COMP 2012H Honors Object-Oriented Programming and Data Structures

Assignment 11 Dictionary using trie

Honor Code

We value academic integrity very highly. Please read the Honor Code section on our course webpage to make sure you understand what is considered as plagiarism and what the penalties are. The following are some of the highlights:

- Do NOT try your "luck" - we use sophisticated plagiarism detection software to find cheaters. We also review codes for potential cases manually.
- The penalty (for **BOTH** the copier and the copiee) is not just getting a zero in your assignment. Please read the Honor Code thoroughly.
- Serious offenders will fail the course immediately, and there will be additional disciplinary actions from the department and university, upto and including expulsion.

End of Honor Code

Objectives & Intended Learning Outcomes

The objective of this assignment is to expose you to manipulating information using the trie data structure. After this assignment you should be able to:

1. Learn the trie data structure and some basic operations
2. Implement algorithms in C++ to manipulate the structure
3. Make use of move operations to enhance the efficiency of your program
4. Understand lambdas and operator overloading in C++

End of Objectives & Intended Learning Outcomes

Menu

Page maintained by

CHAU Yu Hei
yhchau@connect.ust.hk
Last Modified:
11/21/2022 16:40:44

Homepage

Course Homepage

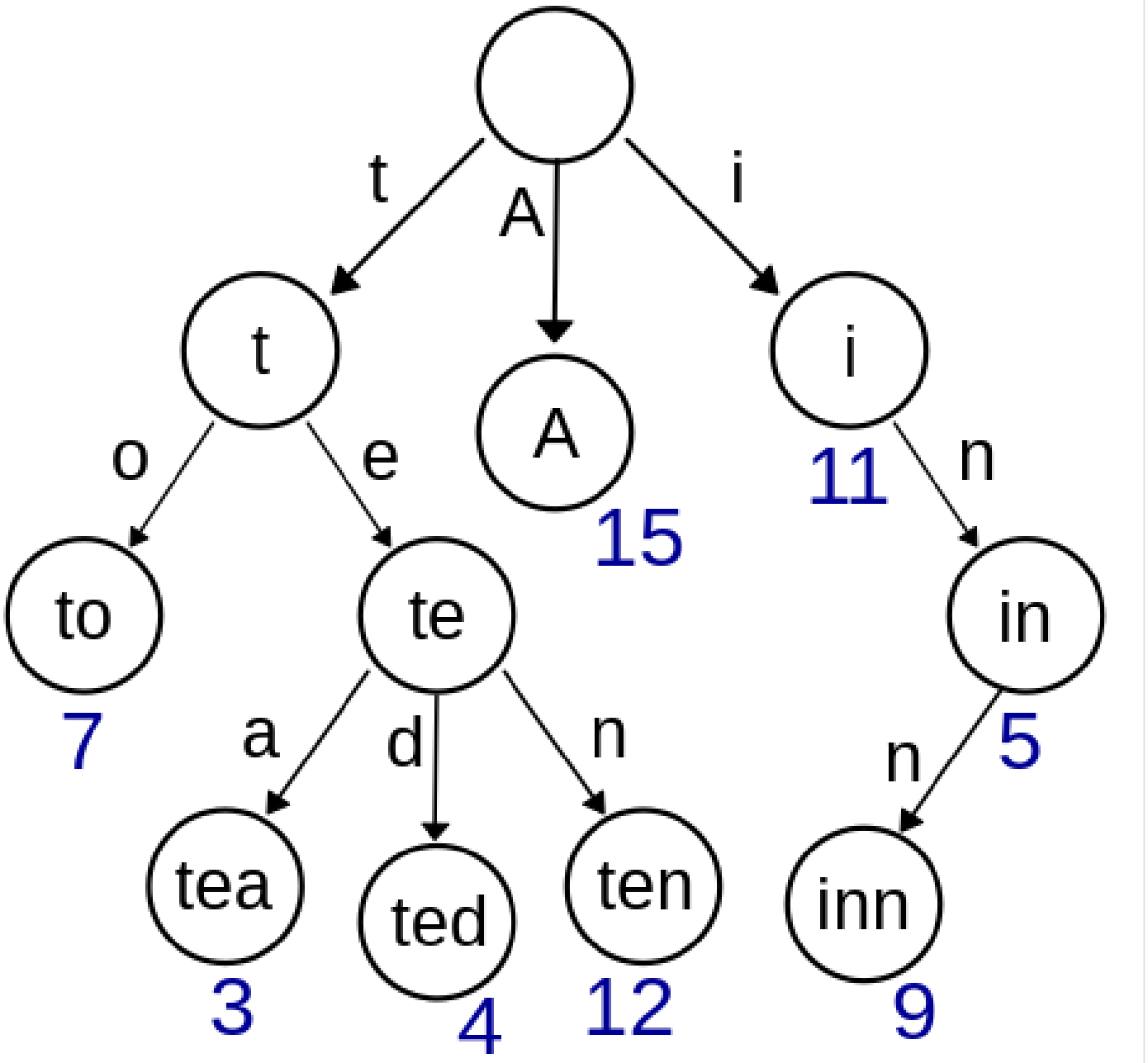


Image Source: Wikipedia https://en.wikipedia.org/wiki/Trie

Introduction

The objective of this assignment is to create a console program which has the basic functions of a 'dictionary library'. The program stores multiple dictionaries in a list. The user can create, delete and modify the saved dictionaries. Most importantly, the program contains word lookup and word suggestion functions with reference to the dictionaries.

- Store a list of dictionaries, each with a unique name.
- Modify the contents of the dictionary by adding/removing new words or updating its meaning.
- Supports the merging of two dictionaries with each other.
- Supports trimming down a dictionary into a smaller one.
- Lookup and word suggestion in a dictionary.

In this assignment, you are not required to:

- Write the code for user input / output, which is provided in the skeleton code.

In this assignment, you will be required to:

- Implement the functions for manipulating the trie structure.
- Understand move operations in C++

End of Introduction

Description

Please read the FAQ section regularly, and do check it one day before the deadline to make sure you don't miss any clarification, even if you have already submitted your work by then. You can also raise questions on Piazza and remember to use the "pa11" tag.

Code structure

The skeleton code structure is as follows:

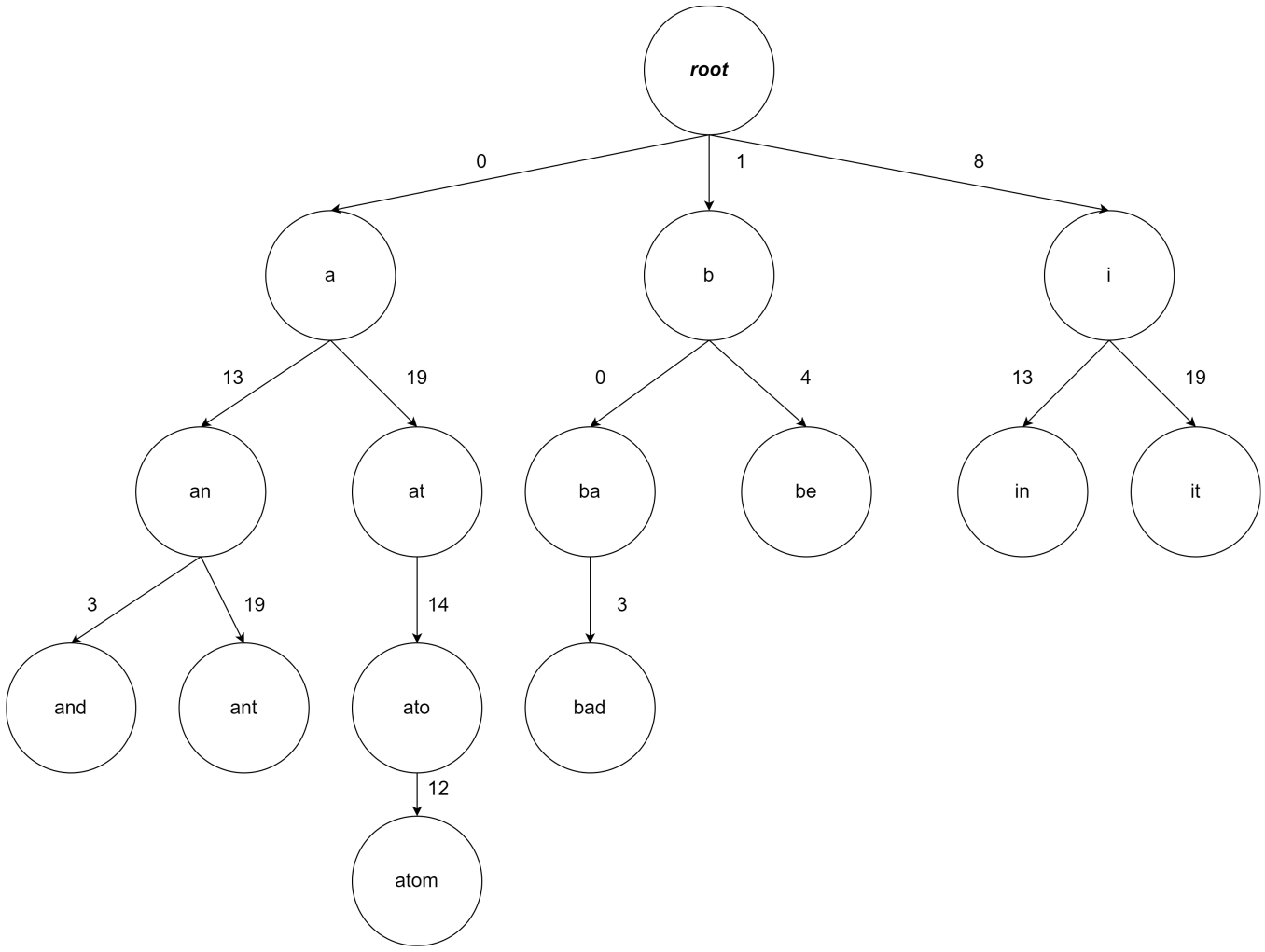
```
PA11
└── main.cpp
└── dictionary.cpp
└── dictionary.h
└── node.cpp
└── tasks.cpp
└── tasks.h
```

Your task for this PA is to implement functions in `dictionary.cpp`. ZINC will only accept the modified `dictionary.cpp` file, modifications to other files will be discarded.

Tries for dictionary

The trie structure is a data structure which stores information in a tree form. In each trie, there is a root node corresponding to the empty string. Each trie node potentially stores 26 addresses, corresponding to each English letter, as potential children. The 'key' of a trie node is the string created by traversing the tree structure from the root to the node.

Tries in detail



The 'string' in the nodes represents the key of the node (except for the root node, which is represented by the empty string). The edges (straight lines) of the diagram indicates the connection from the parent node to the child node. Each node has an array of 26 pointers `Node* Node::children[26]`, where the `i`th pointer corresponds to a child node with key equal to that of the parent node plus a character corresponding to `i`. The index `i` is represented by the numbers in the edges above. The pointer may be `nullptr`, whenever no such child exists. Each node except the root node contains a pointer `Node* Node::parent` to its parent, while being equal to `nullptr` for the root node. To summarize:

- `Node* Node::children[26]` - Pointers to children
- `Node* Node::parent` - If not the root node, pointer to parent

Node that `Node::children[26]` and `Node::parent` are private. You should use the following for access:

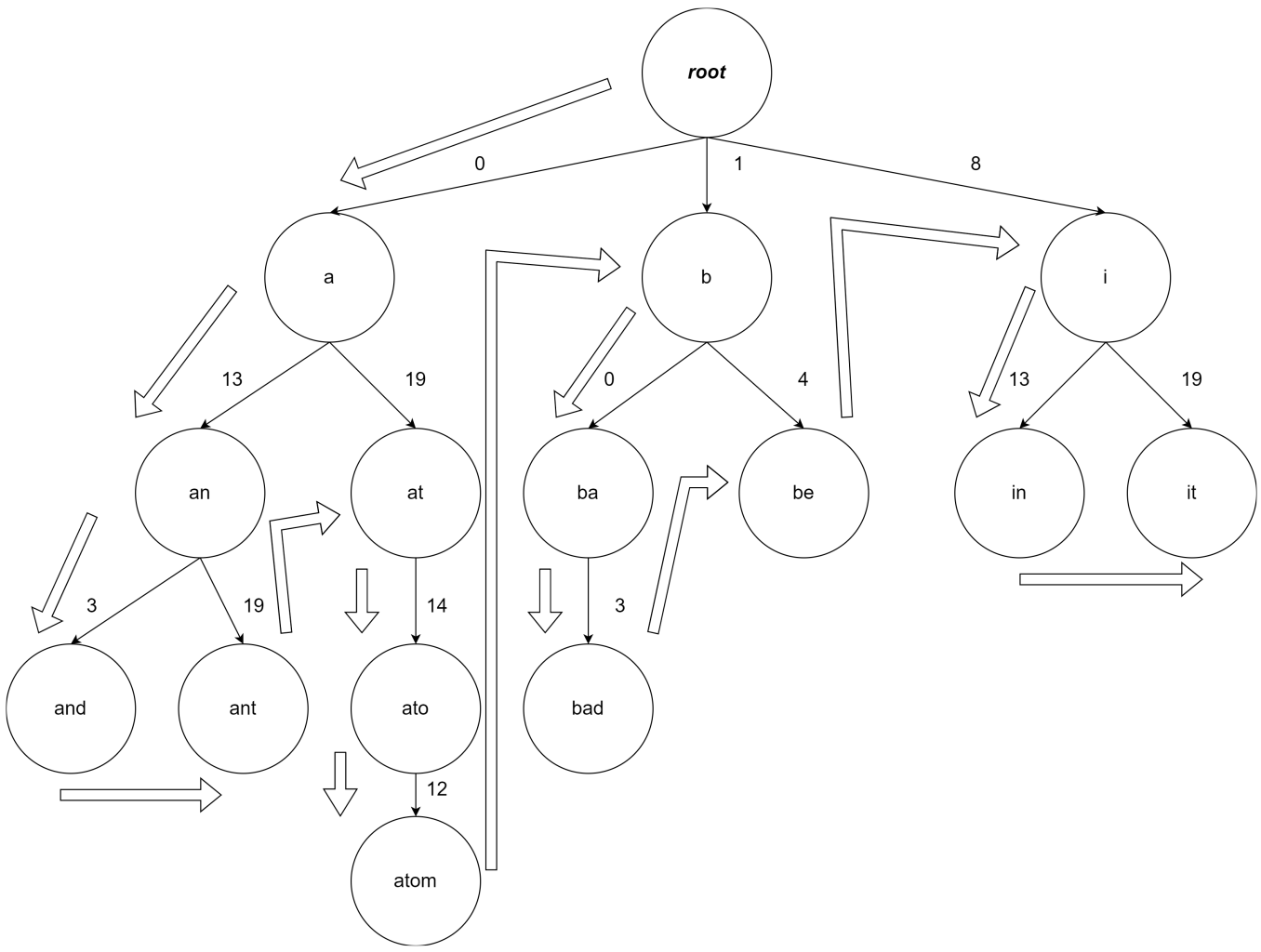
```
Node* Node::operator[](const int& idx);
void Node::set_child(const int& idx, Node* ptr);
Node*& Node::get_parent();
```

Easy conversion between character and index

You can use your own method using ASCII table, or use the provided `static` helper functions:

```
int Dictionary::character_to_index(const char& ch);
char Dictionary::index_to_character(const int& idx);
```

Traversal



In this PA, we mainly use **pre-order traversal** for the trees. The illustration of pre-order traversal is given above. In words, pre-order traversal is a traversal method which traverses the tree depth-first and in alphabetical order, where the contents of the node are retrieved **immediately** when the node is first visited. The 'expected output' of printing the keys of the above example would be (discarding the root):

```
 a -> an -> and -> ant -> at -> ato -> atom -> b -> ba -> bad -> be -> i -
> in -> it
```

Implementation of dictionary

The dictionary consists of two main classes, `Dictionary` and `Node`. The `Dictionary` class contains a pointer to a root node, with all the supporting functions for manipulating and retrieving the contents of the dictionary. The `Node` class contains a pointer to its parent, 26 pointers to potential children, and the type and definition of the word.

- Dictionary
  - Constructor and destructors
  - Copy and move assignment
  - Add node - Adds a node with a specified key to the dictionary
  - Remove node - Remove all nodes starting with the specified key from the dictionary
  - Find node - Finds a node with the specified key, returns nullptr if none found
  - Print all elements - Prints the tree structure of the dictionary, depth first, in alphabetical order
  - Print all elements with type - Prints all words with the given word type
  - Merge - Merges two dictionaries together
  - Filter - Filters the dictionary by returning all elements with key starting with the given string
- Node
  - parent - Pointer to parent of this node, may be null
  - children - Pointers to children of this node, may be null
  - meaning.meaning - The definition of the word
  - meaning.type - The type of the word (noun, verb, adjective etc.)
  - Constructor - The parents and children are set by default to `nullptr`

- Destructor - Deletes all of the children, but does not delete the parent.

Information retrieval from dictionary

Obviously not all nodes are words. To distinguish this, whenever `meaning.meaning` is not the empty string, we call the node a 'word' or a 'valid word'. For example, printing all words belonging to the dictionary means that traversing all nodes of the dictionary, whilst only printing the ones such that `meaning.meaning` is not empty. **You can safely assume** that `meaning.meaning` is non-empty if and only if `meaning.type` is non-empty.

Assumptions for arguments of functions

A common argument in the functions is `const char*`. This represents a string in the usual way, ending in `'\0'`, and in addition the contents of the string are restricted to characters `a-z` (always lowercase). **Do not** `delete` any `const char*` passed to your functions. The given `const char*` **may** be an empty string. The argument would not be `nullptr` **unless** explicitly stated.

Requirements

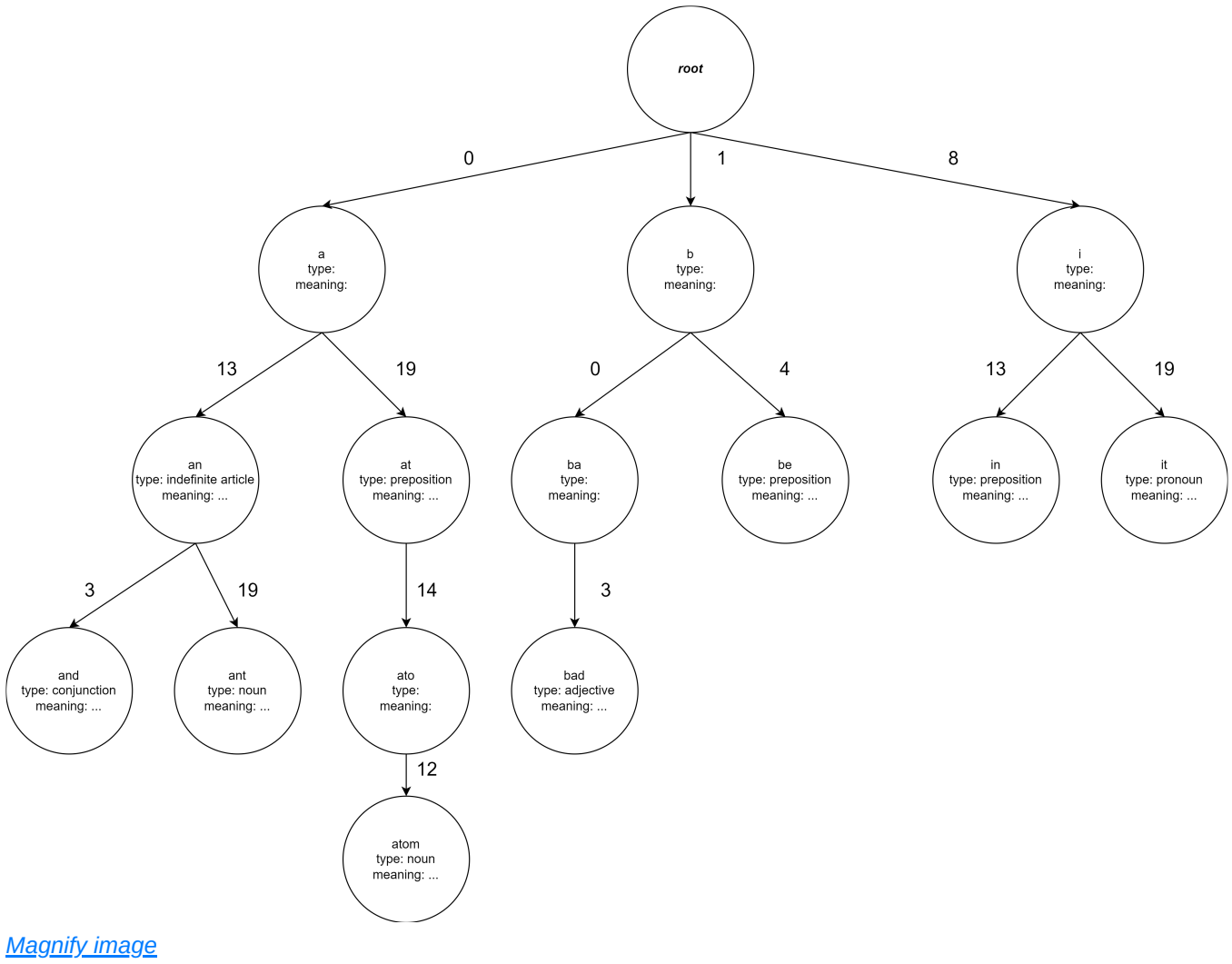
Here are a few clarifications for what is expected in this assignment.

- You can use any C++ standard library, such as `<string>, <vector>, <list>` etc.
- You may define additional global variables, helper functions, custom classes, or whatever you may want.
- There may be various algorithms and implementation methods for doing the tasks below. You can freely choose to implement them in your own way, such as lambda functions, recursive functions and so on, as long as the requirements of the tasks are met.
- Although not recommended, you may modify the provided functions in `Dictionary.cpp`, *at your own risk*.
- Note that some tasks require some conditions on the efficiency of the algorithm. See grading for more details.
- The above points can be summarized as: you can do whatever you want as long as your program compiles and gives the same output as the demo program, with no memory leak.
- You are advised to **only** modify the file `Dictionary.cpp`. ZINC will only make use of your `Dictionary.cpp` file, and modifications of other source files will be overwritten.

End of Description

Tasks

This section describes the functions that you will need to implement. You also have to implement the corresponding operator overloads for the respective functions, which mostly only consists of a single call to the function. They will be explicitly stated. In the following, all examples would refer to the following tree:



Magnify image

When the meaning or type is empty in the above diagram, this means that it is an empty string. Here is the PPT for PA11 in the Lab: PowerPoint

Important: Ensure tree structure

In *all* the functions below, you should ensure the tree structure satisfies the conditions stated by Tries in detail above. To quote:
Each node has an array of 26 pointers `Node* Node::children[26]`*, where the* `i`*th pointer corresponds to a child node with key equal to that of the parent node plus a character corresponding to* `i`*. The index* `i` *is represented by the numbers in the edges above. The pointer may be* `nullptr`*, whenever no such child exists. Each node except the root node contains a pointer* `Node* Node::parent` *to its parent, while being equal to* `nullptr` *for the root node. To summarize:*

- `Node* Node::children[26]` *- Pointers to children*
- `Node* Node::parent` *- If not the root node, pointer to parent*

If the tree structure is **not** totally correct, you may risk segmentation fault errors in ZINC as the hidden test cases use **all** of the properties of the tree structure, which you would get zero marks for that task. There may also be a risk that your program works locally and segfaults on ZINC. As mentioned before, ZINC uses all the properties of the tree structure, while your program may not. It is your responsibility to check that your program runs on ZINC.

Copy constructor

```
Dictionary::Dictionary(const Dictionary& d);
```

Description - Constructor to create a deep copy of `d`.

Copy assignment

```
Dictionary& Dictionary::operator=(const Dictionary& d);
```

Description - Copy assignment to create a deep copy of `d`. Hint: The original data has to be discarded to prevent memory leak!

Add node

```
Node* Dictionary::add_node(const char* key);
Node* Dictionary::operator+=(const char* key);
```

Description - Creates a new node with the given key. Returns the created node. If a node with given key already exists, then return the already existing node. If the key is the empty string, return the root node. You can directly call add_node in the operator overload.

Comments: In the provided skeleton code of the full program, the contents of the Node will be updated *after* the Node with the key is created. It is therefore useful to return a Node* so that the skeleton code obtains a pointer to the created node. However, the `operator+=` usually returns a reference to `this` in most applications. This implementation of `operator+=` is admittedly, an ad-hoc implementation.

The comments are just for your interest, you don't have to worry about updating `Node.meaning` as it is afterwards done by the skeleton code.

Remove node

```
void Dictionary::remove(const char* key);
void Dictionary::operator-=(const char* key);
```

Description - Removes all the nodes with key starting with the given `key`. The node with key exactly equal to the given `key` should also be removed. If no such nodes exists, do nothing. Remember to free up the memory used by all such nodes too. Of course the root node should not be deleted. You can directly call remove in the operator overload.

Find node

```
Node* Dictionary::find_node(const char* key) const;
```

Description - Finds the node with the exact given `key`. Returns `nullptr` if no such node exists. If the key is empty string, return the root node.

For each

```
template<typename T> void Dictionary::foreach(T&& lambda) const;
```

Description - A function to do pre-order traversal on the tree. The function accepts a lambda function as argument, and then the lambda function would be called for every node in the tree (except the root node). The lambda function accepts two arguments, `(Node* current_node, vector& current_key)`. For each node accessed via pre-order traversal (except root node), call the lambda function.

Of course current_node should be the pointer to the node accessed. current_key should contain a list of integers which corresponds to the indices required to travel to current_node from the root node.

With each line indicating the pointer to node, followed by an array representing the vector<int>, the lambda calls for the example are:

```
a [0]
an [0, 13]
and [0, 13, 3]
ant [0, 13, 19]
at [0, 19]
ato [0, 19, 14]
atom [0, 19, 14, 12]
b [1]
ba [1, 0]
bad [1, 0, 3]
be [1, 4]
i [8]
in [8, 13]
it [8, 19]
```

Print all elements

```
void Dictionary::print_all_elements(ostream& o) const;
std::ostream& operator<<(std::ostream& o, const Dictionary& dict) const;
```

Description - Prints out all nodes in pre-order traversal (alphabetical), **including** the nodes that are not words, but not the root node. The syntax of the message should be `key (type) meaning [number of nodes printed]`, one line for each node. For convenience, `cout << key_string_without_spaces << current_node << "[" << count << "]\n";`, where the predefined operator overload `ostream& operator<<(ostream& o, Node* n)` for printing the meaning of the node is used. Referring to the above example,

```
a ()  [1]
an (indefinite article) ... [2]
and (conjunction) ... [3]
ant (noun) ... [4]
at (preposition) ... [5]
ato ()  [6]
atom (noun) ... [7]
b ()  [8]
ba ()  [9]
bad (adjective) ... [10]
be (preposition) ... [11]
i ()  [12]
in (preposition) ... [13]
it (pronoun) ... [14]
```

You are encouraged to use lambda functions and foreach for this task. The overloaded `<<` operator can be used for couting a dictionary directly. You can directly call print_all_elements in the implementation of the operator overload.

Print elements given type

```
void Dictionary::print_elements_given_type(const char* type) const
```

Description - Prints out all words in alphabetical order, and such that the word type is equal to `type`. Recall that a `Node` is a *word* if and only if `meaning.meaning` is not the empty string. If `type == nullptr`, do not impose any restrictions on the word type. For the above example with `type = nullptr`,

```
an (indefinite article) ... [1]
and (conjunction) ... [2]
ant (noun) ... [3]
at (preposition) ... [4]
atom (noun) ... [5]
bad (adjective) ... [6]
be (preposition) ... [7]
in (preposition) ... [8]
it (pronoun) ... [9]
```

For the above example with `type = "preposition"`,

```
at (preposition) ... [1]
be (preposition) ... [2]
in (preposition) ... [3]
```

You are encouraged to use lambda functions and foreach for this task.

Merge dictionaries (copy)

```
Dictionary merge(const Dictionary& d2) const;
```

Description - Merges **all the nodes** of the current dictionary with the other dictionary. That is, the set of keys in the new dictionary is exactly the union of the set of keys in the original dictionaries. The original dictionaries should not be modified. If a word exists in both dictionaries `*this` and `d2`, use the word definition and type in `*this`. The new dictionary should be a deep copy of both dictionaries.

You can initialize the name of the returned dictionary to be empty string. The name is mainly used internally by `main.cpp`, which would be set to be the user input (in the skeleton code).

Merge dictionaries (move)

```
Dictionary merge(Dictionary&& d2) const;
```

Description - Merges **all the nodes** of the current dictionary with the other dictionary. This time, the contents in `d2` should be moved to the new dictionary, and d2 is expected to be destroyed right after. The contents of `*this` should not be modified, and be deep copied to the new dictionary.

You can initialize the name of the returned dictionary to be empty string.

Filter dictionary

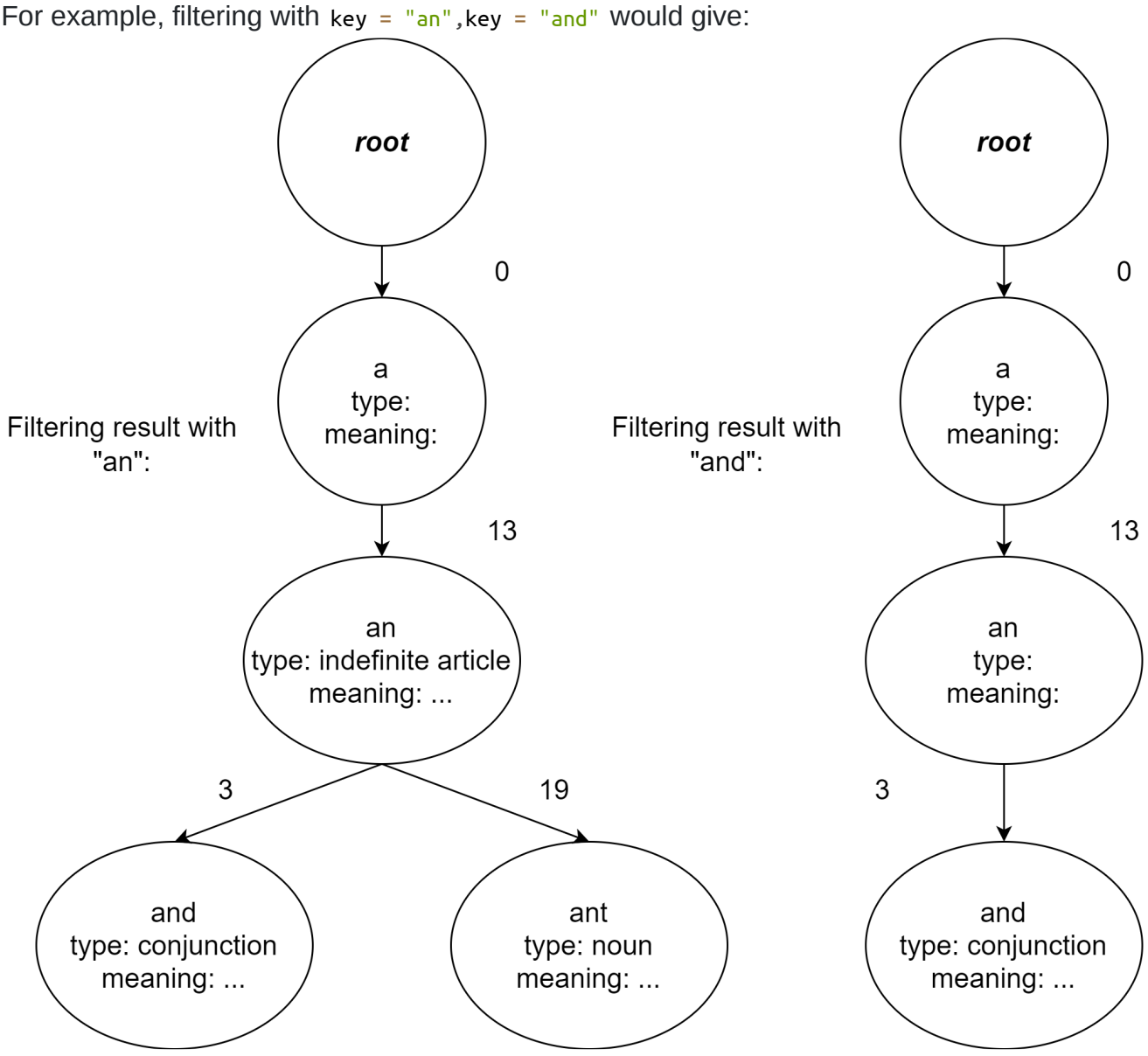
```
Dictionary Dictionary::filter_starting_word(const char* key) const;
```

Description - Filters the dictionary by returning a new dictionary containing all the nodes with key starting with `key`. The set of keys of the resulting dictionary is exactly the subset of keys in the original dictionary such that the key starts with the argument `key`. The necessary ancestor nodes of the node with `key` have to be newly created. If such ancestor nodes exists with `meaning.meaning` not empty, its meaning and word type is to be discarded (set to empty string) in the new dictionary.

For the node with `key` and its children, the meaning and word type have to be copied over. The contents of `*this` should not be modified, and the new dictionary should be a deep copy containing the correct information. If `key` is `nullptr` or empty string, return a deep copy of `*this`.

For a more illustrative description, this is just copying the entire subtree starting with prefix `key` (including key), with the node having key equal to `key` as a (temporary) root of the subtree.

Afterwards, we create necessary parent nodes for the root of this newly created subtree so that the node would have equal key in the new tree. The newly created parent nodes will have meaning and word type to be empty string. If the subtree is empty, return an empty dictionary. For example, filtering with `key = "an",key = "and"` would give:



You can initialize the name of the returned dictionary to be empty string.

Bonus: Efficiency and correctness of implementation of tree traversal

This is not a speed programming competition. You will be awarded bonus marks as long as all tasks 01-14 in ZINC are verified (see below for criteria).

Accessing `Node::operator[](const int& idx)` each time increments `int NodeStats::pointers_accessed` by 1. The goal is to make the number of calls to Node::operator[] as few as possible in the following:

```
Node* Dictionary::find_node(const char* key); //Requirement (1)
void Dictionary::print_all_elements(); //Requirement (2)
Dictionary Dictionary::merge(Dictionary&& d2) const; //Requirement (3)
```

- Requirement 1 - At most the length of the string represented by key
- Requirement 2 - At most 26 x number of nodes in `*this` (including root)
- Requirement 3 - At most [(26 x number of nodes in `*this` (including root)) + number of nodes in `*this` (excluding root)]

The bonus part will be obtained as long as requirements 1-3 are met. We do not time the actual running time or your program, nor do we compare the running time or number of operator[] calls between students.

End of Tasks

Recommendations & Hints

You are recommended to read the grading before starting the assignment to understand how the functions are going to be tested in ZINC. The following functions are useful for debugging your code:

```
int NodeStats::get_nodes_created(); // Total number of Nodes created (constructor)
int NodeStats::get_nodes_deleted(); // Total number of Nodes deleted (destructor)
int NodeStats::get_pointers_accessed(); // Total number of times Node::operator[] is called.
void NodeStats::print_statistics();
```

End of Recommendations & Hints

Resources & Sample I/O

- Skeleton code (last updated on 15:20 9 Nov): PA11_Skeleton.zip
- Demo programs (last updated on 09:00 9 Nov):
  Windows x64 / Windows x86 / Linux / MACOSX x86 / MACOSX arm
- Demo powerpoint in Lab
- Sample program inputs (word definitions by Dictionary.com): Sample Inputs

Changelog:

- 15:20 9 Nov - Fixed a problem in Line 12 of the skeleton code version of `Tasks.cpp`.
- 18:00 11 Nov - Updated sampleIO to include program outputs.

If you encounter problems running the MacOS demo programs, you can try look for a Windows PC on campus or use the Virtual Barn from anywhere to run the Windows demo program.

Dictionary with real words

The test case 7 in sample inputs contain a dictionary with real words. You can try and play with it with the demo programs. You can remove the last 'end' line from test case 7 so the demo program does not exit immediately. The word definitions are from Dictionary.com.

Errors in running demo program

The windows demo programs are compiled in HKUST virtual barn with MSVC compiler. If the programs cannot be run in your computer you can try using the Virtual barn.

Auto comparison of sample inputs and sample outputs

The sample inputs and sample outputs might use `\n, \r\n` as end of line. This might interfere with automatically copy-pasting the sample inputs into the console, or auto comparison of outputs using scripts. Whether `\r\n, \n` is preferred depends on your system, so you should manually check it with notepad++ whether the end of line used in the samples fit your use case. The sample outputs contain the "raw" outputs of the program, without any user inputs. For the "mixed" output containing also the user inputs for readability, you may use the demo program with the sample inputs to generate them.

End of Resources & Sample I/O

Submission & Grading

Deadline: 26 November 2022 Saturday HKT 23:59.
Submit a zip file to ZINC. Compress the single file `dictionary.cpp`, **not a folder containing `dictionary.cpp`**.

Grading Scheme

Your score for this PA will be the sum of all test cases in ZINC. The table below roughly shows the weight of each task. Remember the operator overloading functions have to be completed too. Partial (3-6) means the partial points you will get when Test Cases 3-6 are completed but Test Cases 3-10 are not fully completed. For the other test cases, there is no partial completion. The bonus can **only** be obtained when the other parts are 100%, and there is no memory leak. If there is memory leak in any task, there is a `x0.95` penalty for the total score.

The list below **does not mean** that you are guaranteed to get the stated points if you implement the required functions correctly. Wrong implementations of the rest of the functions may interfere with the program in uncertain ways, such as errors or memory leak. If a dispute regarding ZINC auto grading arises, the COMP2012H teaching team reserves the right to final decision.

Test Case	Weight
01-02 : Copy constructor, Copy assignment	30%
03-10 : Add node, Remove node, Find node, Filter dictionary, For each, Printing	30%
• Partial (03-06) : Add node, Remove node, Find node, Filter dictionary	20%
11-12 : Merge dictionaries - copy and move	40%
Memory leak penalty	x0.95
Bonus (13-14) : Efficiency of implementations	5%

The odd test cases are visible test cases from ZINC. You can view the inputs from sample IO. The even test cases are hidden test cases.

ZINC auto grading

For test cases 01-02 and 11-14, the odd test case is of the same format as the even test case, but they are different. Each of 01-02, 11-12, 13-14 runs the class `Tasks` in `Tasks.cpp` in place of the main program.

For test cases 03-10, the odd test cases are in different format from the even test cases. The odd test cases contain a random permutation of the input from the sample English dictionary (see sample I/O), with some query of the dictionary. The even test cases contains another tree, and elements are to be deleted from the tree. Sample even test cases are provided, while the actual test cases are different, but with the same format.

Your score in ZINC is a 6 digit number. The number starting from the 3rd digit from the right would be your raw score before factoring in the memory leak penalty and bonus marks, out of 1000. For example, 100016 would mean 100%, while 50008 would mean 50%. The first two digits from the right would be the sum of memory leak completions and bonus tasks. Your final grade will be computed using the aforementioned grading scheme with this info.

Auto grading only functions (important)

To facilitate separate grading, some functions in `Dictionary` are copied to the `Tasks` class. Of course those implementations are removed in the skeleton code. The copies in the `Tasks` are provided in ZINC, but they will not be revealed. To test your program, you ***have to*** implement your own functions for `zinc_replacement_` as the corresponding functions in `Dictionary`. You can use CTRL+F on `Tasks.cpp` to find all the instances where the functions starting with `zinc_replacement_` are called.

End of Submission & Grading

Frequently Asked Questions

Q: My code doesn't work, here it is, can you help me fix it?
A: As the assignment is a major course assessment, to be fair, you are supposed to work on it by yourself and we should never finish the tasks for you. We are happy to help with explanations and advice, but we are **not allowed** to directly debug for you.

Q: There seems to be a compilation error for the unit `Tasks.cpp`. How do I fix it?
A: A student in Piazza pointed out there was a typo in the skeleton code version of Tasks.cpp. The skeleton code is updated on 15:20 9 Nov. If you have already written code using the previous version of the skeleton code, you can modify Line 12 of Tasks.cpp from `d.print_all_elements();` to `d.print_all_elements(cout);`

Q: My code works locally but not on ZINC. Why does this happen?
A: According to *Auto grading only functions (important)* in Grading, ZINC replaces some of the codes in Tasks.cpp with the 'official' implementations for separate grading. The reason that your code works locally but not on ZINC is due to the differences in implementation of those functions. Some implementations require some aspects of the tree structure, while others don't. Your functions should make sure (see important message in Tasks) the tree structure is kept **entirely** correct (see description above). As a hint, you could debug your code by accessing a 'processed' dictionary directly with various methods, such as going from top to bottom, or going from leaves to root and so on.

Q: My code works locally but segfaults on ZINC. Why does this happen?
A: This is precisely the above problem.

End of FAQ