CISS350: Data Structures and Advanced Algorithms
Assignment 9

Objectives

1. Implement tree node class using `std::vector`
2. Implement tree node class using `std::list`
3. Implement binary tree node

For this assignment you will be implementing several tree nodes. The classes are template classes so that the nodes are be used for different types of data in the node. (Recall that it's usually a good idea to implement the non-template version first. You might need to experiment with the non-template version from time to time. So I recommend you *keep* the non-template version and develop the template version in parallel.)

Here's a quick description of the classes:

- `TreeNodev`: The tree node uses a `std::vector` of pointers for its children
- `TreeNodel`: The tree node uses a `std::list` of points for its children. `std::list` is the doubly-linked list class in the STL. Use the web to find out more about this class. (See the first section below on a quick tutorial.)
- `BinaryTreeNode`: The tree node uses a left and right pointer to refer to its children.

In the first two cases, there is no restriction on the maximum branching factor, i.e., the number of children. If a tree has a maximum branching factor it would probably be stored in the corresponding tree classes:

- `Treev`
- `Treel`
- `BinaryTree`

However, we won't be using these tree classes in this assignment.

For submission, make sure each question has it's own folder. For instance for `a09q01`, the code must be in folder `a09q01`. All question folders must be in a folder `a09`. Submit using alex.

Note: As in CISS245, skeleton code and pseudocode, where given, are meant to give you ideas. They are not meant to be complete.

Q2. You should know that to find 3-by-3 magic squares you can do this: enumerate all possible 9 digits numbers and use the 9-digit numbers to create 3-by-3 grid and test them to see which fits the requirements of being 3-by-3 magix squares. If you use this to generate the 9-digit numbers:

```
for (int n = 0; n <= 999999999; n++)
{
   ...
}
```

you would have generated 1,000,000,000 grids of 3-by-3 digits. We can shave off lots of redundant numbers like this:

```
for (int n = 123456789; n <= 987654321; n++)
{
   ...
}
```

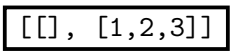
But this method of search for magic squares would still go through lots of numbers which clearly can't be magic squares. For instance when you `i` reaches 124000000, you will also redundantly go over 124000001, ..., 124009999 which you can tell can't be magic squares since they all have at least two zeroes.

One way to speed up the search is to look at one of the requirements for magic squares: the numbers used must be distinct. They are called permutations. Here's an example. The following is a permutation of 123:

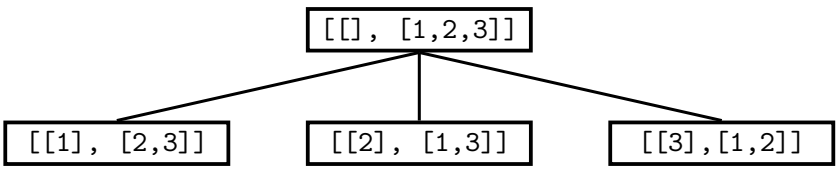
$$123, 132, 213, 231, 312, 321$$

There are only 6 such permutations.

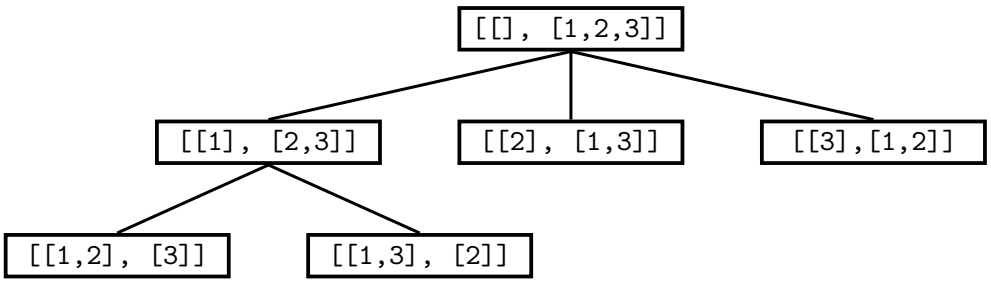
Note that we can build permutations this way: We start with $[1, 2, 3]$ as a list of available symbols. The permutation at this point is empty.



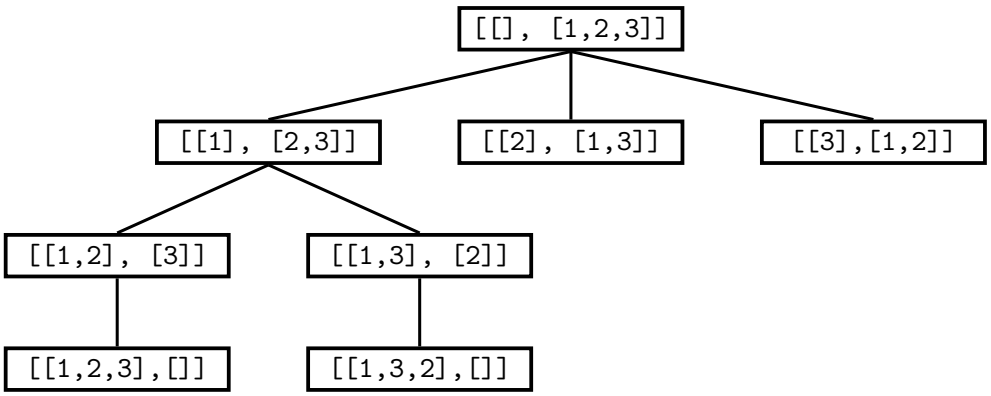
We now create permutations of length 1:



There are three. From the first of the three, I can create two permutations of length 2:



It should be clear what is happening here. We have a tree. Here are two leaves containing one permutations each:



Recall that a 3-by-3 magic is a 2D grid of distinct numbers from 1 to 9 with every row, every column, and every diagonal adds up to the same value. You can generate permuations with the above. The leaves are the permutations. Therefore all you need to do now is to traverse the tree (say using inorder traversal) and when the permutation forms a magic square, you print it (our put is into a container such as a vector).

How big is the tree above? For the permutation of 3 symbols, the size of the tree is

$$1 + 3 + 3 \cdot 2 + 3 \cdot 2 \cdot 1$$

To generate the permutations of 9 symbols (i.e., 1, 2, 3, ..., 9), the total number of nodes is

$$1 + 9 + 9 \cdot 8 + 9 \cdot 8 \cdot 7 + \cdots + 9!$$

This is a huge number. (You can compute that with your calculator.)

But you can do better. Note that for an n-by-n magic square containing numbers $1, 2, 3, ..., n^2$, the sum of each row, column, or diagonal is

$$\frac{n(n^2+1)}{2}$$

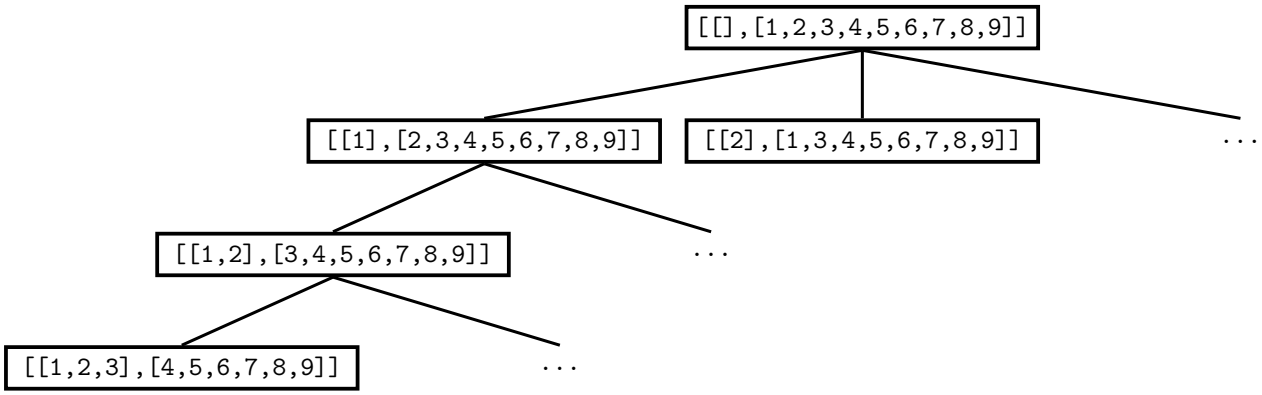
How does this help? For each node in the tree above, once n numbers are generated, you have a row and therefore you can can compute the sum of this row. Once the row does not add up to

$$\frac{n(n^2+1)}{2}$$

you know that there's no point computing its descendents. Likewise once you have $2n$ numbers, you can compute the sum of the second row. If the sum is not

$$\frac{n(n^2+1)}{2}$$

again, you don't have to compute its descendents. Etc. Once you have $(n-1)n$ number you can check the first column and one of the diagonals. Here's an example when $n = 3$.



Note that the node with the partially completed permutation `[[1,2,3],[4,5,6,7,8,9]]` need not be expanded further since $1 + 2 + 3$ is not $3(3^2 + 1)/2 = 15$. Therefore that is a leaf.

Write a program that accepts n as a command-line argument and, using the above method,

1. prints all n-by-n magic squares.
2. prints the number of nodes generated

Each magic square is printed on one line with the numbers in the squares separate by commas. Here's a sample run just to fix the output format:

```
g++ main.cpp
./a.out 3
8,1,6,3,5,7,4,9,2
1
```

The above program discovers one magic square, prints the magic square and prints 1. (The above is only to fix output format. There should be more 3-by-3 magic squares.) Note the

command-line argument `3` tells the program to print all magic squares of size 3.

Note. There are other methods to discover magic squares. You must use the above method since the point is to practice using trees.

Hint.

1. The tree can be build in a depth-first manner using a stack (the idea is very similar to depth-first traversal). (You can use `stl::list` to simulate a stack if you like. STL also comes with a `std::stack` class – you can use this too.) In other words, first create the root pointer pointing to the root node allocated in the heap. Push the root pointer onto the stack. Now in a while loop, as long as the stack is not empty, pop a pointer `p` off the stack. Check if the node that `p` points to is a leaf. If it's not, create children (on the heap) and attach them to the node that `p` is pointing to. Also, push the pointers of these children nodes onto the stack.
2. You can print the magic squares as you build the tree or you can traverse the tree after it's completely built, printing out leaves that are magic squares.
3. Each node has two lists of numbers. You can for instance use `std::vector< int >` for these two lists of numbers. In that case you should probably do this:

```
class Data
{
private:
    std::vector< int > permutation;
    std::vector< int > available;
};
```

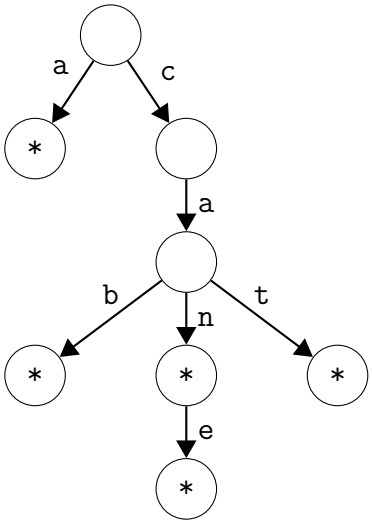
Q5. [Prefix tree/trie]

Suppose I want to write a spellchecker. First I would have to collect list of words. When the spellchecker runs, I will probably have to load the words into the program and then for each word read from a document, I will check against the collection of words loaded into the program's memory.

Let's say that I want my program to recognize the following words `a`, `cab`, `cat`. You can of course store the words into an array of strings. Say you have a collection of 500,000 words and the average length of the words is 10 characters. That means that I need about 5,000,000 bytes. I'll probably sort the array so that I can search for words quickly. The worse case is $\log_2(500000)$ word searches and then about 10 character comparisons for each word.

Let's think about it in a different way. Notice that in the English language, many words share common prefixes, i.e., left substrings. For instance notice that `cab` and `cab` has the same first two characters.

So first look at this picture:



It should be clear what we're doing here. Basically paths from the root in the tree form words. However not all paths form valid words. So to mark valid words with `*` in the sense that when going from the root to a node marked `*`, the edges (or the letters corresponding to the edge) forms a valid word. So in the above, you see word `a`, `cab`, `cane`, and `cat`. How do we assume letters with the edges? We can use the index positions of the pointers. So for instance from the root, the 0–th pointer that points to child-0 represents character `a`. Likewise reading a character `c` is the same as following the pointer to arrive at child-2.

Using the above idea, the above tree is constructed like this

```
int a = int('a' - 'a'); // i.e., 0
int b = int('b' - 'a'); // i.e., 1
int c = int('c' - 'a');
int t = int('t' - 'a');

TreeNodev< char > * p(' ');

p->insert(a, '*');
p->insert(c, ' ');

p->child(c)->insert(a, ' ');

p->child(c)->child(a)->insert(b, '*');
p->child(c)->child(a)->insert(t, '*');
// etc.
```

Basically the point is that at each node, there can be 26 pointers, one pointer for each character. (We only worry about lowercase.) So to check if the tree contains the word `dog`, we go node to node, following the appropriate pointer based on the character of the word. Since `d - a = 3`, `o - a = 14`, `g - a = 6`, we check if `p->child(3)->child(14)->child(6)->data()` is `'*'`.

You are given a word file `word.txt`. Write a program to do the following:

(a) Build the word tree using `word.txt` according to the above scheme.

(b) Print the total size of the tree (i.e., number of nodes) and a newline.

(c) The program then reads the command-line argument for a string input and search the word tree for that string. If the word is found, the program print `*` and then the word (and a newline). For instance:

```
g++ main.cpp
./a.out cab
1000
*cab
```

(Assuming that your tree contains 1000 nodes – the actual number is larger.) If the word is not found, then there are two cases. Suppose your tree can complete the word. Then the output follows this format:

```
g++ main.cpp
./a.out ca
1000
+ca:b,ble,t
```

if your program finds the words `cab`, `cable`, `cat`. Note the order follows dictionary order. If the string from command-line argument is `zz` which cannot be completed to a word, the

program prints this:

```
g++ main.cpp
./a.out zz
1000
?zz
```