CSCI 2132: Software Development

Lab 8: Structs and Dynamic Memory Management

Synopsis

In this lab, you will:

- Learn about C `structs`.
- Learn to use `malloc` and `free` to manage heap-allocated memory.
- Learn a bit about efficient data structures.
- Take your first steps with compiling multi-file C projects.

Contents

Overview

In this lab, you will use two features of C that we have not discussed in class yet: `structs` and dynamic memory management. We will cover them shortly in class. The final product you produce in this course is a balanced binary search tree, an efficient data structure you will hopefully learn more about in either CSCI 2110 or CSCI 3110. In order to implement this data structure, you will make heavy use of pointers.

You will split your code into three source files: a header file *declaring* the necessary data types and functions, a source file *defining* these types and functions, and an application that includes the header file to make use of the tree data structure. Thus, you have two source files that need to be compiled separately and then linked to obtain an executable program.

Note: This lab may seem very long. It is and it isn't. There are 12 pages that simply show the final code you constructed over the previous pages. The code itself is little over 200 lines, which is substantial but should be well within the complexity you should be able to work with in second year. Still, you will probably not be able to finish this lab in one sitting. I encourage you to finish it at home.

Step 0: An introduction to `structs` and dynamic memory management

`structs`

You can think of a `struct` as a Java class without methods and without access protection: it is nothing but a collection of named data fields that can be accessed and manipulated freely. Here's an example:

```
struct student {
    char *first_name;
    char *last_name;
    unsigned int banner_number;
    float gpa;
};
```

As you can see, a `struct` is in a sense orthogonal to an array in its capabilities: an array allows you to store a number of values that does not need to be known at compile time, but all values must be of the same type. A `struct` can store only a fixed number of values, but they can have arbitrary types (which must be specified at compile time).

To access the elements of a `struct`, you use dot-notation as in Java:

```
int main() {
    struct student aplus_gal;
    aplus_gal.first_name = "Jenna";
    aplus_gal.last_name = "Best";
    aplus_gal.banner_number = 1;  // Represents B00000001
    aplus_gal.gpa = 4.3;
    print_student(aplus_gal);
    return 0;
}
```

The invocation `print_student(aplus_gal)` is interesting because it passes `aplus_gal` *by value*; a copy of the entire structure is passed to the function as an argument. This is different from Java where all objects are passed by sharing (i.e., as a pointer) and can get costly quickly. You will soon switch to using pointers to manipulate `structs` to avoid this cost. For now, here is the `print_student` function that completes this example.

```
void print_student(struct student s) {
    printf("First name:    %s\n", s.first_name);
    printf("Last name:     %s\n", s.last_name);
    printf("Banner number: B%08u\n", s.banner_number);
    printf("GPA:           %.2f\n", s.gpa);
}
```

One annoying feature of C `structs` you may have noticed is that, wherever you declare a variable or function argument of a `struct` type, you need to write `struct structname` instead of just `structname`.

This gets tedious very quickly. Thus, a common C idiom is to define an *anonymous* struct, one that does not have a name, and immediately give it a name using `typedef`:

```
typedef struct {
    char *first_name;
    char *last_name;
    unsigned int banner_number;
    float gpa;
} student;
```

Now you can (and must) simply write `student` instead of `struct student` everywhere you used `struct student` in the example above.

Next you will modify the program to use a pointer as the argument of the `print_student` function. `structs` are types as any other. Therefore, you can create pointers to `struct` values and dereference them to gain access to the referenced `structs` and their fields. The following example should hold (almost) no mystery for you at this point:

```
typedef struct {
    char *first_name;
    char *last_name;
    unsigned int banner_number;
    float gpa;
} student;

void print_student(const student *s) {
    printf("First name:    %s\n", (*s).first_name);
    printf("Last name:     %s\n", (*s).last_name);
    printf("Banner number: B%08u\n", (*s).banner_number);
    printf("GPA:           %.2f\n", (*s).gpa);
}

int main() {
    student aplus_gal;
    aplus_gal.first_name = "Jenna";
    aplus_gal.last_name = "Best";
    aplus_gal.banner_number = 1;  // Represents B00000001
    aplus_gal.gpa = 4.3;
    print_student(&aplus_gal);
    return 0;
}
```

Instead of passing `aplus_gal` to `print_student`, you now pass its address in the last line of the main function. Consequently, the `print_student` function needs to dereference this pointer to get access to the referenced `struct`. For example, it now refers to the `first_name` field as `(*s).first_name` instead of `s.first_name` because `s` is no longer a value of type `student` but a pointer to a value of type `student`.

One detail that may not be completely self-explanatory is the use of `const student *` instead of `student *` as the type of the argument of `print_student`. As discussed in class, this is a common idiom

when passing a value as a pointer for efficiency reasons. It guarantees that the called function can *read* the referenced data but not modify it.

The pattern of dereferencing a pointer and accessing a field of the referenced `struct` becomes tedious fairly quickly but is very common. To avoid cluttering your code with excessive parentheses, reduce your typing effort, and make your code more readable, C has an operator that merges pointer dereferencing and field access into one: the "arrow operator" `->`. Writing `s->first_name` has *exactly* the same effect as writing `(*s).first_name`.

This leads to the final version of our little A+ student example and finishes this crash course on `structs` in C:

aplus.c

```
#include <stdio.h>

typedef struct {
    char *first_name;
    char *last_name;
    unsigned int banner_number;
    float gpa;
} student;

void print_student(student *s) {
    printf("First name:    %s\n", s->first_name);
    printf("Last name:     %s\n", s->last_name);
    printf("Banner number: B%08u\n", s->banner_number);
    printf("GPA:           %.2f\n", s->gpa);
}

int main() {
    student aplus_gal;
    aplus_gal.first_name = "Jenna";
    aplus_gal.last_name = "Best";
    aplus_gal.banner_number = 1;  // Represents B00000001
    aplus_gal.gpa = 4.3;
    print_student(&aplus_gal);
    return 0;
}
```

If you are confident that you fully understand the above code, move on to the next part of the lab; there is plenty of work left to do. If you feel you need to try this code out to gain some confidence, then copy the above code into a text file, compile it using gcc and check that it does what you expect it to do.

Dynamic memory management

We discussed in class that local variables of functions are stored in the function call's stack frame and cease to exist the moment the function call returns. Thus, a function cannot create any values that should outlive the duration of the function call.

One alternative would be to store such values in static variables that are stored in the `DATA` segment of your code and thus exist for as long as your program runs. This may be okay if you know the sizes, types, and number of such variables that you need at compile time, but when the number of such variables depends on the input you are processing, this is not a viable solution either.

This is where the heap comes to your rescue: it is a memory space where you can store values whose lifetime is not limited to the duration of a single function call and whose number or size (in the case of arrays) you cannot determine at compile time.

In Java, all objects allocated with `new` are stored on the heap; the variable to which this new object is "assigned" stores only a reference (i.e., pointer) to the memory location on the heap where the object is stored. Java's garbage collector periodically searches the heap for objects that are no longer referenced by any variable and then destroys those objects, that is, it marks the memory occupied by these objects as unused (and thus available for future allocation requests) again.

C's management of heap-allocated data differs from Java's in three ways: First, it makes the fact that you are storing a pointer to a value rather than a value explicit; every time you work with heap-allocated data, pointers abound. Second, allocating a chunk of heap memory is not type-safe as in Java: You simply tell the `malloc` function how many bytes of memory you need; `malloc` reserves a chunk of heap memory of at least this size for you and returns a `void` pointer to it; you then need to assign this `void` pointer to a pointer of the type you really want so you can work with the allocated memory.[1] If you allocate too little memory (the `sizeof` operator helps you avoid this mistake), you will eventually access memory locations outside the allocated heap block and bad things will happen.

The third difference is the most jarring for users of modern programming languages: there is no garbage collector (we discussed why). So, whether you still hold a pointer to a heap-allocated memory block or not, C will make no effort to release this block back to the set of available memory locations on the heap. It is your responsibility as a programmer to tell C when you are done using a given chunk of memory by calling `free` with a pointer to the memory block as an argument.

You will explore this more in the context of building binary search trees in this lab. For now, modify your A+ student example so the student is created on the heap by a function `make_student`. This function returns a pointer to the student record. Your `main` function calls `make_student` to obtain a new student record, then calls `print_student` to print it, and finally frees the allocated memory before exiting. Here is the `make_student` function:

[1] C does *not* require you to cast the pointer type. It is reasonable to assume that this is because `void` pointers do not allow you to do anything with the referenced data, so you simply *must* assign it to a typed pointer. Since this always involves a cast to the pointer type on the left-hand side of the assignment, the cast happens implicitly.

```
student *make_student() {
    student *aplus_gal = malloc(sizeof(student));
    aplus_gal->first_name = "Jenna";
    aplus_gal->last_name = "Best";
    aplus_gal->banner_number = 1;
    aplus_gal->gpa = 4.3;
    return aplus_gal;
}
```

Compared to the way you previously created the `aplus_gal` record, there are two differences: As just said, you request a chunk of memory to hold the student record. You use `malloc` to do so and request `sizeof(student)` bytes of memory. Second, since `aplus_gal` is now a pointer to `student`, not a value of type `student` itself, all fields of the record need to be accessed using "`->`" rather than "`.`". Finally, you return a pointer to the allocated memory block to the caller of `make_student`.

Next, modify your `main` function so it uses `make_student` to get a student record:

```
int main() {
    student *aplus_gal = make_student();
    print_student(aplus_gal);
    free(aplus_gal);
    return 0;
}
```

The most notable difference here is that you use `make_student` to create the student record to print. As a result, `aplus_gal`'s type has changed from `student` to `student *`. After calling `print_student` on this record—this function does not change from its earlier version because it already takes a pointer as an argument—you make sure the memory occupied by the student record is released by calling `free(aplus_gal)`.

The complete code now looks like this:

aplus2.c (1/2)

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    char *first_name;
    char *last_name;
    unsigned int banner_number;
    float gpa;
} student;
```

aplus2.c (2/2)

```
student *make_student() {
    student *aplus_gal = malloc(sizeof(student));
    aplus_gal->first_name = "Jenna";
    aplus_gal->last_name = "Best";
    aplus_gal->banner_number = 1;
    aplus_gal->gpa = 4.3;
    return aplus_gal;
}

void print_student(const student *s) {
    printf("First name:    %s\n", s->first_name);
    printf("Last name:     %s\n", s->last_name);
    printf("Banner number: B%08u\n", s->banner_number);
    printf("GPA:           %.2f\n", s->gpa);
}

int main() {
    student *aplus_gal = make_student();
    print_student(aplus_gal);
    free(aplus_gal);
    return 0;
}
```

If you paid attention, you may wonder why it is okay to assign `"Jenna"` and `"Best"` to the fields of `aplus_gal` without allocating them on the heap. In a production use of the `student` structure, you would store arbitrary strings as `first_name` and `last_name` and, yes, those values would also have to be allocated on the heap because their lifetime must be at least as long as the lifetime of the record that refers to them. Releasing the memory of a `student` record would then most likely involve not only releasing the memory of the record itself but also the memory referenced by pointers stored in the record. Here, you get away without doing this because `"Jenna"` and `"Best"` are string constants and are therefore stored in the `DATA` segment of your code: they exist for as long as your program runs.

Another note: Whenever you use `malloc` to request a chunk of heap-allocated memory, you should normally check its return value. This return value can be `NULL` (the equivalent of Java's `null`) if there is not enough heap memory available to satisfy the allocation request. If the return value is `NULL`, your program should print an error message and terminate gracefully instead of trying to work with the `NULL` pointer and, as a result, most likely crashing with a segmentation fault. You did not do this here in the interest of simplicity.

Binary search trees: Efficient dictionaries

A common problem almost any non-trivial program has to solve is storing a collection of values that can be updated and searched quickly. Such a data structure is called a *dictionary*. A dictionary is what is called an *abstract data type* (ADT), which simply means that it is defined not by its concrete implementation but by the set of operations it must support. In Java terminology, an abstract data type is an `interface`.

A basic dictionary should support six operations:

Create: Create a new dictionary.

Destroy: Destroy the dictionary, that is, release all heap memory it uses. (In a language like Java, the garbage collector would take care of this. In C, you need to call a function to accomplish this.)

Insert(x): Insert an item x into the dictionary.

Delete(k): Delete an item with key k. (The dictionary organizes the records it stores by keys associated with them. For a student record, for example, a reasonable key would the student's banner number. Which part of the record to use as the key is part of the definition of the dictionary.)

Find(k): Decide whether there exists an x with key k in the dictionary and, if so, return it.

List: Provide some means to iterate over all items in the dictionary.

There are many concrete dictionaries. Java has `ArrayLists` (essentially arrays), linked lists, hash tables, and search trees. Each of these data structures supports all the basic dictionary operations and is thus a dictionary, but the cost of these operations differ between concrete dictionary implementations.

For an array, insertions and deletions can take up to linear time no matter whether you keep the array sorted or not. The advantage of sorting the array is that you can use binary search to search for any element in the array in only $O(\lg n)$ time. If you do not sort the array, then the best you can do to locate a given element (or decide that it is not present) is to iterate over all elements in the array until you find the element or reach the end of the array. This takes linear time in expectation. This is also the search cost achieved by a linked list, no matter whether you keep it sorted or not (because binary search requires constant-time access to elements in the middle of the sequence, something that a linked list does not support). Insertions into a linked list, on the other hand, take constant time because you can simply add the new element to the beginning of the list. If the list is a doubly-linked list (with predecessor and successor pointers), then deletions can also be supported in constant time. Hashtables are the most efficient dictionaries in many situations in practice because they allow you to perform both updates (insertions and deletions) and searches in expected constant time. Here, we discuss balanced binary search trees, which achieve $O(\lg n)$ time per update or search, but in the worst case, not only in expectation.

The road map for this lab is as follows:

- First you will implement a basic binary search tree. The cost per operation depends heavily on the order in which you insert elements into the data structure because you will not make any effort to keep the tree balanced yet.
- Then you will write a simple test program that uses the tree to store a set of strings. You will be able to add strings to this set, delete strings from this set, search for a string matching a given pattern, and list all strings currently in the set in sorted order.

- Finally, you will modify your implementation so the tree remains balanced, that is, you will maintain a logarithmic bound on the height of the tree, thereby guaranteeing that the cost per operation is $O(\lg n)$.

  There are, once again, many ways to balance binary trees. Many methods that *guarantee* a height of $O(\lg n)$ are fairly complicated (albeit not nearly as complicated as often claimed); the best one in many ways is the strategy used by *red-black trees*. We will not discuss these here and instead opt to implement *splay trees*. The way you maintain balance in a splay tree is very simple. The penalty you pay for this simplicity is that you cannot *guarantee* that the cost of a single operation on a splay tree is $O(\lg n)$; you can only guarantee that a sequence of n operations takes $O(n \lg n)$ time, which is what you really care about most of the time. However, splay trees enjoy additional properties not enjoyed by the worst-case structures. For example, if you access the elements in the tree in almost sorted order, the cost on a red-black tree is $O(\lg n)$ per element access, $O(n \lg n)$ for the entire access sequence. A splay tree, on the other hand, can be proven to achieve a much lower linear cost for the entire access sequence in this case.

  Exploring different types of binary search trees would be a serious excursion into the theory of algorithms and data structures and thus is not something we do in this course. The main reason I chose splay trees as an example for this lab is that the rebalancing strategy is very simple but still demonstrates the basic operations needed to restore balance.

As a last step before starting your binary search tree implementation, here is the specification of the data structure you will develop:

The tree should be able to store values of arbitrary types, not just integers, floating point numbers or records of a specific type. Thus, the operations of the search tree will take `void *` arguments similar to the `qsort` function in the standard library, which takes a `void *` to the array to be sorted. The tree will not take ownership of the data items it stores, that is, when you delete an item from the tree, it will not be destroyed automatically. Instead, the delete operation will return the deleted item; if this is an item that needs to be deleted, the caller of the delete operation needs to take care of this. This makes the binary search tree more flexible because it allows you to build a binary search tree over a collection of elements stored in an array, for example, which should not be freed individually.

A consequence of wanting to store arbitrary elements in the tree is that the tree implementation cannot know how to compare elements stored in the tree, something that is needed because the tree's efficiency stems from the fact that it stores its elements in sorted order. Thus, the user needs to provide a comparator function that allows the tree to compare the elements it stores. Specifically, the user should provide *two* comparator functions, one to compare records and one to compare a given key with the key of a record. For the correct behaviour of the data structure, it is important that the ordering of two records is consistent with the ordering of their keys. In order to avoid passing these comparator functions to every dictionary operation, they are provided to the function that creates a new dictionary and are stored in the created dictionary. Any operation on the dictionary then uses these functions stored in the dictionary as needed. This gives the following interface to the tree data structure.

First, you need to define a function type that allows you to compare two records:

```
typedef int (*cmp_t)(const void *, const void *);
```

This says that `cmp_t` is a function type that takes two `void *` as arguments and returns an `int`. This `int` is less than, equal to or greater than zero if the first argument is less than, equal to or greater than

the second argument. (This is not pointer comparison. The function should compare the data items referenced by the two pointers.)

The comparison between keys and records is captured by a second function type `key_cmp_t`. Since you do not know anything about these key and record types, they are once again represented as `void *`, so the resulting type looks exactly the same as the above `cmp_t` type:

```
typedef int (*key_cmp_t)(const void *, const void *);
```

The first argument is a pointer to a key. The second argument is a pointer to a record. Even though any function that is a `cmp_t` is also a `key_cmp_t` and vice versa, it is still valuable to have two different names for these two types of functions because using them in the argument list of a function makes it explicit how the function intends to use these function arguments.

To create a tree of type `tree_t` (which will be define later), you need to implement a function

```
tree_t make_tree(cmp_t cmp, key_cmp_t key_cmp);
```

As said before, this creates a tree that uses `cmp` and `key_cmp` to compare records and keys and records.

When you are done using the tree, you need to destroy it using the function

```
void destroy_tree(tree_t tree);
```

Inserting a new element (represented as a `void *` because you want to be able to store values of any type) is done using the function

```
node_t tree_insert(tree_t tree, void *new_item);
```

This function inserts the new item into the tree and returns an identifier of (really a pointer to) the tree node storing this element.

This identifier is used when deleting an element:

```
void *tree_delete(tree_t tree, node_t node);
```

As said before, this function returns the deleted element so the user can free the memory occupied by this element if necessary. (Essentially, the tree returns ownership of the deleted item back to the user.)

To decide whether the tree stores a record with a given key, you use the following function:

```
node_t tree_find(tree_t tree, void *key);
```

If a record with the given key is found, this function returns the ID of the node storing this record. If no such record is found, it returns `NULL` (because `node_t` is just a pointer). Returning the ID of the tree node is useful because it allows you to use, for example, `tree_delete(tree, tree_find(tree, key))` to delete a record with a given key.

The record stored at a given tree node can be accessed using

```
void *node_value(node_t node);
```

The final function you need is one that allows you to iterate over all records in the tree. You will do this by creating an iterator:

```
iter_t make_iterator(tree_t tree);
```

This iterator type `iter_t` supports a single operation:

```
void *iter_next(iter_t iter);
```

advances the iterator to the next record in the tree and returns it. If there is no next element, that is, the iterator has moved past the last record in the tree, the return values is `NULL`.

Similarly to a tree, you need to destroy an iterator once you are done using it. The following function does this:

```
void destroy_iterator(iter_t iter);
```

Now let us get to work and implement the above functions:

Step 1: Create a header file that defines the interface

You will separate the implementation of the tree from the implementation of any code that uses the tree. Thus, you need to split the tree implementation into the C source code file that contains the implementations of all binary tree functions and a header file that only *declares* these functions and can be included by any code that uses your binary tree. Here is the header file:

tree.h

```
#ifndef TREE_H
#define TREE_H

struct _tree_t;
struct _node_t;
struct _iter_t;

typedef struct _tree_t *tree_t;
typedef struct _node_t *node_t;
typedef struct _iter_t *iter_t;

typedef int (*cmp_t)(const void *, const void *);
typedef int (*key_cmp_t)(const void *, const void *);

tree_t make_tree(cmp_t cmp, key_cmp_t key_cmp);
void destroy_tree(tree_t tree);

node_t tree_insert(tree_t tree, void *new_item);
void *tree_delete(tree_t tree, node_t node);

node_t tree_find(tree_t tree, void *key);

void *node_value(node_t node);

iter_t make_iterator(tree_t tree);
void destroy_iterator(iter_t iter);

void *iter_next(iter_t iter);

#endif /* TREE_H */
```

This header file uses two common idioms. First, it wraps the entire header file into

```
#ifndef TREE_H
#define TREE_H
...
#endif
```

As discussed in class, there are situations where you implement multiple header files that should include `tree.h` because they use tree functions. If some C file includes two or more such header files, you end up

including `tree.h` two or more times (because, remember, `#include "tree.h"` includes the text contained in `tree.h` verbatim). This double inclusion can create problems. The `#ifndef-#define-#endif` idiom ensures that the header file is included only once:

The first time you try to include it, `TREE_H` is undefined and `#ifndef` ("if not defined") succeeds. As a result, the code up to the matching `#endif` is included. Part of the included code is to define `TREE_H` (`#define TREE_H`). Thus, the next time you try to include `tree.h`, `TREE_H` is defined, which makes `#ifndef TREE_H` fail and the code between `#ifndef TREE_H` and `#endif` is silently ignored.

The other idiom is to define `tree_t`, `node_t` and `iter_t` as pointers to `structs` that are *declared* in the header file but not defined. Thus, the user of this header file has no way to manipulate trees, tree nodes, and iterators in any way other than using the functions provided in the header file because no information about the inner structure of the tree representation is shared with the user. The tree, tree node, and iterator types are *opaque*. This is one way to ensure proper separation between different components of a program.

Note that this works only because you define *pointers* to these `structs`. In order to define a pointer to some data type, the size of that data type does not have to be known. If you had written, for example,

```
struct _tree_t;
typedef struct _tree_t tree_t;
```

the compiler would have complained because it cannot determine the size of the type `tree_t` defined here because the size of a `struct _tree_t` is not known yet.

Step 2: Define the tree structure

A binary tree is a collection of tree nodes that are all reachable from the root of the tree (see Figure 1a). Thus, it suffices to store a pointer to the root. In addition, as discussed before, the tree stores the comparator functions so they do not need to be passed to each of the functions that work with the tree. This leads to the following definition of the tree type:

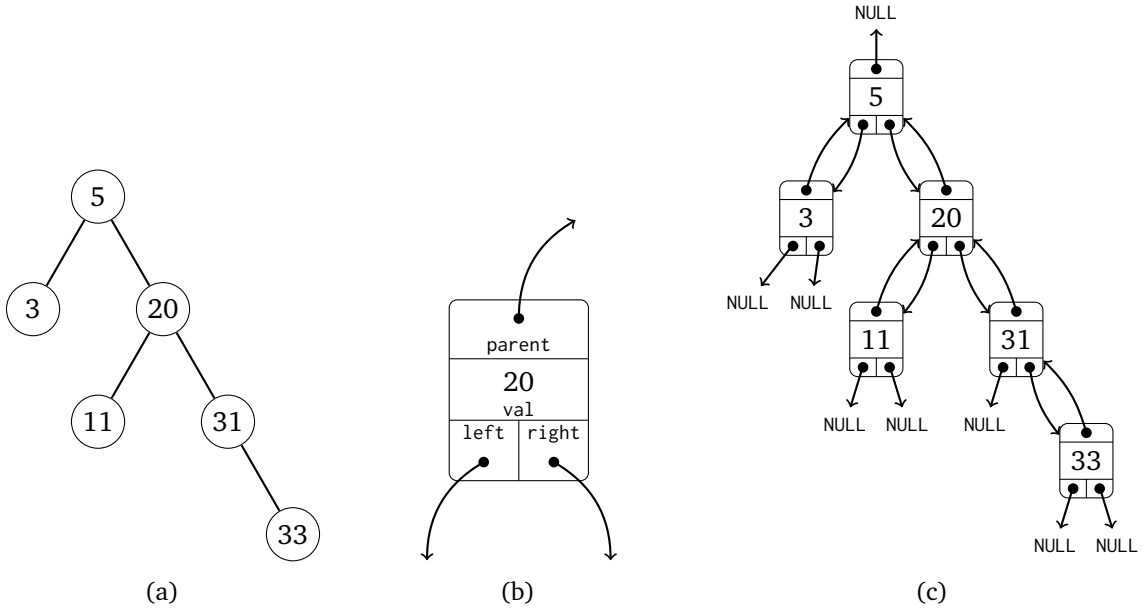
```
struct _tree_t {
    node_t root;
    cmp_t cmp;
    key_cmp_t key_cmp;
};
```

This is part of the tree *implementation* and, along with all the code below, should be added to a file named `tree.c`.

Every tree node stores a record and has a parent, a left child, and a right child (see Figure 1b):

```
struct _node_t {
    void *val;
    node_t parent, left, right;
};
```

For the root of the tree, the parent is `NULL`. For a leaf, the left and right children are `NULL`. This is illustrated in Figure 1c.



(a) (b) (c)

Figure 1: (a) A binary search tree. (b) The representation of a binary search tree node. (c) The representation of the tree in Figure (a).

Step 3: Creating and destroying a tree

Creating an empty tree is easy. This tree has no nodes at all, so even its root is `NULL`. Apart from that, it only needs to store the provided comparators:

```
tree_t make_tree(cmp_t cmp, key_cmp_t key_cmp) {
    tree_t tree = malloc(sizeof(struct _tree_t));
    if (tree) {
        tree->root = NULL;
        tree->cmp = cmp;
        tree->key_cmp = key_cmp;
    }
    return tree;
}
```

Destroying a tree is a bit more challenging: you need to find all the nodes in the tree and destroy them as well. To this end, it is useful to view each node as representing the subtree composed of all its descendants. To destroy this subtree, you need to first destroy the subtrees rooted at the children of the current node and then free the memory used by the current node:

```
void destroy_subtree(node_t node) {
    if (node) {
        destroy_subtree(node->left);
        destroy_subtree(node->right);
        free(node);
    }
}

void destroy_tree(tree_t tree) {
    destroy_subtree(tree->root);
    free(tree);
}
```

Step 4: Insertions and deletions

The key property that allows a binary search tree to be searched efficiently is that it stores the records in sorted order. Specifically, for any node `x` in the tree, the records stored at descendants of `x->left` are no greater than `x->val` and the records stored at descendants of `x->right` are no less than `x->val`. This is illustrated in Figure 1a. When inserting a new record into the tree, you need to ensure that this property is preserved.

Let x be the value to be inserted, let y be the largest value already in the tree that is no greater than x and let z be the smallest value greater than x (y's successor). Then it is not hard to see that (a) y has no right child or z has no left child and (b) making x the right child of y or the left child of z preserves the ordering of the elements in the tree. Figure 2 illustrates both cases. Thus, you only need to find z and, if z has a left child, y and then add x as a child of one of these two nodes. The next procedure does this:

```
node_t tree_insert(tree_t tree, void *new_element) {
    node_t new_node = malloc(sizeof(struct _node_t));
    if (new_node) {
        new_node->val = new_element;
        new_node->left = new_node->right = NULL;
        if (tree->root) {
            node_t node = tree->root;
            while (node) {
                if (tree->cmp(new_element, node->val) < 0) {
                    if (node->left) {
                        node = node->left;
                    } else {
                        node->left = new_node;
                        new_node->parent = node;
                        node = NULL;
                    }
                } else {
                    if (node->right) {
                        node = node->right;
                    } else {
                        node->right = new_node;
                        new_node->parent = node;
                        node = NULL;
                    }
                }
            }
        } else {
            new_node->parent = NULL;
            tree->root = new_node;
        }
    }
    return new_node;
}
```

Figure 2: The two cases of inserting a new item x (blue). (a) x is added after its predecessor in sorted order. (b) x is added before its successor in sorted order.

To delete an element, you do not need to find the node that stores it because the deletion procedure already takes a reference to this node as an argument. However, deletion poses its own challenges. Deleting a leaf is easy: you just free its memory and make sure that its parent's left or right pointer is set to `NULL` depending on whether the deleted node was its left or right child. If the deleted leaf is the last node of the tree, then it has no parent. In this case, the root of the tree has to be set to `NULL`. However, the deleted node is not guaranteed to be a leaf. You have to deal with a number of cases:

- If the node to be deleted does not have a left child, replacing it with its right child gives a valid tree again (see Figure 3a).
- If the node to be deleted does not have a right child, replacing it with its left child gives a valid tree again (see Figure 3b).
- It the node to be deleted has both a left and a right child, the leftmost node in its right subtree by definition does not have a left child. Deleting this leftmost node from the right subtree using the first case above and then replacing the node to be deleted with this leftmost node once again produces a valid tree (see Figure 3c).

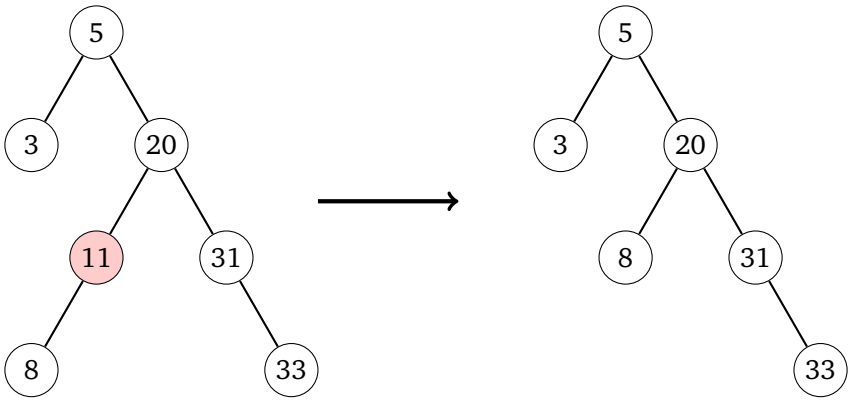
This gives the following code:

```
void *tree_delete(tree_t tree, node_t node) {
    void *val = node->val;
    if (node->left && node->right) {
        node_t repl = node->right;
        while (repl->left) {
            repl = repl->left;
        }
        node_suppress(tree, repl);
        repl->left = node->left;
        if (repl->left) {
            repl->left->parent = repl;
        }
        repl->right = node->right;
        if (repl->right) {
            repl->right->parent = repl;
        }
        node_replace(tree, node, repl);
    } else {
        node_suppress(tree, node);
    }
    free(node);
    return val;
}
```

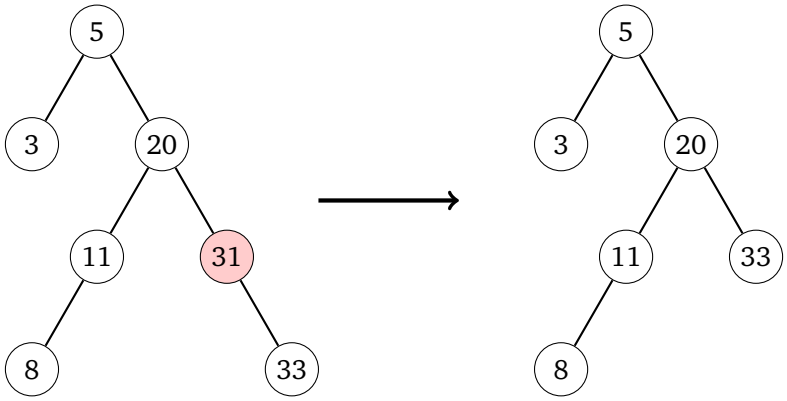
The helper function `node_suppress` removes a node with at most one child from the tree by making its only child a child of its parent (see Figures 3ab):

```
void node_suppress(tree_t tree, node_t node) {
    if (node->left) {
        node_replace(tree, node, node->left);
    } else {
        node_replace(tree, node, node->right);
    }
}
```

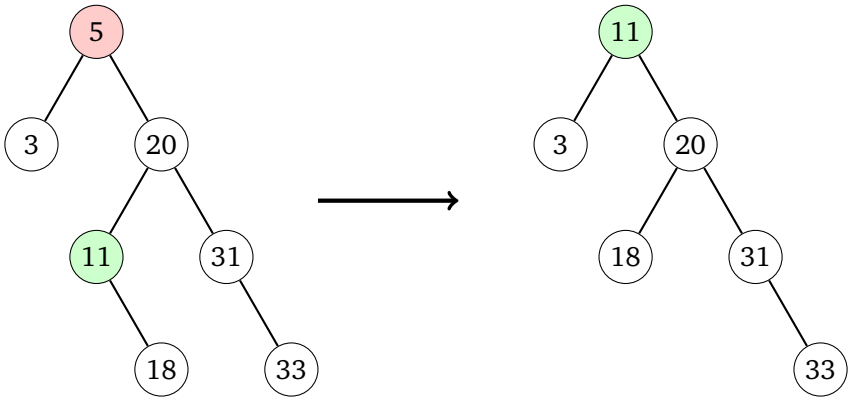
The call `node_replace(node, repl)` used in both `tree_delete` and `node_suppress` replaces `node` with `repl` as the child of `node`'s parent or as the root of the tree if `node` has no parent:



(a)



(b)



(c)

Figure 3: The three cases of deleting a node. The deleted node is shown in red on the left. The resulting tree is shown on the right. (a) The deleted node has no left child. The deleted node is suppressed. (b) The deleted node has no right child. The deleted node is suppressed. (c) The deleted node has two children. The green node on the left is the leftmost node in the deleted node's right subtree and replaces the deleted node in the resulting tree.

```
void node_replace(tree_t tree, node_t node, node_t repl) {
    if (repl) {
        repl->parent = node->parent;
    }
    if (node->parent) {
        if (node->parent->left == node) {
            node->parent->left = repl;
        } else {
            node->parent->right = repl;
        }
    } else {
        tree->root = repl;
    }
}
```

Step 5: Finding a key in the tree and accessing a node's value

The procedure for finding a node whose record's key matches a given search key is very similar to the procedure for looking for the spot where to insert a new value: If the search key matches the key of the current node, this node is the answer. Otherwise, the search has to continue in the left subtree or in the right subtree depending on whether the search key is less than or greater than the key of the current node:

```
node_t tree_find(tree_t tree, void *key) {
    node_t node = tree->root;
    while (node) {
        int dir = tree->key_cmp(key, node->val);
        if (dir == 0) {
            break;
        } else if (dir < 0) {
            node = node->left;
        } else {
            node = node->right;
        }
    }
    return node;
}
```

Accessing the value stored at a node is straightforward:

```
void *node_value(node_t node) {
    return node->val;
}
```

Step 6: Iterating over the values in a tree, in sorted order

In order to iterate over the elements in a tree in order, you first need to visit the nodes in the root's left subtree in order (recursively), followed by the root, followed by the nodes in the root's right subtree. The first visited node is thus the leftmost node of the tree, which is easy to find. The next node after the current node is

- The leftmost node in the current node's right subtree if this subtree is not empty or
- The lowest ancestor of the current node that has the current node in its left subtree otherwise.

This is illustrated in Figure 4. Based on this, an iterator over the elements in the tree can be implemented as a pointer to the current tree node:

```
struct _iter_t {
    node_t node;
};
```

The iterator is initialized by finding the leftmost node in the tree:

```
iter_t make_iterator(tree_t tree) {
    iter_t iter = malloc(sizeof(struct _iter_t));
    if (iter) {
        iter->node = tree->root;
        if (iter->node) {
            while (iter->node->left) {
                iter->node = iter->node->left;
            }
        }
    }
    return iter;
}
```

Destroying an iterator simply frees it:

```
void destroy_iter(iter_t iter) {
    free(iter);
}
```

The `iter_next` function updates `iter->node` to be the successor of `iter->node` using the rules above:

```
void *iter_next(iter_t iter) {
    if (iter->node) {
        void *val = iter->node->val;
        if (iter->node->right) {
            iter->node = iter->node->right;
            while (iter->node->left) {
                iter->node = iter->node->left;
            }
        } else {
            while (iter->node &&
                   (!iter->node->parent ||
                    iter->node == iter->node->parent->right)) {
                iter->node = iter->node->parent;
            }
            if (iter->node) {
                iter->node = iter->node->parent;
            }
        }
        return val;
    } else {
        return NULL;
    }
}
```

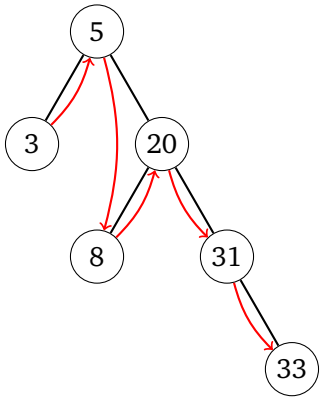


Figure 4: The red arrows indicate the order in which an iterator needs to visit the nodes in the tree. You should verify that this ordering can be found using the rules implemented in `iter_next`.

The code so far

Here is the complete `tree.c` file so far:

tree.c (1/5)

```
#include <stdlib.h>
#include "tree.h"

struct _tree_t {
    node_t root;
    cmp_t cmp;
    key_cmp_t key_cmp;
};

struct _node_t {
    void *val;
    node_t parent, left, right;
};

tree_t make_tree(cmp_t cmp, key_cmp_t key_cmp) {
    tree_t tree = malloc(sizeof(struct _tree_t));
    if (tree) {
        tree->root = NULL;
        tree->cmp = cmp;
        tree->key_cmp = key_cmp;
    }
    return tree;
}

void destroy_subtree(node_t node) {
    if (node) {
        destroy_subtree(node->left);
        destroy_subtree(node->right);
        free(node);
    }
}

void destroy_tree(tree_t tree) {
    destroy_subtree(tree->root);
    free(tree);
}
```

tree.c (2/5)

```
node_t tree_insert(tree_t tree, void *new_element) {
    node_t new_node = malloc(sizeof(struct _node_t));
    if (new_node) {
        new_node->val = new_element;
        new_node->left = new_node->right = NULL;
        if (tree->root) {
            node_t node = tree->root;
            while (node) {
                if (tree->cmp(new_element, node->val) < 0) {
                    if (node->left) {
                        node = node->left;
                    } else {
                        node->left = new_node;
                        new_node->parent = node;
                        node = NULL;
                    }
                } else {
                    if (node->right) {
                        node = node->right;
                    } else {
                        node->right = new_node;
                        new_node->parent = node;
                        node = NULL;
                    }
                }
            }
        } else {
            new_node->parent = NULL;
            tree->root = new_node;
        }
    }
    return new_node;
}
```

tree.c (3/5)

```
void node_replace(tree_t tree, node_t node, node_t repl) {
    if (repl) {
        repl->parent = node->parent;
    }
    if (node->parent) {
        if (node->parent->left == node) {
            node->parent->left = repl;
        } else {
            node->parent->right = repl;
        }
    } else {
        tree->root = repl;
    }
}

void node_suppress(tree_t tree, node_t node) {
    if (node->left) {
        node_replace(tree, node, node->left);
    } else {
        node_replace(tree, node, node->right);
    }
}

void *tree_delete(tree_t tree, node_t node) {
    void *val = node->val;
    if (node->left && node->right) {
        node_t repl = node->right;
        while (repl->left) {
            repl = repl->left;
        }
        node_suppress(tree, repl);
        repl->left = node->left;
        if (repl->left) {
            repl->left->parent = repl;
        }
        repl->right = node->right;
        if (repl->right) {
            repl->right->parent = repl;
        }
        node_replace(tree, node, repl);
    } else {
        node_suppress(tree, node);
    }
    free(node);
    return val;
}
```

tree.c (4/5)

```
node_t tree_find(tree_t tree, void *key) {
    node_t node = tree->root;
    while (node) {
        int dir = tree->key_cmp(key, node->val);
        if (dir == 0) {
            break;
        } else if (dir < 0) {
            node = node->left;
        } else {
            node = node->right;
        }
    }
    return node;
}

void *node_value(node_t node) {
    return node->val;
}

struct _iter_t {
    node_t node;
};

iter_t make_iterator(tree_t tree) {
    iter_t iter = malloc(sizeof(struct _iter_t));
    if (iter) {
        iter->node = tree->root;
        if (iter->node) {
            while (iter->node->left) {
                iter->node = iter->node->left;
            }
        }
    }
    return iter;
}

void destroy_iterator(iter_t iter) {
    free(iter);
}
```

tree.c (5/5)

```
void *iter_next(iter_t iter) {
    if (iter->node) {
        void *val = iter->node->val;
        if (iter->node->right) {
            iter->node = iter->node->right;
            while (iter->node->left) {
                iter->node = iter->node->left;
            }
        } else {
            while (iter->node &&
                   (!iter->node->parent ||
                    iter->node == iter->node->parent->right)) {
                iter->node = iter->node->parent;
            }
            if (iter->node) {
                iter->node = iter->node->parent;
            }
        }
        return val;
    } else {
        return NULL;
    }
}
```

Step 7: A test case

Before adding code to keep the tree balanced, you should test that the current tree implementation works. Search trees are very useful for solving a range of problems efficiently. For example, you can sort by inserting the elements to be sorted into a binary search tree and then iterating over the elements in sorted order. If you keep the tree balanced, this takes $O(n \lg n)$ time just as if you had used Mergesort or Quicksort.

Here, you will use a much more mundane application as an example: You will store a set of strings and support four operations on this set:

- Add a string to the set.
- Remove a string from the set. To do so, you will provide a pattern and remove an arbitrary string in the set that starts with this pattern.
- Find a string. Again, you will report an arbitrary string that starts with the given pattern.
- List the current set of strings *in sorted order*. (This is something that binary search trees support easily. If you used a hashtable to store the set of strings, updates would be faster but reporting the strings in the hashtable in sorted order would involve an additional sorting step.)

Your program will start with the empty set and will print a prompt "? ". At this prompt, you can enter one of five commands:

- `a <string>` to add the given string to the set.
- `d <pattern>` to delete a string starting with the given pattern.
- `f <pattern>` to find a string starting with the given pattern. Both this operation and the delete operation print `STRING NOT FOUND` if there is no match.
- `l` to list all strings in sorted order.
- `q` to quit.

The following code implements this functionality. Three details are worth discussing. First, the `tree.h` header file is included using `#include "tree.h"` instead of `#include <tree.h>`. This difference is significant. The `#include <...>` form searches for the header file to be included in "system locations" determined as part of the installation of the C compiler, by setting the environment variable `C_INCLUDE_PATH` or by providing the `-I` option to `gcc`. The `#include "..."` form also searches in the current working directory and thus finds the `tree.h` file you created earlier.

Second, the two comparison functions of the dictionary are used to implement exactly the functionality you want: The sorting of the strings in the tree is done using `strcmp`, which ensures that the strings are sorted lexicographically. The search for a pattern (the "key" here) is done using a thin wrapper around `strncmp`. `strncmp` is a variant of `strcmp` that takes an additional argument to limit the number of characters used in the string comparison. By setting this argument to the length of the pattern, you ensure that a string that starts with this pattern is deemed to be equal to the pattern during the search while any other string is correctly found to be less than or greater than the pattern.

Third, these comparison functions take `char` pointers as arguments while a `cmp_t` or `key_cmp_t` takes two `void` pointers as arguments. To silence compiler warnings pointing out this mismatch, you cast `strcmp` and `prefix_strcmp` to to `cmp_t` and `key_cmp_t`, respectively.

stringset.c (1/3)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "tree.h"

int prefix_strcmp(const char *key, const char *val) {
    return strncmp(key, val, strlen(key));
}

int main() {
    tree_t tree = make_tree((cmp_t) strcmp, (key_cmp_t) prefix_strcmp);
    if (!tree) {
        printf("ERROR ALLOCATING TREE\n");
        return 1;
    }
    int quit = 0;
    size_t line_cap = 0;
    char *input = NULL;
    while (!quit) {
        printf("? ");
        int num_chars = getline(&input, &line_cap, stdin);
        if (num_chars < 0) {
            quit = 1;
        } else if (num_chars < 2) {
            printf("INVALID INPUT\n");
        } else if (input[0] == 'a') {
            if (num_chars < 4 || input[1] != ' ') {
                printf("INVALID INPUT\n");
            } else {
                size_t len = strlen(input);
                char *string = malloc(len - 2);
                input[len - 1] = 0;
                if (string) {
                    strcpy(string, input + 2);
                    tree_insert(tree, string);
                }
            }
```

stringset.c (2/3)

```
        } else if (input[0] == 'd') {
            if (num_chars < 4 || input[1] != ' ') {
                printf("INVALID INPUT\n");
            } else {
                input[strlen(input) - 1] = 0;
                node_t node = tree_find(tree, input + 2);
                if (node) {
                    free(tree_delete(tree, node));
                } else {
                    printf("STRING NOT FOUND\n");
                }
            }
        } else if (input[0] == 'f') {
            if (num_chars < 4 || input[1] != ' ') {
                printf("INVALID INPUT\n");
            } else {
                input[strlen(input) - 1] = 0;
                node_t node = tree_find(tree, input + 2);
                if (node) {
                    printf("%s\n", (char *) node_value(node));
                } else {
                    printf("STRING NOT FOUND\n");
                }
            }
        } else if (input[0] == 'l') {
            iter_t iter = make_iterator(tree);
            if (iter) {
                char *string;
                while ((string = iter_next(iter))) {
                    printf("%s\n", string);
                }
                destroy_iterator(iter);
            }
        } else if (input[0] == 'q') {
            quit = 1;
        } else {
            printf("INVALID INPUT\n");
        }
    }
```

stringset.c (3/3)

```
    if (input) {
        free(input);
    }
    iter_t iter = make_iterator(tree);
    if (iter) {
        void *string;
        while ((string = iter_next(iter))) {
            free(string);
        }
        destroy_iterator(iter);
    }
    destroy_tree(tree);
    return 0;
}
```

Step 8: Compile and test your work

This is the first example of a program with more than one source file. We will discuss later how to automate the compilation of such multi-file projects and ensure that every time you recompile your code, only the files that have changed since the last compilation are compiled again. Here, you simply compile things manually:

```
$ gcc -c tree.c
$ gcc -c stringset.c
```

The `-c` option runs the given `.c` file through the preprocessor and compiler to produce an object file (`.o`) with the same name but does not perform the linking step.

You should not get any compiler errors from these two steps if you entered the code in the previous steps exactly as listed here. If you do get errors, fix them and recompile until the compilation does not produce any errors.

To obtain an executable, you need to link the two object files produced by the above compilation steps. You do this by running `gcc` as you have done so far, only you provide object files rather than C files as arguments and you provide *multiple* object files:

```
$ gcc -o stringset stringset.o tree.o
```

Again, this should not give any errors. You can now run your code using

```
$ ./stringset
```

Congratulations! You made it to the first milestone in this lab and have succeeded in implementing a non-trivial pointer-based data structure. Celebrate by adding and deleting some strings, searching for strings, and listing all strings currently in the tree. Verify that the find and list operations produce the ouput you'd expect.

Step 9: The basic tool for rebalancing

The icing on the cake will be that you keep the tree balanced to ensure searches are fast. (The cost of traversing the tree to list all items it contains is always linear no matter the shape of the tree.) The tool used by all balanced binary search trees to limit the height of the tree is rotations. If the left subtree of a given node is very deep compared to the right subtree, then it is helpful to restructure the tree so the left child of x becomes the new root of the subtree. This is called a *right rotation*. If the right subtree is deeper, a *left rotation* restructures the tree in the opposite direction. These two operations are inverses of each other and are illustrated in Figure 5.

By referring to the node that must become the new root of the subtree, they both simply become rotations. The direction of the rotation is determined by whether the referenced node is the left child or right child of its parent:

```
void left_rotate(tree_t tree, node_t node) {
    node_t parent = node->parent;
    node_replace(tree, parent, node);
    parent->right = node->left;
    if (node->left) {
        node->left->parent = parent;
    }
    node->left = parent;
    parent->parent = node;
}

void right_rotate(tree_t tree, node_t node) {
    node_t parent = node->parent;
    node_replace(tree, parent, node);
    parent->left = node->right;
    if (node->right) {
        node->right->parent = parent;
    }
    node->right = parent;
    parent->parent = node;
}

void rotate(tree_t tree, node_t node) {
    if (node == node->parent->left) {
        right_rotate(tree, node);
    } else {
        left_rotate(tree, node);
    }
}
```

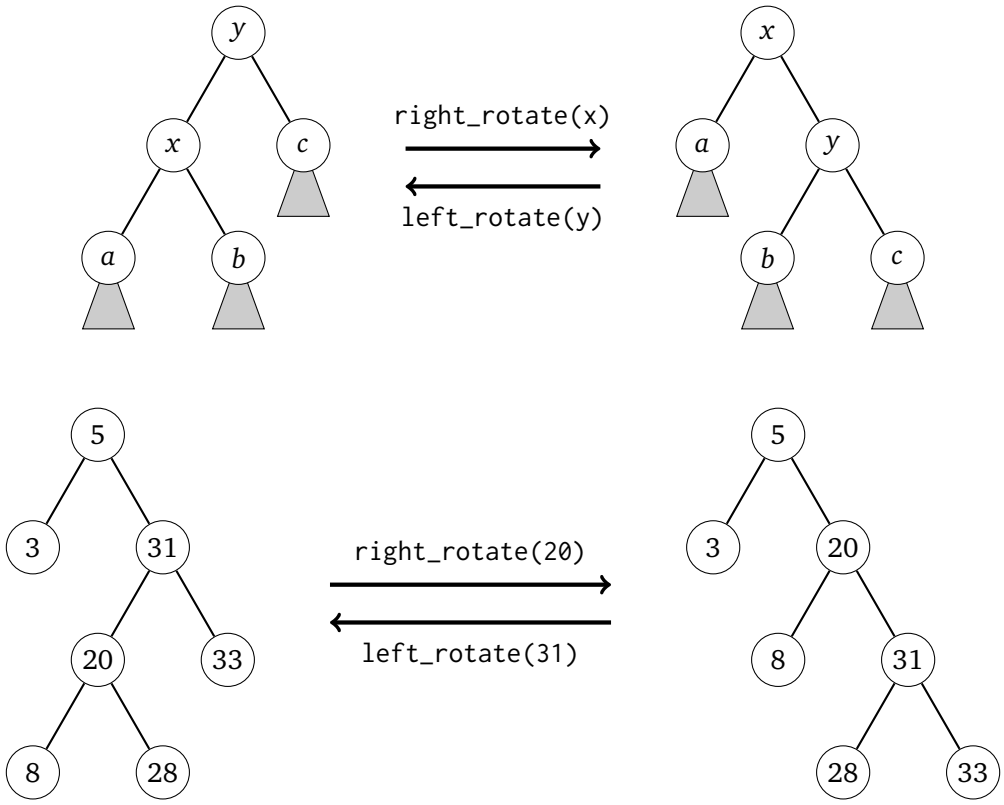


Figure 5: Rotations. The top part of the figure shows left and right rotations abstractly. The bottom part of the figure illustrates these two operations using a concrete example.

Step 10: Splaying

The operation used by a splay tree to maintain balance is splaying. Splaying a node means that the tree is restructured using rotations so the node becomes the root of the tree (see Figure 6).

Splaying is implemented using three basic operations:

Zig: If the node to be splayed is a child of the root, a single rotation makes it the root of the tree (see the last step in Figure 6).

Zig-zig: If the node to be splayed and its parent are both left children of their parents or both right children of their parents, you first rotate the parent and then the node to move the node two levels up the tree (see the second step in Figure 6).

Zig-zag: If the node to be splayed is the left child of its parent and the parent is the right child of its parent or vice versa, you rotate the node twice to move it two levels up the tree (see the first step in Figure 6).

A splay operation applies zig-zig and zig-zag operations to move the node to be splayed up the tree until it is either the root or a child of the root. If it is the child of the root, you finish with a single zig operation. The code of the splay operation looks as follows:

```
void splay(tree_t tree, node_t node) {
    while (node->parent) {
        if (node->parent->parent) {
            if ((node->parent == node->parent->parent->left &&
                 node == node->parent->left) ||
                (node->parent == node->parent->parent->right &&
                 node == node->parent->right)) {
                rotate(tree, node->parent);
            } else {
                rotate(tree, node);
            }
        }
        rotate(tree, node);
    }
}
```

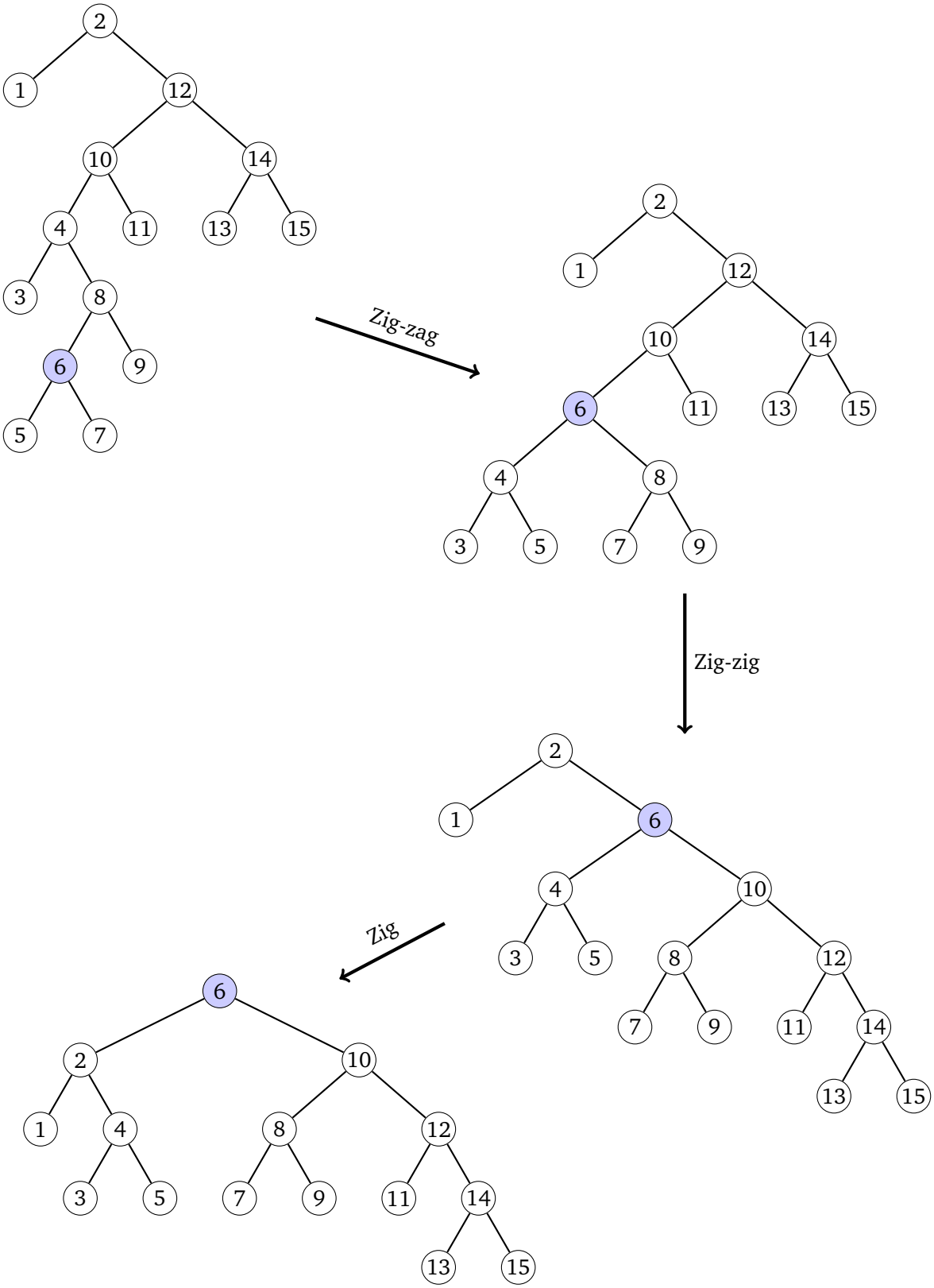


Figure 6: An example of a splay operation that involves a zig-zag, a zig-zig, and a zig operation.

Step 11: Splaying after basic operations

The final piece in the puzzle is to invoke the splay operation on the appropriate nodes as part of every insertion, deletion or search:

- After a search operation, the node storing the sought record (or, if no such node exists, the last node visited by the search) is splayed.
- After an insertion, the new leaf is splayed.
- After a deletion, the parent of the removed node is splayed. The notion of "removed node" depends on the situation that applies to the delete operation. If the node to be deleted has at most one child, its parent is the node that is splayed. If the node to be deleted has two children, it cannot be removed directly. Instead, it is replaced with the leftmost node in its right subtree. In this case, this leftmost node in the right subtree is the one that is considered to be removed as far as splaying goes, that is, the parent of this replacement node is splayed.

Modifying search and insert operations to add these splay operations is fairy straightforward:

```
node_t tree_find(tree_t tree, void *key) {
    node_t node = tree->root;
    if (!node) {
        return NULL;
    }
    for (;;) {
        int dir = tree->key_cmp(key, node->val);
        if (dir == 0) {
            splay(tree, node);
            return node;
        } else if (dir < 0) {
            if (node->left) {
                node = node->left;
            } else {
                splay(tree, node);
                return NULL;
            }
        } else {
            if (node->right) {
                node = node->right;
            } else {
                splay(tree, node);
                return NULL;
            }
        }
    }
}
```

```
node_t tree_insert(tree_t tree, void *new_element) {
    node_t new_node = malloc(sizeof(struct _node_t));
    if (new_node) {
        new_node->val = new_element;
        new_node->left = new_node->right = NULL;
        if (tree->root) {
            node_t node = tree->root;
            while (node) {
                if (tree->cmp(new_element, node->val) < 0) {
                    if (node->left) {
                        node = node->left;
                    } else {
                        node->left = new_node;
                        new_node->parent = node;
                        node = NULL;
                    }
                } else {
                    if (node->right) {
                        node = node->right;
                    } else {
                        node->right = new_node;
                        new_node->parent = node;
                        node = NULL;
                    }
                }
            }
        } else {
            new_node->parent = NULL;
            tree->root = new_node;
        }
    }
    splay(tree, new_node);
    return new_node;
}
```

The delete operation is complicated slightly by the fact that you need to decide whether to splay the parent of the deleted node or the parent of the leftmost node in its right subtree:

```
void *tree_delete(tree_t tree, node_t node) {
    void *val = node->val;
    node_t splay_node;
    if (node->left && node->right) {
        node_t repl = node->right;
        while (repl->left) {
            repl = repl->left;
        }
        if (repl->parent == node) {
            splay_node = repl;
        } else {
            splay_node = repl->parent;
        }
        node_suppress(tree, repl);
        repl->left = node->left;
        if (repl->left) {
            repl->left->parent = repl;
        }
        repl->right = node->right;
        if (repl->right) {
            repl->right->parent = repl;
        }
        node_replace(tree, node, repl);
    } else {
        splay_node = node->parent;
        node_suppress(tree, node);
    }
    if (splay_node) {
        splay(tree, splay_node);
    }
    free(node);
    return val;
}
```

The final code

This is the final version of the `tree.c` file:

tree.c (1/7)

```
#include <stdlib.h>
#include "tree.h"

struct _tree_t {
    node_t root;
    cmp_t cmp;
    key_cmp_t key_cmp;
};

struct _node_t {
    void *val;
    node_t parent, left, right;
};

tree_t make_tree(cmp_t cmp, key_cmp_t key_cmp) {
    tree_t tree = malloc(sizeof(struct _tree_t));
    if (tree) {
        tree->root = NULL;
        tree->cmp = cmp;
        tree->key_cmp = key_cmp;
    }
    return tree;
}

void destroy_subtree(node_t node) {
    if (node) {
        destroy_subtree(node->left);
        destroy_subtree(node->right);
        free(node);
    }
}

void destroy_tree(tree_t tree) {
    destroy_subtree(tree->root);
    free(tree);
}
```

tree.c (2/7)

```
void node_replace(tree_t tree, node_t node, node_t repl) {
    if (repl) {
        repl->parent = node->parent;
    }
    if (node->parent) {
        if (node->parent->left == node) {
            node->parent->left = repl;
        } else {
            node->parent->right = repl;
        }
    } else {
        tree->root = repl;
    }
}

void node_suppress(tree_t tree, node_t node) {
    if (node->left) {
        node_replace(tree, node, node->left);
    } else {
        node_replace(tree, node, node->right);
    }
}

void left_rotate(tree_t tree, node_t node) {
    node_t parent = node->parent;
    node_replace(tree, parent, node);
    parent->right = node->left;
    if (node->left) {
        node->left->parent = parent;
    }
    node->left = parent;
    parent->parent = node;
}

void right_rotate(tree_t tree, node_t node) {
    node_t parent = node->parent;
    node_replace(tree, parent, node);
    parent->left = node->right;
    if (node->right) {
        node->right->parent = parent;
    }
    node->right = parent;
    parent->parent = node;
}
```

tree.c (3/7)

```
void rotate(tree_t tree, node_t node) {
    if (node == node->parent->left) {
        right_rotate(tree, node);
    } else {
        left_rotate(tree, node);
    }
}

void splay(tree_t tree, node_t node) {
    while (node->parent) {
        if (node->parent->parent) {
            if ((node->parent == node->parent->parent->left &&
                 node == node->parent->left) ||
                (node->parent == node->parent->parent->right &&
                 node == node->parent->right)) {
                rotate(tree, node->parent);
            } else {
                rotate(tree, node);
            }
        }
        rotate(tree, node);
    }
}
```

tree.c (4/7)

```
node_t tree_insert(tree_t tree, void *new_element) {
    node_t new_node = malloc(sizeof(struct _node_t));
    if (new_node) {
        new_node->val = new_element;
        new_node->left = new_node->right = NULL;
        if (tree->root) {
            node_t node = tree->root;
            while (node) {
                if (tree->cmp(new_element, node->val) < 0) {
                    if (node->left) {
                        node = node->left;
                    } else {
                        node->left = new_node;
                        new_node->parent = node;
                        node = NULL;
                    }
                } else {
                    if (node->right) {
                        node = node->right;
                    } else {
                        node->right = new_node;
                        new_node->parent = node;
                        node = NULL;
                    }
                }
            }
        } else {
            new_node->parent = NULL;
            tree->root = new_node;
        }
    }
    splay(tree, new_node);
    return new_node;
}
```

tree.c (5/7)

```
void *tree_delete(tree_t tree, node_t node) {
    void *val = node->val;
    node_t splay_node;
    if (node->left && node->right) {
        node_t repl = node->right;
        while (repl->left) {
            repl = repl->left;
        }
        if (repl->parent == node) {
            splay_node = repl;
        } else {
            splay_node = repl->parent;
        }
        node_suppress(tree, repl);
        repl->left = node->left;
        if (repl->left) {
            repl->left->parent = repl;
        }
        repl->right = node->right;
        if (repl->right) {
            repl->right->parent = repl;
        }
        node_replace(tree, node, repl);
    } else {
        splay_node = node->parent;
        node_suppress(tree, node);
    }
    if (splay_node) {
        splay(tree, splay_node);
    }
    free(node);
    return val;
}
```

tree.c (6/7)

```
node_t tree_find(tree_t tree, void *key) {
    node_t node = tree->root;
    if (!node) {
        return null;
    }
    for (;;) {
        int dir = tree->key_cmp(key, node->val);
        if (dir == 0) {
            splay(tree, node);
            return node;
        } else if (dir < 0) {
            if (node->left) {
                node = node->left;
            } else {
                splay(tree, node);
                return NULL;
            }
        } else {
            if (node->right) {
                node = node->right;
            } else {
                splay(tree, node);
                return NULL;
            }
        }
    }
}

void *node_value(node_t node) {
    return node->val;
}
```

tree.c (7/7)

```
struct _iter_t {
    node_t node;
};

iter_t make_iterator(tree_t tree) {
    iter_t iter = malloc(sizeof(struct _iter_t));
    if (iter) {
        iter->node = tree->root;
        if (iter->node) {
            while (iter->node->left) {
                iter->node = iter->node->left;
            }
        }
    }
    return iter;
}

void destroy_iterator(iter_t iter) {
    free(iter);
}

void *iter_next(iter_t iter) {
    if (iter->node) {
        void *val = iter->node->val;
        if (iter->node->right) {
            iter->node = iter->node->right;
            while (iter->node->left) {
                iter->node = iter->node->left;
            }
        } else {
            while (iter->node &&
                   (!iter->node->parent ||
                    iter->node == iter->node->parent->right)) {
                iter->node = iter->node->parent;
            }
            if (iter->node) {
                iter->node = iter->node->parent;
            }
        }
        return val;
    } else {
        return NULL;
    }
}
```

Step 12: Test the final version of your code

Recompile you `test.c` file

```
$ gcc -c test.c
```

and link it with the `stringset.o` file

```
$ gcc -o stringset stringset.o tree.o
```

There is no need to recompile `stringset.c` because its code has not changed. Again, if the compiler reports errors, fix them by checking how your code deviates from the code in these notes. Then experiment with the program by adding, deleting, and searching for different strings and listing all strings in the string set.

Step 13: Commit your work

Commit your work to SVN. The files you should commit are

```
lab8
├─ tree.h
├─ tree.c
└─ stringset.c
```