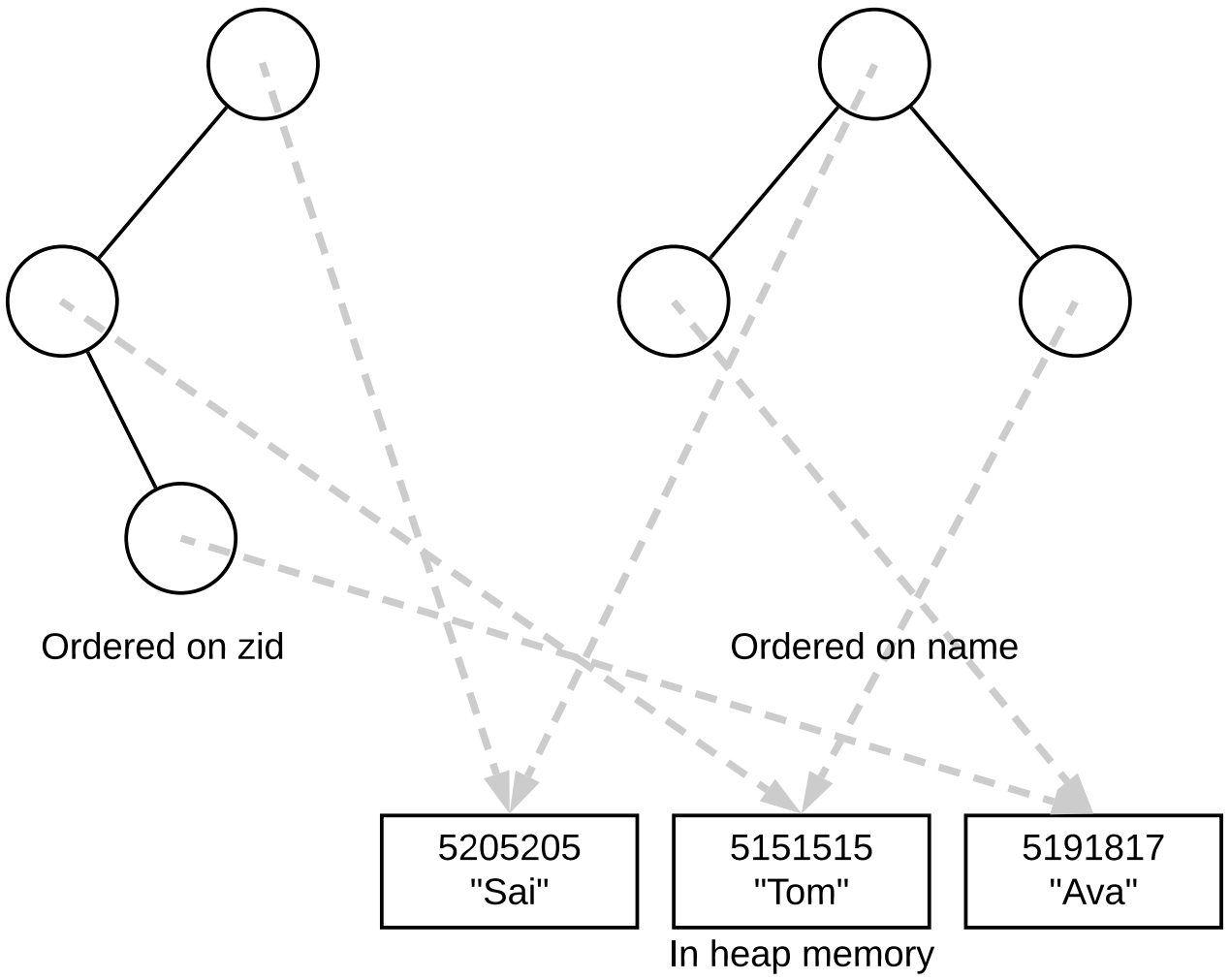
```
Node doTreeInsert(Tree t, Node n, struct record r) {
    if (n == NULL) {
        return newNode(r);
    }

    int cmp = t->compare(r, n->value);
    if (cmp < 0) {
        n->left = doTreeInsert(t, n->left, r);
    } else if (cmp > 0) {
        n->right = doTreeInsert(t, n->right, r);
    }
    return n;
}
```

Saving Space

Above, we assumed that each tree node contains a record struct. This means if we create multiple search trees (ordered on different fields or attributes), there will be multiple copies of each record, which will consume a lot of space.



To save space, we can let each tree node store a pointer to a record rather than the full record itself. That way, only one copy of each record needs to exist.



Setting Up

Create a directory for this lab, change into it, and run the following command:

```
$ unzip /web/cs2521/21T1/labs/week04/downloads/lab.zip
```

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

If you've done the above correctly, you should now have the following files:

`Makefile`	a set of dependencies used to control compilation
`runDb.c`	a program that provides a command-line interface to the StudentDb ADT
`runtest`	a script that runs a few tests
`test.c`	a program that produces output for a few tests
`List.c`	implementation of the `List` ADT (complete)
`List.h`	interface to the `List` ADT
`Record.c`	implementation of the `Record` ADT (complete)
`Record.h`	interface to the `Record` ADT
`StudentDb.c`	implementation of the `StudentDb` ADT (incomplete)
`StudentDb.h`	interface to the `StudentDb` ADT
`Tree.c`	implementation of the `Tree` ADT (incomplete)
`Tree.h`	interface to the `Tree` ADT
`tests/`	a sub-directory containing expected output for some tests

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: `./runDb` and `./test`.

File Walkthrough

`runDb.c`

`runDb.c` provides a command-line interface to the StudentDb ADT. It creates a StudentDb object, and then accepts commands to interact with it, including inserting student records into the database, searching for records, and deleting records. Here is an example session with the program:

```
$ ./runDb
StudentDB v1.0
Enter ? to see the list of commands.
> ?
Commands:
    + <zid> <family name> <given name>     add a student record
   lz                                      list all records in order of zid
   ln                                      list all records in order of name
    d <zid>                                delete a student record
   fz <zid>                                find a student record by zid
   fn <family name> <given name>           find student records by name
    ?                                      show this message
    q                                      quit

> + 1 Stark Tony
Successfully inserted record!
> + 2 Banner Bruce
Successfully inserted record!
> + 6 Rogers Steve
Successfully inserted record!
> + 4 Pym Hank
Successfully inserted record!
> + 3 Romanoff Natasha
Successfully inserted record!
> lz
1|Stark|Tony
2|Banner|Bruce
3|Romanoff|Natasha
4|Pym|Hank
6|Rogers|Steve
> ln
> fz 1
Found a record:
1|Stark|Tony
> fz 3
Found a record:
3|Romanoff|Natasha
> fn Rogers Steve
No records found
> d 1
Successfully deleted record!
> lz
2|Banner|Bruce
3|Romanoff|Natasha
4|Pym|Hank
6|Rogers|Steve
> fz 1
No records with zid '1'
> q
$
```

Note that the program doesn't correctly perform operations involving names. We will add code to fix this later.

`StudentDb.c`

`StudentDb.c` implements the StudentDb ADT, which handles student records. A user can attempt to insert a student record by calling the `DbInsert()` function, and the StudentDb ADT will insert the record if there is not already a record with the same zid. The StudentDb ADT also provides other useful operations, such as deleting a record with a given zid, finding a record with a given zid or name, and listing all the records in order of zid or name.

You should have a read of the functions in `StudentDb.c` to understand how they use the Tree and Record ADTs. Note that at the moment, only the operations involving insertion and zid work.

`Tree.c`

`Tree.c` implements the Tree ADT, which is used to enable efficient searching of records. Notice that the `TreeNew` function (which creates a new tree) takes in a function pointer to a comparison function which determines how the records should be ordered. You should read the `DbNew` function in `StudentDb.c` to see how it passes the comparison functions to `TreeNew`. You should also have a quick read of the search, insertion and deletion algorithms in `Tree.c` to see how they access and call the comparison function.

`List.c`

`List.c` implements the List ADT, which is used to create a list of records. Note that the list has no sorting capabilities, so if you want the records to be ordered in a particular way, you must ensure they are appended in that order.

`Record.c`

`Record.c` implements the Record ADT. You should read the definition of `struct record` to see what a student record consists of, but since this is an ADT, you won't be able to access the fields directly - you'll need to use the `RecordGetXYZ` functions listed in `Record.h` to access them. This ADT is fully implemented, so you do not need to modify it.

Header files

StudentDb, Tree, List and Record are ADTs, and their associated header files contain extensive descriptions of their interface functions. If you are unsure about what a certain ADT function does or is supposed to do, you should read its description in the relevant header file.

Additional Notes

Dummy Records

If you have read through the functions in `StudentDb.c`, you would have noticed the use of so-called "dummy records" in `DbFindByZid` and `DbDeleteByZid`. What are they and why do we need them?

Suppose that the StudentDb contains a number of records and we want to find a record with a given zid. We should search the tree that is ordered on zid, but `TreeSearch` takes in a record, not a zid. So what should we pass into `TreeSearch`?

That is where the dummy record comes in! Because the comparison function used by the zid-ordered tree only inspects the zid of each record (see `compareByZid`), two records that have the same zid will be considered equal, no matter what names they contain. Hence, we can create a dummy record that contains the zid we are searching for, along with some dummy names (an empty string is fine), and pass this into `TreeSearch`. If the tree does contain a record with that zid, it will consider this record as being equal to the dummy record, and return the real record.

This is why it is important that `compareByZid` only compares the zids of the records, and not any other fields. If `compareByZid` also compared the names of the records, this wouldn't work.

Task 1

Implement the `compareByName()` comparison function in `StudentDb.c`. The function should compare the two given records by name (family name first, then given name) and then by zid if the names are equal.

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

```
$ ./runtest 1
```

The command generates observed output for the `compareByName` tests in `test.c` and compares them with the expected output in `tests/1.exp`.

Once you pass the tests for `compareByName`, modify the `DbNew` function in `StudentDb.c` so that the `db->byName` tree actually uses the new comparison function.

> **NOTE:**
>
> Tasks 1 and 2 can be completed independently of each other. However, both tasks need to be completed before attempting Task 3.

Task 2

Implement the `TreeSearchBetween()` function in `Tree.c`. This function should search the tree for all records that are considered (by the tree's comparison function) to be between the records `lower` and `upper` (inclusive), and return them all in a list in order. In order for the search to be as efficient as possible, the function should visit as few nodes as possible.

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

```
$ ./runtest 2
```

The command generates observed output for the `TreeSearchBetween` tests in `test.c` and compares them with the expected output in `tests/2.exp`.

> **HINT:**
>
> We have provided a stub helper function, `doTreeSearchBetween()`, to help you get started. It is recommended that you implement and use this function, but you can write your own helper function if you want.

> **NOTE:**
>
> What does "visit as few nodes as possible" mean? Essentially, it means that you shouldn't visit nodes unnecessarily. A basic in-order traversal that adds the requested records to the list would be extremely easy to implement, but also very inefficient, as it would need to visit all nodes in the tree.

Task 3

Implement the `DbFindByName()` function in `StudentDb.c`. This function should find all the records that have the same family name *and* given name as the provided family name and given name, and return them in a list in increasing order of zid.

Also implement the `DbListByName()` function in `StudentDb.c`. This function should display all the records in order of name (family name first).

> **HINT:**
>
> Use the `TreeSearchBetween` function that you implemented in Task 2.

> **HINT:**
>
> If you're trying to use `TreeSearchBetween` but you are unsure about what `lower` and `upper` should be, think about how you could use dummy records to extract just the range of records that you need.

> **HINT:**
>
> Don't introduce any memory leaks! If you create a record in `DbFindByName` but don't need it anymore after the function returns, you should make sure to free it. (Don't just call the `free` function - `Record` is an ADT so you should use an ADT function to free it.)

Once you are done, recompile the `runDb` program with the `make` command and run it. Use commands to insert records, search for records (by zid and by name), and list records (by zid and by name). Does your `DbFindByName` function work when there are multiple records with the same name? Do the records get returned in the correct order (i.e., in increasing order of zid)? What if there are no records with the given name? Test thoroughly.

Task 4

To make testing a little less tedious, we can create a file containing commands and make the program read in commands from the file. Create a file called `commands.txt` that contains commands to test your `DbFindByName` and `DbListByName` functions from Part 3. Here is an example `commands.txt`:

```
+ 1 Stark Tony
+ 2 Banner Bruce
fz 1
```

Now run the following command:

```
$ ./runDb -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?

Submission

You need to submit three files: `StudentDb.c`, `Tree.c` and `commands.txt`. You can submit these via the command line using `give` using the command:

```
$ give cs2521 lab04 StudentDb.c Tree.c commands.txt
```

or you can submit them via WebCMS. After submitting them (either in this lab class or the next) show your tutor, who'll give you feedback on your code, ask you about your tests, and award a mark.

> **NOTE:**
>
> If you did not complete all the tasks, you must still submit *all* the required files. This could mean creating an empty file if a required file did not exist before.

Assessment

To receive a mark for this lab, you *must* demonstrate your lab to your tutor during your Week 04 or Week 05 lab session. You will be marked based on the following criteria:

Comparison function (1 mark)
This mark is for the correctness of your comparison function in Task 1.

`TreeSearchBetween` (2 marks)
These marks are for the correctness of your `TreeSearchBetween` function in Task 2 (1 mark), as well as whether you were able to avoid unnecessarily visiting nodes (1 mark).

`DbFindByName` (1 mark)
This mark is for the correctness of your `DbFindByName` function in Task 3. The correctness of your function will be determined by tests that we will run against your submission.

Testing (1 mark)
This mark is for whether your `commands.txt` file contains adequate tests for your `DbFindByName` function. These tests should cover at least a few interesting test cases.