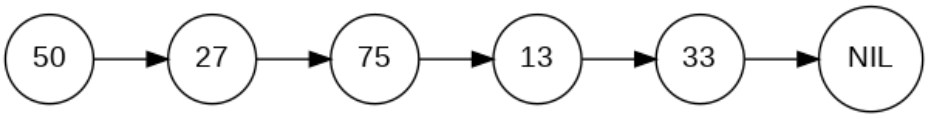
Programming Workshop Split Winter 2018

Problem Space

You are in charge of a comparative study on sponges. In this study, you will be collecting samples and analyzing the unique proteins of each sample. Every time a new protein is found, you will need to add it to your knowledge base. To make things easier, let us assume that the proteins are enumerated (i.e.- each unique protein is assigned a value $i \in \mathbb{Z}$).

The easiest solution, of course, is to keep a list of encountered unique proteins. Every protein that you encounter, p_i, will have to be checked against this list. If protein p_i is in the list, you do not have any new data to collect. However, if p_i is not in the list you will need to add it.

The list data structure is represented in memory as a linear chain of nodes. Structurally, every node in the chain will contain a value as well as the location of the next element. After reading in a few samples, our list may look like:



Notice the inductive nature of the data type. If we were to characterize this

mathematically, we could write

$$L_a = 1 + a \times L_a$$

with L_a being the list that contains elements $x \in a$. The list node that subsumes all other sublists is known as the *head* of the list. The remainder of the list is known as the *tail*.

This solution works well for the first few thousand samples, but then it gets slow. Painfully slow. Why is this happening to you?

Question 1a What is the time complexity of searching the list?

Solution Since we only have access to the immediate predecessor for any given node, we must traverse the list link by link. Assuming that the elements in the list are distributed uniformly, the expected distance to travel to find an element is half of the list, or $O(\frac{n}{2})$. However, as we are concerned with asymptotic complexity, we remove all constants and non-dominating factors. The final solution is $O(n)$.

Question 1b What is the time complexity of inserting at the head of the list?

Solution In this case, we can simply create a new node, and use the old list as this node's tail. This takes constant, or $O(1)$, time.

Question 1c What is the time complexity of inserting at the tail of the list?

Solution Notice that in this case we must traverse the whole list to access the last element. This takes $O(n)$ time.

Acknowledge Shortcomings

Upon closer analysis, you start to grasp part of your algorithms limitations. If your knowledge base was a dictionary, you would never flip through it page by

page to find your word! Instead, you would open to a page somewhere in the middle of the book, and decide if you need to search before or after this page, and repeat as necessary. This search technique is made possible because the words in the dictionary are *sorted*.

Thinking that you will be clever, you decide to sort the knowledge base list.

Question 2a What effect will this have on your solution? Why?

Solution Sorting the list will actually make operations slightly slower! This is because we will be maintaining a sorted list, but will not get the benefits of accessing from the middle of the list. We must still enter the list from the head, and traverse through the list links.

Question 2b What is the time complexity of searching the sorted list?

Solution Nothing has changed from the case of question 1a. This operation will still take $O(n)$ time.

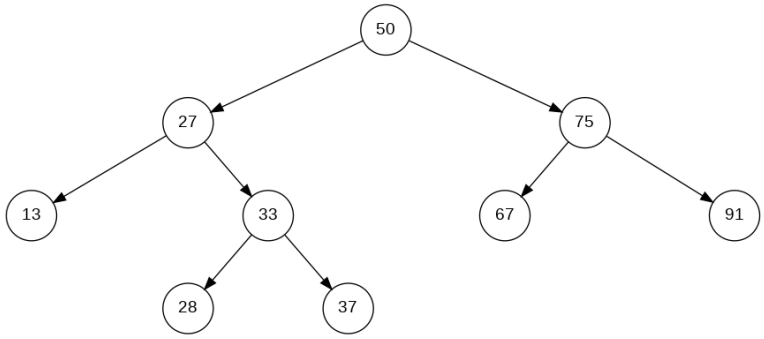
Question 2c What is the time complexity of inserting into the sorted list?

Solution Instead of inserting at the head or tail of the list, we must now find the correct location to insert and element as to preserve the order of the list. Assuming that the element to insert has been chosen from a uniform distribution, the expected number of operations is $O(\frac{n}{2})$. Removing constants, this takes $O(n)$ time.

Making Improvements

Instead of a 1-dimensional list structure, lets lift our data structure into 2-dimensions. For every protein, p_i, present in out knowledge base, we will also keep track of the proteins lexically less than p_i, and the proteins that are lexically greater than p_i. This type of structure is known as a *binary search tree* (which

will be referred to as a BST in the rest of the lab). After reading in a few samples, our BST may look like this:



Notice the inductive nature of the data type. If we were to characterize this mathematically, we could write

$$T_a = 1 + a \times (T_a)^2$$

with T_a being the BST that contains elements $x \in a$. The BST node that subsumes all other subtrees is known as the *root* of the tree. Subtrees that have no child subtrees are called *terminals*, and internal BST nodes that have child subtrees are known as *non-terminals*. A *balanced tree* is a BST that has the property that all paths from the root node to terminal nodes can differ in length by at most 1.

Question 3a What is the worst case time complexity of searching an binary tree?

Solution The longest path, and therefore worst case, will always be from the root of the tree to a terminal node. The pathological case that maximizes the ratio of longest path to total number of nodes is when we get no benefit from splitting the subtrees. In this case we essentially have a list, and the worst case search time is $O(n)$.

Question 3b What is the worst case time complexity of searching a balanced binary tree?

Solution In a balanced tree, all paths from the root to terminal nodes can differ by at most 1. As in question 3a, the worst case is when we travel the whole *height* of the tree. In order to determine this *height* in terms of the number of nodes that are in the tree, observe the following sequence:

- A tree of height 1 can have 1 element
- A tree of height 2 can have 3 elements
- A tree of height 3 can have 7 elements
- A tree of height 4 can have 15 elements
- ...

In more concrete terms, we say that a tree of height h can have $2^h - 1$ elements. With some minor algebraic manipulation, we can determine what the height of a balanced binary tree should be for a given size n.

$$\begin{aligned} 2^h - 1 &= n \\ 2^h &= n + 1 \\ h &= log_2(n + 1) \end{aligned}$$

After removing constants and non-dominating factors, we obtain the final result of $h \approx log(n)$. Therefore, the worst case search time for a balanced BST is $O(log(n))$.

Question 3c What is the worst case time complexity of inserting into a binary tree? A balanced binary tree?

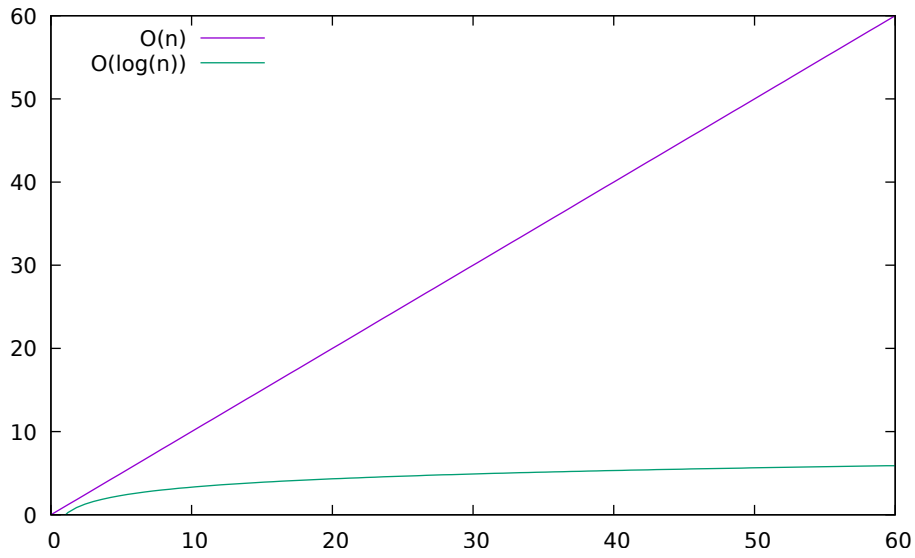
Solution To insert a node x into a BST, we can follow a path from the root to the first empty subtree, and then insert to this location. In order to preserve the properties of a BST, where smaller nodes are to the left and larger nodes are to the right, we need to be smart how we traverse the tree. Anytime x is smaller than the node we are comparing, traverse left. Anytime x is greater than the node we are comparing, go right. In the case that both are equal, do nothing as the value is already in the BST.

Just like questions 3a and 3b, the worst case running time is when the entire height of the tree must be traversed. When the tree shares structural similarity to a list, the running time degrades to $O(n)$. When the tree is perfectly balanced, the running time is $O(log(n))$.

Measuring Improvement

You may be thinking to yourself that improving the asymptotic bounds from $O(n)$ to $O(log(n))$ is not that big of a deal; so what if it is a log function now? Trust me, it is a big deal!

Lets say that you want to find the last element in a collection that has $1e6$ members. A list would need 1 million steps, whereas a balanced BST would only need $log_2(1e6) \approx 20$ steps! Below is a plot comparing the running times of these two structures as input size increases.



As you can see, having a time complexity of $O(log(n))$ is much more desirable than $O(n)$! We should be using a BST everywhere, right? This is not the case

in practice however. Consider what happens to a tree if the data that is being read into the BST is already sorted. We would build a BST whose structure is isomorphic to a list! The time complexity of many operations would degrade to $O(n)$, and we would not gain any benefit for using the more complex BST structure over a simple linked list.

For this reason, numerous tree data structures have been created to *provably ensure* that the BST structure remains balanced, and the $O(log(n))$ time complexity is guaranteed. Some notable examples of these structures include *Red-Black Tree*, *AVL Tree*, and *AA Tree*. You should do further reading on these trees if you have an interest in data structure design.

Implementation

Now that we have the theory out of the way, lets build our own trees! For this section of the lab it would be best to split into groups of two or three, and each group needs to have at least one computer with Python installed.

Instead of just giving you the solutions in Python, we will work through each requirement in both C and Haskell to give you an intuition into how the problems could be solved. The C language operates at a much lower level of abstraction than Python, and requires explicit memory management and has ungaurded pointers. Despite the differences, imperative and mutable structure algorithms are similar to how they would be implemented in Python. Unlike C, Haskell is a language that operates at a much higher level of abstraction, and contains many features that are not supported in Python. Haskell can still serve as a valuable model to demonstrate declarative and immutable structure algorithms.

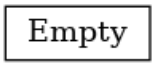
Your end goal for this lab is to produce an equivalent Python program. Full solutions can be seen in the git repository for C and Haskell. After the lab session, the full Python solution will be posted as well. The C code will use a tree fixed to

the Integer type and the Haskell code will be fully polymorphic. Your solution can be implemented using either polymorphism or using a fixed Integer type.

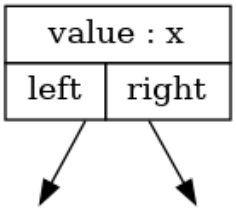
The example code for C and Haskell are introduced without significant explanations of syntax, or general philosophy of programming paradigms. If you are interested in the content, have a question on what is being presented, or just want to learn more about a topic, **please ask!!** This lab is extremely flexible and should be catered to what *you* want to learn.

Structure

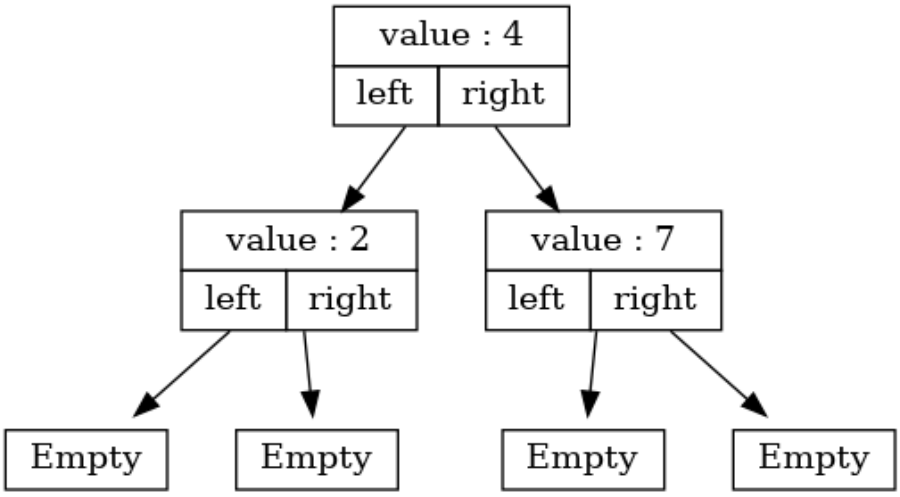
The very first thing that we will need to do is define the data type that we will be operating on. Remember back to the mathematical definition of a BST: $T_a = 1 + (T_a)^2$. The first step to make this type is to distinguish the possible *shapes* the BST type can take by splitting on +. Visually, a BST could look like:



for the 1 (also called *unit*) side of the equation, or it could look like:



for the product side of the equation. These components are chained together to build the complete BST data structure. An example of such a structure can be seen below:



C Language

In C we do not readily have access to means of creating algebraic data types, so we will need to be a bit creative. The C type of `NULL` is inhabited by a single element, meaning that this is a unit type. We will use this to represent our BST unit type of the empty tree.

Product types in C are created using something called `struct`. The language natively supports the representation of Integers, but does not have built in support for declaring and using custom data types. Instead, we must keep track of actual memory location addresses, called pointers in computer science vocabulary. The block of code declaring this structure is written as:

```
struct node {
    int key;
    struct node* left;
    struct node* right;
};
```

The syntactic form `struct xxx { ... };` declares a new product type that is

labeled as *xxx*. The body of the structure, or interior of the curly braces, contains the individual elements of the product. In our case, we have a key of type integer, the left subtree, and the right subtree. As stated previously, these subtrees are actually pointers in the structure, which can be seen from the `*` after the structure label. The `struct node*` type means that this is a memory address to a `node` structure. Because pointers point to raw memory locations, we have to be careful about where we access and need to manually allocate and free memory blocks.

To help us create a node in the BST, we make an auxiliary function that takes in an Integer, allocates the appropriate memory, and returns a node structure. The function `malloc` allocates memory for the structure, and `X->Y` syntax is getting the location of projection `Y` of structure `X`. The keyword `return` exits out of a function call, returning whatever value occurs immediately after itself.

```
struct node* makeNode(int item) {
    struct node* temp = (struct node*) malloc(sizeof(struct node));
    temp->key   = item;
    temp->left  = NULL;
    temp->right = NULL;
    return temp;
}
```

Haskell

The Haskell language directly supports algebraic data types, and can be declared with the `data` keyword. Below is the algebraic data type of a BST.

```
data BST a = E | T a (BST a) (BST a)
```

Here we can see that the `BST` type is parameterized by some type `a`. This data structure is polymorphic, and the `a` serves the same purpose as a variable in mathematics; it is just a placeholder for a concrete type. The `|` is Haskell's way

for declaring sum types, with the first lexeme of either side of the sum serving as labels to identify which constructor to use. In our case, we have `E` for the empty BST label, and `T` for the nonempty node label. Product types are implicit, so the `T` constructor can be read as the product of some type `a`, the left BST, and the right BST.

Display

Once we have a way to represent the BST data structure, we need a way to display it as a String. Because of the inductive nature of the data type, we will use a recursive function to display the tree. The simplest way to display a BST tree in a manner that remains simple to verify the required invariants is to display the BST in order from least to greatest. This is also known as an *in-order traversal*.

The algorithm for printing a tree can be outlined as follows, starting with the root of the tree as the initial element:

- If the left subtree is not empty, print it.
- Print the value of the node.
- If the right subtree is not empty, print it.

This is sounds overwhelming vague, so lets look at some code!

C language

In C we declare functions using the following syntax:

```
<return> <name> (<args>) { ... }
```

where `<return>` is the type returned by the function, `<name>` is the label of the function, `<args>` is a comma separated list of *type-variable name* pairs, and

everything inside of the curly braces is the function body. An example function

```
int f(int x, int y) { return x / y; }
```

could be written in more mathematical terms as

$f : (x \in \mathbb{Z}) \times (y \in \mathbb{Z}) \to \mathbb{Z} \cup \mathbb{Z}_{\perp}$

The final $\mathbb{Z}_{\perp}$ represents the error state of the function. We need this in the mathematical definition since C cannot prove that a function will always return without hitting an error state. Our example function f will evaluate to this state when $y = 0$.

Now that we have the basics down, lets write our function to display the BST. This function will simply display a tree, so there will not be a return type. C has a keyword `void` to represent a function that does not have a return. The argument to the function will be an address to a location of a BST, or more explicitly, it will be of type `struct node*`. This is enough information to write the signature of the C function.

Next we need to fill in the body following our outline of the algorithm. Just as it was described earlier, we we be using recursion in this function. First, we check to make sure that the tree (or subtree) is not empty. If this tree is not empty, then we need to print the values smaller than the current node, then print the current node, and finally print the values larger than the current node. This can be a bit difficult to grasp at first. Try tracing through a call of this function on paper.

```
void show(struct node* root) {
    if (root != NULL) {
        show(root->left);
        printf("%d ", root->key);
        show(root->right);
    }
```

```
}
```

Haskell

Haskell is a purely functional language, and does its best to resemble mathematics. Function signatures are defined with

```
<name> :: <type_1> -> <type_1> -> ... -> <type_n>
```

Our previous division example could be written as

```
f :: Integer -> Integer -> Integer
```

or more safely as

```
f :: Integer -> Integer -> Maybe Integer
```

The language also supports pattern matching on structures using `case` statements. This will be one of the primary control structures, along with conditional branching and recursion. Lastly, Haskell handles the overloading of operators using *type classes*, which are defined using the `class` keyword and instantiated with the `instance` keyword.

Ok, time to write our display function! In Haskell there is a type class called *Show* that ensures the function `show : a -> String` exists for a given type *a*. Lets make an instance of *Show* for our BST. Fist we will split up the possible cases of our data structure with the `case` keyword. For the case of the empty tree, `E`, we print the empty string. For the case of a tree node, `T`, we will print all of the values smaller than the current node, print the current node, and then print all of the values larger than the present node. The `++` operator defines catenation over strings (and is also a *Semigroup* for Strings, but that is outside the scope of the lab).

```
instance Show a => Show (BST a) where
```

```
show t = case t of
  E       -> ""
  T k l r -> show l ++ " " ++ show k ++ " " ++ show r
```

Try tracing through this display function for an arbitrary test input. Discuss with your lab group both the similarities and differences between the C and Haskell way of displaying trees.

Lookup

Because of the invariants imposed on BSTs, we can find elements in the tree efficiently using recursion. When searching for value x in tree T, we simply follow the procedure:

- If the current node T is empty, the value x does not occur in the tree.
- If x is equal to the value of the current node T, our value was found in the tree.
- If x is less than the value of the current node T, our value must be within the left subtree (if it exists).
- If x is greater than the value of the current node T, our value must be within the right subtree (if it exists).

C language

Finding values in the C implementation is simple, but since C does not include boolean values by default we must use 0 and 1 to represent `False` and `True` respectively.

```
int find(int key, struct node* root) {
    if (root == NULL) return 0;
```

```
    if (root->key == key) return 1;

    if (key < root->key) {
        find(key, root->left);
    } else {
        find(key, root->right);
    }
}
```

Haskell

The Haskell function that we will use for `find` includes some basic error checking. Notice that the return type is `Maybe a` and not `a`. With this we are telling the language that this function may not return a value; it could either return `Nothing` or a value of `Just a`. In other words, $Maybe_a = 1 + a$.

```
find :: Ord a => a -> BST a -> Maybe a
find x t = case t of
  E        -> Nothing
  T k l r -> case compare x k of
               LT -> find x l
               EQ -> Just k
               GT -> find x r
```

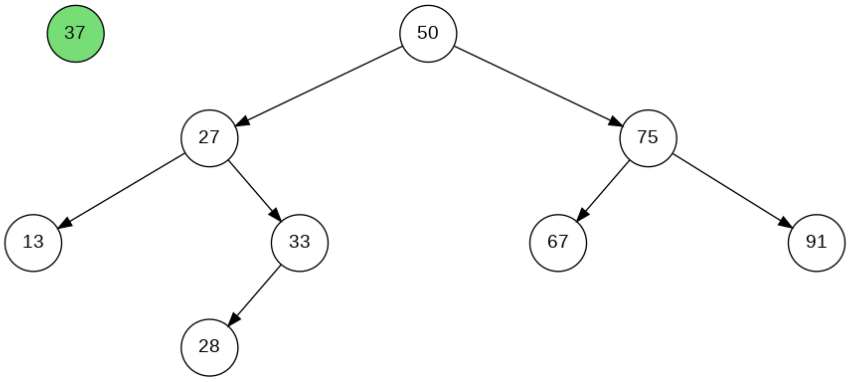
Insert

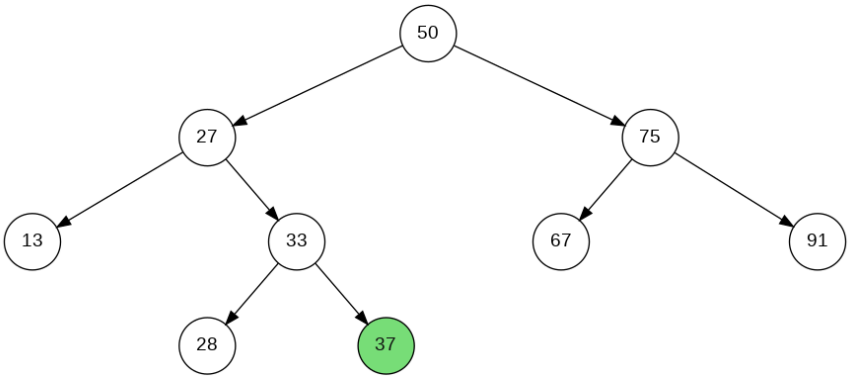
Inserting into a BST is a lot like the story of Goldilocks trying to find the perfect bowl of porridge. We will be looking for just the right terminal node to place the new value; it can't be too big, it can't be too small, it must be just right!

The general idea is to do comparisons to find the correct path from the root node to a terminal node that satisfies the invariants of a BST. We will choose this path by doing comparisons between the value to insert, x, and the value of the current node in the path, y. We will adhere to the following procedure:

- If $x < y$, we travel along the left subtree.
- If $x > y$ we travel along the right subtree.
- If $x == y$, we don't have to do anything, since the value is already in the tree.
- Repeat the above steps until an empty node is reached. This is the correct location to insert, and we create a new node at this location for the value x.

The following tree diagrams illustrate the structure of our example tree before and after the value of 37 is inserted.





C language

The C implementation for tree insertion follows our outline closely. The case for an empty tree is identical to our description. However, the traversal portion for a non-empty tree must be slightly modified. In addition to traversing down the left or right subtrees, we must also *update* these trees to point to the modified subtrees *after* the insertion has been performed. This can be seen where we set the left or right subtree pointers to whatever will be returned on the subsequent iterations down the BST structure.

```
struct node* insert(int key, struct node* root) {
    if (root == NULL) return makeNode(key);
    if (key < root->key) {
        root->left = insert(key, root->left);
    } else {
        root->right = insert(key, root->right);
    }
    return root;
}
```

Haskell

The Haskell solution nearly follows the same steps as the C solution, but instead of mutating the left and right subtree pointers, we will create new tree instances. There is also another subtle difference: this version of the BST is polymorphic, but it cannot work on *all* types. In order to be contained in a BST, data needs to possess a natural ordering (meaning `<`, `>`, and `==` are well defined). In Haskell this can be ensured by forcing our carrier type a to be a member of the `Ord` type class.

This can be seen in the function signature for `insert`.

```
insert :: Ord a => a -> BST a -> BST a
insert x t = case t of
  E       -> T x E E
  T k l r -> case compare x k of
               LT -> T k (insert x l) r
               EQ -> t
               GT -> T k l (insert x r)
```

Delete

Removing entries from a BST is a bit more involved than we have seen for the previously defined operations. Navigating to the node that we want to delete follows the same procedure as the `find` and `insert` operations, so we will omit an explanation for it here. Once the node to delete is found, there are three cases we must consider:

1. The node has no child elements - In this case we can simply delete the node.

2. The node has a single child element - In this case we will simply replace the deleted node with the child node. In this way, all invariants of the BST are preserved.

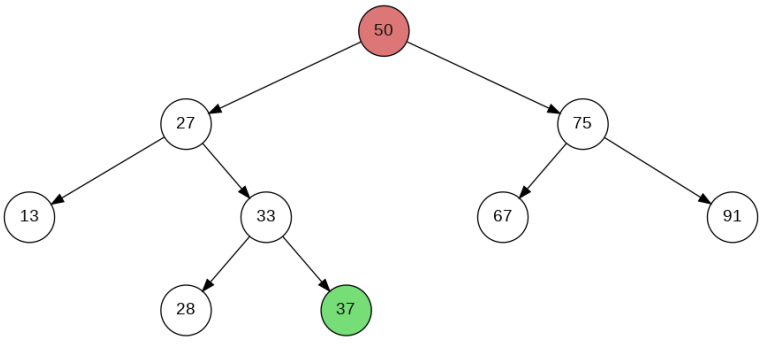
3. The node has 2 child elements - This is the difficult case. Unlike in case 2, we cannot simply replace the deleted node with one of the child nodes; we would be left with a dangling subtree! The most efficient course of action would be to:

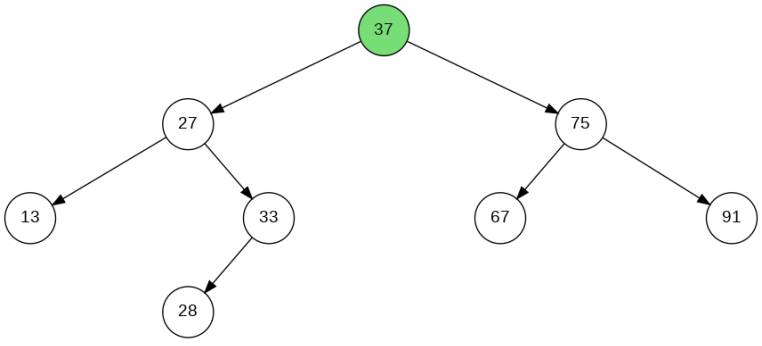
    - Leave the node x in place, but replace the value contained by x with a value y that is already in the tree *and* is in a position to be easily deleted

- Delete the node that contains y from the tree.

You may be asking yourself *how do we choose y to maintain the BST invariants*? This is a good question, and has a simple answer when we consider what the BST should look like when flattened to a list: a *sorted* list. If we want to maintain the invariants of the BST, we should be replacing the deleted value with either of the adjacent nodes in the sorted list. In other words: the largest element in the left subtree or the smallest element of the right subtree. These nodes are called *successor* nodes.

The following example shows what our example tree looks like before and after deleting the value 50. The value to be deleted is highlighted in red, and the successor node is highlighted in green.





C language

The C function `delete` is very much the same as our code for `find`, with the exception that instead of returning a `True` if we find the node, the `delNode` function is called to purge the value from the tree and adjust the branches as to satisfy the BST invariants.

```
struct node* delete(int key, struct node* root) {
    if (root == NULL) return root;
    if (key < root->key) {
        root->left = delete(key, root->left);
    } else if (key > root->key) {
        root->right = delete(key, root->right);
    } else {
        root = delNode(root);
    }
    return root;
}
```

The `delNode` function is where the interesting bits are found. Case 1 is straightforward: if there are no child elements free the memory allocated for the node and set return `NULL` to update the parent node from the previous call from the `delete` function.

Case 2 occurs when only one of the left or right subtree pointers is not `NULL`. In this case we create a temporary node containing the non-empty subtree, free the memory for the current node, and return the temporary node to update the parent node.

Case 3 is the tricky one, when both the left and right subtree pointers are not `NULL`. First we will find a successor node (in the example the *right* successor), by successively iterating down the left children of the right subtree. This can be seen

in the `while` loop. Once this node is found, we update the current nodes values and free the memory allocating the successor node. Lastly, we must clean up the pointer to the deceased successor node.

```
struct node* delNode(struct node* root) {
    if (root->left == NULL && root->right == NULL) {
        free(root);
        return NULL;
    }

    else if (root->right == NULL) {
        struct node* tmp = root->left;
        free(root);
        return tmp;
    } else if (root->left == NULL) {
        struct node* tmp = root->right;
        free(root);
        return tmp;
    }

    else {
        struct node* successor = root->right;
        struct node* parent = NULL;
        while (successor->left != NULL) {
            successor = successor->right;
            parent = successor;
        }
        root->key = successor->key;
        free(successor);
        if (parent != NULL) {
```

```
            parent->left = NULL;
        } else {
            root->right = NULL;
        }
        return root;
    }
}
```

Haskell

The Haskell implementation for `delete` is straightforward compared to the C implementation. Simply put, we follow the procedure:

- If the tree is empty, do nothing.
- If the tree is not empty and the value of the tree is not what we want to delete, recurse the appropriate subtree.
- If the current node is what we want to delete, perform the deletion.
  - When only 1 subtree is non-empty, return the subtree.
  - When both subtrees are non-empty, find the value of the successor node and the subtree with the successor node removed. Update the current tree with the new node value and new subtree.

In the example below, we are using the *left* successor, and therefore will be updating the *left* subtree of the current node. Also note that we do not explicitly check for 2 empty subtrees, this case is implicit when recursing on the first or second if statements in the `EQ` case. If you are still unsure, try tracing through the algorithm on paper!

```
delete :: Ord a => a -> BST a -> BST a
```

```
delete x t = case t of
  E        -> E
  T k l r -> case compare x k of
               LT -> T k (delete x l) r
               EQ -> if isEmpty r then l
                     else if isEmpty l then r
                     else let (k',l') = delMax l
                          in T k' l' r
               GT -> T k l (delete x r)
```

To help make the code clearer, the predicate to check for empty trees `isEmpty` has been created.

```
isEmpty :: BST a -> Bool
isEmpty t = case t of
  E -> True
  _ -> False
```

The `delMax` function is the auxiliary function that returns the max value of a tree, and the input tree with this max value removed. When called on a nodes left subtree, the return value of this function is the successor and updated left subtree after deletion.

```
delMax :: Ord a => BST a -> (a, BST a)
delMax t = case t of
  T k l E -> (k,l)
  T k l r -> let (k',r') = delMax r
             in (k', T k l r')
```

Python Binary Search Tree

Now it is your turn! With your group, roll your very own binary search tree library. You should have the following functionality implemented:

- A main function where you have created test instances of BSTs, and *test* your functions on these trees to check for erroneous behavior.
- A way to create BST objects (also known as a constructor).
- A function to check element membership in the BST.
- A display function that prints a tree *in order*.
- An insert function that will correctly insert an element into a BST.
- A delete function that will correctly remove an element from a BST.

If you finish early, here are some more advanced exercises that you can implement:

- Set union, which is one way of merging 2 trees together into a new BST.
- Set intersection, which is one way of merging 2 trees together into a new BST.
- Set difference, which is one way of merging 2 trees together into a new BST.
- A *finite mapping*, sometimes called a *dictionary*, using a BST tree. (Hint: use a tuple as the tree elements)

Good luck!