National University of Singapore
School of Continuing and Lifelong Education

TIC1002: Introduction to Computing and Programming II
Semester II, 2020/2021

Tutorial 6
Recursive Containers

1. In previous lectures, we learnt about the Set ADT found in the C++ STL. For this question, we will limit our discussion to these functions that support a Set.

   - `Set<T>` will construct and create a new set.
   - `void insert(Set<T> s, T element)` inserts element into the set
   - `bool element_of(Set<T> s, T element)` returns true if element is contained in the set, and false otherwise.
   - `Set<T> intersect(Set<T> s, Set<T> t)` returns a new Set that is the intersection of sets s and t.

   We will examine three different implementations to store the elements of a set ADT.

   Implementation 1 Using a vector to contain the elements of the set. Note that a set does not contain any duplicates.

   Implementation 2 Using a vector that is sorted. (What assumptions had we made?)

   Implementation 3 Using a Binary Search Tree.

   Tasks:

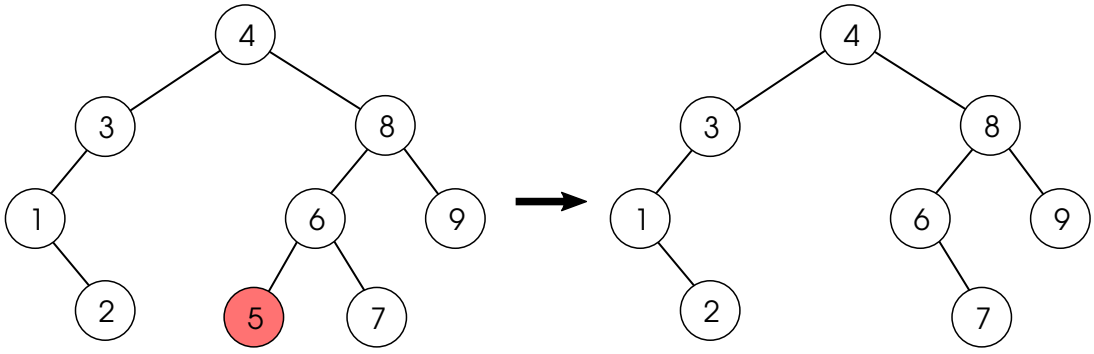
   (a) For each of the three implementation, implement the supporting functions listed above.

   (b) Compare the time complexity of the supporting functions across all the three implementations.

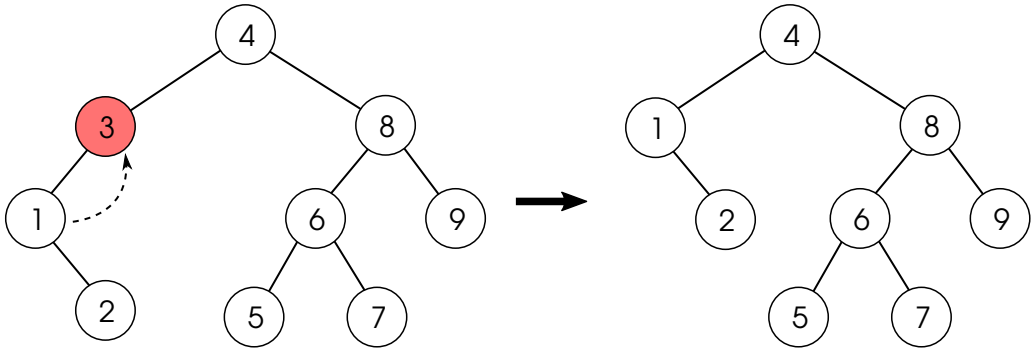
2. From the previous question, you would find that adding new elements to a BST is fairly straightforward. Removing an element, on the other hand, is not so.

   Because the property of a BST should be maintained after removing a node, we need to following this algorithm for removing the node:

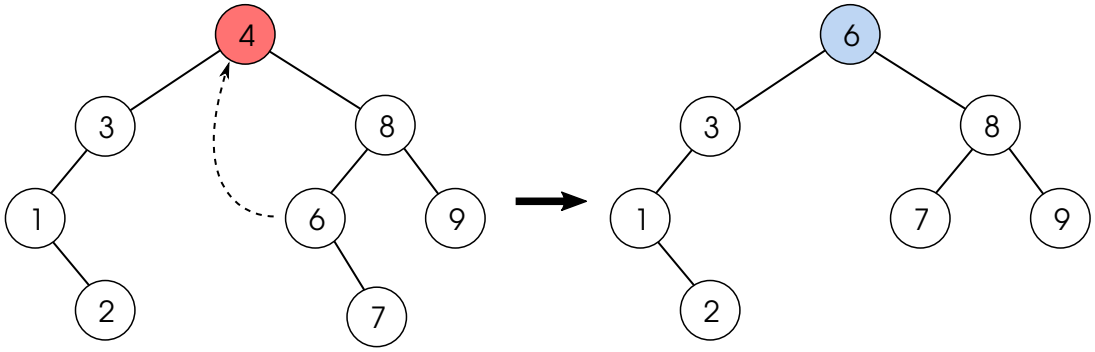
   - If the node has no children, then simply remove the node.



   - If the node has only one left child, replace the node with the left child.



- Otherwise, replace the node with the next largest node, which is the *left-most descendant* of the right child.



Provide an implementation `void remove_node(BSTNode &node)` to remove the given node from the BST. For simplicity, you may assume that `BSTNode` has an additional field `BSTNode *parent` which contains reference to its parent (if any).

3. Consider the following ADT:

```
struct Airport {
    string name;
    set<Airport *> connections;
}
```

It models an airport, and contains the airport name, and a set of other airports that it has connecting flights to.

(a) Given a particular airport, travellers want to know if they will be able to directly connect to another given airport.

Implement the function `bool direct_connect(Airport src, Airport dst)` which returns true if there is a direct connection from src to dst.

Things to note:

i. Compiler will complain if you try to compare two structs using `==`, because it does not know how to compare between two structs. In most modern languages, there is an implicit meaning of equivalence but it is not supported in C++.

ii. Thus, to simplify things, you might assume that every airport has a unique name, and perform your own comparison using names.

iii. Or if you are really want to do more, you can compare addresses instead.

(b) Travellers are still able to travel to a destination airport that has no direct connection by transiting through other airports.

Implement the function `bool indirect_connect(Airport src, Airport dst)` which returns true if there is a route from src to dst by transiting through any number of airports.

Things to note:

i. A direct connection also implies an indirect connection.

ii. The connection is not bi-directional, i.e. airport A connects to B, does not necessary mean B has a connection back to A.

iii. In fact, first try solving the problem assuming there are no cycles in the connections. This will prevent infinite loop. Once done, then try to modify your solution to handle cycles.

(c) There is a limit to the number of transits that a traveller can take.

Implement the function `bool indirect_connect(Airport src, Airport dst, int hops)`, where *hops* is the maximum number of "hops" which the traveller is willing to take. Note that hops should at least be 1 in order to reach the immediate next destination.