CISS350: Data Structures and Advanced Algorithms
Assignment 9

Objectives

(a) Implement tree node class using `std::vector`
(b) Implement tree node class using `std::list`
(c) Implement binary tree node

For this assignment you will be implementing several tree nodes. The classes are template classes so that the nodes can 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 pointers 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.

Graphviz

Graphviz is a graph visualization software. You can find out more about this research project first initiated by AT&T Labs. It allows you to specify a graph including the nodes and the edges between nodes. The software produces image files drawing the graph, attempting to place the nodes in such a way that there is minimal edge crossings. You can also specify shapes of the nodes (circle, square, etc.) as well as color. Besides putting into the nodes, you can also label edges. This is probably among the most famous graph visualization software and is used by many computer scientists

To install the program on your fedora virtual machine do this (as root)

```
dnf -y install graphviz
```

This assumes that you're using a fedora machine.

Here's an example on how to create a directed graph. Create a text file named `graph.dot` (or any name you choose) with the following contents:

```
digraph G
{
   a -> b;
   a -> c;
   a -> d;
   b -> e;
}
```

You are creating nodes a, b, c, d, e with a joined to b, etc.

Run the following command from your shell:

```
dot -Tps graph.dot -o graph.ps
```

and you will get an image file `graph.ps` with the graph. Of course you can choose any other filename for your image file. This generates the image file as a postscript file. You can convert that to a JPG file doing this in linux:

```
convert graph.ps graph.jpg
```

To draw an (undirected) graph change your file to this:

```
graph G
{
   a -- b;
   a -- c;
   a -- d;
   b -- e;
}
```

You can change the label of a node and the shape like this:

```
digraph G
{
   a -> b;
   a -> c;
   a -> d;
   b -> e;

   a [shape=box];
   b [label="hello\nworld"];
}
```

You can find more about graphviz on the web.

Writing to a disk file

Refer to your CISS245 notes or textbook on file I/O. The following is a quick review on writing to disk files:

```
#include <iostream>
#include <fstream>

int main()
{
    std::ofstream f;
    f.open("output.txt");
    f << "hello world\n";
    f << 42 << '\n';
    f << 3.14159 << '\n';
    f << true << '\n';
    f.close();

    return 0;
}
```

After running this program, go to your shell and check the file `output.txt`.

Here's an example of reading contents of a file. I'll use the file created above.

```
#include <iostream>
#include <fstream>

int main()
{
    std::ifstream f;
    f.open("output.txt");
    std::string s;
    char t[100];
    int i;
    double d;
    bool b;
    f >> s;
    std::cout << s << '\n';
    f >> t;
    std::cout << t << '\n';
    f >> i;;
    std::cout << i << '\n';
    f >> d;
    std::cout << d << '\n';
    f >> b;
    std::cout << b << '\n';
    f.close();
```

```
    return 0;
}
```

The STL vector and list classes

Make sure your study and run the following program. Also, check my notes. Search for C++ references on the web – example: `www.cppreference.com` or `www.cplusplus.com`.

```
#include <iostream>
#include <list>
#include <vector>

//--------------------------------------------------------------------------
// An iterator is like a pointer. STL container classes can give you
// iterators. The point of iterator objects is to allow you to traverse
// the container using operator()++ and access the value by de-referencing.
//
// To get an iterator that points to the first value in the container, you do
// this:
//
//     std::list< int >::iterator p = list.begin();
//
// In the code below, because there's a template parameter, C++ will ask you
// to put "typename" before the declaration. So you have to do this:
//
//     typename std::list< int >::iterator p = list.begin();
//
// Make sure you remove "typename" below and compile the code to see the
// error message.
//
// You can compare your iterator p against the "end-of-list" doing this:
//
//     p != list.end()
//
// (You can think of list.end() as returning the pointer value just beyond
// the last value in the container.)
//
// The following therefore allows you to traverse the std::list, xs, printing
// its values:
//
//     for (typename std::list< int >::iterator p = xs.begin();
//          p != xs.end();
//          p++)
//     {
//         std::cout << (*p) << ' ';
//     }
//
// If the list xs is constant, then you need to use a constant iterator
// (which cannot modify the value that it points to):
//
//     for (typename std::list< int >::const_iterator p = xs.begin();
//          p != xs.end();
//          p++)
```

```
//     {
//         std::cout << (*p) << ' ';
//     }
//
// Make sure you change the iterator below to a non-constant iterator,
// compile your code and see the error message returned by the compiler.
//
// The loop in the template operator below is very standard when working
// with STL containers. Make sure you study it carefully.
//
// The code to print the values in a std::vector class is exactly the same.
// So if v is a std::vector<int> object, this prints the values in v:
//
//     for (typename std::vector< int >::const_iterator p = v.begin();
//          p != v.end();
//          p++)
//     {
//         std::cout << (*p) << ' ';
//     }
//--------------------------------------------------------------------------
template <typename T >
std::ostream & operator<<(std::ostream & cout, const std::list< T > & list)
{
    for (typename std::list< T >::const_iterator p = list.begin();
         p != list.end();
         p++)
    {
        cout << (*p) << ' ';
    }
    return cout;
}

int main()
{
    std::list< int > list;

    list.push_back(5);
    std::cout << list << std::endl;

    list.push_back(3);
    std::cout << list << std::endl;

    list.push_back(7);
    std::cout << list << std::endl;

    list.push_front(0);
    std::cout << list << std::endl;

    list.push_front(3);
    std::cout << list << std::endl;
```

```
    std::cout << "front: " << list.front() << std::endl;
    std::cout << "back: " << list.back() << std::endl;

    list.front() = -999;
    list.back() = 999;
    std::cout << list << std::endl;


    std::cout << "size: " << list.size() << '\n';
    std::cout << "empty?: " << list.empty() << '\n';

    list.pop_front();
    std::cout << list << std::endl;

    list.pop_back();
    std::cout << list << std::endl;
    std::cout << list << std::endl;

    list.clear();
    std::cout << list.size() << std::endl;

    return 0;
}
```

The STL `list` class is implemented using double-ended queues. The STL `vector` class is implemented using dynamic arrays. However the methods/operators supported by the vector class are very similar.

Although they have many similarly named methods/operators, you have to understand that the implementation is different and therefore the runtime complexities can be different. For instance the `push_back` of the `std::list` class has a worst runtime of $O(1)$. The worst runtime for the `push_back` method of the `std::vector` class is of course $O(n)$ where n is the size since for the vector class, inserting at the back when the size reaches the capacity, requires allocating a new dynamic array in the heap and copying n values from the old array to the new

Make sure you convert the above program to the vector version. You will see that the vector class does not have a `push_front` method.

Methods for tree nodes

Now I'm going to describe all the methods/functions/operators that you must implement for all tree node classes. First let me talk about the instance variables.

Each node class has a `key_` member and a `parent_` pointer. (Recall that there are situations where a parent pointer is not necesary, but for this assignment, we will include parent pointers.) Besides that, we need to talk about the member variables for the children pointers. The following are the options that we'll deal with for this assignment.

The class that uses `std::vector` of pointers for children looks like this:

```
template < typename T >
class TreeNodev
{
public:
    const T & key() const
    {
        return key_;
    }
    T & key()
    {
        return key_;
    }
private:
    T key_;
    TreeNodev * parent_;
    std::vector< TreeNodev * > child_;
};
```

(Memory aid: `TreeNodev` with `v = vector`.)

The class that uses `std::list` looks like this:

```
template < typename T >
class TreeNodel
{
private:
    T key_;
    TreeNodel * parent_;
    std::list< TreeNodel * > child_;
};
```

(Memory aid: `TreeNodel` with `l = list`.)

The class for binary tree node looks like this:

```
template < typename T >
class BinaryTreeNode
{
private:
    T key_;
    BinaryTreeNode * parent_;
    BinaryTreeNode * left_;
    BinaryTreeNode * right_;
};
```

The following is a list of methods that must be supported by all the tree node classes. In the following, let `node` be an object from any of the tree node classes. There are also corresponding functions that do the same thing. except that the functions accept pointers. For instance there's a `height` *function* that accepts a pointer to a node object; the return value is of course an `int`. In other words the prototype of the `height` *function* looks like this:

```
template < typename T >
int height(const TreeNodev< T > * const p);
```

Of course each node must be able to return the `key_` as a reference:

```
node.key()          key_ as a reference
```

If the `node` is constant, the reference returned is a constant. This has nothing to do with trees.

Now for the common methods for all tree node class.

```
node.is_root()      true iff node is a root
node.is_leaf()      true iff node is a leaf
node.is_nonleaf()   true iff node is non-leaf
node.height()       height of node
node.depth()        depth of node (note: level and depth are the same)
node.size()         number of descendents + 1

node.parent()       pointer to parent (returned as a reference)
node.root()         pointer to root

node.next()         pointer to next sibling,
                    NULL if there's no next sibling. In the case of binary
                    tree node, assume left pointer to the 0--th child and
                    right points to the 1-st child..
node.prev()         pointer to previous sibling,
                    NULL if there's no next sibling. In the case of binary
                    tree node, assume left pointer to the 0--th child and
```

```
                    right points to the 1-st child.

node.num_children()  number of children.
node.first_child()   pointer to first child (can be NULL)
node.last_child()    pointer to last child (can be NULL)
node.child(i)        pointer to i-th child (i = 0 is the first, can be
                     NULL)
node.left()          left pointer (only for case of binary tree node)
node.right()         right pointer (only for case of binary tree node)
node.leftmost()      pointer to leftmost child
node.rightmost()     pointer to rightmost child

node.insert(i, data) create a new node n, set n.key_ to data and attach n
                     as i-th child of node. Note that this might require
                     adding NULL pointers to fill up to child i - 1.
                     If there is already an i-th child, ValueError
                     exception object is thrown.
node.insert_parent(data)
                     create a new node n, set n.key_ to data and attach n
                     as parent. If there is already a parent, a ValueError
                     exception object is thrown.
node.insert_left(data)
                     create a new node n, set n.key_ to data and attach n
                     as left child of node.
                     If there is already a left child (i.e., left pointer
                     is not NULL) ValueError exception object is thrown.
                     (Only for case of binary tree node.)
node.insert_right(data)
                     create a new node n, set n.key_ to data and attach n
                     as right child of node.
                     If there is already a right child (i.e., right pointer
                     is not NULL) ValueError exception object is thrown.
                     (Only for case of binary tree node.)

node.remove(i)       remove the subtree at child i.
node.remove_left()   remove the subtree at left child
                     (Only for case of binary tree node.)
node.remove_right()  remove the subtree at right child
                     (Only for case of binary tree node.)
```

In case of an error, a `ValueError` exception object is thrown. The following is the `ValueError` class:

```
class ValueError
{};
```

Of course the classes also have obvious methods such as constructors, destructors, `operator=()`, `operator==()`, `operator!=()`. This is also the case for any tree class.

Note that for copy constructor and `operator=()`, make sure that objects are actually created. In other words, suppose a node `node2` has exactly a child pointer that is pointing to a node (i.e., it is not `NULL`). Then after the copy constructor

```
node1 = node2;
```

is called, `node1` also has exactly one child pointer that points a node.

When the destructor is called, all the memory used must be deallocated correctly. This applies to the whole tree structure, i.e., the memory used by all descendents. Note that this implies that the memory used by a child must be deallocated before memory used by a parent is deallocated.

Of course

```
node1 == node2
```

returns `true` exactly when `node1.key_` has the same value as `node2.key_`, both have the same tree structure, and the corresponding descendents have the same `key_` values as well. Otherwise `false` is returned.

With the above methods, if `node` is a tree node object, then one can for instance print the children of `node` like this:

```
for (int i = 0; child.child(i) != NULL; i++)
{
    std::cout << *(node.child(i)) << std::endl;
}
```

or this

```
p = node.first_child()
while (p != NULL)
{
    std::cout << *p << std::endl;
    p = p->next();
}
```

where `p` is a pointer of the appropriate type.

With the above methods/operators/functions, one can quickly build a binary tree

like this:

```
BinaryTreeNode< std::string > * p = new BinaryTreeNode("+");
```

```
p->insert_left("*");
p->insert_right("-");

p->left()->insert_left("1");
p->left()->insert_right("2");

p->right()->insert_left("3");
p->right()->insert_right("4");
```

Using the functions instead, we would do this:

```
BinaryTreeNode< std::string > * p = new BinaryTreeNode("+");
insert_left(p, "*");
insert_right(p, "-");

insert_left(left(p), "1");
insert_right(left(p), "2");

insert_left(left(p), "3");
insert_right(right(p), "4");
```

Print

The printing of the nodes for all tree node classes follow the same format.

For instance here is the `TreeNodev` class:

```
template< typename T >
class TreeNodev
{
public:
    ...
private:
    T key_;
    TreeNodev * parent_;
    std::vector< TreeNodev * > child_;
};
```

On executing this:

```
TreeNodev< int > node(5); // key_ = 5, parent_ = NULL,
                          // child_.size() = 0

std::cout << node << std::endl;
```

you get the following output:

```
<TreeNodev 0x12341234 key:5, parent:0,
    child:
>
```

If `node.child_` is not empty, say it has a size of 4, the printout would look something like this:

```
<TreeNodev 0x12341234 key:5, parent:0,
    child:
        0x12341238
        0
        0x12341242
        0x12341244
>
```

Inheritance

It's actually possible to subclass class templates. You do not have to do it for this assignment. You can read up on it. If you do use inheritance, it will save you some code since many methods and operators and functions for all the above three classes are actually very similar.

(My recommendation is that do *not* try template inheritance first. I strongly suggest you finish the assignment first – or at least 2 problems – before you try to combine them into an inheritance hierarchy.)

Q1. Implement the `TreeNodev` class with supporting methods, operators, and functions. Here's the class template that you should start with. This is skeleton code and is obviously not complete.

```
#ifndef TREENODEV_H
#define TREENODEV_H

#include <iostream>
#include <vector>

template < typename T > class TreeNodev;

template < typename T >
bool is_root(const TreeNodev< T > * p)
{
    return (p->parent() == NULL);
}

template < typename T >
class TreeNodev
{
public:
    TreeNodev(const T & key,
              TreeNodev * parent=NULL)
              : key_(key), parent_(parent)
    {}

    const TreeNodev * const parent() const
    {
        return parent_;
    }

    bool is_root() const
    {
        return ::is_root(this);
    }

private:
    T key_;
    TreeNodev * parent_;
    std::vector< TreeNodev * > child_;
};

#endif
```

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++)
{
   ...
}
```

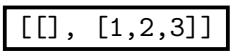
But this method of search for magic squares would still go through lots of numbers which clearly can't be magic squares. For instance when your `i` reaches 124000000, you will also redundantly go over 124000001, ..., 124009999 which you can tell can't be magic squares since they all have at least two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

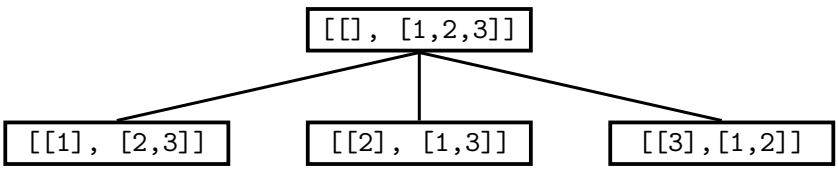
$$123, 132, 213, 231, 312, 321$$

There are only 6 such permutations.

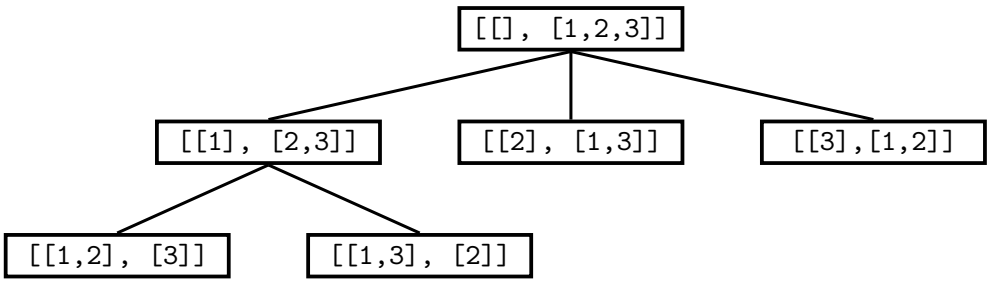
Note that we can build permutations this way: We start with $[1, 2, 3]$ as a list of available symbols. The permutation at this point is empty.



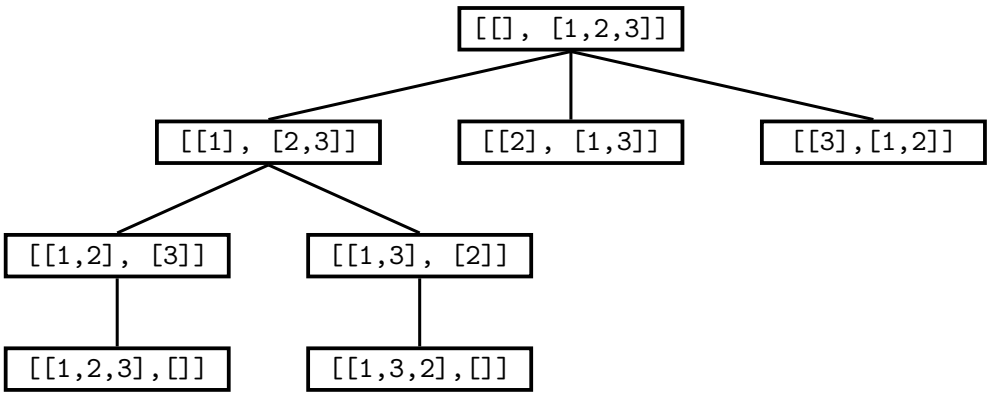
We now create permutations of length 1:



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 square 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

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

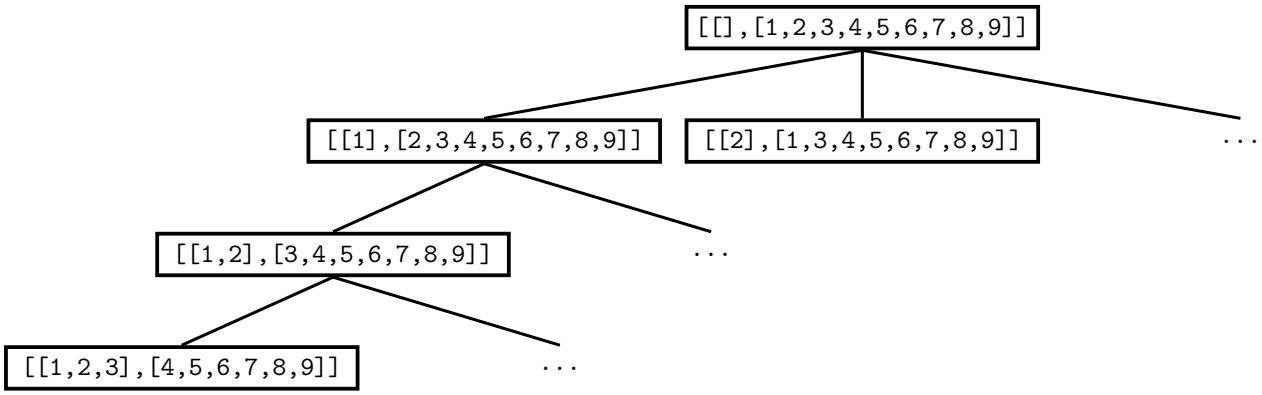
How does this help? For each node in the tree above, once n numbers are generated, you have a row and therefore you can can compute the sum of this row. Once the row does not add up to

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

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.

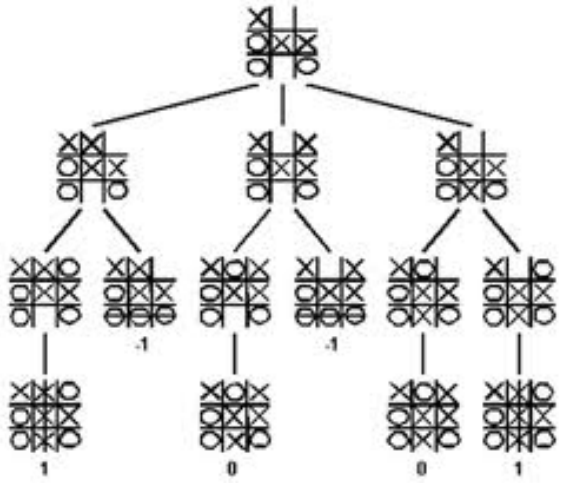
1. The tree can be build in a depth-first manner using a stack (the idea is very similar to depth-first traversal). (You can use `stl::list` to simulate a stack if you like. STL also comes with a `std::stack` class – you can use this too.) In other words, first create the root pointer pointing to the 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;
};
```

Q3. Implement the `TreeNodeI` class with supporting methods, operators, and functions.

```
#ifndef TREENODEI_H
#define TREENODEI_H

#include <list>

template < typename T >
class TreeNodeI
{
public:
    TreeNodeI(const T & key,
              TreeNodeI * parent=NULL)
        : key_(key), parent_(parent)
    {}

private:
    T key_;
    TreeNodeI * parent_;
    std::list< TreeNodeI * > child_;
};

#endif
```

Q4. The following is a game tree:



It's a tree where the nodes contain game states, describing all the possible scenarios of a game, in this case tic-tac-toe. The leaves are of course when there's a draw or when there's a winner.

Write a program that accepts (as command-line arguments) an integer `n`, a character `c` that is either `X` or `O` (that's uppercase letter and not zero), a string representing an `n`-by-`n` tic-tac-toe game state, print the game tree of states highlighting the states with guaranteed wins. Character `c` represeents the player to make the next move. You must also generate a graphviz dot file. Details on console and dot file printout is below. For this question, you must use the `TreeNodeI` class.

You may assume that the inputs are correct. For instance if the user enters 4 for `n` then the game state as a string has a length of $\texttt{n}^2 = 16$.

The children for a game state is basically taking an available cell on the game board for a player. The children are ordered in the usual way (i.e., left to right, top to bottom for available cells.)

Here's how the user will enter a game state. To enter the empty 3-by-3 board, the user enters `_________`, i.e., use `_` (underscore) for a space. For this game state:

```
X|O|X
-+-+-
 | |O
-+-+-
 | |X
```

the user enters `XOX__O__X`.

Your printout must print the tree using pre-order traversal of the game tree. For each edge down the tree, indent by 4 spaces. This printout is to the console/shell window.

Here's an input requesting for all winning moves for `X` (underlined text is user input). Only the first few lines of output is shown. In this case the player making the move is `X`. (Yes I know there are more `X`'s than `O`'s.)

```
g++ main.cpp
./a.out 3 X XOX__O__X
XOX__O__X
    XOXX_O__X
        XOXXOO__X
            XOXXOOX_X (X)
            XOXXOO_XX
                XOXXOOOXX (_)
    XOX_XO__X (X)(*)
    ...
```

Next to each leaf game states, you will see `(X)` indicated that the winner is `X`. If you see `(O)`, then `O` is the winner. If you see `(_)`, then it's a draw.

Also, next to the children states of the root, you will see `(*)` which tells the user that that's a good move, i.e, it's either a winning move or is guaranteed to make a winning move in the future. For instance the child-2 of the root is a child state that will produce a winning move:

```
g++ main.cpp
./a.out 3 X XOX__O__X
XOX__O__X
    ...
    XOX__OX_X (*)
    ...
```

That's because the descendents of this state looks like this:

```
g++ main.cpp
./a.out 3 X XOX__O__X
XOX__O__X
    ...
    XOX__OX_X (*)
        XOXO_OX_X
            XOXOXOX_X (X)
            XOXO_OXXX (X)
        XOX_OOX_X
            XOXXOOX_X (X)
            XOX_OOXXX (X)
        XOX__OXOX
            XOXX_OXOX (X)
            XOX_XOXOX (X)
    ...
```

Note that all paths lead to winning states. Also, note that when X is making a move and there is one move that will force a win in the future, X is still the winner. X does not need to have all leaf nodes to be winning states.

As for the graphviz dot file, here's an example on how to label the nodes.

```
digraph G
{
   ...

   XOX_X_XX_ [shape=box, fontsize=6, label="XOX\n_X_\nXX_"];
}
```

Obviously this should be generated by your program.

Note. A game tree is essentially the intelligence behind AI game agents. For games even with moderate complexity, their game trees are extremely huge. For instance there are approximately 10^{120} possible game states in chess. This is even more than the number of atoms in the observable universe, i.e., 10^{80}. This means that there is absolutely *no* way one can create the whole game tree of chess and then play a game of chess intelligently.

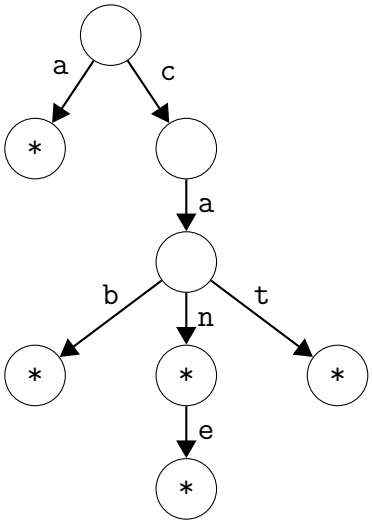
Q5. [Prefix tree/trie]

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 worst 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 `cat` have 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
```

Q6. [Implement binary tree]

Implement the `BinaryTreeNode` class with supporting methods, operators, and functions.

Q7. [Expression tree]

You must use `BinaryTreeNode` from the previous question.

Write a program that accepts an arithmetic expression in this form: If the user wants to compute

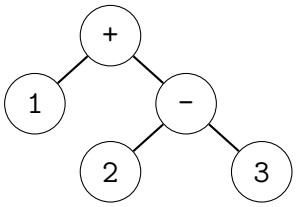
$$1 + (2 - 3)$$

he enters

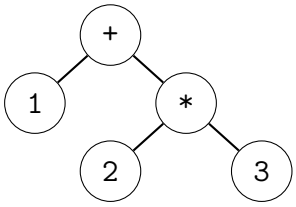
```
1+(2-3)
```

(i.e., no spaces). The arithmetic expression operates only on single-digit numbers from `0` to `9` (it's pretty easy to include other multi-digit numbers but you don't have to worry about these cases) and the operators are `+,-,*,/,%`.

The program then builds a tree



Your program must generate the tree according to standard precedence rules. For instance multiplication goes before addition. Therefore for the string `1+2*3`, the tree is



(Note that some nodes hold operators and some hold integer values.)

Your program then generates a dot file for graphviz, prints the arithmetic expression in RPN (in the console window), and print the result of the expression (in the console window). You then generate a `ps` file to display the tree. For the above expression `1+2*3`, in RPN, it becomes `123*+`.

Note.

1. It's OK to use the textbook or the web to find a suitable algorithm to create the tree from the arithmetic expression string. You can also refer to my notes or the textbook. But you must write the code yourself, of course.
2. Of course you also have unary operators (example: "negative of" and "positive of") and ternary operators (example: `(:?)`).